Symbolic Generation of Parallel Solvers for Inverse Imaging Problems

Jessica L. M. Pavlin pavlinjl@mcmaster.ca
Christopher Kumar Anand anandc@mcmaster.ca
Department of Computing and Software, McMaster University
Hamilton, Ontario, Canada L8S 4K1

Medical Image Reconstruction is characterized by requirements to (1) process high volumes of data quickly so sick patients need not wait or be called back for re-imaging, and (2) provide the highest level of software quality, including both correctness and safety. Parallelization today allows very high levels of performance, at the cost of duplicating hardware resources and increasing software complexity. Software complexity makes it harder to ensure correctness and safety in a domain where errors can easily arise out of miscommunication between domain experts and software developers.

To solve these problems, we propose a development environment which transforms simple mathematical models into parallel programs via term graph transformations and code generation to an intermediate language we call Atomic Verifiable Operations (AVOps), which we have previously introduced together with a linear-time verification method. In this paper, we will describe a variation on AVOps which supports multi-core systems, and, for the first time, the Coconut Expression Library, a Domain Specific Language (DSL) for domain experts to specify mathematical models in the form of objective functions, and for performance experts to provide rule-based transformations to compute gradients, algebraic simplifications and finally parallelizations expressed using AVOps.

Computing and Software Report CAS-14-05-CA

Contents

The authors thank NSERC, MITACS, Optimal Computational Algorithms, Inc., and IBM Canada for financial support.

The Coconut (COde CONstructing User Tool) Project aims to provide each domain expert collaborating on high-performance, safety-critical scientific software their own interface allowing for the separate specification of all ingredients from mathematical models down to efficient instruction scheduling, with all transformations providing proofs of correctness and safety. Initial work focussed on instruction selection and scheduling [1], already providing novel views on abstraction, including a declarative assembly language. Later components have added higher-level abstractions, and intermediate languages essential for proofs of safe parallelization.

This paper provides the first description of the highest level of abstraction, the Coconut Expression Library (CEL), in which domain experts can transparently specify mathematical models and regularizers, independent of implementation considerations, and of subsequent optimized code generation—including algebraic simplifications, symbolic differentiation, common subexpression elimination (CSE), and parallelization—via term graph transformation rules. In the following, we define by example the type of model for which we would like to generate efficient code; we exhibit examples of domain specific languages for modelling and term graph transformation; we explain a novel approach to differentiation required; we specify a verification method for eliminating parallel hazards which extends our previous framework for distributed computation; and finally, we describe parallelization rules targeting this framework.

1 Motivation: Using Phase Contrast Angiography (PCA)

We use Magnetic Resonance Imaging (MRI) as a use case. MRI is interesting from a computational stand point, as the manner in which each MRI imaging experiment is carried out allows for encoding of a wide variety of physically important properties, which can be fit with varying complexity of mathematicla models. For this particular application, we focus on a problem in phase contrast angiography, or PCA. In PCA, velocity fields are encoded as a phase modulation to a complex-valued image. Velocity profiles generated via PCA can help physicians diagnose and track congenital heart disorders, brain aneurisms, and circulatory diseases, and plan brain surgery to avoid major blood vessels. Unfortunately, due to the nature of the data, image acquisition times are prohibitive, leaving PCA techniques under-utilized in current practice. Using techniques collectively known as parallel imaging, image acquisition times can be reduced by selectively undersampling the

MRI data, instead using data from multiple, geometrically distinct receivers to supplement the usual Fourier encoding.

While collecting the full MRI data for this experiment requires the patient to be in the machine for upwards of 40 minutes. Recent reports of time to reconstruct the flow fields times are even worse: for example, an image size of 128×128 required 13 minutes for computation [7], and a 3-dimensional $128 \times 128 \times 128$ volume required between 180 minutes and 780 minutes for computation [8]. Some experiments collecting less data and using better mathematical optimization techniques have demonstrated some promising results for low resolution images. For example, the problem has been solved using 28% of the data at a resolution of 128×128 (meaning the patient spends about 12 minutes in the MRI machine) in five minutes [6]. Typically, however, integrating parallel imaging with model based reconstructions adds significant complexity to the model both in form and in the size of the accompanying data.

PCA is especially interesting because, by adding multiple phase modulations to a base MRI image for comparison, it contains redundancy which lends itself to sparse sampling and regularization. Next we explain the physical model for PCA, exhibiting (in CEL) a problem-specific regularizer which can benefit from independent mathematical models and code optimization strategies.

