FPGA-based Electrical Impedance Tomography

Patrick Suggate, Tim Molteno, and Colin Fox
Electronics Research Group, Department of Physics
University of Otago
Dunedin, New Zealand
Email: {patrick,tim,fox}@physics.otago.ac.nz

***Abstract*—We are building a compiler to help map large computations into FPGAs because we are also developing hardware to perform Electrical Impedance Tomography (EIT) in real-time. The EIT inverse problem is non-linear and severely ill-posed and the sampling algorithms that calculate good solutions have a high computational cost. These algorithms require performing many forward-map solves and this is the calculation that is being mapped into dedicated hardware. We expect that the FPGA implementation will be significantly faster, and use much less power, when compared to implementations that use only general-purpose processors. This is because the raw compute-throughput of the hardware will be very high, and code-analysis of our compiled computation shows that the whole computation will fit within just the static RAMs inside a moderate-sized FPGA, therefore eliminating any memory-access I/O bottle-necks created by having to access external RAMs. Analysis has also shown that about 90% of all operations can be of the form of fused multiply-and-add operations and, by using a floating-point/fixed-point hybrid multiply-and-add functional unit, can be mapped efficiently into Xilinx FPGAs. The compiler is then needed to schedule our computation across an array hundreds of parallel functional-units, and efficiently utilising the available buses and static RAMs, to maximise hardware utilisation and, therefore, solve-rate.**

I. Introduction

We are building a compiler to assist in mapping a very-large computation into a FPGA (Field-Programmable Gate Array). This computation is a statistical inference algorithm that is used to evaluate the inverse-problem arising from Electrical Impedance Tomography (EIT). Generating the desired number of samples requires (as many as) hundreds-of-thousands of solves of the forward-map of the simulated physical system. The forward-map is evaluated to calculate the likelihood of a proposed set of conductivities at each step of a Markov process. We are building hardware to evaluate this algorithm in real-time as part of an EIT imaging system that can be used for monitoring industrial processes. This requires extremely high-performance compute-hardware and an efficient mapping of our algorithm into this hardware. A high-level overview of our proposed design is shown in Figure 1.

We have decided to implement this algorithm within a FPGA because FPGA compute-throughput can be very good, both in terms of performance-per-dollar and performance-per-watt [1], [2]. A downside to using FPGAs is that obtaining real-world performance that is close to this potential performance can be very difficult [2]. What follows is a method of using the partial evaluation of additional information that is available at compile-time to produce a simplified computation to map into the FPGA.

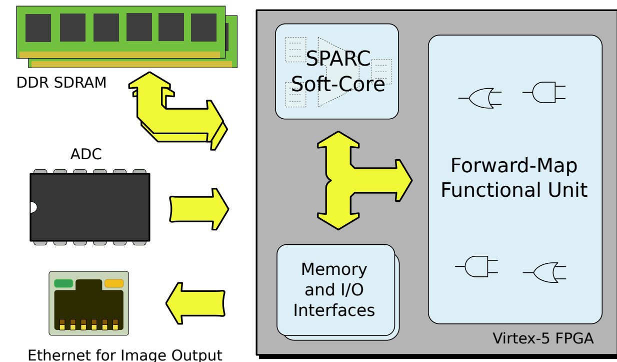

Fig. 1. An overview of our proposed design for real-time EIT.

II. The Electrial Impedance Tomography Algorithm

The EIT inverse problem can be informally stated as: given a set of voltages and currents measured at electrodes at the boundary of a region, what are the conductivities within this region? An example of an EIT setup is shown in Figure 2. A Metropolis-Hastings MCMC (Markov-chain Monte Carlo) algorithm is used to generate samples from the posterior distribution of solutions. At each transition of the Markov-chain, an accept probability needs to be calculated by solving the forward-map for the proposed state [3]. Evaluating the forward-map requires solving a system of equations within the FPGA, and these are equations are represented by a sparse-matrix. This system of equations is generated from the proposed conductivities and then solved at every iteration of the algorithm. To generate the desired number of independent samples requires at least tens-of-thousands of solves of the forward-map of the simulated system (and currently more than 100,000 steps are performed).

