FPGAs for accelerating HPC engineering workloads: the why and the how

Keywords: *FPGAs, HPC, supercomputing*
Topic/Industry sector: *Innovative hardware*
Target audience: *HPC developers and code users*
Key codes/use cases involved: *Nekbone, Alya, MONC*
Organisations involved: *EPCC*
Contact for further questions: *Nick Brown, n.brown@epcc.ed.ac.uk*

Introduction

Running high performance workloads on Field Programmable Gate Arrays (FPGAs) has been explored but is yet to demonstrate widespread success. Software developers have traditionally felt a significant disconnect from the knowledge required to effectively exploit FPGAs, which included the esoteric programming technologies, long build times, and lack of familiar software tooling. Furthermore, for the few developers that invested time and effort into FPGAs, from a performance perspective the hardware historically struggled to compete against latest generation CPUs and GPUs when it came to Floating Point Operations per Second (FLOPS).

In addition to significant developments in FPGA hardware of the past several years, there have also been large improvements in the software eco-system. High Level Synthesis (HLS) is a key aspect, not only enabling developers to write code in C or C++, and for this to be synthesised down to the underlying Hardware Description Language (HDL), but also allowing for programmers to reason about their code on the FPGA at the algorithmic level. Moreover, AMD Xilinx and Intel, who are the two major FPGA vendors, have built a software environment around HLS, Vitis and Quartus Prime respectively. This not only automates lower level aspects but also provides some profiling and debugger support. Whilst significant advances, such developments are not a silver bullet in enabling easy exploitation of the technology, not least because whilst the tooling is reliably able to generate correct code, if such programs are still based upon the CPU code then this is seldom fast.

Consequently programmers must adopt a different style of programming when it comes to FPGAs, and an important question is whether FPGAs can follow the same trajectory in the HPC community as GPUs have over the past 15 years, from highly specialist hardware that demonstrates promise for a small number of applications, to widespread use. If such is to occur, then not only is support and leadership required from the FPGA community, but furthermore two general questions must be strongly answered, firstly *why would one ever choose to accelerate their HPC application on an FPGA compared to other hardware?* and secondly *how can we best design our high performance FPGA codes algorithmically so that they are fast by construction?*

In this white paper we use the lessons learnt during the EXCELLERAT CoE to explore these two questions and use five diverse HPC kernels to drive our discussion. These are:

- **MONC PW advection** kernel which calculates the movement of quantities through the air due to kinetic forces. MONC is a popular atmospheric model in the UK and advection represents around 40% of the runtime. It is stencil based and was previously ported to the AlphaData ADM8K5 in [1] and [2], and an Alveo U280 in [3].

- **Nekbone AX kernel** which applies the Poisson operator as part of the Conjugate Gradient (CG) iterative method. Nekbone is a mini-app which represents the principal computational structure of the highly popular Nek5000 application [4], and is also representative of many other Navier Stokes based workloads. This has been ported to both the Xilinx Alveo [5] and Intel Stratix FPGAs [6].
- **Alya incompressible flow matrix assembly engine** which accounts for around 64% of the model runtime for Alya benchmarks. Alya itself is a high performance computational mechanics code used to solve complex coupled multi-physics, multi-scale, and multi-domain problems and used extensively in industrial engineering simulation. This incompressible flow engine was ported to the Alveo U280 in [7].
- **Himeno benchmark** which measures the performance of a linear solve of the Poisson equation using a point-Jacobi iterative method. A popular benchmark, it has been ported to Xilinx Alveo U280 [8]and Intel [9] FPGAs .
- **Credit Default Swap (CDS) engine** used for calculating financial credit risk. Based on the industry standard Quantlib CPU library, we have ported this engine to the FPGA [10]. Furthermore, Xilinx have developed an open source version in their Vitis library [11].

Why FPGAs for HPC engineering codes?

Why would one ever choose to accelerate their HPC application on an FPGA compared to other hardware is a simple but critically important question that the community must answer. HPC developers already know, generally speaking, what sort of code properties lend themselves to running on CPUs or GPUs, and if FPGAs are to become more widespread in HPC then knowledge about exactly what sort of codes they suit, and benefits can deliver must be understood. Whilst modern FPGA hardware can provide a respectable level of floating point performance via the on-chip components known as Digital Signal Processing (DSP) slices, realistically the continual advances made in GPU and CPU technologies means that if one's code is computationally bound, then FPGAs will likely fall short of the performance delivered by these other technologies and as-such are not the best option.