MRI samples magnetic fields created by precessing hydrogen nuclei. The frequency of precession is proportional to the strength of an external magnetic field, just as a top spins faster on earth than on the moon. We use linear variations in field strength to encode geometric information, and by balancing opposite variations in time, create phase variations proportional to velocity. See [9] for details. The important point is that all MRI samples the Fourier Transform (FT) of the model variables we want to know, and in the case of PCA, the model variables are further modulated by the velocity:

$$\widetilde{m}_G = \rho \cdot e^{iG \cdot V} \tag{1}$$

where the known gradient G, and unknown velocity V and base image ρ combine to produce the measurement $m_G = \mathrm{ft}(\widetilde{m}_G)$.

This formulation is sufficient when complete datasets are collected, but for accelerated (sparsely sampled) experiments, our model incorporates the physical measurements, which come from the Fourier Transform of the complex image. Since MRI measurements contain independent, identical, normally distributed errors, the least-squares solution is a maximum likelihood

estimate (MLE).

$$\sum_{G\in\mathcal{G}} \left\| m_G - \pi_G \,\mathrm{ft}(\rho \cdot e^{iG\cdot V}) \right\|^2, \tag{2}$$

where the norm is the ℓ_2 vector norm summing the individual differences over the subset of measurements represented here by projecting, for each G—via π_G onto a linear subspace of the complete measurements of the Fourier Transform, and $\mathcal{G}$ is a set of gradient sensitizations—which are parameterized by vectors in $\mathbb{R}^3$. Realistic models are further complicated by the fact that multiple antennae are used to measure modulated views of the data, introducing exploitable redundancy.

While sparsely sampling the Fourier Transforms saves time, it leads to underdetermined or numerically ill-conditioned models. This can be mitigated by regularizing—adding a penalty term to the MLE (2)—as introduced by Tikhonov [10]:

$$\min \left\{ \sum_{G\in\mathcal{G}} \left\| m_G - \pi_G \,\mathrm{ft}(\rho \cdot e^{iG\cdot V}) \right\|^2 + \lambda R(V)^2 \right\}, \tag{3}$$

with λ providing a relative weighting of the a priori information represented by the regularizer R versus the need to fit the data. The simplest such penalty is the ℓ_2-norm of the model variables. Other common penalties involve ℓ_2- or ℓ_1-norms of differences of neighbours.

Such penalties can and should be predefined functions in CEL, but this would exclude the definition of novel problem-specific penalties. In the case of PCA, Conservation of Mass (CoM) can be formulated as a penalty, since any flow into the volume of interest should be balanced by outflow. This property has been used as a regularizer in meteorological optimization problems [5, 2], where it has reduced solving time, but not to our knowledge, in medical imaging. Consider a cubic sample at coordinates (i, j, k) in the discretization of the imaging volume. The difference between the velocity component entering versus the component leaving opposite faces results in a gain or loss of material. Summing over three pairs of faces, we obtain a quantity which should be zero. There are different ways of approximating this, but the simplest is

$$(v_{x(i+1,j,k)} - v_{x(i,j,k)}) + (v_{y(i,j+1,k)} - v_{y(i,j,k)}) + (v_{z(i,j,k+1)} - v_{z(i,j,k)}) \tag{4}$$

which gives a scalar quantity at each sample. Since the discretization is an approximation, we do not expect this quantity to be exactly zero, so we

form a penalty function by summing the squares of this quantity over all samples.

To accommodate standard as well as such problem-specific regularizers in a transparent and extensible way, we provide, in CEL, a structured higher-order function we call a Sparse-Convolution-Zip (SCZ), which forms new discretized quantities by combining neighbouring values at fixed offsets from one or more input discretized quantities. In the study of partial differential equations, such operations are called stencil computations. Providing such a construct is allows us to treat discretization vectors as first-class objects and compute efficiently using them.

With this problem as a guide, we set the goals for the **Coconut Expression Library (CEL)**, in order to capture such problems:

1. create a DSL close to current use by applied mathematicians and scientists in informal model development, see § 2; this must include
2. first-class vector variables, § 2,
3. derivation with respect to vector variables, § 3,
4. built-in linear operators with optimized implementations like the FFT, § 2,
5. user-defined zip-shift-map operations generalizing stencils, § 2; and
6. an interface for computer scientists to specify simplification and parallelization operations algorithmically, §5;
7. an efficient parallel run-time system to execute generated code, § 4;
8. an efficiently verifiable model of synchronization, § 4;
9. an efficient parallel unconstrained optimizer into which to plug generated function and gradient evaluators, which we will describe in a future paper.

