An Algebra of Array Combinators and its Applications

Mary Sheeran

1 Purpose and Aims

Recent successes in machine learning have been built not only on new algorithms such as deep learning, but also on work with systems and programming languages to make the link between user programs and the practical low level computations needed to produce useful results. Much of this work is concerned with so-called tensors, a new name for multi-dimensional arrays. The scale of the tensor programs in question is often huge, and is continuously growing. For example, GPT-3, a transformer model that produces human-like text, has 175 billion parameters. AlphaFold from DeepMind, which recently had unprecedented success in predicting protein structures, has underlined the potential value to science and humanity of deep learning, and the rush to build on those results is only just starting. It is important to find better ways to run tensor computations efficiently on available computing resources. This involves partitioning the computations so that sub-parts can run on accelerator devices that are well suited to the load, for example that have enough memory. Until recently, it took months to retarget important ML models (or programs) to new accelerator configurations. DeepMind has had considerable success in their current approach based on a special language for specifying partitions and transformations, but much remains to be done. One possible approach is to view tensors as data types and from that perspective reconsider operations, to better enable the specification of suitable partitioning, and the building of suitable abstractions both for programming and partitioning. This proposal aims to do exactly that. Based on our earlier work, we expect that a declarative approach to arrays and partitioning will also have application in hardware design, in particular for the direct implementation of array or tensor processing datapaths. Thus, we consider both hardware and software implementation of array programs. It is essential that hardware correctly implement a specification, so we place strong emphasis on proving functional correctness.

Computer science has long studied ways to construct programs from smaller programs, and to reason about the results. In functional programming, combinators or higher order functions (functions that take functions as input and produce functions as results) are used for this purpose, backed by static type checking. Backus elucidated the importance of designing a set of program construction combinators simultaneously with their algebra, enabling reasoning about the resulting programs. By algebra, we mean a set of laws relating the various operators of the language, for example that map f ; map g is equal to map (f ; g), where ; denotes forward function composition, and map f applies a function to each element of a list. Such laws enable not only reasoning by the user about programs, but also compiler transformations that eliminate unnecessary intermediate data structures. A Domain Specific Languages (DSL) is often a set of combinators with a desired algebra, with Henderson's Functional Geometry being an illustrative example [15], and my own muFP, for structural hardware description based on combinators, another [24].

It is customary to index sequences (whether lists or arrays) using integer indices and to freely add integers to such indices when operating on pairs or groups of elements. Much work has been done on analysis of array access patterns and data dependences for parallelisation or optimisation of loops, including the well known polyhedral methods. However, unrestricted access makes reasoning about the resulting programs (whether by the programmer or the compiler) more difficult. Here, we propose a new approach to the design of sequence combinators, by supporting a sufficiently rich algebra to enable reasoning but retain considerable expressive power. This is done by supporting a restricted form of arithmetic on array indices. Having the algebra enables not only reasoning about the resulting computation but also the analysis of choices in implementation, relating to resource use and how to map the computation into hardware or onto processors in a distributed system.

We consider *whole array* operations with index arithmetic operating uniformly on index bits viewed as vectors over GF(2) (the finite field of two elements), and thus also permutations as matrices over GF(2). This gives a whole new lens with which to view the problem of how to express and reason about array computations and access patterns. This proposal aims to investigate this topic in depth, and to demonstrate the results in hardware design and tensor programming.

I have developed array and list combinators in the context of the design of regular structures such as data-independent sorters, the Fast Fourier Transform and prefix and median networks [25]. I recently realised, however, that there is even greater regularity than has been documented in the literature; in the sorters for instance, a new form of arithmetic on array indices reveals pleasing regularity. Consider the sorter shown in a standard notation for sorting networks (see for example Knuth [19]) on the left in Figure 1. In the column of comparators pointed out with an arrow below, normal index arithmetic indicates comparisions between index pairs (0,13), (1,12), (6,11) and (7,10), for instance, with those paired indices thus being 13,11,5 and 3 apart. However, switching to indices regarded as length 4 vectors over GF(2) reveals that in all cases, for pair (a,b), $a \oplus b = 13$ or equivalently $a \oplus 13 = b$, where $\oplus$ is *bitwise exclusive or* (that is addition of vectors over GF(2)). For example, $6 = [0,1,1,0]$, $11 = [1,0,1,1]$, so $6 \oplus 11 = [1,1,0,1] = 13$. Indeed, every column has a single number (like 13 here) that is the *distance* between each pair of compared index vectors, as shown above the diagram. For distance k, we call such a column k*-regular*. The same is true of sorter B on the right in Fig. 1, and of a multitude of other sorting networks. It was this previously unknown form of regularity that triggered initial work on an alternative view of arithmetic for array indexing, in a bid to better understand regular structures and the combinators that describe them.