There are, in our view, four major benefits to using FPGAs for HPC workloads from the perspective of the applications developer:

- **Non-compute bound codes**, where the code on the CPU is bound by either fetching from memory and/or other stalls by the micro-architecture.
- **Performance predictability**, where due to the lack of a black-box micro-architecture, the programmer can far more easily determine the realistic theoretical performance that their code should deliver
- **Power efficiency**, where the lower clock frequency and lack of hardware infrastructure common on other technologies, results in significantly reduced power draw. This is summarised for our case studies in Table 1 with more information in [3], [5], [6], [8], and [10] available for interested readers.
- **Efficient handling of smaller problem sizes**, where research [12] has shown that FPGAs can be especially competitive against other architectures for smaller problem sizes. This can significantly help with strong scaling, with FPGAs still providing performance where other architectures would already be experiencing diminishing returns due to overheads.

Non-compute bound codes

A significant proportion of HPC codes are, to some extent, bound by the fact that the compute units cannot be continually fed with data. Whilst there are numerous causes, for our purposes, it is enough to categorise these as **memory stalls** and **other micro-architecture stalls**. The former is where memory access, typically via DRAM, on is a bottleneck and data cannot be fetched fast enough. Such is due to one or both of exhausting external memory bandwidth and/or application memory access patterns which make poor use of the cache. Conversely, micro-architecture stalls represent situations where the backend slots of the CPU cannot remain adequately filled and can be caused by a variety of issues including heavy branch prediction misses, a lack of independent instructions to run concurrently, or exhausting non-floating point execution units.

CPU and GPU technologies attempt to tackle these issues in different ways, but the fixed general purpose nature of such architectures impose limits, which is where FPGAs can potentially provide an important benefit for HPC. The reconfigurable nature of FPGAs means that we can often arrange the logic in such a way to minimise such stalls. Taking memory stalls are an example, FPGAs enable a bespoke memory mechanism at the kernel level to be adopted which fully suits the application in question. It is important to stress that we are not talking about developing bespoke memory controller IP blocks here, but instead the programmer specialising their HLS application level code to pipeline data access most effectively by utilising the appropriate on-chip memory space and concurrently fetching data, performing any reordering, and computing. Rent's rule [13], which is concerned with the ratio between the complexity (e.g. number of gates) in a logic block and the number of external connections to that block, also applies favourably to FPGAs, where this ratio is typically an order of magnitude greater than CPUs or GPUs, resulting in much higher bandwidth into the logic.

Table 1 describes, for each of our case studies, the percentage of cycles undertaking useful computation, the percentage stalled due to memory issues, and the percentage stalled due to other micro-architecture issues. Our case studies are varied, and Table 1 also reports the ultimate speed up on the FPGA against the CPU. For all these case studies there is some performance benefit to running on the FPGA, and whilst there are clearly numerous factors at play that determine performance, there is a loose correlation between the best speed up on the FPGA being obtained when the CPU is excessively stalled. Whilst it might not seem surprising that if the CPU is performing badly then there is more opportunity for speed ups on the FPGA, our argument is that it is precisely the bespoke nature of the architecture provided by FPGAs that enable such stalling issues to be ameliorated. Interested readers can refer to [2], [3], [5], [6], [8], [7], and [10] for more information about our individual case study performance.

Case Study	CPU Cycles computing	CPU Cycles stalled due to memory	CPU Cycles stalled due to other	Speed up on FPGA	Times FPGA more power efficient than CPU
MONC PW advection	48%	17%	35%	3.92	31.06
Nekbone AX kernel	27%	57%	16%	4.44	10.86
Alya matrix assembly	57%	32%	11%	3.45	6.14

Himeno bench-mark	62%	11%	27%	2.31	2.87
CDS engine	72%	1%	27%	1.37	4.50

Table 1 - Summary of case study performance on the CPU, ultimate speed up on the FPGA, and the number of times the FPGA is more power efficient than the CPU. All runs completed on Intel Xeon Platinum Cascade Lake 8260M CPU and Alveo U280 FPGA.

