Investigating how hardware architectures are expressed in high-level languages for an SKA algorithm

Krystine Dawn Sherwin*, Ben Stappers†, Prabu Thiagaraj‡, Kevin I-Kai Wang*, and Oliver Sinnen*
*Department of Electrical and Computer Engineering, University of Auckland
tshe835@aucklanduni.ac.nz, k.wang@auckland.ac.nz, o.sinnen@auckland.ac.nz
†School of Physics and Astronomy, University of Manchester
ben.stappers@manchester.ac.uk
‡University of Manchester; Raman Research Institute, Bangalore
prabu@rri.res.in

***Abstract*—High-level approaches to hardware development can expedite the design process, allowing for rapid design space exploration. However, in order to generate optimised solutions expert intervention is often still required. This work seeks to explore the relationship between high-level descriptions and the resulting hardware architecture. This aims to reduce the barrier to entry for software developers (without hardware expertise) to produce optimised hardware designs through application of classical loop optimisation techniques. An algorithm from the Square Kilometre Array (SKA) is chosen to demonstrate the effects of such changes in a real world, real-time application requiring high throughput and low power consumption, taking a systematic approach in order to achieve an optimised result. A systolic array design is also discussed and compared with the software style changes. The Intel FPGA SDK for OpenCL (AOCL) Offline Compiler (AOC) is used here for verification and synthesis of the designs being examined, targeting an Arria-10 FPGA accelerator.**

I. Introduction

Being able to develop algorithms at a high level, agnostic to the hardware being used, allows for rapid exploration of the impact that changes in the structure of the algorithm can have. For large scale applications, as in the Square Kilometre Array (SKA), such exploration is vital to finding an optimised solution across a range of options, particularly when heterogeneous systems are introduced with multiple accelerators. The high performance requirements of the SKA, along with the nature of the algorithms needed, presents an opportunity to explore loop descriptions in software and the resulting output, as well as how to use OpenCL to describe a systolic array structure.

The main contributions of this paper are as follows, (1) Systematic exploration of code manipulations at a high level and how these translate to hardware, (2) An experimental evaluation of such manipulations, and (3) A real world investigation of how to represent systolic arrays in OpenCL.

Section II explores the current state-of-the-art for optimisation, looking in particular at compiler level techniques. The algorithm being explored is introduced in Section III, followed by discussion of the high-level optimisations explored here in Section IV. An evaluation is presented in Section V before conclusions are drawn in Section VI.

II. Related work

High level approaches attempt to make FPGA development more accessible. Approaches such as LegUp utilise the familiar C programming language for hardware description [1], while others such as OpenCL provide more explicit control over hardware generation without relying on tool specific pragmas and optimisations [2]. In addition, OpenCL is available across multiple hardware platforms [3] with support from a variety of manufacturers and vendors, leading to its focus here.

While high level approaches are capable of producing functionally correct hardware, current techniques often still require manual intervention to achieve high performance [4]. This requires similar levels of expert knowledge as low level approaches in order to most effectively utilise the resources available [5]. Previous work discusses system design with OpenCL [6], showing a number of techniques with no effort made to compare them. Further work is required to reduce this barrier to entry, and provide an increased ability to convey information to the compiler for optimisation.

III. Folding and optimisation module

The SKA project seeks to answer big questions about the Universe through the use of a very large, intercontintental array of radio telescopes. Within the SKA, there exists a sub-element called the Pulsar Search Engine (PSS). This sub-element takes raw data input, outputting filtered data with pulsar candidates identified for further processing [7]. For processing the large amounts of data coming in, current designs for the PSS incorporate a number of heterogeneous nodes, each one containing significant amounts of memory, along with a pair of multicore CPUs, and both a GPU and FPGA accelerator [7].

The Folding and Optimisation (FLDO) module of the PSS is used as a case study for this work. This module, positioned at the end of the PSS takes all pulsar candidates for each chunk of data, and finds optimal values for three parameters. Each candidate will be output as a folded data cube, consisting of array dimensions up to $128 \times 128 \times 64$. The first of these dimensions corresponds to frequency, while the other two correspond to time. Additional information is also output for tracking the signal-to-noise ratio (SNR) across the search space.