The EIT forward-map solve, which is performed at each step of the MCMC algorithm, is a fairly-large computation itself and at each evaluation it solves a large system-of-equations, therefore requiring many floating-point operations (FLOPs). The system of equations is generated by applying the Finite-Element Method (FEM) to the Boundary Value Problem (BVP) of the conductivies of a fixed mesh spanning this region, which also includes the electrodes on the boundary of this region.

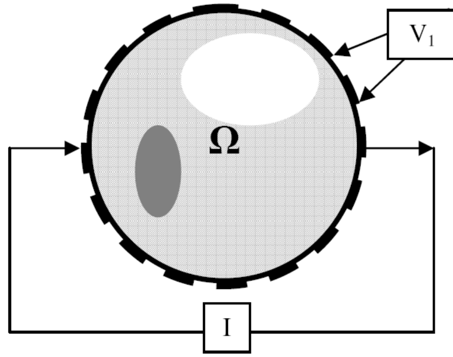

Fig. 2. EIT Diagram of 16 electrode EIT experiment. The potential V1 is measured after a current I has been imposed across the core . (Molinari 2003)

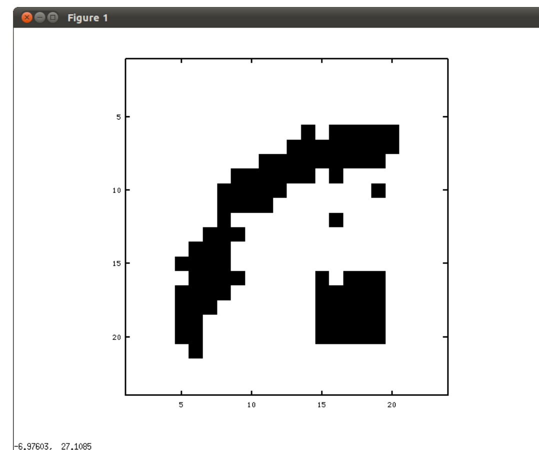

Fig. 3. A sample from the posterior distribution calculated by the EIT testbed, which is used for testing and development.

The EIT forward-map consists of a fixed-mesh, finite-element system corresponding to the conductances of each of the elements within the domain. Generating samples from the posterior distribution (the probability distribution of the viable solutions) involves:

- Assembly of the system matrix and applying the boundary conditions,
- Performing a Cholesky factorisation,
- Triangular solves for each of the multiple right-hand-side vectors (of boundary currents),
- Calculating the proposal likelihood (by performing a sum-of-squared-differences on the measured and simulated electrode-potentials).

For the EIT testbed (see Figure 3), that we are using for compiler development, the total FLOP-count for each solve of the FEM system is about 2.2 million (see Table I), and this solve-step is performed at every iteration of the Metropolis-Hastings MCMC algorithm. The calculated (but unnormalised) likelihood value represents the (scaled) probability that the set of conductances passed to the solver is a solution to the EIT inverse problem. This likelihood value is then combined with any prior knowledge of the problem to give an acceptance probability. If the proposed set of values for the conductances (the current state of the Markov chain) is accepted then this becomes a conditional sample.

With the current implementation of this algorithm it can take more than one hundred of these conditional samples to generate just one independent sample from the posterior distribution, but the conditional-sample acceptance-rate at each iteration is only about 0.1, so more than a thousand solves of the forward-map are needed for just one independent sample[1] . Generating the desired number of samples from the posterior distribution typically involves performing hundreds of thousands of these solves as the Markov chain traverses the state-space of the problem[2]. Additional details of the algorithms discussed here can be found within [3].

[1]Also, thousands of initial iterations and solves are needed before the Markov-chain even converges to the target distribution, and therefore before we can even start taking conditional samples, and this is called the ”burn-in” period [3]

[2]Yet this algorithm is very advanced and represents the state-of-the-art [3].

The computation outlined above has properties that appear well-suited to calculation within a FPGA:

- The size of the intermediate data (a sparse-matrix and some working memory) are less than one MB, so can be stored entirely within an FPGA's internal SRAMs.
- The algorithms used within the solver have a high degree of parallelism [4], [5].
- Due to the mesh being the same at each iteration, the sparsity pattern of the system-matrix remains the same, so the calculation has the same "shape" for each solve-step, and so the same sequences of operations are repeated many, many times.
- Fused Mulitply-and-Add (FMA) functional units, which tend to be the dominant arithmetic operations used by matrix algorithms, can be built moderately cheaply within Xilinx FPGAs by using a hybrid floating-point/fixed-point FMA unit. The accumulator of this FU can use fixed-point partial sums, and renormalised only as necessary, as was shown in [6].