Performance predictability

Calculating a realistic performance figure that a kernel should deliver on a CPU or GPU is very difficult from the code alone due to the black-box nature of the micro-architecture. By contrast, the transformations performed by HLS to synthesise C or C++ to the HDL layer are transparent and discoverable enough such that one can more accurately predict floating point performance on an FPGA from code. A common error is for developers to calculate hardware peak performance and use this as context for their application's performance. But such estimates involve numerous assumptions and can be highly inaccurate (for example see the discrepancy between *Rpeak* and *Rmax* in the Top500!) Instead, a far more accurate figure can be determined on the FPGA by focusing on the code level.

An example of this is illustrated in Listing 1 which is a very simple kernel working with data from three arrays, *u_vals*, *v_vals*, and *w_vals*. The kernel involves two double precision floating point operations, a multiplication, and an addition. Because the loop is pipelined, these operations will run concurrently for different data, and as such from the code we can determine that per clock cycle there will be two double precision floating point operations. Multiplying this by the clock frequency, we calculate the number of floating point operations per second, for instance if the FPGA is running at 300 Mhz then the code in Listing 1 can obtain 600 MFLOPS.

```
for (int j=0;j<NX;j++) {
#pragma HLS PIPELINE II=1
  double temp=u_vals[j] * v_vals[j];
  result[j]=temp + w_vals[j];
}
```

Listing 1 - Simple example HLS code

Equation 1 calculates such at-best theoretical performance for a kernel, where *fpc* is the number of floating point operations per cycle and *frequency* the clock frequency. Whilst it is realistic to expect that one could get close to this figure, there are often code implementation issues which require optimisation to do so. For instance, in the example of this section it might be that the *vals* arrays of Listing 1 are held in external HBM or DDR memory and there is overhead in accessing the data, which is causing the pipeline to stall. By considering this theoretical performance the programmer can obtain a clear idea of how much opportunity there is for optimisation, and whether the programming level effort being invested is worthwhile.

$$FLOPS = fpc * frequence$$

Equation 1 - Prediction of theoretical performance

This at-best theoretical performance is also very useful in providing an upper bound on the performance that an FPGA kernel will deliver. Irrespective how much optimisation of the code in Listing 1 is undertaken, at 300Mhz, 600 MFLOPS is the maximum performance. If, for instance, the CPU were achieving a much higher level of performance then the FPGA programmer would need to increase either the number of floating point operations concurrently operating per cycle, or the clock frequency. As an example, if the CPU delivered 10 GFLOPS for this code, then 17 instances of the loop would need to be operating concurrency (at 300 MHz) to match CPU performance.

How to write high performance FPGA codes

Due in part to their lower clock frequency, in comparison to other hardware FPGAs are typically less forgiving of poorly tuned code. Where, to obtain even adequate performance on the FPGA with HPC workloads, the programmer must structure their code in the correct manner. After selecting a suitable kernel, matching an application's bottlenecks with the benefits that can be delivered by the architecture as described previously, there are three rules for achieving good performance with HPC FPGA codes which are summarised below and are explored in more detail in this section:

1. Calculate the at-best theoretical performance of a design and ensure this is acceptable before writing the code. If not, then modify the design.
2. ***Design top-down*** by structuring the code to suit FPGAs by enabling separate parts of the kernel to run concurrently. This is important because FPGAs will only provide acceptable performance if one takes advantage of the massive amounts of on-chip concurrency. Whilst this is easily achievable for simple codes by loop pipelining, it is more difficult for non-trivial real-world kernels.
3. ***Optimise bottom-up*** by concentrating on individual code components to ensure that they are free from stalls and each stage can continually progress, consuming input data and generating results. The following are examples of common algorithmic anti-patterns which limit performance and should be avoided:
    a. External data accesses which cannot be read or written contiguously, as this is especially expensive on FPGAs
    b. Data is produced and consumed by different parts of the kernel in different orders, requiring careful consideration to handle efficiency and correctly
    c. Spatial dependencies where inputs to one iteration require results from previous

These points link to the benefits discussed in the previous section, where if a kernel is non-compute bound and excessively stalling on the CPU, then if an FPGA design meets the required performance (rule one), by adhering to rules two and three on the FPGA, the programmer has effectively solved their performance bottlenecks. Put simply, ***to obtain good performance on an FPGA then the golden rule is to keep the data flowing*** and this is what rules two and three aim to achieve.

