Excalibur H&ES Interim Report

This project was designed to improve the use of FPGAs in HPC codes in research by enabling the effective use of a novel testbed, including multiple FPGAs from different vendors, and providing software support for new development or the porting of existing codes to FPGAs. This report details the development of the software associated with the testbed.

We have chosen two major subprojects that we believe will enable a range of users wishing to exploit FPGAs: a Markov Chain Monte-Carlo (MCMC) application with wide-ranging applications in statistics, implemented on both the FPGA and CPU for comparison, and an iterative solver library, providing simple access to cross-platform, FPGA-optimised iterative solvers. These two projects have helped inform the development of good practice guidance and workshops which will be finalised near the end of the project in August.

1 Use-cases and preliminary results

1.1 Markov Chain Monte-Carlo application

Markov Chain Monte-Carlo methods are widely used in scientific analysis as a form of parameter inference. For a parameterised model, these methods calculate the posterior distribution of parameter values, given a set of observed or computed data-points. They generally involve generating hundreds of thousands, or more, samples from some parameter space; for each sample a (potentially costly) likelihood and prior function must be evaluated. As is common in scientific applications, our likelihood function is modelled as a multivariate Gaussian. The evaluation of this function has two potential bottlenecks, depending on the size of the matrices involved: the generation of the covariance matrix from the covariance function ($O(N^2)$ in complexity, where N is the number of data-points and the covariance matrix is $N \times N$), and the solution to the matrix-vector equation ($O(N^3)$). This linear solver step, which includes a Cholesky decomposition, is therefore expected to be the dominant factor for large matrices.

Development of the MCMC implementation has been two-pronged: the CPU implementation using C++ with an LAPACK back-end for matrix solver methods, and an FPGA implentation in C and C++, utilising the Vitis HLS tools and methods from the Vitis libraries. There is also a python script for generating data for test-cases.

The CPU and FPGA versions in many respects mirror one another, although the FPGA version diverges considerably from the CPU code where necessary for optimisation. Presently, though both codes are working correctly for test problems, the FPGA code lags behind the CPU implementation in performance. FPGA implementation must be highly optimised to provide satisfactory performance due to the slow clock speed of the FPGA (around six times slower than a typical CPU); potential problems currently being investigated include under-utilisation of parallelisation or inefficient use of accesses to slower memory impeding efficient pipelining. These improvements must be balanced with the availability of resources in the FPGA fabric, particularly for large matrices where data may be forced into slower memory banks.

1.2 Iterative solver library

The iterative solver library (currently in an early stage) is being developed partially (potentially wholly) in a novel, high-level, dataflow-focused extension to Python: data-centric Python or DaCe. Iterative solvers were chosen specifically due to their wide-spread use in HPC and in multiple fields of science. Existing,

lower-level implementations of solvers (including FPGA-specific optimisations) can inform the development of this higher-level, user-focused library. Beyond being of direct use to scientific developers, this library will provide useful performance data on various solver algorithms which can inform future development of FPGA-accelerated iterative solver implementations in existing, widely-used numerical libraries such as PETSc and SciPy.

1.3 Development of teaching materials and good practice guidance

Near the completion of the two sub-projects listed above, we will begin the collation of good practices and general guidance for FPGA development; these will be presented both as documentation and as two workshops. We currently plan to target one workshop towards an HPC audience (that is, developers who are experienced with HPC or maintain existing HPC codes), and will focus on lower-level software stacks and optimisations (such as C++ with Vitis HLS). The other planned workshop will target researchers wishing to explore the potential of FPGAs in their work and will likely present DaCe as the preferred development environment.

2 Review of tooling in FPGA development

2.1 Intel OpenCL and SYCL

In lieu of an Intel FPGA (which has been made available to us recently), the Intel FPGA devCloud provided the integrated hardware & software platform to investigate Intel's FPGA development environment. While both OpenCL and SYCL are language extensions for use in programming heterogeneous systems, SYCL is higher-level and generally more accessible.

The developers identified and underwent two training and development courses:

- Introduction to OpenCL for FPGAs on the Coursera MOOC platform
- Pre-recorded webinars and videos from Intel's FPGA academic program

The Intel implementation of the SYCL specification, known as Data Parallel C++, is part of their OneAPI project. In SYCL, the host and kernel code are typically written in the same C++ file, and compiled together into a single executable. The platform detection routines are available as C++ APIs via the device selector and command queue handler classes. The developer experience was that the overloaded methods in these classes provided sufficient flexibility to implement arbitrary device-selection logic (e.g. a hierarchy of preference of accelerators to use) with a clear, readable, and succinct syntax. As an alternative to manual memory movement, SYCL2020 features Unified Shared Memory (USM) where any pointer allocated on the host is also a valid pointer on the device. This is a useful tool in the porting of existing CPU/GPU SYCL codes to FPGA architectures. Additionally, SYCL provides ways to interface with OpenCL, providing a clear migration path for existing OpenCL codes.