Moving to a Vector of Index Bits (VIB) view of indices and supported operations on them brings the question of how to deal with permutations (of the elements of arrays or lists) in this setting. The answer is to adopt the Bit Matrix Multiplication and Complement (BMMC) permutations that were extensively studied by Cormen in the context of data parallel programming and of memory management in disk based computations [9]. These permutations are represented by a nonsingular matrix A over GF(2) and a complement vector c, so that $y = Ax \oplus c$. Many standard permutations belong to BMMC, including reverse, bit-reversal, shuffling and grey codes. The BMMC permutations were called Affine Index Transformations by Edelman et al [12]. I will encode the linear algebra on index digits in combinators, revealing new forms of regularity and expressing intricate access patterns. I am combining two sets of old ideas and I believe that they still have much to give.

Expected results

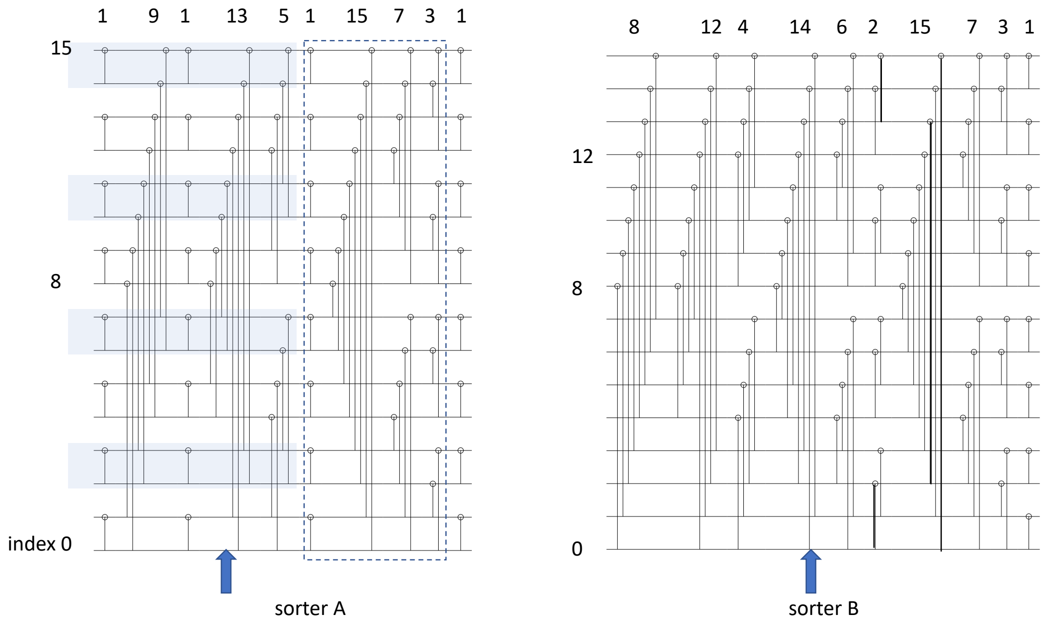
1. Array and list combinators with an interesting and useful algebra, and a prototype parallel functional implementation.

2. New ways to express, reason about and implement partitioning of data to and from multi-dimensional arrays, particularly for partitioning of tensor computations onto computational units, both at large scale and in resource constrained settings. Evaluation through standard case studies such as those used in the FlexFlow project [17].