Leveraging theoretical performance metric

Unlike other technologies, the transparency of FPGAs provides a significant advantage where the programmer can calculate a realistic theoretical performance measure and be confident that, given the appropriate code level support, they will achieve close to this. This is especially important for FPGAs because, due to the long associated build times, it is time consuming to experiment iteratively with code versions, so building upon a solid design is very beneficial. Our Nekbone AX kernel case study is an example of where, for the initial design, the at-best theoretical performance was calculated to be 6.9 GFLOPS. However, on a 24-core Xeon Platinum CPU the code was achieving almost ten times this level of performance, and as such even with a perfect implementation our initial FPGA design was never going to get near the performance required. Put simply, the initial design needed a rethink, either increasing the amount of concurrency and/or the clock frequency.

At this point the code was performing 23 floating point operations per cycle, and by refactoring the calculations, we increased concurrency so that each of the six matrix multiplications could perform 31 floating point operations per cycle. This increased the total number of floating point operations performed per cycle to 203, and at 300MHz resulted in a new theoretical performance of 61 GFLOPS. This is still slightly short of the 65 GFLOPS provided by the CPU and-so the clock frequency was raised to 400MHz, boosting the theoretical performance to 81.2 GFLOPS.

It should be highlighted this is at a single kernel level, and whilst ultimately four kernels could fit on the FPGA and run concurrently, high performance at the individual kernel level is a crucial foundation to such endeavours. This brings us to the other benefit of such a metric, where it can be used by the programmer to understand how well their code has been implemented, effectively whether code is stall free or not. This is especially important for FPGAs because, whilst vendors are making some progress, profiling tools provide little insight about performance inside one's kernel. Leveraging such metrics provide a clear indication about potential stalls (our third rule described above) and hence areas for improvement. For instance, with Nekbone the initial Von-Neumann based code achieved only 0.29% of the (6.9 GFLOPS) theoretical performance which indicated there was significant low hanging fruit for optimisation. Ultimately the optimised kernel achieved 95% of the theoretical performance (77.73 GFLOPS achieved from an 81.2 GFLOPS theoretical maximum), which indicated that it was performing well.

Designing top-down: Application Specific Dataflow Machine (ASDM)

It won't surprise readers that, due to FPGAs being founded on dataflow rather than Von-Neumann principals, to obtain best performance one must embrace this different way of programming. This addresses rules two and three described above, and whilst the HLS tooling is generally advanced enough to correctly synthesise C or C++ based Von-Neumann based code, this will likely result in significant stalling and low concurrency in the resulting execution for non-trivial codes. Such is illustrated by Table 2 which depicts the performance difference for each of our case studies delivered when moving from a Von-Neumann based version of the code running on the FPGA to a dataflow algorithm. There are some very significant performance differences, especially for those kernels that ultimately end up performing well and making best use of the FPGA compared to other technologies. This illustrates both the importance of embracing the dataflow approach, and the danger that the uninitiated might assume poor performance based upon their initial code without undertaking such optimisations. Moreover, the CDS dataflow engine we developed is 27.3 times faster on an Alveo U280 than Xilinx's version from the Vitis open source library [11] which they have developed closer to the Von-Neumann model and targeted at ease of integration but does not fully implement our second and third rules.

Case Study	**Runtime speed up between Von-Neumann and dataflow**
MONC PW advection	811.1
Nekbone AX kernel	3886.5
Alya matrix assembly	589.2
Himeno benchmark	111.3
CDS engine	49.3

Table 2 - Kernel level performance difference (runtime speed up) for each case study when moving from a Von-Neumann to dataflow based algorithm on Alveo U280 FPGA

To motivate discussions in this section, Listing 2 illustrates a very simple HLS code snippet where, in a single pipelined loop, data is being read from external memory (e.g. DRAM or HBM), used for computation, and the result written back to the external memory. Because such simple code can all reside in a single loop then the performance will be reasonable as it can be effectively pipelined and run concurrently for each element.

```
for (int j=0;j<NX;j++) {
#pragma HLS PIPELINE II=1
  double temp=external_input[j];
  double result=temp+10;
  external_output[j]=result;
}
```

Listing 2 - A simple HLS code where pipelining provides good performance but is limited for more complex kernels