But mapping this computation into an FPGA, and effectively exploiting the available parallelism, would likely be extremely time-consuming using existing tools.

III. Compile-Time Partial Evaluation of the Problem

At program compile-time there is often useful information available that could be used to compile a more efficient computation, but such information is often domain-specific, and therefore difficult for general-purpose compilers to make use of. Here we present a method to use some of this additional information when compiling a large computation into a form more-suitable for FPGA-based computation. In the example presented below, properties known at compile-time of the sparse-matrices are used to simplify the resulting computation. The method first evaluates the algorithm within an appropriate monad[3], which in this case numerically evaluates any known

[3]A full discussion of monads is beyond the scope of this paper, see [7] for more information, but within the scope of this document consider a monad to be a computational context, i.e. the computation occurs within this context, and according to the properties of that context.

values, and performing symbolic evaluation on the unknowns. This process is called partial evaluation [8] and it produces a simplified Intermediate Representation (IR) of the problem, which is then further transformed by subsequent stages into a form more suitable for FPGA implementation.

By using information available at compile-time, partial evaluation of the given algorithm allowed extremely-aggressive code-unrolling to be performed as well as eliminating all memory indirection[4]. This is then followed by multiple transformation and optimisation stages, and then passed to an appropriate compiler back-end, which currently generates C code for testing purposes. For FPGA-based implementations, the compiler backend needs to schedule the task across hundreds of functional units, and task-clustering and scheduling algorithms seem suited to this task [9].

IV. Optimisations

The compiler presented here has an additional partial-evaluation stage that typical compilers do not have. This extra information allows certain optimisations to be performed more aggressively than typically done by other more-traditional compilers. This section briefly outlines some of these optimisations that are currently implemented and those that are planned for the compiler.

A. Unused-Calculation Elimination

As an example, a naive global-stiffness-matrix assembly step would calculate the full symmetric sparse matrix, even though both the upper- and lower-triangular are identical, and the Cholesky factorisation that is performed during the solve step only needs a diagonal matrix, i.e. just over half of the matrix elements, the elements along the main diagonal and either the strictly upper or lower elements. A compiler that is given a dataset with the correct shape can automatically prune all unneeded calculations that would otherwise be performed by code generated by a naive compiler. In the above case of assembling the system matrix, the eliminated calculations would be the calculation of any of the stricly-upper values of the system matrix since only the lower-triangular entries of the matrix are passed to the solve-step.

B. Constant Propagation

Any constants known at compile-time are propagated throughout the computation, and many can subsequently be simplified or eliminated during partial-evaluation and optimisation stages. An example of constants that were combined or elminated were those used for stiffness-matrix assembly.

[4]Sparse matrices avoid storing (most) zeros of a matrix at the expense of requiring additional data-structures which contain information necessary to determine the locations of the non-zero elements of the sparse-matrix. This typically results in matrix-element accesses requiring additional steps to calculate the memory address, and this process usually consists of memory indirection [5], i.e. first fetching the memory-address (or offset) of the desired element from a table, which is also stored within system memory, before the value of the element itself can then be fetched.

C. Strength Reduction

Functional Units (FU) for some operations can be expensive in gates, like square-root or divide, but can be converted to other operations leading to simpler and faster FUs. During testing, the following conversions typically lead to better Directed Acyclic Graph (DAG) optimisations or more efficient FGPA implementation:

- Conversion of subtracts into adds and negates allows more flexible reordering of inputs into tree-adders, and this can be easilty done since
$$a - b = a + (-b). \tag{1}$$
- Fusion of negate-operations into other operations to save a clock-cycle of latency, as a floating-point negation requires just inverting the most-significant bit of the bit-field of the number.
- Division-by-a-square-root operations, which are performed during Cholesky factorisation, are transformed into multiply-by-the-inverse-square-root operations, i.e.
$$\frac{a}{\sqrt{b}} = a \times b^{-\frac{1}{2}}. \tag{2}$$
- When possible, floating-point additions are converted to fixed-point accumulations, and in most cases these can then use the DSP48E prmitives built into Xilinx FPGAs.
- Conversion of divisions into multiply-by-the-recipocal operations, because from analysis of the code, divisions were typically performed many times with the same denominator, and multiplications are cheaper in FPGAs, so this is then:
$$\frac{a}{b} = a \times b^{-1}. \tag{3}$$

D. Critical-Path Length Reduction and Increasing Parallelism

The desired computation can never take less time than the sum of the FU-latencies of the longest chain of data-dependencies for the computation. The length of this chain is called the Critical Path Length (CPL) of the computation. A short CPL requires good initial choices of algorithms to be passed to the compiler as well as appropriate optimisations that are subsequently performed by the compiler's transformation stages.