3. Better understanding of regular structures in both hardware and software. This should support both formal verification and the use of search to find new such structures. Evaluation through case studies in hardware design [27] and in software (standard ML benchmarks).

4. The addition of a notion of time to the partitionable array data type, or possibly to a separate partitioning language, to enable verified folding of algorithms into hardware by trading off space and time. Demonstration on case studies from signal processing or machine learning.

Figure 1: On the left, a sorting network for 16 inputs, built from a Balanced Merger [10]. It has 10 phases each of 8 2-sorters, pictured as vertical lines receiving 2 inputs on the left and producing the min and max on the right, with the min output circled. The last 4 phases in the dotted box are the balanced merger, the first 6 consist of two half-sized sorters combined by the que combinator so that one works on the shaded horizontal lines, see section 4. On the right is a sorting network made with the same merger, but with the half-size sorters interleaved using the ilv combinator.



2 State-of-the-art

Backus' classic paper introducing his FP language, its combinators and their algebra, inspired generations of functional programmers [3]. I adapted his ideas for hardware description, verification and refinement, in relational and functional settings [24, 4, 18], while Bird, Meertens and others studied the theory of lists, and cemented functional programming's heavy reliance on higher order functions like map and fold. Lists are not ideally suited to parallelism and arrays are increasingly investigated, for example in the Accelerate library in Haskell, and in standalone languages like Futhark [16] for GPU programming. Java provides high performance (functional) data parallel programming, using familiar higher order functions like *map*, *fold* and *scan*. Blelloch's NESL was seminal [5].

Renewed interest in array programming has been driven by the need to exploit data parallelism for scalability, and by the rise of Machine learning (ML), based on multidimensional array computations. Array languages are often implemented as Embedded Domain Specific Languages in Haskell [7, 2, 22], Scala [29], C++ [21] or other high level host languages, and MatLab, NumPy and Julia have all been influential. APL remains influential, see for example work on Remora [28], with its static rank polymorphism that lifts base operators to higher dimensional array structures; it has the explicit aim of solving Backus' *von Neumann bottleneck* via loop and recursion free aggregate operations, but is, however, difficult to implement. I am also interested in aggregate operators, but with lesss emphasis on lifting to multiple dimensionalities. There has been considerable focus on ways to enable array fusion in domain specific compilers, including our work on pull and push arrays [7]. We studied array representations in the context of the Feldspar project, in which we developed an embedded domain specific language for Digital Signal Processing (DSP), in collaboration with Ericsson [2]. One of the main motivations of the project was the high cost of retargeting DSP code to new architectures or platforms, something that often took months (in the C programming language). Our DSL expressed parallelism using higher order functions. It did not express partitioning and deployment separately from the algorithm description; perhaps it should have done.

Tensorflow, and systems built upon it, are widely used within Machine Learning [1]. The most recent version has moved to the "define-by-run" or eager automatic differentiation scheme popularised in PyTorch, and thus away from the original compiled data flow graph approach. Although now archived, very recent work on Swift for Tensorflow made a pioneering first attempt to to expose strong types for tensor programming [23]. Interestingly, it opens the possibility to have several different tensor libraries, all wrapped in a well designed type system. Unfortunately, the implementation of Automatic Differentiation proved challenging in the SwiftIR. Similar work on strongly typed tensor programming is being done on DiffSharp in F#. My interest in types for tensor programming was piqued by Vytiniotis' invited talk at PPDP on work at DeepMind and Google on an Intermediate Representation for partitioning of tensor computations, partIR [31]. partIR, which sits on top of a tensor backend compiler and runtime (such as XLA), regards partitioning as a program transformation, and provides abstractions to express partitioned machine learning models on accelerator systems, as well as declarative transformation rules for partitioning actions and fusion. The annotations may be added manually, as interactive tactics or using search and reinforcement learning. The results from partIR are promising, although much remains to be done.