The performance of SYCL against native hardware description languages or finely controlled OpenCL code needs to be carefully evaluated, although its enhanced portability and more radid development cycle is an attractive feature even if it requires some trade-off in performance. Overall, we believe that the SYCL standard greatly lowers the accessibility barrier for FPGA computations. In general, OpenCL has seen poor adoption in GPU programming, with Nvidia better supporting CUDA and AMD strongly supporting HIP. While Intel have used OpenCL as their framework of choice for programming FPGAs, they appear to be encouraging the use of SYCL instead. As a result we cannot recommend OpenCL as a future-proof framework for heterogeneous computing.

2.2 Xilinx HLS

High Level Synthesis (HLS) is the framework used when developing higher level code for Xilinx FPGAs, as an alternative to using hardware description languages such as VHDL. Code is instead written in C or C++ (with some language restrictions) augmented with `#pragma HLS` statements, which is converted into a hardware design by the Vitis compiler. This allows scientific programmers familiar with C++ to begin experimenting

with FPGAs with relative ease. The programmer can specify certain optimising transformations (e.g. loop pipelining or unrolling) or memory placements (e.g. DDR, URAM) using the `#pragma` statements; other optimisations may be applied automatically by the compiler.

As mentioned above, the slow clock speeds of the FPGA mean that solutions must be strongly optimised in order to see a benefit, and that not all problems will admit the degree of parallelisation necessary to beat a CPU implementation. FPGA optimisation generally relies on pipelining data: individual elements moving through a multi-step process one after the other in such a way that the second step of one element overlaps with the first step of the next element. Perfect pipelining necessitates that the steps in this process do not conflict by requiring the same resources. Changes within the code can cause unanticipated knock-on effects, such as a change to the kind of memory to which a variable has been allocated, which can then drastically change the efficiency of a pipeline and necessitate changes to the parallelisation. Furthermore, because the resources on any given device differ, solutions may need to be significantly modified to achieve peak performance on different devices.

Developing efficient code can be aided by using the Vitis HLS GUI application, which allows one to analyse compiled hardware emulations of one's code. This can include estimates of latency, iteration intervals (crucial for pipelining), and, perhaps most importantly, error inducing problems such as timing violations which will cause the hardware solution to fail. Changes to the code can be implemented at this stage, and the effect on the solution (for example, how many calculations are done in parallel) can be confirmed and visualised. The tool also provides guidance for some problems that is is able to identify automatically, which can be particularly helpful for new developers.

Drawbacks to development using HLS include the restriction to Xilinx devices, and a generally steep learning curve. Although the tools allow software developers to use C++ to develop their solutions, the development of efficient code for FPGAs still requires gaining understanding of an architecture which is very different from common CPU and GPU architectures. The connection between the software code and the final hardware product can be obscured by the high level approach and degree of automation. Xilinx provides tutorials and documentation for their tool set, but these tend not to feel comprehensive for a beginner, and error messages from the compiler can appear opaque and confusing for new users. Lengthy compilation times, ranging from tens of minutes for harware emulation to many hours for the full hardware solution, are a barrier to experimentation. It is also worth bearing in mind that the Vitis libraries provided by Xilinx, even in their most up to date versions, can themselves cause errors through timing violations or deprecated statements, and thus cannot necessarily be relied upon as an external resource without modification and re-packaging with your solution.

2.3 Dace

DaCe was chosen as it is significantly higher-level than the HLS software stack used to develop the MCMC code. This allows the team to compare general ease-of-use with alternative software stacks and provides useful experience which will inform the development of the training materials. Additionally, DaCe targets both Intel and Xilinx FPGAs, allowing direct performance comparison using a single source. The framework translates annotated Python code directly into HLS (for Xilinx devices) or OpenCL (for Intel) and uses the vendor's supplied toolchain to compile to hardware. As such, there is a negligible performance penalty for using such a high-level tool, at least in reported benchmarks. Indeed, it is possible to extract the translated (and optimised) low-level code and use that as the basis for further lower-level development and optimisation. Given the high barrier to entry of traditional FPGA development environments (even those using higher-level abstractions than, for example, RTL or Verilog), DaCe's code generation provides an alternative way to learn a tricky-to-use software stack.

Initial use of DaCe has shown it to be much more user-friendly than lower-level FPGA development environments, requiring less boilerplate code for similar functionality and performance. However, due to DaCe being high-level and a relatively young project, it offers less flexibility and transparency. This may be improved as the tool develops and we are in direct contact with DaCe developers to provide feedback and receive guidance. This has so far proved very useful.

3 Plan for remaining work

- MCMC code finished by end of June
- Iterative solver algorithms chosen and initial development started end of June
- Iterative solver development completed by mid August
- Good practice guidance and workshop development begins start of August, completed end of August