

Feasibility Study of Porting a Particle Transport Code to FPGA

Iakovos Panourgias¹(✉), Michele Weiland¹, Mark Parsons¹,
David Turland², Dave Barrett², and Wayne Gaudin²

¹ EPCC, James Clerk Maxwell Building, Peter Guthrie Tait Road, Edinburgh
EH9 3FD, UK

{i.panourgias, m.weiland, m.parsons}@epcc.ed.ac.uk

² AWE, Aldermaston, Reading RG7 4PR, UK

{David.Turland, Dave.Barrett, Wayne.Gaudin}@awe.co.uk

Abstract. In this paper we discuss porting a particle transport code, which is based on a wavefront sweep algorithm, to FPGA. The original code is written in Fortran90. We describe the key differences between general purpose CPUs and Field Programmable Gate Arrays (FPGAs) and provide a detailed performance model of the FPGA. We describe the steps we took when porting the Fortran90 code to FPGA. Finally, the paper will present results from an extensive benchmarking exercise using a Virtex 6 FPGA.

Keywords: FPGA · Particle transport · Wavefront sweep

1 Introduction

Chimaera-2 is a particle transport code based on a wavefront algorithm. The code has been developed and maintained by the UK Atomic Weapons Establishment (AWE) and is used primarily for benchmarking during procurement processes. Chimaera-2 is written in Fortran90 and MPI/OpenMP; it scales well to thousands of cores for large problem sizes and is used to calculate nuclear criticality. It is one of the highest priority application codes in the benchmark suite used by AWE for procurement of HPC systems.

This paper will discuss the programming strategies required to port a CPU-centric HPC code to Field Programmable Gate Arrays (FPGAs) by an application developer with no prior knowledge of FPGA/embedded programming. This feasibility study will provide a detailed performance model of the FPGA implementation of Chimaera-2 and we will show how the model can forecast the runtime of an FPGA ported application. A mini-app, which closely matches the Chimaera-2 code, was implemented and used to evaluate our porting strategy and the performance model predictions. Finally, we will describe the steps we took when porting parts of the Fortran90 code to FPGA using Maxeler's FPGA solution. The limiting factors of the algorithm will be discussed and we will provide solutions for future development.

1.1 Contributions

Although writing applications for FPGAs and embedded devices dates back to the 1990s and the difficulties and steep learning curve have been mentioned in research papers feasibility studies of using new methodologies have not been sufficiently explored. This paper is amongst the first to consider new abstract methods of porting HPC CPU applications to FPGA. Furthermore, this paper studies the feasibility of an application developer with no knowledge of FPGA or embedded computing being able to port a large and complex HPC application. This paper makes the following contributions:

- We discuss and evaluate a set of techniques which enable an application developer to predict the performance characteristics of an FPGA application;
- We describe the conceptual differences between CPU and FPGA programming;
- We describe the implementation of a mini-app that closely matches Chimaera-2 to test different porting strategies and to evaluate the performance model predictions.

2 Related Previous Work

This section will discuss previous attempts of porting a particle sweep transport code to accelerators. The Denovo code [1], produced by the Scientific Computing Group at Oak Ridge National Laboratory, serves a similar purpose to Chimaera-2 – it is used for advanced nuclear reactor design and as a benchmark code. Denovo also works on a three dimensional domain, but unlike Chimaera it is written in C++. Denovo has been ported to work with GPUs using CUDA for the three dimensional case with performances that achieve from 1 to 10 % of the peak GPU performance; however these are for three dimensional systems that parallelise more extensively, including groups (or energy levels), which expose additional levels of parallelism. Work by Gong et al. [2, 3] attempted to convert another three dimensional sweep code, Sweep3D¹, a benchmark code produced by the Los Alamos National Laboratory, to work on GPUs. Their results obtained overall performance speedups for a single NVIDIA Tesla M2050 GPU ranging from 2.56, compared to an Intel Xeon X5670 CPU, to 8.14, compared to an Intel Core Q6600 CPU with no flux fix-up (with flux fix-up on an M2050 was 1.23 times faster than on one X5670). In [4], Gong et al. go on to look at 2D problems using a “Discontinuous Finite Element – Sn” (DFE-Sn) method [5], using a 2D Lagrangian geometry. The performance speed-ups they obtain for simulations on an M2050 GPU range from 11.03 to 17.96 compared with an NTXY2D [6] on Intel Xeon X5355 and Intel Core Q6600. In their case the energy groups in the calculation are regarded as inherently independent and thus can be executed in parallel.

3 Chimaera-2

The full problem space for a Chimaera execution is a fixed uniform rectilinear 3D spatial mesh of grid points. All faces of the problem space are squares or rectangles.

¹ The code can be downloaded from <http://www3.lanl.gov/pal/software/sweep3d>.

For running in parallel using MPI, Chimaera performs a simple domain decomposition of this mesh as shown in Fig. 1. The front face is divided into N rectangles (with N being the number of MPI tasks) that are projected the whole way from the front face to the back face to form 3D elongated cuboids. These 3D domains are often referred as “pencils”.