The sharding specification is kept separate from the algorithm description, as in Halidee [21]. There is strong interest from DeepMind in informal collaboration around new tensor abstractions (types, operators and transformations). I see an interesting parallel between the problem of partitioning tensor computations onto distributed accelerators such as TPUs and GPUs and the problem of implementing such a computation directly in hardware on an ASIC or FPGA. I will explore ways to specify algorithms, partitioning and schedules in the context of hardware implementation of (possibly machine learning oriented) data paths. Ranges in partIR represent contiguous, non-overlapping intervals, and here I am interested in broadening to cover non-contiguous sub-sequences. partIR is built using the new, open source MLIR compiler infrastructure, which should prove useful for my work also [20].

Cormen derived asymtotically equal lower and upper bounds for the number of IO operations required to perform BMMC permutations on parallel disk systems [9], assisted by his emphasis on general bit index manipulations and the matrix-vector view [12, 13].

Finally, higher order functions for structural hardware description have long been advocated by the applicant [26], influencing other researchers [14, 6], although there has been limited success in transferring these ideas into mainstream hardware development. However, significant recent worries about security and privacy of hardware is spurring research in fully formally verified hardware *development*, for example at Google in the Silveroak research project [27], which includes Cava, a version of Lava (on which Singh and I worked for many years [4]) in Coq. This is rather different from the typical post-hoc verification, and gives the opportunity to consider the integration of design and verification at quite a low level of abstraction, since the aim is to verify from the ground up. The work is very much inspired by Chlipala's book on *Certified Programming with Dependent Types*. A less extreme approach, which I hope to explore with my colleague Carl-Johan Seger, formerly of Intel, is to again integrate design and verification, but with the emphasis on studying (and recording) transformations made during refinement from specification to implementation, and on localised formal verification using SAT and SMT solvers. A big question is how to introduce details of timing, and of reuse in both space and time, see for example initial work on Aetherling [11], which introduces an intermediate representation including space- and time-arrays for this purpose. Earlier versions of the Cryptol DSL for designing cryptographic algorihms generated hardware and made use of retiming and other space-time transformations [6]. The latest release does not generate hardware.

3 Significance and scientific novelty

This proposal is motivated by research questions related to design of industrial hardware and software, particularly for machine learning, and by a the problem of understanding, describing and generating code for algorithms with a high degree of regularity.

The success of AlphaFold in protein structure prediction has reverberated beyond the normal rearch of AI and machine learning, and a big move towards ML-assisted science seems not only inevitable but full of hope. The push towards autonomous driving raises many questions, including some about the low level technology of machine learning and how it can be incorporated into safety critical systems. But still, it too attracts ever more people and resources. The demand for computation related to ML seems to be almost insatiable, also changing computer science research agendas from computer architecture to algorithms.

ML development systems rely heavily on research on programming languages and systems, and more such work is needed to supply ever increasing needs. Central to progress is the question of how to partition large tensor computations efficiently onto computing resources. I will concentrate on a type-based approach to this problem by reconsidering the design of tensor data types.

OpenTitan is the first open source project building a transparent, high-quality reference design and integration guidelines for silicon root of trust (RoT) chips, aiming to provide the security foundation for the future internet – an important societal problem in a global world. Google's Silveroak research project is investigating methods of developing fully verified, low level hardware in this context [27], working with a version of Lava within the Coq theorem prover, yet aiming to produce subsytems of the same quality and efficiency as expert, manual designers. Given the concentration on crypto algorithms like AES, this results in an emphasis on the verified design of low level, *datapath* rather than control oriented hardware, while most work on formal hardware design concentrates on raising abstraction levels, and on processor design. This gives an excellent match with this proposal.

Although I have found preliminary work and potential for informal collaboration with colleagues at Google and DeepMind, the work proposed here fills a gap that has not been much explored by academic researchers. I see strong parallels between the partitioning problem for tensor computations and the problem of implementing algorithmic datapaths in hardware, and investigating such a link is typical academic research, aiming for understanding and general principles. I believe that I have succeeded in finding a niche where I can apply insights from declarative programming and from my work on structural combinators *close to the machine* in both of these areas.

The tensor types currently in use in ML are rarely questioned, given their undoubted practical success. But it makes sense to question this aspect of the status quo and to look for better alternatives, amenable to more forms of partitioning. Our intended approach, inspired by a restricted form of index arithmetic, associated combinators and a set of permutations studied by Cormen, is novel.