However, combining all program elements into a single pipelined loop as in Listing 2 is only realistic for trivial codes, and more complex kernels require different loops and code facets. This includes each of our four case studies, whereby default such codes often work in phases as is illustrated in Listing 3. Whilst this example is contrived for simplicity, even though the individual loops are pipelined, the three phases themselves are running sequentially and violates our second rule. The sequential nature imposed means that the concurrency provided by the FPGA is not being exploited.

```
for (int j=0;j<NX;j++) {
#pragma HLS PIPELINE II=1
  internal_in[j]=external_input[j];
}
for (int j=0;j<NX;j++) {
#pragma HLS PIPELINE II=1
  internal_out[j]=internal_in[j]+10;
}
for (int j=0;j<NX;j++) {
#pragma HLS PIPELINE II=1
  external_output[j]=internal_out[j];
}
```

Listing 3 - HLS code running in three sequential phases which is often required for non-trivial kernels

Listing 4 illustrates a modified version of the code where the programmer has explicitly adopted the dataflow paradigm by splitting their code into three separate functions *read_data*, *compute*, and *write_result*, which are connected via streams. Whilst this has increased the code complexity, they are running concurrently and making much better use of the FPGA resources, continually streaming data from one to the next. Put simply, when the programmer specialises their code for the FPGA, they are effectively moving closer to the dataflow paradigm, but this is not necessarily obvious or helped by the fact that they are still using an imperative-based language.

Thinking about this conceptually, we can draw parallels to activities in the 1970, 80s and early 90s, around general-purpose dataflow CPU architectures where code is decomposed into distinct concurrently executing stages, and these connected, such that data is continually flowing between them. This matches closely with the ideas behind the code in Listing 4, where an FPGA kernel transforms data whilst it is flowing and the reconfigurable nature of FPGAs enables programmers to develop a bespoke dataflow machine which is specialised for each application. Ultimately this means that we can design the constituent components of the dataflow machine (the stages, streams connecting them, and usage of memory) in an entirely application specific manner. Consequently it is our proposition that viewing one's FPGA code as an *Application Specific Dataflow Machine (ASDM)* is helpful from the design perspective, and it is a way of thinking about the design of one's code to comply with our second rule. Some FPGA programming technologies, such as Maxeler's MaxJ, CAL, and MATLAB's Simulink more readily expose the dataflow abstraction than others, but HLS C/C++ is most

```
void run_code(double * external_input, double * external_output) {
  hls_stream<double> in, out;
#pragma HLS STREAM variable=in depth=1
#pragma HLS STREAM variable=out depth=1

#pragma HLS DATAFLOW
  read_data(external_input, in);
  compute(in, out);
  write_result(out, external_output);
}

void read_data(double * external_input, hls_stream<double> * in) {
  for (int j=0;j<NX;j++) {
#pragma HLS PIPELINE II=1
    in.write(external_input[j]);
  }
}

void compute(hls_stream<double> * in, hls_stream<double> * out) {
  for (int j=0;j<NX;j++) {
#pragma HLS PIPELINE II=1
    out.write(in.read()+10);
  }
}

void write_result(hls_stream<double> * out, double * external_output) {
  for (int j=0;j<NX;j++) {
#pragma HLS PIPELINE II=1
    external_output[j]=out.read();
  }
}
```

Listing 4 - HLS code moved to dataflow approach, enabling parts to run concurrently (conforming with our second rule)

common and irrespective a dataflow view should be a first class concern to enable fast by construction FPGA codes.

Each of our use-cases has adopted this ASDM abstraction, and to explore some of the reasons behind these performance differences we highlight the ASDMs for the Nekbone use-case. Figure 1 illustrates the ASDM design of the Nekbone AX kernel, where each box is a separate dataflow function and these are connected by data streams (a more complex version of Listing 4). This meets our second rule of all stages running concurrently, but a challenge is that the *U* external input data is consumed in different orders by each of our first three matrix multiplications, and likewise the order in which data is generated by the local accumulation of values is different from the order in which it is consumed by the last three matrix multiplications. Whilst there are a number of possible solutions, the optimal one adopted here in order to *keep the data flowing* is to leverage the internal FPGA memories (e.g. BRAM, UltraRAM) in avoiding such overheads. Effectively we are tailoring the on-chip memory to suit our application, rather than on a CPU or GPU having to use it in the way prescribed by the hardware designer. This on-chip memory is arranged in a ping-pong data buffering style, where the Nekbone kernel of Figure 1 is running in separate phases, the first reading data contiguously and filling three internal buffers (one for each matrix multiplication). The second is then reading from these internal buffers in the order required by each specific matrix multiplication. As these are ping-pong buffers