Two properties of addition and multiplication, commutivity and associativity, allow inputs into these operators to be exchanged arbitrarily[5]. Likewise, subtractions and divisions can be transformed into additions and multiplications by applying the appropriate inverse-operators, negate and reciprocal, to the second-input. If these lead to a reduction in CPL and/or operation count, the compiler will perform these transformations.

Operations that reduce CPL typically increase parallelism as well, so exploiting the commutative and associative properties of addition and multiplication, to reduce CPL, leads to the transformation of chains of sequential-adds, and sequential-multiplies, into tree-adders, and tree-multipliers, respectively. The CPL of a sequential adder is O(N) whereas a tree-adder has path-length of O(log$N$) . Another important gain

[5]Floating-point addition is not associative in general, as order of addition can effect the numerical accuracy of the answer [10].

from this transformation is that the parallelism of a sequential adder is O(1) but the parallelism of a tree-adder is O(N) (see Figures 4 & 5 to see what the relative "shapes" of these operations are).

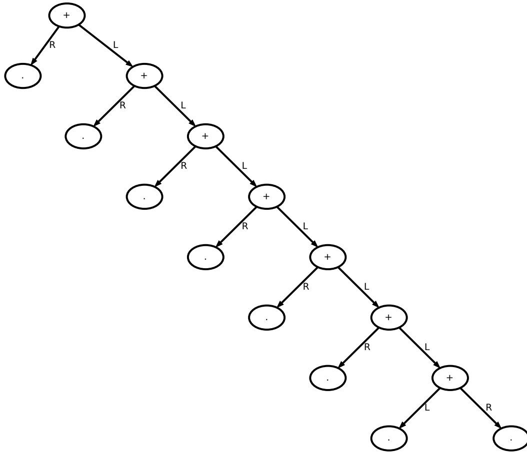

Fig. 4. This sequential adder, which adds eight numbers together, has a maximum path-length of seven additions.

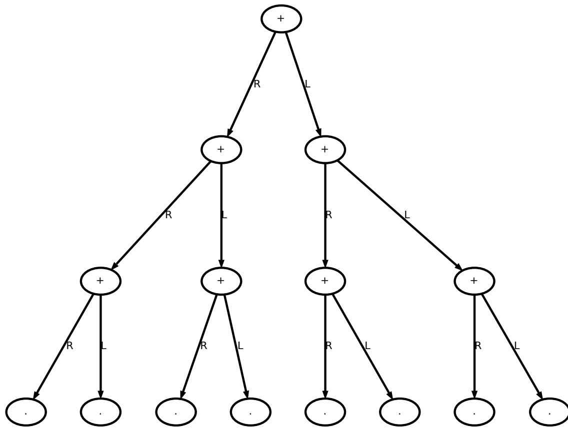

Fig. 5. This tree adder, which adds eight numbers together, has a maximum path-length of just three additions.

E. Data-Locality Optimisations

Appropriate partitioning algorithms allow the the results of computations to be produced both temporally and spatially closer to where they are needed, reducing data transportation requirements. The best case is when as a calculation proceeds, the data needed at each step, the inputs into each FU, are stored locally in a cache or SRAM, or sitting on the outputs of a neighbouring FU, so then it can be transferred only across small distances and with little latency. Because the sparsity-pattern of the input matrix is known at compile-time, all data-locality calculations can be done at this time too, potentially allowing this compiler to produce more efficient code, than can typically be done.

In terms of an FPGA-based computation, if the needed data has only just been calculated, and by a FU across the other side of the FPGA, it has to be transferred via buses to the FU where it is needed, requiring the use of a limited resource (on-chip buses) and incurring a latency cost. These cross-chip data-transfers can reduce the efficiency of the hardware if FUs have to sit idle while they wait upon data, and simply increasing bus bandwidth, and reducing bus latencies, requires more chip area, potentially reducing the number of FUs.