Each iteration of the solver algorithm used in Chimaera-2 involves sweeping through the whole spatial mesh eight times, each of the sweeps starting at one of the eight vertices of the mesh and proceeding to the diametrically opposed vertex. As the boundaries between pencils are crossed, MPI message passing is used to transmit the updated data to adjacent pencils.

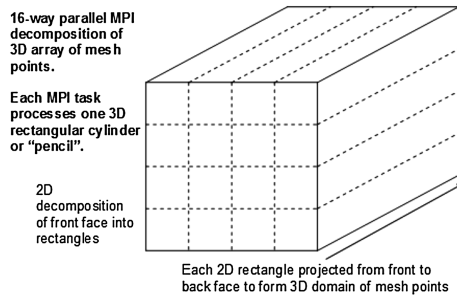


Fig. 1. Chimaera domain decomposition for a 16-way parallel run

Figure 2 shows a single pencil (MPI domain) with a 6×6 square cross section of mesh points. 10^3 geometry sizes can be used to test correctness; however, realistic problems are likely to be larger. 200^3 to 500^3 geometries are common and sizes over $1,000^3$ are considered very large. The sweep direction being illustrated starts at the front top-right vertex; therefore this MPI task will receive data from the pencil to its immediate right and from the one immediately above and transmit data to the left and downwards. Within the pencil, the order of computation is shown by the following code fragment in Code Listing 1.

Note that this assumes that the data in the 3D arrays are held in the same sequence as the sweep direction; that is that the front top-right element has the index (1,1,1) in Fortran syntax. With the data still in this sequence for other sweep directions, one or more of the loops end up being computed in reverse order. For example, “DO K = 1, KEND” might become “DO K = KEND,1,-1” when we compute sweep “Front, Top, Left”. This is because each of the J, K, and L loops exhibits sequential dependence where every iteration depends on the previous iteration and must be computed in the order corresponding to the sweep direction.

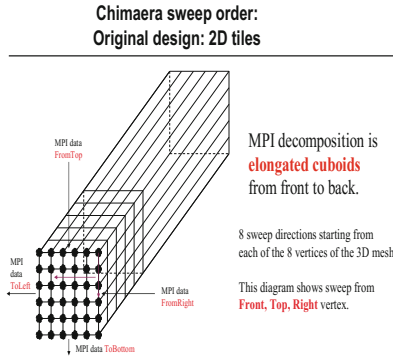


Fig. 2. Original 2D tile design, where computation order across the tile is a simple x and y pair of loops.

```

DO ISW=1,8          ! ISW: sweep direction
DO L=1,LEND        ! L: Front/Back index
  MPI calls
  -----Kernel
  DO K=1,KEND      ! K: Top/Bottom index
  DO J=1,JEND      ! J: Right/Left index
    DO M=1,ISNX   ! M is independent of the structure
      Computation
    ENDDO
  ENDDO
ENDDO
ENDDO
ENDDO
-----
ENDDO
ENDDO

```

Code Listing 1: Basic structure of algorithm.

Although all three spatial loops (i.e. J, K and L) have this sequential dependence, the order of nesting can be changed in any way without affecting the results (except for the complication that it has been chosen to perform the MPI communication after each iteration of the L-loop). The M-loop is concerned with the directions of the particles being tracked and is independent of the J, K and L loops. It does not have sequential dependence and can be logically nested at any point. With the above algorithm, the computational Kernel within the MPI layer processes a single 2D tile (indexed by K and J), which moves from front to back as the outer L-loop is iterated. The Chimaera-2 application is used for procurement purposes at AWE and has been maintained and optimised for the past 10 years. The algorithm has been optimised for serial, MPI, OpenMP and GPU runs.

3.1 Mini-app

Our goal was to port the computationally expensive Kernel of the Chimaera-2 application to the FPGA. However, in order to better understand the effects of porting different parts of the code using different techniques we implemented a mini-app. The mini-app uses less memory than the full Chimaera-2 application; however, the computational Kernel is identical to the full application showing the same data dependencies. It also uses the same number of arrays as input and output.

The mini-app does not implement 8 different sweeps; rather a single sweep (Front, Top, Left) is used. Furthermore, it does not use MPI or OpenMP. The initial values of the arrays are populated programmatically rather than reading an input file. The mini-app allowed us to experiment with different porting techniques and to run large simulations. Chimaera-2 uses more than 40 GBs of RAM for a 120^3 simulation; whereas, the mini-app only uses 12 GBs of RAM for a $1,000^3$ simulation.

4 Porting from CPU to FPGA

Porting to an FPGA historically required writing code in VHSIC Hardware Description Language (VHDL) or Verilog. VHDL and Verilog are hardware description languages (HDLs) and are used to describe the structure, design and operation of electronic circuits. Unlike an Application-Specific Integrated Circuit (ASIC), an FPGA is a semiconductor device that can be configured by the end user. FPGAs contain programmable logic blocks, that can perform complex Boolean logic functions or simple AND or XOR operations. These logic blocks can be connected in order to provide more complex functionalities. The logic blocks are made up of two basic components: flip-flops (FF) and lookup tables (LUT). Many logic blocks can be combined to create simple arithmetic operations (addition, multiplication) or higher math operations like *abs* or even trigonometry operations like *sin* and *cos*.