Initial estimates allow for the time-series data to be folded into the data cube during the Folding (FOLD) stage. The pulsar parameters map the input values to be accumulated in each bin, normalising for the count in each. From here, the parameters are adjusted to find the maximum SNR in the Optimisation (OPT) stage. This adjustment is implemented as a realignment of the bins in the data cube.

TABLE I
CODE MODIFICATIONS AND EXPECTED RESULTS

Name	Technique	Brief description	Expected benefit	Expected negative
$A=\#$	Unroll	Perform # iterations of loop A in parallel		Significantly increased resources, reduced fmax, increased cycle latency
$B=\#$	Unroll	Perform # iterations of loop B in parallel		Increased resources, reduced fmax, potentially increased cycle latency
$C=\#$	Unroll	Perform # iterations of loop C in parallel	Potentially reduced memory, reduced cycle latency	Increased logic
$D=\#$	Unroll	Perform # iterations of loop D in parallel	Potentially reduced memory, reduced cycle latency	
MOV	Code motion	Move code outside of loop to reduce recalculation	Increase fmax and/or reduced cycle latency	Increased memory
C+D	Fusion	Merging loops C and D into one, additional check for final iteration	Reduced cycle latency	Increased logic, potentially reduced fmax
B&C	Flatten	Reduce nesting by iterating B and C together	Reduced complexity	Increased calculaton redundancy
A<>B	Interchange	Rearrange the order of A and B loops such that A is nested in B	Reduced cycle latency	Reduced data sharing
copy	Copy only	Copy input directly to output without performing bin shifting	Reduced resources, reduced cycle latency	No calculations performed (incorrect results)

The optimisation is carried out through a heavily nested loop structure in order to maximise data reuse. This work focuses on the inner-most parameter, performing an exploration of high-level descriptions and the corresponding hardware architecture.

At this stage, the data cube has been reduced to the two time dimensions, with the inner-most parameter corresponding to the pulsar period derivative, $\dot{P}$, resulting in an acceleration of the period. Within each trial, the degree of bin shifting is determined by the current $\dot{P}$ and sub integration, with calculations in loops A and B respectively. Loop C then further reduces data to a single dimension for calculating SNR, with loop D being used for output. This is illustrated in Algorithm 1.

The change in bin shift between $\dot{P}$ trials for each sub integration is linear. Along with the accumulating nature required for reducing the data to a one dimensional array, this leads to the potential for a systolic array approach for this algorithm.

IV. HIGH-LEVEL OPTIMISATIONS

The code modifications investigated are summarised in Table I with a brief description as well as the expected change in hardware. Starting with optimisations provided by OpenCL, pipelines can be automatically generated based on dependency analysis, and can produce hardware with multiple operations in each clock cycle [2]. Next is loop unrolling, whereby multiple loop iterations can be explicitly run in parallel.

Changes to code structure can also be implemented, and are generally used in order to present the underlying algorithm in such a way as to allow for the compiled code to operate in a more optimal way. For example better memory access patterns, or more exposed parallelism may be achieved this way [8]. The first technique, MOV, takes code from a nested loop which does not change across iterations of the loop and moves it outside of the loop. Next, sequential loops can be combined (fusion) or split (fission), changing the loop bounds as necessary. Similarly, loop flattening can be used to combine nested loops. Interchanging the order of nested loops alters the order of iterations. In addition, certain techniques can be combined with each other, in some cases providing a result which is not just the sum of its parts. As an example, when performing both unrolling and code motion the amount of logic to replicate may be reduced.