F. Subtree Pattern Matching and Operator Fusion

The code-size of the completely unrolled and optimised computation can be extremely large, since all program-loops have been unrolled and replaced with the many similar statements. Contained within this optimised IR are many smaller and repeated computations of the same shape, like tree-adders (see Figure 5) of various sizes. Subtree pattern-matching algorithms [11] can then be used to recognise and count these repeated sequences of operations.

After subtree-pattern matching has been performed, repeated sequences of operations can be combined into higher-order operators, if an operation is repeated frequently enough. This can also reduce the required code-size, because a sequence of operations can be replaced with just one new higher-order operator. High-order operators, like a tree-adder, can also increase the efficiency of the functional units (in terms of area and latency) because some intermediate logic can be eliminated, for example some of the denormalisation and renormalisation stages that are required for floating-point addition.

V. The C Backend and Compiler Testing

For testing and development, we generate C code from the internal representation used by the compiler. The internal format used by the compiler is basically an array containing the nodes, edges, and their labels, of the computation graph. From this representation, it is straight-forward to generate C code [6] because the internal representation is roughly equivalent to a simplified Static Single Assignment (SSA) graph repesentation, for example, like that used by the LLVM compiler [12].

The motivation for the C backend was so that any changes to the compiler that produced incorrect output could be identified automatically. The code generated by the C backend can be compiled and evaulated using the same data-sets as both the original Octave code and the Haskell code, and the results compared for correctness. Errors produced during optimisation by the compiler (either by bugs, or by faulty assumptions about which transformations can be applied, and when) can be difficult to find unless equivalence with the original code can be easily (and regularly) checked. Another complication is that, with large numerical computations, even the order of the operations can be critical for preserving numerical-stability, i.e. even if the results would be identical when using infinite-precision arithmetic, two algorithms could produce completely different answers when using finite precision, so an automated "sanity-check" is extremely useful.

[6]Emitting basic C-code is amazingly easy, the total number of lines of code, including comments and spaces, for the C-code generator is only 52 lines. The compiled C-code runs quite slowly though, as it is far too unrolled to fit in the processor's L1-cache.

VI. Results

The finite-element solve-step was compiled to C to evaluate any improvements produced by our compiler, both with and without optimisations enabled. The results of this are shown in Table I. The optimisations performed were:

- Common sub-expression elimination phases, and these were repeated after any signficant IR transformations by other optimisation stages.
- Replacement of subtract operations with additions and negations, for the given heuristic.
- Fused negate operations into other operations.
- Replaced divisions with both a multiplication and a reciprocal.
- Reordering sequences of repeated operators, either multiple additions or multiple multiplications, to reduce tree-height (see Figures 4 & 5).
- Pruning any code that was not used for calculating the final outputs of the computation, or were not used by any subsequent operations.
- Paired off any repeated inputs into either tree-adders or tree-multipliers to reduce memory bandwidth requirements.
- Reordering of the multiple-inputs into tree-adders, or tree-multipliers, that were more likely to produce duplicated sub-expressions which could then be eliminated.

TABLE I
Comparison of critical-path lengths and floating-point operation counts of the output of the compiler for various parts of the finite-element solve, and with optimisations either off or on.

Computation	Critical Path	kFLOPs
Optimisations Off		
Matrix Assembly	7	36
Cholesky Factorisation	703	1056
Triangular Solve	10481	1097
Partitioned Inverses	7437	1720
Optimisations On		
Matrix Assembly	6	19
Cholesky Factorisation	695	1037
Triangular Solve	681	675
Partitioned Inverses	287	1210

A. Preliminary Hardware Synthesis

So far, only floating-point addition FUs have been built, and these have not yet been fully optimised and debugged, as getting the Leading-Zero Anticipation [13] circuit right can be fiddly and error-prone. Floating-point addition FUs were created first as these are the most difficult to build and debug, both in terms of complexity and performance, and would give a good indicaton of the upper-limit for performance. Table II show the advantages that can be gained by specifying additional placement information, and also by only supporting a subset of the IEEE-754 specification. The Xilinx cores are "black-boxes" so it is unknown as to how much placement information that they contain[7].