There is a growing interest in using FPGAs as energy efficient accelerators and new programming models designed to make FPGA programming more widely accessible are gradually emerging. The work presented in this paper uses Maxeler's Multiscale Dataflow solution to port Chimaera-2 to FPGA. This approach allows an application programmer to use a high-level (Java-like) language, called MaxJ, as an intermediary step before using the Maxeler tool chain to translate the code to VHDL. The VHDL files are then used by a set of third party vendor tools, which build the configuration bitstream that is uploaded to the FPGA. The FPGA used here is a Max3 solution which is based on a Xilinx Virtex-6 (V6-SXT475) FPGA with 24 GB RAM. Maxeler's solution allows an application to be split into three parts:

- Kernel(s), which implement the computational components of the application in hardware;
- Manager configuration, which connects Kernels to the CPU, engine RAM, other Kernels and other dataflow engines via MaxRing;
- CPU application, which interacts with the dataflow engines to read and write data to the Kernels and engine RAM.

- Dataflow variable type

FPGAs allow the representation of a number by using an arbitrary number of bits. It is possible to either use pre-defined primitive variable types (such as 32-bit integers or 64-bit floats) or define a custom data format. For instance, if it is known that an algorithm requires 48-bit precision a matching custom type can be created.

- Streams of data

Dataflow computing operates on windows into data streams. The data window can be held in the on-chip memory of the FPGA and thus minimises data transfer costs. We can use static or variable stream offsets to look back or forward into a stream of data. Variable stream offsets can be set once for the duration of a Kernel, or they can be modified per FPGA clock cycle.

- Loops/Counters

Loop statements are heavily used in High Performance Computing. Simple loops can be easily ported to an FPGA. A simple loop applies the same operation to a group of data elements. In an FPGA this translates to streaming data element through the same logic block. Unfortunately, not all loop statements are simple. In order to implement more complex loop statements on an FPGA it is necessary to keep track of where we are in the stream. Furthermore, many HPC applications use nested loops. We use counters to port nested or loops with boundary conditions.

- Data Dependency Loops

So far discussion has been limited to simple loops, where each iteration of the loop is using values from the incoming stream(s) and loop counters. However, many loops in HPC applications have dependencies and each iteration relies on the values calculated in previous iterations. In order to port these loops, it is necessary to either unroll the loops or create cyclic graphs. Unrolling loops is only possible if the number of iterations is known (or if it is known that the number of iterations will be N at most). Loop unrolling on the FPGA uses replicated hardware resources and the limit is therefore the size of the FPGA. Unfortunately in many HPC applications the number of iterations per loop will be large enough to consume all available hardware resources. Therefore the better option is to create a cyclic data graph, in which the output of the loop stream can be connected to the input stream. Thus, the output of a calculation is available as input.

- Scalar inputs and outputs

In addition to streaming input and output data between CPU and FPGA, it is also possible to transfer read-only configuration values at runtime.

- Boolean control logic

Since FPGA designs are built in advance, FPGA code must be able to handle all possible execution paths through the code. For every conditional “if/then/else” statement a multiplexer is created, which diverts the data streams (using multiplexers) to the correct logic block path.

5 Performance Model

Estimating the performance of a dataflow implementation is simpler than for a CPU implementation due to the static scheduling of the FPGA. The FPGA design is built out-of-band using the MaxCompiler and the FPGA vendor tools. The bitstream is then loaded on the FPGA and can be used by an HPC application. We can estimate the performance of an application with a known runtime in cycles (like Chimaera 2) executing an FPGA design. If the runtime is not known, it is still possible to model the performance of the application; as long as the application uses a regular pattern for reading and writing data to external interfaces. The FPGA has a set of well-defined external interfaces (PCIe and DRAM) with fixed performance capabilities. Furthermore, the FPGA does not create or handle any dynamic events, like interrupts. Also the FPGA does not use pre-emption for interrupting the kernels since only a specific set of kernels can be executed at one time. Finally, out of order execution and cache misses are not relevant on an FPGA. The FPGA design dictates the compute nodes that will be used and the data that will be computed.

Thus, the performance of an FPGA design depends on the following factors:

- Number of cycles per kernel
- Data read from and written to DRAM
- Data read from and written to PCIe
- Data read from and written to Network Interface

The total time that an application spends in a dataflow execution is computed by:

$$T_{Total} = \max(T_{compute}, T_{DRAM}, T_{PCIe}, T_{MaxRing}) \quad (1)$$

5.1 Model Compute

As mentioned earlier, the dataflow design uses static scheduling. Therefore $T_{COMPUTE}$ is the time that it takes the input data to run through the dataflow design:

$$T_{compute} = \frac{Cycles}{Frequency} \quad (2)$$

The frequency of the design determines the compute speed. For simple single pipe designs the number of cycles is equal to the number of inputs, i.e. 1. However, for multi-pipe designs the number of cycles is equal to the number of input divided by the number of pipes.