Algorithm 1: Base algorithm

```
Input: p, pdot_0, pdot_step, t_0, t_step, nudot, deltaNu
A: for trial=0 to TRIALS do
    bin_sums = {0};
    /* pdot calculations                    */
    B: for sub_int=0 to SUB_INTS do
        /* bin shift calculations           */
        C: for bin=0 to PHASE_BINS do
            input_bin = (bin - bin_shift) % PHASE_BINS;
            bin_sum[bin] += input[sub_int][input_bin];
    D: for bin=0 to PHASE_BINS do
        output bin_sum[bin];
```

Systolic arrays present another option for implementation. These are well suited for hardware, but are only possible with certain algorithm structures. Here, this allows for usage of local memory to reduce loads from global memory and improve data reuse, providing for a highly parallelised design, theoretically capable of outputting at a rate of one trial every cycle. In order to achieve this, the algorithm can be restructured in order to explicitly describe the desired hardware structure, shown in Fig. 1. In contrast to the initial algorithm which requires a loop nest of depth 4 to cover all P trials, the systolic array is described here using a loop nest of depth 3. This loops through the full 2D systolic array inside of an iterator over the full P and $\dot{P}$ search space. This approach results in a relatively complex code base compared to the above, but should be capable of significantly improved performance.

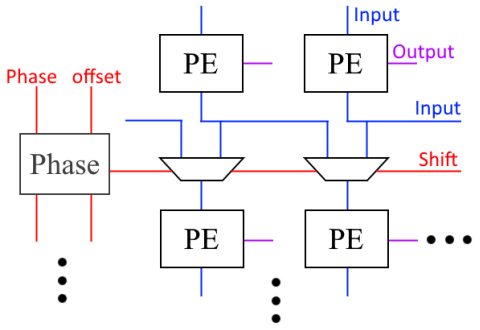

Fig. 1. Desired systolic array structure

TABLE II
HARDWARE RESULTS, MIRRORING THE METRICS DESCRIBED BY NANE ET AL. [9]

Optimisation	Frequency	Total_time_ms	Cycles	trials/s	Logic	Memory bits	RAM
base	262.2	66.02	17.3M	993k	59444	4017346	389
MOV	240	72.43	17.4M	905k	65466	4004866	383
C+D	250	41.72	10.4M	1.57M	57243	4006658	389
B&C	264.1	16.04	4.24M	4.09M	58286	4028418	393
A<>B	279.5	15.05	4.21M	4.35M	60611	4017154	398
B&(C+D)	266.7	16.02	4.27M	4.09M	55556	4002626	387
B<>(C+D)	259.3	16.21	4.2M	4.04M	56544	4004482	386
copy	261.5	3.06	800k	21.4M	54105	2773506	307

V. EVALUATION

In this section, compilation results are compared with our prior expectations: first for the baseline implementation, comparing with the software reference; then for the loop transformations; and finally looking at the systolic array implementation.

A. Hardware/software setup

For the prototype designs and experiments we employed the Nallatech 385A FPGA PCIe accelerator card in conjunction with an Intel Core i7-6700K CPU@4.0GHz, and 64GB of RAM. CentOS version 7.3.1611 is used, along with Intel FPGA SDK for OpenCL (AOCL) version 16.0.2.

For this initial investigation, the dimensions of the array have been reduced from 64×128 to 8×8, and the $\dot{P}$ search space reduced from 512 possibilities to 8. This was done primarily to reduce the time taken for synthesis and verification. To prevent loops from being automatically unrolled, explicit pragmas have been added to the base code. The exact number of P trials was controlled by the host without requiring recompilation, however each test was carried out to provide a consistent sample size of 65536 $\dot{P}$ trials in total. This was done to allow the hardware pipelines to operate for a time under full load, giving a more representative measure of the expected real performance.

B. Baselines

A throughput of 65536 $\dot{P}$ trials in 0.26ms is required for the real system. An initial version of C++ code is able to perform the necessary calculations in 60ms. While this is significantly slower than required, it was sufficient for verifying values produced by hardware. The base hardware version is able to provide the required values in 66ms, slightly slower than the software performance. This time does not include the overheads for launching and copying data to/from the accelerator. Initial experiments showed this to generally contribute an additional 1-3ms across all tests. Hardware details are given in Table II, along with the other code modifications examined here. Performance is measured using hardware performance counters on memory accesses which are inserted during initial compilation, providing a minor additional resource overhead.