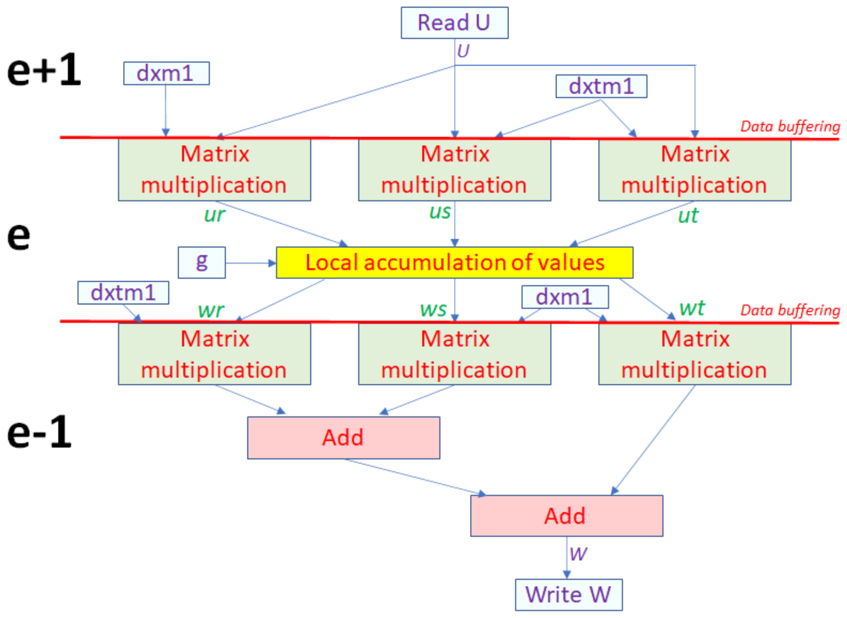

Figure 1 - Application Specific Dataflow Machine (ASDM) design of NekBone AX kernel

then each buffer has two copies which enables both phases to run concurrently, the first phase is fetching data for the next element (e+1) and the second phase computing on data for the current element (e).

This same solution can be adopted to solve the other issue where intermediate results are generated in a different order from which they are consumed by the last three matrix multiplications. This is achieved by adding a third concurrent phase to the kernel (e-1). The benefit of the ASDM approach is that the stages are running concurrently by design (rule two) and the programmer is continually questioning whether data is moving or stalled. By focusing on this algorithmic level, mitigations can be implemented which are likely not obvious based on a Von-Neumann view. Effectively by following our second rule has resulted in a kernel design on the FPGA which means that all constituent components can run concurrently. Furthermore, this view is portable between architectures, where running on an Intel Stratix-10 and using OpenCL channels to connect dataflow regions, we achieved performance within around 15% of the Alveo U280.

Optimising bottom-up: Individual code level concerns

Once the design has been organised into the top-down ASDM view that we have described, then programmers must focus on optimising the individual regions that make up each dataflow stage at a code level. These optimisations ensure that data can continually flow through each dataflow stage, ideally for progress to be made at each cycle. Whilst these code level techniques are numerous and beyond the scope of this white paper, although interested readers can refer to [14], [15], and [16], there are some generalisations that can be drawn which are highlighted here.

There are two general approaches that one can use to ensure that the data is kept flowing at the bottom-up code level, firstly pipelining and secondly loop unrolling. These are illustrated in Listing 5, where pipelining (left hand code) will aim to run the constituent components of the loop concurrently for different iterations. The argument *II=1* to the *PIPELINE* pragma sets what is known as the *initiation interval* and this instructs the tooling to inject a new loop iteration into the pipeline each cycle, for example setting this to two would inject a new iteration every other cycle. The right hand example of Listing 5 illustrates loop unrolling, where copies (in this case 4) of the loop are made and run concurrently, processing different iterations in parallel to improve performance.

```
for (int j=0;j<NX;j++) {
#pragma HLS PIPELINE II=1
  double d=x*y;
  double j=d*z;
  double p=d*j;
  result=p;
}
```

```
for (int j=0;j<NX;j++) {
#pragma HLS PIPELINE II=1
#pragma UNROLL factor=4
  double d=x*y;
  double j=d*z;
  double p=d*j;
  result=p;
}
```