As an example, in order to compute the loops K, J and M (from Code Listing 1) we need to run for $K*J*M$ cycles. Thus, for the 10^3 simulation with $M = 36$, the FPGA will run for 3,600 cycles. Using a frequency of 150 MHz we estimate that the FPGA will run for 24 μ s. If we port the L loop, the FPGA will run for $L*K*J*M$ cycles. Thus, for the 10^3 simulation the FPGA will run for 36,000 cycles or for 240 μ s. However, the LKJ FPGA port will be called L fewer times than the KJ FPGA port. It is estimated that the overhead of calling the FPGA is close to 7 ms. As the size of the simulation

increases, the overheads of calling the FPGA many times for small runtimes also increases.

5.2 Model PCIe

PCIe is one of the three different ways of getting data into an FPGA. PCIe is a bi-directional link which can send and receive data in parallel without sharing bandwidth. To estimate the PCIe transfer rate we can use the following formula:

$$T_{PCIe} = \max\left(\frac{BytesIn_{PCIe}}{BandwidthIn_{PCIe}}, \frac{BytesOut_{PCIe}}{BandwidthOut_{PCIe}}\right) \quad (3)$$

The number of bytes transferred to and from the FPGA is known at compile time. For the Chimaera-2 application it is known that the KJ FPGA port is streaming $Bstream = K * J * M * sizeof(double) * number\ of\ array$ bytes to the FPGA. The values for K, J and M depend on the size and configuration of the simulation. The “number of arrays” value depends on the implementation and is equal to 8 for the KJ FPGA port. The naïve implementation copies redundant data to the FPGA. This approach simplifies the logic on the FPGA, but the cost of copying more data slows down the application. The optimised version uses more logic on the FPGA in order to re-arrange arrays which hold replicated data. For example, the naïve implementation copies three 2D arrays as three 3D arrays. The optimised implementation copies three 2D arrays.

We also know that the design is streaming out to the host Bstream Bytes. The naïve approach copies all output values to the host (even if some of them are temporary and not useful). The optimised version on the other hand only copies the minimum data back to the host that is needed for future iterations. Table 1 shows the two versions:

Table 1. GBs transferred for KJ FPGA port

| Size | Total naïve (GBs) | Total optimised (GBs) |
|-----------|-------------------|-----------------------|
| 10^3 | 0.00038 | 0.00014 |
| 60^3 | 0.01352 | 0.00405 |
| 500^3 | 0.93877 | 0.27142 |
| $1,000^3$ | 3.75509 | 1.08302 |

The ideal PCIe bandwidth is 1.8 GBs/s; however, for our calculations we also used a non-ideal value of 1.5 GBs/s. The non-ideal bandwidth reflects real life benchmarks using Chimaera-2 and the mini-app. Table 2 shows the PCIe transfer times in seconds. It can be seen that the optimised KJ FPGA port reduces the time spend on PCIe data transfers. The reason behind the performance improvement is that the optimised KJ FPGA port transfers less data to the FPGA. The naïve KJ FPGA implementation copies superfluous data, thus spending more time on the host in order to create the data arrays and spending more time when transferring the data to/from the FPGA.

Table 2. Model KJ FPGA port PCIe performance in seconds

| Size | Naïve In | Naïve Out | Optimised IN | Optimised OUT |
|-----------|----------|-----------|--------------|---------------|
| 10^3 | 0.00011 | 0.000080 | 0.000037 | 0.000034 |
| 60^3 | 0.00429 | 0.00321 | 0.00115 | 0.0011 |
| 500^3 | 0.29802 | 0.22357 | 0.07606 | 0.07472 |
| $1,000^3$ | 1.19209 | 0.89407 | 0.3032 | 0.29847 |

5.3 Model DRAM

The FPGA has two different types of memory: DRAM and BRAM. The BRAM memory is a very fast close to the FPGA logic chips memory of a very limited size. The BRAM on the FPGA is less than 10 MBs in size. The FPGA also has 24 GBs of DDR3 DRAM on-board. Like any DRAM, it shares read and write bandwidth on a single bus (unlike the PCIe bus, which has different bandwidths for read and write). The time spent on writing and reading to the DRAM can be expressed by the following formula:

$$T_{DRAM} = \frac{BytesIn_{DRAM} + BytesOut_{DRAM}}{Bandwidth_{DRAM}} \quad (4)$$

The performance of writing to and reading from the DRAM is also affected by two other factors: memory frequency and memory access pattern.

Memory frequency is chosen at build time. A faster memory frequency usually yields a higher bandwidth; however, a faster memory frequency makes the design harder to build due to the stricter timing requirements imposed by the faster frequency.