TABLE II
Provisional and approximate results for the relative performance of our custom floating-point functional-units versus those that were generated using the Xilinx Coregen™ logic-core generation tool. The performance numbers are for the slowest (-1) speed-grade of Virtex-5 FGPAs.

Functional Unit	Pipeline Stages	Frequency (MHz)	Latency (ns)	Logic Used
24-bit Adders				
Coregen 24-bit add	5	200	25	350
Coregen 24-bit add	11	350	31	400
Lava HDL 24-bit add	5	370	14	280
Lava HDL 24-bit add	6	440	14	336
32-bit Adders				
Coregen 32-bit add	5	190	26	440
Coregen 32-bit add	12	350	34	600
Lava HDL 32-bit add	5	350	14	432

VII. Discussion

The results presented in Table I show that the partially-evaluated and optimised computation is able to successfully exploit far more parallelism, while also reducing FLOP-count and calculation latency, and completely eliminating all memory indirection. The most significant gains are from the improved triangular-solve step. Since only a subset of the electrodes are used, a signficant number of calculations have been eliminated outright, and reordering has vastly-reduced the critical path length of the computation.

Improvements were less significant for the Cholesky factorisation step as this is a heavily-studied problem [5], [14], [15], as its computational complexity is greater than the subsequent solve-step, and we were already using an advanced and heavily-refined algorithm [5]. But since we have the required shape information available for the Cholesky factorisation, the small gains made here are still useful as they are effectively free.

The results shown in Table I contain the FLOP-counts and CPL for two different matrix-vector solve algorithms. The first is the standard triangular-solve algorithm [5] which has a fairly-long critical-path, as it is by its nature a very linear algorithm. We have managed to reduce its long paths of subtractions by transforming the subtractions into additions which is then followed by sequential-adder to tree-adder transformations (see Figures 4 & 5 to see what this looks like).

The second solve algorithm uses an algorithm that calculates the partioned-inverse of a triangular matrix [4], so the subsequent solve-step is now just two matrix-vector multiplications, but this clearly requires more FLOPs (see Table I). There is another algorithm which uses a heuristic to perform a combination of the above two algorithms, achieving a different trade-off between path-length and FLOP-count [16], but evaluating

[7]The logic-used statistic for the Xilinx floating-point FUs is extremely generous to Xilinx because these logic-cores have significant "spill-over" into the surrounding logic, as well as there are many "holes" within the regions that they are placed, but this spare logic is unlikely to be utilised efficiently by anything else.

the algorithm leading to efficient hardware implementation is future work.

A. Floating-Point Hardware Considerations

We have simulated many EIT computations at varying levels of numerical precision (but this is currently unpublished) to verify that the EIT algorithm will still converge when using far lower numerical precision than the commonly-used IEEE-754 double-precision floating-point format. Use of lower precision allows far more FUs to be added to the design, and 24-bit floating-point units could even be considered "overkill" for the current task. Unfortunately, even the fastest Xilinx floating-point units, that can be generated using the Coregen™ tool, have very long pipeline-lengths, as shown in Table II. To achieve high solve-rates, total latency is important, as the solve-time is sum of the latencies of all operations within the critical-path of the computation. By only implementing the necessary subset of the IEEE-754 specification (and this has been experimentally verified that this subset still gives correct results), and by specifying additional placement information, total latency has been approximately halved. The final design is expected to operate at 500 MHz, and with just five pipeline stages for the floating-point FUs, on the upcoming Xilinx Kintex7™ FPGA architecture (and this has been checked using the Xilinx synthesis tool).

Parameterised floating-point units are being built using the Lava Hardware Description Language[8] (Lava HDL). The Lava HDL allows combinators to be used to specify relative-placement information for all of the logic elements, so that even parameterised circuits can be generated that have performance close to hand-tuned implementations, and performance far superior to the circuits generated by the Xilinx automatic Place-And-Route (PAR) tool (see Table II). Parameterisable-width floating-point units allow the size of the floating-point operations to be changed, depending upon the precision requirements. The Lava HDL placement-combinators means that the rectangular tile-shape is maintained even as the width-parameter is varied, and so the resulting FUs can then be easily tiled into arrays of memories, buses, and floating-point FUs.