C. Simple loop transformations

From the results in Table II, comparisons can be made between the resource utilisation and cycle latencies for the modifications being made. However, there is significant variation in frequency which does not appear to be consistent between combinations, giving overall performance variation of ±20%.

Table IV presents additional compilations with unrolls enabled. As can be seen from this table and Table II there are some interesting remarks to be made about particular combinations. When performing both loop fusion and loop flattening without unrolling, i.e. B&(C+D), the results appear as expected with the effects of both individual optimisations being combined. However, attempting to unroll B results in the compiler adding extremely large delays (over 1200 cycles) between iterations, as well as significantly higher logic utilisation, in an attempt to compensate for what the compiler perceives as inter-iteration dependencies. Unrolling both A and B also results in a drop in performance, though not as severe. This is due to serialisation of the C loop performed by the compiler in order to avoid inter-iteration dependencies. It should also be noted that the best performing result is achieved purely through full unrolling of certain loops. This may be somewhat misleading as full unrolling is only possible due to the reduced size requirements explored here and thus producing a disproportionate effect on the throughput and resource usage due to reduced overheads.

Table III provides an overview of the effect each of the code modifications has on the resulting hardware. Where it was expected that code motion would result in increased memory the actual result is a reduction in memory at the cost of increased

TABLE III
GENERALISATIONS FROM TABLES II AND IV

Name	Apparent benefit	Apparent negative
$A=\#$		Increased resources, increased cycle latency, increased complexity
$B=\#$	Reduced cycle latency	Increased resources
$C=\#$		Increased resources, increased cycle latency
$D=\#$	Reduced resources	Increased cycle latency
MOV	Reduced memory	Increased logic
C+D	Reduced resources, reduced cycle latency	
B&C	Reduced logic, reduced cycle latency	Increased memory, reduced parallelism capability
A<>B	Reduced cycle latency	Increased logic, reduced parallelism capability

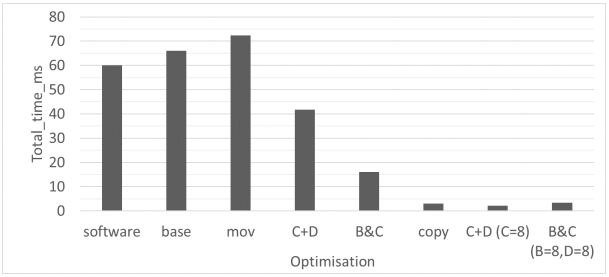

Fig. 2. Graph showing performance comparisons between selected optimisations, including both the software reference and hardware copy-only versions.

TABLE IV
SELECTED HARDWARE RESULTS WITH UNROLLING

Optimisation	Frequency	Total_time_ms	Cycles	trials/s	Logic	Memory bits	RAM
C+D $C=8$	255.1	2.21	564k	29.7M	56559	4023170	398
B&C $B=8, D=8$	244.4	3.39	829k	19.3M	63626	4020290	402
B&(C+D) $B=8$	207.4	3670	761M	17.9k	86382	4997757	501
A<>B $C=8, D=8$	260.9	3.98	1.04M	16.5M	59921	4045826	402
copy $D=8$	252.8	1.18	298k	55.5M	61752	2837250	334
$A=2$	229.3	78.27	17.9M	837k	69499	4135426	408
$B=2$	229	9.76	2.24M	6.71M	66699	4093186	408
$C=2$	255.5	67.74	17.3M	967k	60007	4044994	388
$D=2$	260.3	66.51	17.3M	985k	59377	3968066	389
$A=2, B=2$	228.62	120.12	27.5M	546k	82364	4256322	433
$C=8, D=8$	264.1	2.15	568k	30.5M	57356	4028610	400

logic. This is believed to be a product of using more registers for storage rather than RAM blocks. In general, it is believed that these differences are a result of the compiler performing unexpected optimisations. Selected optimisations are also shown in Fig. 2, allowing comparisons between their performance.