Memory access patterns play a more significant role to the DRAM performance. If we try to use a semi-random access pattern, e.g. one where the data is not located sequentially on DRAM, the performance will be very poor. The maximum bandwidth that can be achieved is expressed by the following formula:

$$Bandwidth = Bytes_{DIMM} \times No_DIMMS \times Frequency_{DRAM} \times Efficiency \quad (5)$$

“Bytes per DIMM” is 8 Bytes on the MAX3. However, it can be configured and set during the build phase of the design. The “No_DIMMS” is based on the actual FPGA hardware that is used (in our case this is 6 DIMMS). The frequency of the DRAM controller is also set during the build phase of the design. It can vary from 606 up to 800 MHz (for both read and write). The efficiency depends on the size of the transfer and the access pattern. Assuming a perfect efficiency and using the maximum frequency for both read and write operations a MAX3 DRAM subsystem can achieve a theoretical bandwidth of 38.4 GB/s. More modern FPGAs can achieve higher frequencies and are also able to accommodate more complicated designs.

Our implementation used a streaming approach rather than copying data to the DRAM. Thus, we will not use T_{DRAM} and $T_{NETWORK}$ in our calculations. The algorithm which we used did not require the use of DRAM, however in order to inform design choices it is useful to estimate the time spent on the DRAM. As long as we know the amount of required data we can model the performance of the DRAM subsystem.

6 Results

This section discusses the performance results of porting the mini-app and Chimaera-2 to FPGA. Due to the memory requirements of the Chimaera-2 code we were able to run simulations with geometry sizes up to 120^3 . The mini-app enabled us to run simulations with geometry sizes up-to $1,000^3$.

The CPU run-times were obtained on a workstation with an Intel Core i7-2600S CPU @ 2.8 GHz (3.8 GHz with turbo mode) and running CentOS release 6.6. The workstation had 32 GBs of DDR3 RAM and the mini-app and Chimaera-2 applications were using a single core. The Fortran90 code was compiled using the GNU Fortran compiler (v4.4.7) and the O3 optimisation flag. The C code was compiled using the GNU Compiler Collection compiler (v4.4.7) and the O3 optimisation flag.

Both the mini-app and Chimaera-2 code stream data to/from the FPGA. Hence, we did not use the on-board DRAM. The input data are streamed to the FPGA before the execution of the compute Kernel and the computed values are streamed back to the host. The amount of data copied to/from the FPGA depends on the port implementation (naïve or optimised) and the number of ported directional loops (KJ or LKJ).

6.1 Mini-app Naïve KJ Implementation

Using the performance model calculations it is possible to estimate T_{COMPUTE} and T_{PCIe} . Based on this model, we created two versions of the application: a naïve and an optimised implementation. The first 2 columns of Table 3 show the performance in seconds for streaming data (T_{PCIe}) to/from the FPGA using the naïve implementation and using 1.8 GB/s bandwidth. The third column shows the performance (in seconds) of running the FPGA at 100 MHz. The next two columns select the maximum of each phase (T_{PCIe} to/from and T_{COMPUTE}) and use it to model the performance of the naïve mini-app using either 1.8 GB/s and 1.5 GB/s PCIe rates.

Table 3. Mini-app (KJ) Naïve T_{PCIe} , T_{COMPUTE} , modelled and actual duration

| Size | To FPGA | From FPGA | Compute total | Duration 1.8 GB/s | Duration 1.5 GB/s | Actual duration | Actual duration* |
|-----------|---------|-----------|---------------|-------------------|-------------------|-----------------|------------------|
| 10^3 | 0.0008 | 0.0006 | 0.0736 | 0.0008 | 0.0008 | 0.00657 | 0.00453 |
| 100^3 | 0.895 | 0.5960 | 0.43 | 0.895 | 1.0747 | 1.1515 | 0.9213 |
| 500^3 | 111.80 | 74.51 | 48.50 | 111.80 | 134.16 | 124.56 | 115.37 |
| $1,000^3$ | 894.22 | 596.05 | 367.00 | 894.22 | 1,073.06 | 963.66 | 950.08 |

Table 3 shows the naïve implementation of the mini-app is PCIe bound. For small simulations the overhead of calling the FPGA multiple times makes the runtime compute bound. However, for simulations larger than 100^3 the application becomes PCIe bound. The last two columns of Table 3 show the actual duration of the naïve mini-app in seconds (for two different designs; using 100 MHz and 170 MHz speeds). It can be seen that increasing the operation speed of the FPGA by 70 % does not affect the duration of the mini-app; since it is PCIe bound. For example, for the $1,000^3$

simulation the 100 MHz design runs for 963 s; whereas the 170 MHz design runs for 950 s (a performance improvement of 1.3 %).

Furthermore, the performance model is able to accurately predict the actual performance of the application. For the $1,000^3$ simulation, the performance model predicts a duration between 894 and 1,073 s using a speed of 1.8 GB/S and a less favourable speed of 1.5 GB/s. The actual duration is 963 and 950 s (for the 100 MHz and 170 MHz designs). The actual duration corresponds to a ~ 1.65 GB/s PCIe transfer rate.

6.2 Mini-app Optimised KJ Implementation

The optimised mini-app application reduces the need for redundant data copies to/from the FPGA. In order to reduce the amount of data that needs to be copied it is necessary to implement more logic on the FPGA. Three of the input arrays are used as read-only memory and it was possible to reduce the size of the arrays from three dimensions to two. The three read-only arrays are accessed using a constant pattern. The pattern is not the same for each array; however, it is the same for the runtime of the application. We implemented the pattern on the FPGA using logic blocks and several fast memory blocks to store the data. The effect that this reduction has on T_{PCIe} can be seen in the first two columns of Table 4. The third column shows that the optimised version of the mini-app is compute bound. Since the optimised version of the mini-app is compute bound, increasing the operational speed of the FPGA by 70 % (from 100 MHz to 170 MHz) also improves the runtime performance of the mini-app.