VIII. Conclusion

The cost of partial-evaluation of algorithms that are repeated many times, and with some of the same values each time, can be amortised in at least some situations, as we have shown here. This seems to be particularly true for FPGA implementations since the development effort tends to be large anyway, and any reduction in complexity of the final system should easily justify the extra pre-processing steps. More work is needed to determine how beneficial any improvements due to the simplification of the final calculation really are, and this is future work.

[8]Unfortunately, the status of Lava is "experimental", and the documentation is very old, so Lava is very hard to learn and work with. For more information about the Lava HDL, though now slightly out-of-date, see:
http://raintown.org/lava/

A. Future Work

Some directions for future work include:

- Partitioning the DAG using ParMETIS.
- Subtree isomorphisms to Reduce Code Size
- Complete the mapping of the computation into hardware. The final imlementation is planned to consist of: SIMD (Single-Instruction Multiple-Data) architecture processors optimised for floating-point throughput; fetch-and-store FUs, which support gather, permute, and scatter operations, to control the flow of data within the FPGA; and special-purpose FUs for specific sequences of frequently-repeated higher-order operations.
- Parallelising the Compiler to reduce compilation time.
- FPGA-assisted compilation for subtree pattern matching. And maybe even hardware assisted Just-In-Time (JIT) compilation.
- Efficient CPU compiler backends (for example, supporting the x86 architecture, and probably using another compiler backend, like LLVM's), and possibly even support for GPUs, using an OpenCL backend.

References

[1] P. Suggate and T. Molteno, "N-Body Gravitational Simulation Using Dedicated Hardware," in *Proceedings of the 13th Electronics New Zealand Conference, ENZCon'06*, 2006, pp. 147–150.

[2] T. El-Ghazawi, "The Promise of High-Performance Reconfigurable Computing," *IEEE Computer*, 2008.

[3] G. Nicholls and C. Fox, "Prior Modelling and Posterior Sampling in Impedance Imaging," *Bayesian Inference for Inverse Problems*, pp. 116–127, 1998.

[4] F. Alvarado and R. Schreiber, "Optimal Parallel Solution of Sparse Triangular Systems," *SIAM Journal on Scientific Computing*, vol. 14, pp. 446–460, 1990.

[5] I. Duff and J. Reid, "The multifrontal solution of indefinite sparse symmetric linear equations," *ACM Trans. Math. Software*, vol. 6, pp. 302–325, 1983.

[6] F. de Dinechin, B. Pasca, O. Cret, and R. Tudoran, "An FPGA-specific Approach to Floating-Point Accumulation and Sum-of-Products," in *ICECE Technology, 2008. FPT 2008. International Conference on*, 2008, pp. 33–40.

[7] P. Wadler, "Comprehending Monads," *Mathematical Structures in Computer Science*, vol. 2, pp. 461–493, 1992.

[8] Y. Futamura, "Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler," *Higher-Order and Symbolic Computation*, vol. 12, pp. 381–391, 1999.

[9] M. A. Palis, J.-C. Liou, and D. S. Wei, "Task clustering and scheduling for distributed memory parallel architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, pp. 46–55, 1996.

[10] IEEE, "754-2008 IEEE Standard for Floating-Point Arithmetic," *IEEE Standards*, pp. 1–58, 2008.

[11] R. Cole and R. Hariharan, "Tree Pattern Matching to Subset Matching in Linear Time," *SIAM Journal on Computing*, vol. 32, pp. 1056–1066, 2003.

[12] E. Eckstein, O. Knig, B. Scholz, and A. Krall, "Code Instruction Selection Based on SSA-Graphs," *Lecture Notes in Computer Science*, vol. 2826, pp. 49–65, 2003.

[13] M. S. Schmookler and K. J. Nowka, "Leading zero anticipation and detection?a comparison of methods," *Computer Arithmetic, IEEE Symposium on*, vol. 0, p. 0007, 2001.

[14] D. Irony, G. Shklarski, and S. Toledo, "Parallel and fully recursive multifrontal sparse Cholesky," *Future Generation Computer Systems*, vol. 20, no. 3, pp. 425–440, 2004.

[15] N. Gould, J. Scott, and Y. Hu, "A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 2, p. 10, 2007.

[16] P. Raghavan, "Efficient Parallel Sparse Triangular Solution with Selective Inversion," 1995.