Listing 5 - Illustration of loop pipelining and unrolling

For performance it is important that data is continually flowing which most often means that the pipeline's initiation interval should be one. However numerous code level facets can impact the ability to achieve this, and these are illustrated in Listing 6. The left-hand example is accumulating a double precision floating point value and the challenge is that there is a dependency from one iteration to the next on this double precision floating point number, but it takes multiple cycles (eight on the Alveo U280) to complete a double precision addition. Consequently, the initiation interval must be 8, as the calculation must have completely finished for one iteration before the next starts to process. The right hand side example illustrates dependencies on memory, where *external* is some external memory port and there is a maximum of one access per cycle on these. When the loop is pipelined then both lines will execute concurrently, for different iterations, and consequently it requires two accesses per cycle which is not supported. Therefore, the initiation interval here will be two, and the same is also true when using internal on-chip memories which are at-most dual ported.

```
double val=0;
for (int j=0;j<NX;j++) {
#pragma HLS PIPELINE II=1
  val=val+external[i];
}
```

```
for (int j=0;j<NX;j++) {
#pragma HLS PIPELINE II=1
  double a=external[i];
  double b=external[i+1];
}
```

Listing 6 - Illustration of spatial and port dependencies that will lower the initiation interval

In the previous section we highlighted that, for our Nekbone case study, data read from *U* is consumed in different orders for each of the first three matrix multiplications and how we addressed that. Another potential solution would have been for each of these separate matrix multiplications to read the external data concurrently in the order that they required. However, from a bottom-up perspective this would be sub-optimal for two reasons, firstly this would require three reads on the memory port whereas only one per cycle is supported (the issue highlighted in Listing 6 right-hand code) and secondly two of the three reads would be non-contiguous. On the FPGA non-contiguous external memory accesses must be avoided because, as described in [2], such external data accesses impose significant performance overheads due to requiring expensive read requests (over 20 cycles) for each individual element rather than a single read request for the entire block of data in the contiguous case. This illustrates one of the challenges of designing codes for FPGAs, because whilst we have described the top-down design and bottom-up optimisations of rules two and three separately, effectively as independent steps, in reality there is a feedback cycle and they are far more interlinked. Insights and activities from rule three feedback to rule two and a design will evolve based upon activities undertaken at both these viewpoints.

The purpose of this section has not been to describe in detail how to undertake bottom-up code optimisations, but instead to give a brief view of some of the challenges involved and the approach developers must adopt when porting their codes to the architecture. For more detail on this point then the interested reader can refer to our case-studies, for instance in the MONC PW advection kernel a shift-buffer was adopted to match the requirements of the memory, in Nekbone we partitioned the memories to support multiple accesses, in the Alya matrix assembly code the ordering of operations modified, and in the CDS engine the order of operations changed to fix spatial dependencies. For such details readers are pointed to [3], [5], [10], and [7].

Conclusions and outlook

In this white paper we have explored the role of FPGAs for accelerating HPC codes, identifying the three main areas of benefit that the technology can deliver over and above other hardware. We have argued that, whilst the tooling has significantly developed in the past decade, pursuing a Von-Neumann model of computation will seldom result in good performance. Instead, one must embrace the dataflow paradigm to fully benefit from an FPGA, and it is especially helpful to consider one's code as an *Application Specific Dataflow Machine (ASDM)*.

We have explored three rules for HPC development on FPGAs, with the overarching objective being to *keep the data flowing* by ensure that there is a sound initial design, all facets of code can run concurrently across the FPGA, and for data to be moving continually. When describing the benefits that FPGAs can provide to HPC it might be surprising that we omitted the flexibility of floating point arithmetic. Whilst such techniques are promising we have found on current generation FPGAs, because such IP cores are pipelined anyway, given a good dataflow design then the performance and energy benefits are limited [17] with precision lower then 32-bit. However next-generation hardware technologies are becoming available that combine specialist vectorisation engines with the reconfigurable fabric, such as Xilinx's Versal ACAP with AI engines and Intel's Stratix-10 NX with AI tensor blocks. This could significantly accelerate such reduced precision floating point and fixed-point operations and should be watched by the community.