4 Preliminary and previous results

I have pioneered the use of domain specific languages in hardware design and for code generation from functional descriptions [18, 4, 2, 30], partially with help of VR funding (reg. no. 2011-6234, and two earlier). Obsidian, a Haskell-embedded, code-generating DSL for GPU programming, provided the means to experiment both with array combinators for describing regular structures such as sorters, and for work on datatypes such as pull and push arrays that enable array fusion in embedded languages [7]. Let us concentrate on *length homogeneous n-functions* that take a sequence of length n to an output of the same length. The ilv combinator applies a 2^k-function f separately to the even and odd indexed elements of a sequence of length 2^{k+1}. We can generalise this to a function parm that divides the inputs based on the parity of a chosen set of bits of the index, indicated by a mask. Then, ilv is parm 1, while que is parm 2 and vee is parm 3.

Applying any of these combinators to a function that applies a 2-function to array elements a fixed distance apart (the k-regularity illustrated earlier) preserves the regularity property, possibly changeing k. The balanced merger and sorter A (Fig. 1) can be defined recursively as

```
bal 1 s2 = s2
bal n s2 = vee (bal (n−1) s2) −>− evens s2

sort 1 s2 = s2
sort n s2 = que (sort (n−1) s2) −>− bal n s2
```

The two half-size mergers are combined with *vee* and composed (−>−) with the final column (evens), which applies the 2-sorter to elements at indices i and $i+1$ for even i. evens can be made by repeated application of *que*. The two half-size sorters are combined using *que*. The sorter on the right of Figure 1 uses the same merger, but combines the half-size sorters using ilv rather than que. The diagrams are generated from these descriptions, which can also generate input to a SAT solver to enable formal verification (based on the zero one principle) of fixed size sorters[8].

I have shown that the combinators ilv , vee and que are sufficient to make any m-regular column from a 2-function f, and indeed there is a normal form for their application, as

```
    vee (ilv f) = que (ilv f) = ilv (ilv f)
```

For example mask 14 or $[1,1,1,0]$ corresponds to ilv (vee (vee f)), read from $[vee, vee, 1, ilv]$ and such a column is indicated by an arrow in sorter B in Fig. 1. These combinators are in some sense immediately flattenable. Let inc be a predicate on sequences indicating whether a sequence is non-decreasing. In sorting networks, it is known that the rightmost column must compare adjacent elements. When it is evens s2 as in Fig. 1, then to produce a sorted output, the input to that column must be sorted in the two subsequences selected by ilv and in those selected by vee, which we write as $vee(ilv) \wedge ilv(inc)$.The final column odds s2 compares elements at indices i and $(i+1)$ for *odd* i *odds s2*, and is used in Batcher's odd even merge sort, for example. Then, the input should rather satisfy $que(inc) \wedge ilv(inc)$, to produce sorted output.

One might be tempted to work with only these three combinators, but there are reasons to consider more when studying the algebra of the combinators and of m-regular columns. Composing a BMMC permutation p (with matrix A and complement vector c) and its converse on both sides of a k-regular computation again results in a regular computation, with distance $A\ k$. Therefore, we can push the permutation through a regular column, changing the distance, but keeping the regularity. So the BMMC permutations play nicely with our combinators. However, this also reveals that we should also record which indices correspond to the first inputs of each 2-function, since the permutation may flip over some of the 2-functions, for example if it is *reverse*, made from the identity matrix and all ones as complement vector. But we already know how to select half of the elements of an array in various ways using dot product with a mask. We extend regularity to include two masks, with the second indicating the first inputs of the instances of the 2-function. Again, the linear algebra view indicates how to calculate with these masks, which is relevant also for parallel code generation. Starting with ilv and the two permutations that can be composed on either side of it to make vee and que, one can build everything else, all of the BMMC permuations, and thus all of the combinators built using parm. For parameterised verification of regular hardware structures in Coq, having such a small basis should reduce the number of lemmas and the complexity of the inductive proofs.