Table 4. Mini-app (KJ) T_{PCIe} , T_{COMPUTE} , modeled and actual duration

| Size | To FPGA | From FPGA | T_{COMPUT} | Duration 1.8 GBs | Duration 1.5 GBs | Actual duration | Actual duration* |
|-----------|---------|-----------|---------------------|------------------|------------------|-----------------|------------------|
| 10^3 | 3.4E-4 | 2.8E-4 | 0.0736 | 0.07036 | 0.07036 | 0.00587 | 0.0041 |
| 100^3 | 0.306 | 0.2995 | 0.43 | 0.43 | 0.43 | 0.11267 | 0.0970 |
| 500^3 | 37.88 | 37.29 | 45.35 | 45.35 | 45.46 | 52.2 | 39.9 |
| $1,000^3$ | 302.6 | 298.2 | 360.7 | 360.7 | 363.13 | 375.5 | 318.1 |

As mentioned earlier, the performance of an application on an FPGA can be improved by increasing the number of pipes that are used, which in turn increases the number of calculations per cycle. In order to perform more calculations the FPGA design increases the width between logic blocks. Instead of streaming 64 bits of data between logic blocks, multiples of 64 bits are streamed (i.e. 128 bits for 2 pipes, 256 bits for 4 pipes, and so on) using more logic blocks (such as adders or multipliers); thus, more physical space is required on the FPGA. Fortunately, there is an initial cost for each resource; however, doubling the resource usage does not double the actual hardware usage. We experienced 50 % increase of actual resource usage whenever we doubled the number of pipes.

The T_{PCIe} performance model of the “double” implementation is identical to the model of the “single”, because the same amount of data is being streamed. However,

T_{COMPUTE} is now reduced to 50 % as two cells are computed per cycle and the FPGA Kernel will therefore run for half the time. Unfortunately in that case the problem is again PCIe bound. Figure 3 shows a comparison between the various versions of the mini-app.

If the double pipes mini-app application was not PCIe bound a performance increase of $\sim 100\%$ could be expected. And since there is enough physical space on the FPGA to use even more pipes, a doubling of performance with each step could be expected.

Since the problem is now PCIe bound, an option is to employ compression on the host side to reduce the amount of data that needs to be transferred. Uncompressing the data on the FPGA is free in terms of compute cost. However, one has to take into account that compressing the data on the host side would increase the runtime of the host application.

6.3 Mini-app LKJ Implementation

Using the same methodology as above we model T_{PCIe} and T_{COMPUTE} ; the results are shown in Table 5. We can see that porting the L loop, the application becomes compute bound. However, the amount of data that needs to be streamed to/from the FPGA has been reduced. For example, for the $1,000^3$ simulation the LKJ version streams 7.9 GBs and streams back 0.53 GBs (as opposed to 0.54 GBs and 0.53 GBs for the KJ version). The LKJ version streams the data only once, whereas the KJ version has to repeat the streaming operation 1,000 times.

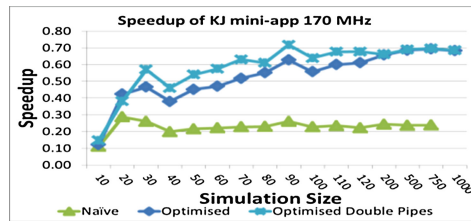


Fig. 3. Mini-app (KJ) speedup comparison

The last two columns of Table 5 show the actual duration and the modelled duration of the mini-app LKJ port. Since the mini-app LKJ port is compute bound, a speed increase from 100 MHz to 170 MHz reduces the runtime of the application.

Table 5. Mini-app (LKJ) T_{PCIe} , T_{COMPUTE} , modeled and actual duration

| Size | To FPGA | From FPGA | T_{COMP} | Duration 1.8 GBs | Duration 1.5 GBs | Actual Duration | Actual duration* |
|-----------|---------|-----------|-------------------|------------------|------------------|-----------------|------------------|
| 10^3 | 4.0E-05 | 3.0E-05 | 0.0011 | 0.0011 | 0.0011 | 0.00195 | 0.00123 |
| 60^3 | 0.0020 | 0.00108 | 0.0785 | 0.0785 | 0.0785 | 0.07962 | 0.04636 |
| 500^3 | 0.59 | 0.07 | 45.01 | 45.01 | 45.01 | | |
| $1,000^3$ | 4.44 | 0.30 | 360.01 | 360.01 | 360.01 | | |

We are not able to run simulations larger than 60^3 . The M loop calculations are independent; however, the K, J and L loop calculation are not. Our design creates a cyclic loop on the K, J and L output streams in order to drive the output calculation as input for the next iteration. The Virtex FPGA allows ~ 1 MB of look-back buffers per stream. For the 60^3 simulation 0.9887 MBs ($K*J*M$ entries) are used as buffer. For the 70^3 we need 1.34 MBs, which is larger than the available buffer. In order to be able to execute problems that are larger than 60^3 it is necessary to modify the algorithm and use the DRAM as a temporary buffer. Furthermore, if the size of the simulation increases to more than the available DRAM there will not enough space on the FPGA to hold the temporary data. Another option to support larger problem sizes is to modify the computational algorithm. The current algorithm computes a 2D face and then moves back in the L (Z) dimension. However, it is possible to instead compute a whole pencil in the L (Z) dimension and then move to the next pencil in the J (X) dimension. This modification would reduce the look back buffer to $L*M$ entries. Thus, we could run $3,600^3$ simulations.

