Algorithms and Arithmetic: Choose Wisely

Invited Paper

George A. Constantinides
Imperial College London

***Abstract*—I will introduce a semi-formalism to allow us to conceptually reason about the differences between customised arithmetic design, as one might see in FPGA-based compute, and general purpose arithmetic, as one might find in microprocessor design. This framework will, I hope, expose to the reader the reason that we should be thinking carefully about appropriate data representations when designing custom hardware for compute, as well as clearly showing the link between these decisions and algorithmic ones. I will then provide a concrete example from the literature on matrix computation where some careful algorithmic tweaking results in the ability to use fixed-point arithmetic and, hence, far higher performance than would otherwise be achieved.**

I. Problem Setting

Consider a numerically intensive computational task, such as often found in matrix computation. This will frequently come to the designer as a pre-existing algorithm $\mathcal{A}$, often pre-parameterised by types for manipulating approximations to real numbers, most commonly `floats` or `doubles`. The task is: 'implement this algorithm in hardware'. Let us denote this algorithm (using floating point types) as $\mathcal{A}(\mathbb{F})$. Sometimes the task will instead arrive as a problem description $\mathcal{P}$: 'I need some hardware to solve this problem'. We can imagine the two as related by $\mathcal{A}(\mathbb{F}) \in \mathcal{P}$, where we envisage that $\mathcal{P}$ corresponds to a whole set of acceptable algorithms. In practice, when working with a pre-existing algorithm, we often imagine that 'if only we were working in arbitrary precision', then the result of the algorithm would give an exact solution to the problem *i.e.* that $\mathcal{A}(\mathbb{R})$ together with some notion of and bounds on `Quality` uniquely define $\mathcal{P}$. Quality here can consider numerical error but also properties such as the silicon area and energy consumed, and the execution time. Of course this is not the case when starting with a problem description, for example an arbitrary precision implementation of a fixed number of Jacobi iterations does not exactly solve a system of linear equations! Nevertheless, no matter how we come across our problem $\mathcal{P}$, so long as we conceptually have a quality function $\texttt{Quality}_{\mathcal{P}}(\mathcal{A}(\mathcal{D}))$ defining how well Algorithm $\mathcal{A}$ operating with data type $\mathcal{D}$ implements a solution to problem $\mathcal{P}$, we can start our design process.

If we were designing a general purpose microprocessor, then we are not clear beforehand what algorithms will be executed on the microprocessor. Our goal must therefore be to design our numerical representation to work well across a range of problems. One can envisage this as a multi-objective optimization, where $\mathbb{E}$ denotes some kind of expectation operator:

$$\text{maximise}_{\mathcal{D}} : \mathbb{E}_{\mathcal{P},\mathcal{A}(\mathcal{D})\in\mathcal{P}} \left\{\texttt{Quality}_{\mathcal{P}}(\mathcal{A}(\mathcal{D}))\right\} \quad (1)$$

We can interpret this as: 'I'm trying to find a datatype so that problems generally get solved to high quality'. The history of IEEE-754 illustrates the complexity of solving this problem in practice [1]. I argue here that taking these lessons directly to custom hardware is not appropriate, because in this context we should be solving a *different problem*:

$$\text{maximise}_{\mathcal{A}(\mathcal{D})\in\mathcal{P}} : \texttt{Quality}_{\mathcal{P}}(\mathcal{A}(\mathcal{D})) \quad (2)$$

Note the differences: $\mathcal{P}$ is now a free variable, indicating that our hardware is *problem specific*. We are free to choose any algorithm / data type *combination* to maximise quality for this algorithm. It follows directly that the optimal value attained in the latter case is always at least that obtained in the former case, and *this is one of the main reasons we use FPGAs for compute acceleration*; pick your algorithm and datatype to suit your application, rather than trying to recreate processor hardware in an FPGA.

II. Case Study

One very common algorithmic kernel in matrix computation is the Lanczos iteration [2]. This iteration forms the basis of various well-known methods like the method of Conjugate Gradients (CG) [3] and the Minimum Residual method (MINRES) [4] which are in turn used in many large-scale numerical codes. We will use this kernel as a case study, basing this section on [5].

This code is typically implemented using floating-point arithmetic, because the dynamic range of the variables involved can be significant and is data dependent.