The algebra of the combinators is supportive even when we need to compose m-regular columns. The properties of addition over vectors of GF(2) mean that pairs of adjacent columns are guaranteed to match pleasingly, in that groups of four 2-functions, two from each column, fit together exactly, with exactly four indices in play in each group. Again, this is related to the properties of the VIB arithmetic. So, for example, consider the two adjacent columns with distance 2 and 15 in sorter B in Fig. 1. The darker vertical lines indicate such a well-fitting pair of pairs of comparators. One can describe the composition not only as the original ilv (que (que s2)) −>− vee (vee (vee s2)) but also as parm 5 (parm 6 (que s2 −>− vee s2)), a column containing a composition. This is exactly the kind of rearrangement that is relevant when generating GPU code for such structures, or when looking for sequential tasks of a suitable size, and I have on occasion done this kind of fusion of columns manually. Compilers need to do this kind of transformation to control memory use. So there are good reasons to go beyond the three core combinators. Such rearrangements are relevant in hardware design by refinement too. In ML, one also needs to be able to partition inputs to models in various ways. In partIR, a small number of partitioning transformations is currently used. I do not foresee problems in reproducing those, and then extending to allow more forms, beyond tiling.

Being able to describe very regular structures is practically useful even when designing less regular ones. Often, a less regular structure can be derived, during circuit generation, for instance, from a regular one using a method that I have called *clever circuits* [25]. For example, I used this method to generate small median circuits, by including only those parts of a regular sorter needed to produce the middle output, keeping track of what is known about each potential output and its relationship to the others during the generation. This is an example of the use of non standard interpretation during generation, so that different variants of the final circuit can be tried out in the decision about what exatly to include. Could one use similar techniques in generating deep neural nets? I believe so, and would like to experiment with this, inspired by work on simplifying neural net architectures. My postdoc Y. Yu is working on the use of property based testing (PBT) in neural network development, and PBT could perhaps be used to make choices between different possible architectures for sub-parts of the network.

The above examples are a first illustration of the combination of BMMC permutations and combinators for 2-functions. For 4 inputs, 2 index bits, BMMC permutations can express all 24 possible permutations. (There are 6 non-singular 2 by 2 matrices over GF(2), and 4 possible complement vectors.) But in general, at size $n \times n$, there are $\prod_{j=0}^{n-1}(2^n - 2^j)$ non-singular matrices, as each row or column must be linearly independent of the previous ones, and there are 2^n complement vectors. So for 4 index bits, say, there are $16 \times 15 \times 14 \times 12 \times 8 = 322560$ BMMC permutations, of the $16! = 20922789888000$ possible. The BMMC permutations are the well behaved ones, and deserve further study and formalisation. I have performed initial experiments in Haskell, using Chalmers Lava, for hardware generation and verification.

The big questions are how to generalise beyond binary, and to multiple dimensions in a way that is compatible with practical programming and compilers. In ML, deep learning networks are built from rather regular, linear algebra-oriented computations, matrix multiplications, convolutions and other regular functions, We have experience of generating code from such functions in the context of our Feldspar DSL, but we did not explore novel array indexing (beyond pull and push arrays). We will start by implementing a DSL for linear algebra, concentrating on extending the above algebra on index digits to multidimensional array types.

5 Project description

5.1 Theory and method

The project rests on the theory of affine index digit transformations and associated combinators, aiming to partially restrict array accesses, to give reasoning power while covering a rich set of intricate but regular access patterns. We will implement the combinators and the associated partitioning languages as domain specific languages.

5.2 Time plan and implementation

Year 1

1. Extend the range of combinators beyond those described in section 4 and study the resulting algebra (first for single dimensional arrays). For example, the combinators above work on all the array elements, but one must be able to express working on only some, but maintain the algebra. This is a fine line and combinator design needs to be driven by case studies (linear algebra, the Silveroak project [27], and possibly also collaboration with Intel).

2. Implement software prototype for data parallel programming (in the classic rather than ML sense) both as an EDSL in Haskell for multicore (Sheeran) and in Futhark [16] (postdoc).

3. Extend to multidimensional arrays. Case studies in ML and linear algebra.

Months 13-30

Expressing and implementing partioning.