6.4 Chimaera-2 Port

We implemented two versions of the K, J and M loop FPGA port for the Chimaera-2 application: the naïve and the optimised versions. The Chimaera-2 kernel uses 8 input arrays and 6 output arrays.

The first three columns of Table 6 show T_{PCIe} and T_{COMPUTE} for the optimised version of Chimaera-2 kJ port, which reduces the amount of data that needs to be copied to/from the FPGA. As an example, the PCIe duration of the optimised version compared to the naïve for the 120^3 simulation dropped from 8,261 to 2,154 s, which represents an improvement of almost 4x.

Table 6. Chimaera-2 (KJ) T_{PCIe} , T_{COMPUTE} , modeled and actual Duration

| Size | To FPGA | From FPGA | T_{COMPUTE} | Duration 1.8 GBs/s | Duration 1.5GBs/s | Actual Duration |
|-----------|-----------|-----------|----------------------|--------------------|-------------------|-----------------|
| 10^3 | 1.69 | 1.43 | 3.9 | 3.9 | 3.9 | 10.93 |
| 60^3 | 305.91 | 292.47 | 248.37 | 305.91 | 367.09 | 403.81 |
| 100^3 | 1,390.41 | 1,345.56 | 1,098.66 | 1,390.41 | 1,668.50 | 1,588.80 |
| 500^3 | 169,162 | 166,198 | 133,595 | 169,162 | 202,994 | |
| $1,000^3$ | 1,348,658 | 1,327,596 | 1,067,831 | 1,348,658 | 1,618,389 | |

The last column of Table 6 shows the actual duration of the FPGA kernel calls. For small simulation sizes the overheads cause the model to drift. However, as the simulation sizes increase the predictions of the model match the duration of the Kernel. Figure 4 (top left) shows the overall runtime of the Chimaera-2 application. The difference between the naïve and optimised approach grows larger as the simulation increases in size.

Figure 4 (top right and bottom left) show the performance improvement of reducing data transfers to/from the FPGA. The first figure shows the speedup of the overall application runtime whereas the second compares only the durations of the Kernels.

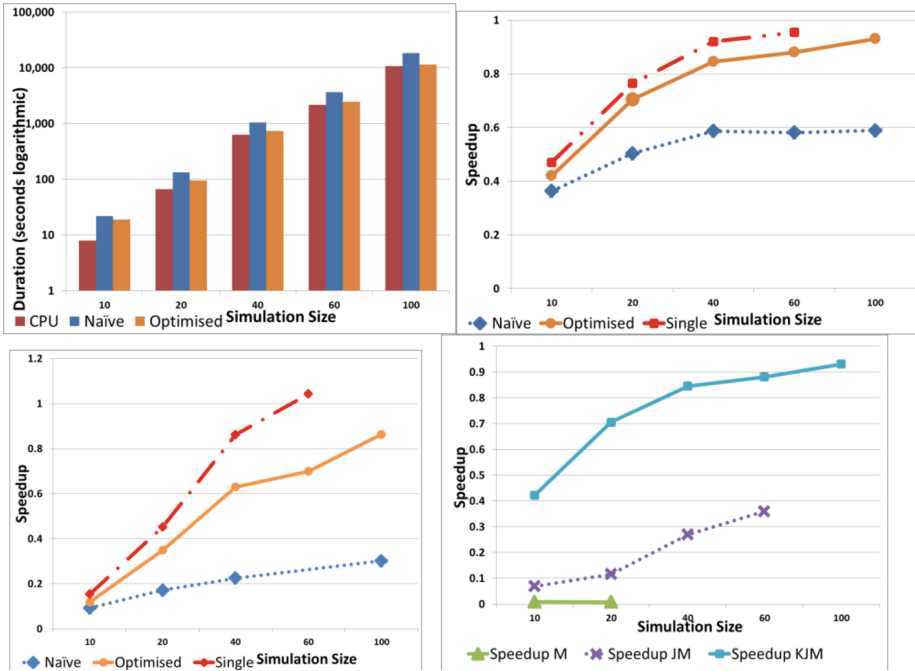


Fig. 4. Overall duration (CPU, naïve and optimised) (top left); overall speedup (naïve, optimised and single precision) compared to single core CPU run (top right); kernel only speedup (naïve and optimised) compared to single core CPU run (bottom left); speedup of M, JM and KJM loop implementations (bottom right)

Both implementations (naïve and optimised) are PCIe bound. It can be seen that as the size of the simulation increases, both implementations perform better. However, this improvement will only last until the PCIe links become saturated. Once the PCIe links are saturated performance improvements will plateau. It can be seen that the naïve implementation speedup plateaus with the 60^3 simulation.