D. Systolic array approach

Through implementation of a systolic array, it was possible to reach significantly higher performance, without a significant increase in logic or memory requirements. This was able to be achieved for a single kernel design, using 60939 logic elements, 487 RAM blocks, and 14 DSPs, operating at a frequency of 280.7MHz. As would be expected, as the total number of $\dot{P}$ trials increases, the number of cycles per trial approaches 1. This is shown in Table V. When set to 65536 total trials as before, the systolic array was able to produce the results in only 0.25ms, slightly faster than that required by the SKA. However in the current implementation it is not possible to directly extend this to the full scale. This is primarily due to automatic usage of RAM blocks by the compiler. Each RAM block currently only contains 2 values. This does not change with the array size, resulting in significant over allocation of resources. It is unknown as to why the arrays have been assigned to RAM blocks rather than utilising registers. This issue must be resolved in order to achieve the necessary performance at the required scale.

VI. CONCLUSIONS

In this paper we carried out a systematic exploration of high-level code manipulations to reason about hardware translations for an FPGA accelerator. This was carried out on a scaled down, real world algorithm from the SKA PSS sub-element. A variety of code optimisation techniques were applied individually, as well as a selection of combinations. Changes in synthesised results were compared with expectations based on the changes made at the software level, with differences highlighted and possible reasons provided. With a real world performance target of 0.26ms for 65536 trials, these optimisation techniques were unable to achieve a total time lower than 2.15ms, even at a significantly reduced data scale. In contrast, a highly specialised systolic array approach was able to achieve a total time as low as 0.25ms. However, at present the code used is not suitable for operation at the full scale required and more work is yet to be done.

Future work aims to incorporate optimisations explored here into the compiler level, rather than requiring significant code level modifications by a designer. This is intended to improve the portability of OpenCL with pragmas for such optimisations, reducing the requirement for manual structural changes . Improved design space exploration is also possible through modelling of optimisation effects, reducing the need for lengthy hardware synthesis.

ACKNOWLEGMENT

We gratefully acknowledge that this research was financially supported by the SKA funding of the New Zealand Government through the Ministry of Business, Innovation and Employment (MBIE). The authors would also like to thank the rest of the SKA-TDT for their input and work on the pulsar search pipeline.

REFERENCES

[1] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson *et al.*, "Legup: High-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36.

[2] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto *et al.*, "From opencl to high-performance hardware on fpgas," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2012, pp. 531–534.

[3] H. Wang, J. Gante, M. Zhang, G. Falcao, L. Sousa, and O. Sinnen, "High-level designs of complex fir filters on fpgas for the ska," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Dec 2016, pp. 797–804.

[4] Y. Zhang, M. Sinclair, and A. A. Chien, "Improving performance portability in opencl programs," in *Supercomputing*, J. M. Kunkel, T. Ludwig, and H. W. Meuer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 136–150.

[5] M. A. Ozkan, O. Reiche, F. Hannig, and J. Teich, "Fpga-based accelerator design from a domain-specific language," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–9.

[6] I. Janik, Q. Tang, and M. Khalid, "An overview of altera sdk for opencl: A user perspective," in *2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE)*, May 2015, pp. 559–564.

[7] L. Levin, W. Armour, C. Baffa, E. Barr, S. Cooper, R. Eatough *et al.*, "Pulsar searches with the ska," *arXiv preprint arXiv:1712.01008*, 2017. [Online]. Available: https://arxiv.org/abs/1712.01008

[8] R. Sotomayor, L. M. Sanchez, J. G. Blas, A. Calderon, and J. Fernandez, "Aki: Automatic kernel identification and annotation tool based on c++ attributes," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 3, Aug 2015, pp. 148–153.

[9] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis *et al.*, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct 2016.

TABLE V
HARDWARE RESULTS FOR SYSTOLIC ARRAY DESCRIPTION

Total $\dot{P}$ trials	Total_time_ms	Cycles	trials/s
4096	0.03	8.42k	137M
16384	0.07	19.6k	234M
32768	0.14	39.3k	234M
65536	0.25	70.2k	262M
524288	1.89	531k	277M