1. Design and implement a partitioning language, influenced by partIR, to match the combinators and multidimensional array data type, going beyond tiling above a tensor IR.

2. Demonstrate its use in both an ML case study, and in signal processing examples from the earlier Feldspar project. Catalogue known partitioning methods [17].

3. For hardware design, add a notion of clock to the array language. Develop space-time transformations and a notion of refinement. It is currently not clear if the partitioning should be done in the hardware description itself or in a separate partitioning language as in partIR. Again, image processing or DSP case studies (or possibly examples from Intel) will help to decide.

Months 31-48

Further development and new applications of the tools and ideas developed earlier. For example, can clever generation methods such as search or *clever circuits* [25] also be used in an ML setting? Can we efficiently implement distributed ML algorithms due to Johansson at KTH (see below)? What is the set of abstractions (such as pipelining) that should be above the partitioning language, particularly for ML? Can we express previously unexplored ML networks with richer forms of striding?

5.3 Project organisation

I will coordinate and execute the research in language design, development (in Haskell) and case studies. One postdoc (80%) will start mid-2022, concentrating on partitioning in software/ML.

6 Equipment

CHAIR AI infrastructure (Chalmers). Possibly also infrastructure provided by SNIC or AI Sweden.

7 Need for research infrastructure: Not Applicable

8 International and National Collaboration

The collaborations described below will give access to case studies (Google, DeepMind, Intel and Johansson at KTH) and to compiler expertise (DeepMind and Haridi et al at KTH (partIR and MLIR), Henriksen (Futhark), Keller (Embedded DSLs)). Should funding for work with KTH not be obtained, we will try to arrange an extended visit by the postdoc to KTH.

There will be informal collaboration with Google (Singh et al) and DeepMind (Vytiniotis). Singh was my student at Glasgow and we have collaborated earlier. The Futhark developer, Troels Henriksen at DIKU, Copenhagen, has expressed interest in this work and in mutual visits. I have ongoing discussions with Prof. Gabriele Keller from Utrecht University about possible collaboration around shared interests in DSLs for parallel functional programming.

I am co-PI on a proposal to Intel with Carl-Johan Seger (PI, formerly of Intel, now in our group) to work on integrating formal verification into hardware design. If funded, this project will yield case studies for the work described here, though the main emphasis (for my part) is on a higher level DSL and on refinment methods. I am also part of a proposed consortium with Stenstrom (CSE,Chalmers), Johansson, Haridi and Carbone (KTH) aiming to rethink the entire ML stack from the computer architecture upwards. There, again, my role will be to provide high level DSLs linking the abstraction layers, while in the proposed research I will concentrate on low level aspects. However, new distributed ML algorithms by Johansson will be very interesting case studies in this project, and the joint project will also be based on the MLIR compiler infrastructure [20]. Funding has been sought from the WASP NEST programme and from VR (research environment).

I have been involved with both the Array workshop (with PLDI) and the Workshop on Functional High Performance and Numeric Computing (FHPNC, with ICFP). They are building a supportive research community in array programming.

9 Other applications and grants

See section 8 for grant proposals under consideration, including one to VR. Although the proposal has an emphasis on ML in common with this one, my proposed work does not overlap.

I was co-PI on the VR research environment SyTec (testing of cyber physical systems) in our group, but have not been funded by it recently, due to fragmentation and the demands of my work with gender equality (the Genie initiative) at Chalmers. I am now seeking a return to high activity in research related to my core interests: (20%) on this project, 20% in the consortium project with KTH and 15% in the Intel project (should either of these be funded). I will work 10% on the SSF Octopi project (on secure programming of the Internet of Things, finishing 2023).

10 Independent line of research

This project is independent of others in our group. See section 8 for further details. The FP group's main running projects are SyTec and Octopi (see above), both unrelated to this proposal.

References

[1] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available at `https://www.tensorflow.org/`.

[2] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The Design and Implementation of Feldspar. In *22nd Int. Symp. on Implementation and Application of Functional Languages (IFL), Revised Selected Papers*. Springer, 2011.