We also implemented a version of the optimised code, which replaced double floating point numbers with single precision numbers in order to further reduce the amount of data that needs to be transferred. The single precision version is faster than the double precision implementation. However, it is not twice as fast even though data transfer is reduced by 50%. From Table 6 it can be seen that even though the KJ optimised implementation is PCIe bound, the value of TPC_{IE} is very close to $T_{COMPUTE}$. Thus, when using single precision TPC_{IE} is reduced and the performance limiting factor is again $T_{COMPUTE}$.

7 Conclusions

We have shown how an application programmer with limited knowledge of FPGAs and embedded computing can port a complicated production-ready Fortran90 code to an FPGA and achieve acceptable performance. We have also seen that porting a greater

proportion of the code to the FPGA improves performance. Our latest design uses around 20 % of the available resources (thus we have enough physical space on the FPGA to port more code or to use multiple pipes).

We have also shown how an application programmer can use a mini-app to test different porting strategies. We used the mini-app to implement a streaming version; a version which first copies to DRAM and then streams the data back; a version which copies to DRAM the input and output arrays and a version which copies some of the small arrays to the BRAM. Using the mini-app results, we decided to implement the data streaming version for the full application.

Figure 4 (bottom right) shows the speedups of the M, JM and KJM loop implementations. The M loop implementation clocked a “speedup” of 0.0076, whereas the KJM implementation reached 0.93 of the single CPU performance. Furthermore, as we minimise data transfers and allow the FPGA to re-use data that are already on-chip we realise that one of the most important optimisations that an application programmer can implement is to reduce data traffic. FPGAs can provide Terabytes/sec of data bandwidth between the logic blocks and BRAM. The bandwidth drops to tens of gigabytes/sec for DRAM memory. Furthermore, newer FPGAs increase the available physical space for logic blocks thus allowing even more code to be ported. Also, they run at faster speeds and include more and faster memory (BRAM and DRAM).

We plan to change the algorithm and port the L loop on the FPGA. We need to modify the algorithm due to the limited physical resources of the FPGA. The existing algorithm solves a 2D plane and then moves (backwards or forwards) to the next plane in the Z dimension. However, this approach requires “ $X*Y*M$ ” look-back entries on several streams. We will modify the algorithm and solve a pencil of Z length before moving to the next pencil in the X plane. Thus we will reduce the amount of look-back buffer to $Z*M$ entries. If we have enough physical resources we plan to use multiple pipes in order to compute multiple cells per cycle. Since, the LKJ port will be compute bound, each extra pipe will improve performance.

In order to reduce the amount of data being copied to/from the FPGA we can use compression. We plan to measure the effects of compression and perform a cost benefit analysis to help us decide if the extra time spent on the host CPU to compress data reduces the overall runtime of the application.

Finally, once the performance of the FPGA port is on par with the performance of a compute node we plan to measure and compare the energy consumption of the FPGA and the host platform.

References

1. Joubert, W.: Oak Ridge National Laboratory. Presentation given at the OLCF Titan Summit 2011. Porting the Denovo Radiation Transport Code to Titan: Lessons Learned. http://www.olcf.ornl.gov/wp-content/uploads/2011/08/TitanSummit2011_Joubert.pdf
2. Gong, C., Liu, J., Chi, L., Huang, H., Fang, J., Gong, Z.: Accelerated simulations of 3D deterministic particle transport using discrete ordinates method. *J. Comput. Phys.* **230**, 6010–6022 (2011). <http://www.sciencedirect.com/science/article/pii/S0021999111002348>

3. Gong, C., Liu, J., Chen, H., Xie, J., Gong, Z.: Accelerating the Sweep3D for a graphic processor unit. *J. Inf. Process. Syst.* **7**(1), 63–74 (2011). doi:[10.3745/JIPS.2011.7.1.063](https://doi.org/10.3745/JIPS.2011.7.1.063)
4. Gong, C., Liu, J., Chi, L., Huang, H., Gong, Z.: Particle transport with unstructured grid on GPU. *Comput. Phys. Commun.* **183**, 588–593 (2012). <http://www.sciencedirect.com/science/article/pii/S0010465511003870>
5. Plimpton, S., Hendrickson, B., Burns, S., McLendon, W., Rauchwerger, L.: Parallel Sn sweeps on unstructured grids: Algorithms for prioritization, grid partitioning, and cycle detection. *Nuclear Science and Engineering*, vol. 150, p. 267 (2005). <http://www.sandia.gov/~bahendr/papers/Rad-Transport.pdf>
6. Fu, L., Yang, S.: Researches on 2-D neutron transport solver NTXY2D, Technical report, Institute of Applied Physics and Computational Mathematics, Beijing, China (1999)
7. Maxeler, MaxCompiler Tutorial, v2014.1.1