Algorithm 1 Lanczos Iteration

```
Require: Initial iterate $r_1$ such that $||r_1||_2 = 1$, $q_0 = 0$,
    $\beta_0 := 1$.

1: for $i = 1$ to $i_{\max}$ do
2:     $q_i \leftarrow r_i/\beta_{i-1}$
3:     $z_i \leftarrow Aq_i$
4:     $\alpha_i \leftarrow q_i^T z_i$
5:     $r_{i+1} \leftarrow z_i - \alpha_i q_i - \beta_{i-1} q_{i-1}$
6:     $\beta_i \leftarrow ||r_{i+1}||_2$
7: end for
```

For an FPGA implementation, fixed-point arithmetic might be preferred for reasons of efficiency, yet in fixed-point absence of overflow cannot be guaranteed for this code. There are tools [6], [7], [8] that could be used to prove bounds on the growth of internal variables, allowing for *a priori* variable scaling, but only for a fixed iteration number i or maximum iteration count $i_{\max}$, not independent of or parametric in these variables.

However, the algorithm itself can be modified to make it more 'fixed-point friendly'. It is shown in [5] that if we replace the matrix $A \in \mathbb{R}^{N\times N}$ with $\hat{A} = MAM$ where M is a diagonal matrix with elements $M_{ii} = \left(\sum_{j=1}^{N} |A_{ij}|\right)^{-1}$, then overflow never occurs and we can safely use fixed-point arithmetic. This transformation is just a change of coordinate system, meaning that all the standard uses of the Lanczos iteration, such as solving a system of linear equations, are equally valid in this new coordinate system. In the paper, it is shown that this led to a sustained FPGA fixed-point performance outstripping the peak theoretical GPU performance by a factor of four.

I believe there are two lessons we can directly draw from this case study: (i) revisiting algorithmic assumptions from scratch can provide significant benefit when deciding on a number representation, and (ii) there is still considerable work to be done on automating such a process. The two alternatives to automation of numerical representation selection are, of course, human effort and reversion to a lack of customisation, *i.e.* back to formulation (1) presented in this paper. Both are commonly used in practice today.

III. Conclusion

By far the most considerable automation work has been done on automating the selection of a data representation while keeping the algorithm fixed. This problem was considered by the VLSI design community and then by the FPGA community, initially in the context of circuits for digital signal processing [9]. This corresponds to a third formulation:

$$\text{maximise}_{\mathcal{D}\text{ s.t. }\mathcal{A}(\mathcal{D})\in\mathcal{P}} : \texttt{Quality}_{\mathcal{P}}(\mathcal{A}(\mathcal{D})) \qquad (3)$$

Note here that $\mathcal{A}$ and $\mathcal{P}$ are free variables, indicating that we are interested in a datatype for a particular algorithm solving a particular problem. Automation of this problem is often differentiated by the class of algorithm considered, and there has been little work on algorithms containing fully fledged control structures [10].

Recently we have made some progress on automating formulation (2) by incorporating automated rewriting rules to allow code refactorisation [11], but there remains much to do in this space.

The introduction of hardened floating-point units in Intel FPGAs provides an interesting opportunity to identify those parts of an algorithm truly benefiting from floating-point versus those opportunistically using floating-point 'because it's available', potentially leading to exciting performance enhancements.

The future of automated analysis for the co-design of algorithm and data representation is very bright. I do hope the interested reader will get in touch - we have work to do together!

References

[1] C. Severance, "IEEE 754: An interview with William Kahan," *IEEE Computer*, vol. 1, 1998.

[2] C. Lanczos, "An iteration method for the solution of the eigenvalue problem of linear differential and integral operators," *J. Res. Nat. B. Stand.*, vol. 45, 1950.

[3] M. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *J. Res. Nat. B. Stand.*, vol. 49, 1952.

[4] C. Paige and M. Saunders, "Solution of sparse indefinite systems of linear equations," *SIAM J. Num. Anal.*, vol. 12, 1975.

[5] J. Jerez, G. Constantinides, and E. Kerrigan, "A low complexity scaling method for the lanczos kernel in fixed-point arithmetic," *IEEE T. Comp.*, vol. 64, 2015.

[6] M. Daumas and G. Melquiond, "Certification of bounds on expressions involving rounded operators," *ACM TOMS*, vol. 37, 2010.

[7] D. Boland and G. Constantinides, "Bounding variable values and round-off effects using handelman representations," *IEEE T. Comp.*, vol. 30, 2011.

[8] V. Magron, G. Constantinides, and A. Donaldson, "Certified roundoff error bounds using semidefinite programming," *ACM TOMS*, vol. 43, 2017.

[9] G. Constantinides, P. Cheung, and W. Luk, "Wordlength optimization for linear digital signal processing," *IEEE T. Comp.*, vol. 22, 2003.

[10] D. Boland and G. Constantinides, "Word-length optimization beyond straight line code," in *Proc. FPGA 2013*.

[11] X. Gao and G. Constantinides, in *Proc. FPGA 2015*.