Whilst HLS has lowered the barrier to entry in enabling *correct-by-construction* code on FPGAs, developing high performance code is another question entirely and as we have described here is still a manual undertaking requiring significant experience and expertise. There is opportunity to further develop the FPGA programming languages, frameworks, and compilation technologies to better suit writing *fast-by-construction* codes for FPGAs, where specialist dataflow languages or Domain Specific Languages (DSLs) might be a potential solution. There is a keenness in the HPC community to develop unified programming abstractions which cover CPUs, GPUs, and FPGAs, and it remains an open question whether we can ever expect to write performance portable codes between such diverse architectures.

Acknowledgements

The authors of this white paper would like to thank the ExCALIBUR H&ES FPGA testbed for access to compute resource used in this work. This work was funded under the EU EXCELLERAT CoE, grant agreement number 823691

References

[1] N. Brown, “Exploring the acceleration of the Met Office NERC Cloud model using FPGAs,” in *ISC High Performance*, Frankfurt, 2019.

[2] N. Brown and D. Dolman, “It’s all about data movement: Optimising FPGA data access to boost performance,” in *IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, Denver, 2019.

[3] N. Brown, “Accelerating advection for atmospheric modelling on Xilinx and Intel FPGAs,” in *IEEE Cluster workshop on FPGAs for HPC*, 2021, Online.

[4] P. Fischer, J. Lottes and S. Kerkemeier, “Nek5000 website,” 2022. [Online]. Available: http://nek5000.mcs.anl.gov. [Accessed 26 05 2022].

[5] N. Brown, “Exploring the acceleration of Nekbone on reconfigurable architectures,” in *IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, Online, 2020.

[6] M. Karp, A. Podobas, N. Jansson, T. Kenter, C. Plessl, P. Schlatter and S. Markidis, “High-Performance Spectral Element Methods on Field-Programmable Gate Arrays: Implementation, Evaluation, and Future Projection,” in *In 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Online, 2020.

[7] N. Brown, “Porting incompressible flow matrix assembly to FPGAs for accelerating HPC engineering simulations,” in *IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, St. Louis, 2021.

[8] N. Brown, “Weighing up the new kid on the block: Impressions of using Vitis for HPC software development,” in *In 30th International Conference on Field Programmable Logic and Applications (FPL)*, Online, 2020.

[9] Y. Sato, “Evaluating reconfigurable dataflow computing using the himeno benchmark,” in *IEEE International Conference on Reconfigurable Computing and FPGAs*, 2012.

[10] N. Brown, M. Klaisoongnoen and O. Brown, “Optimisation of an FPGA Credit Default Swap engine by embracing dataflow techniques,” in *In IEEE Cluster workshop on FPGAs for HPC*, 2021, Online.

[11] Xilinx, “Vitis Accelerated Libraries,” 01 04 2022. [Online]. Available: https://github.com/Xilinx/Vitis_Libraries. [Accessed 26 05 2022].

[12] J. Menzel, C. Plessl and T. Kenter, “The Strong Scaling Advantage of FPGAs in HPC for N-body Simulations,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS),* vol. 1, no. 15, pp. 1-30, 2021.

[13] B. Landman and R. Russo, “On a pin versus block relationship for partitions of logic graphs,” *IEEE Transactions on computers 100,* pp. 1469--1479, 1971.

[14] Intel, “Intel High Level Synthesis Compiler: Best Practices Guide,” Intel, 2022. [Online]. Available:

https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/ug-hls-best-practices.pdf. [Accessed 26 05 2022].

[15] Xilinx, “Best Practices for Acceleration with Vitis,” Intel, 2022. [Online]. Available: https://docs.xilinx.com/r/2021.2-English/ug1393-vitis-application-acceleration/Best-Practices-for-Acceleration-with-Vitis. [Accessed 26 05 2022].

[16] J. Hrica, “Floating-Point Design with Vivado HLS,” 20 09 2012. [Online]. Available: https://docs.xilinx.com/v/u/en-US/xapp599-floating-point-vivado-hls. [Accessed 26 05 2022].

[17] M. Klaisoongnoen, N. Brown and O. Brown, “Low-power option Greeks: Efficiency-driven market risk analysis using FPGAs]{Low-power option Greeks: Efficiency-driven market risk analysis using FPGAs,” in *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2022)*, Tsukuba, 2022.

The EXCELLERAT project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 823691.