[3] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[4] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *Proc. Third ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*. ACM, 1998.

[5] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *Int. Conf. on Functional Programming (ICFP)*. ACM, 1996.

[6] S. Browning and P Weaver. Designing Tunable, Verifiable Cryptographic Hardware Using Cryptol. In D. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance*. Springer, 2010.

[7] K. Claessen, M. Sheeran, and B.J. Svensson. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *Proc. 7th Workshop on Declarative Aspects and Applications of Multicore Programming (DAMP)*. ACM, 2012.

[8] Koen Claessen, Mary Sheeran, and Satnam Singh. The design and verification of a sorter core. In *Correct Hardware Design and Verification Methods (CHARME)*. Springer, 2001.

[9] T.H. Cormen. Fast Permuting on Disk Arrays. *Journal of Parallel and Distributed Computing*, 17(1-2), January 1993.

[10] Martin Dowd, Yehoshua Perl, Larry Rudolph, and Michael Saks. The periodic balanced sorting network. *J. ACM*, 36(4), October 1989.

[11] David Durst et al. Type-directed scheduling of streaming accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020. ACM, 2020.

[12] Alan Edelman, Steve Heller, and S. Lennart Johnsson. Index transformation algorithms in a linear algebra framework. *IEEE Trans. Parallel Distrib. Syst.*, 5(12), December 1994.

[13] Donald Fraser. Array permutation by index-digit permutation. *J. ACM*, 23(2), April 1976.

[14] Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. Introducing Kansas Lava. In *Implementation and Application of Functional Languages*. Springer Berlin Heidelberg, 2010.

[15] Peter Henderson. Functional geometry. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, LFP '82, page 179–187. ACM, 1982.

[16] Troels Henriksen et al. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proc. 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI. ACM, 2017.

[17] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Machine Learning and Systems (MLSys)*. mlsys.org, 2019.

[18] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In *Formal Methods for VLSI Design, ed. J. Staunstrup*. North-Holland, 1990.

[19] Donald E. Knuth. *The art of computer programming, volume 3: sorting and searching*. Addison-Wesley, Reading, 2nd edition, 1998.

[20] Chris Lattner et al. MLIR: scaling compiler infrastructure for domain specific computation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO*. IEEE, 2021.

[21] Tzu-Mao Li et al. Differentiable programming for image processing and deep learning in Halide. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 37(4):139:1–139:13, 2018.

[22] D. Maclaurin, Alexey Radul, M. Johnson, and Dimitrios Vytiniotis. Dex: array programming with typed indices. In *Workshop on Program Transformations for Machine Learning (poster), NeurIPS 2019*, 2019. Software `https://github.com/google-research/dex-lang`.

[23] Brennan Saeta et al. Swift for TensorFlow: A portable, flexible platform for deep learning. In *MLSys pre-proceedings*, 2021. `https://proceedings.mlsys.org/paper/2021`.

[24] Mary Sheeran. muFP, A language for VLSI design. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming, LFP*. ACM, 1984.

[25] Mary Sheeran. Finding regularity: Describing and analysing circuits that are not quite regular. In *Correct Hardware Design and Verification Methods (CHARME)*. Springer, 2003.

[26] Mary Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7):1135–1158, 2005.

[27] Satnam Singh et al. Silveroak project at Google, Formal specification and verification of hardware, especially for security and privacy, 2021. Software `https://github.com/project-oak/silveroak`.

[28] Justin Slepak et al. An array-oriented language with static rank polymorphism. In *Proc. 23rd European Symposium on Programming, ESOP*, volume 8410 of *LNCS*. Springer, 2014.

[29] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s), April 2014.

[30] Bo Joel Svensson, Mary Sheeran, and Ryan R. Newton. Design Exploration Through Code-generating DSLs. *Commun. ACM*, 57(6), June 2014.

[31] Dimitrios Vytiniotis. Declarative Abstractions for Tensor Program Partitioning, Invited talk. In *Proc. 22nd International Symposium on Principles and Practice of Declarative Programming*, PPDP '20. ACM, 2020. Slides `https://dimitriv.github.io/partir-ppdp.pdf`.