2 Domain Specific Languages (DSLs)

Domain Specific Languages trade off general expressivity for the right level of abstraction for a particular domain. *Embedded* DSLs are really libraries in another language [4], often a functional language like Haskell. We use two DSLs to organize the work of CEL, each tuned to a different set of domain experts.

2.0.1 Model Expressions

The first DSL is the declarative algebraic expression language designed to look like applied mathematics as used in physical models and optimization problems. To save trouble later—when efficient code generation will depend on not recalculating common subexpressions—each expression is encoded as a node in a hash table, where each entry is either a basic node (a variable, differential, or constant) or an operation combining multiple other nodes. The hash table is implemented using a Haskell `IntMap`, but this type is wrapped in classes which hide the implementation from the user. The user can even work interactively to build up an expression, such as a 3D Fourier transform, $\mathrm{ft}(x + iy)$ of a complex vector composed of real vectors x and y:

```
> let x = var3d (16,16,16) "x"
> let y = var3d (16,16,16) "y"
> ft (x +: y)
(FT((x(16,16,16)+:y(16,16,16))))
```

This DSL has classes for real and complex vector spaces, and instances of these corresponding to regular hexahedral (cubic) discretizations of one-, two-, three- and four-dimensional spaces. For example, the x component of tissue velocity is a field in $\mathbb{R}^3$, which we approximate using a discretization `var3d` with specified resolution. Another example is the problem-specific conservation-of-mass regularizer is

```
massConservation (ThreeD vx, ThreeD vy, ThreeD vz)
                    = norm2 ( conv3Zip1ZM com vx vy vz )
  where
    com (vX,vY,vZ) = (vX[1,0,0] - vX[-1,0,0])
                   + (vY[0,1,0] - vY[0,-1,0])
                   + (vZ[0,0,1] - vZ[0,0,-1])
```

where `conv3ZipZM` constructs a stencil computation with assumed zero margins for out-of-bounds references, `norm2` is the norm squared, and we use familiar array subscript notation for relative indexing of neighbouring values for the three velocity components.

The result is an environment for constructing term graphs also called directed acyclic graphs (DAGs), which looks to the mathematician a lot like expressions appearing in papers and technical reports.

2.0.2 Term Graph Transformations

Once the model is set, it is up to a second set of experts to transform the declarative expressions, encoded via expression hashes, into a parallel program. Simplification, factoring, differentiation and pre-parallelization make

up 4500 lines of literate Haskell, kept manageable (and readable) by the development of a second DSL for term graph rewriting, with two components: pattern matching and construction. We do not have room to adequately describe this language, but we can convey its flavour with two examples:

```
| Just (x) <- (M.invFt `o` M.ft) exprs node    = x exprs
| Just (x,_) <- (M.xRe `o` M.reIm) exprs node  = x exprs
```

which perform rewrites $\mathrm{ft} \circ \mathrm{ft}^{-1}(x) \mapsto x$ and $\Re(x + iy) \mapsto x$. The *matching* module is imported qualified (*i.e.,* `M.*`) with each function matching an operation and returning constructor(s) for source nodes (*i.e.,* `x`). Pattern matchers can be chained together using composition syntax (*i.e.,* `` `o` ``), and alternative matches are chained together using Haskell pattern guards (*i.e.,* `|`). Composition is designed to have syntax very close to that of the modelling DSL, although the semantics are subtly different. Currently, complicated and conditional term rewriting must access the underlying Haskell data structures, but we are working on improving the expressivity of the matching calculus.

3 Differentiation in CEL

While first-class symbolic vectors have many advantages, it is not possible to build up gradients (and higher-order derivatives) from partial derivatives, as taught in vector calculus. Instead, we must use the *exterior derivative* or a functional equivalent involving implicit differentiation, together with algebraic simplifications to put the resulting expressions into normal forms, from which gradients can be extracted. Since we are targeting large problems for which exact second derivatives would be computationally inefficient, we have only implemented first-order differentiation, but higher-order derivatives could be similarly computed.

To preserve common subexpressions for later code generation, differentiation is implemented by adding subexpressions to the hash table of subexpressions in the original function. Although existing symbolic calculators implement exterior derivatives, *e.g.,* Maple's `Forms` package, we are not aware of an implementation capable of calculating derivatives with respect to vector variable, and of collecting subexpressions common to both the function and gradient computations and preserving the multi-dimensional discretization structure, to enable more efficient code generation than would be expected using Automatic Differentiation.

Consider the scalar function $\left\|\mathrm{ft}(x + iy)\right\|_2$ given by the ℓ_2-norm of the Fourier Transform of a complex vector. We can calculate the exterior deriva-

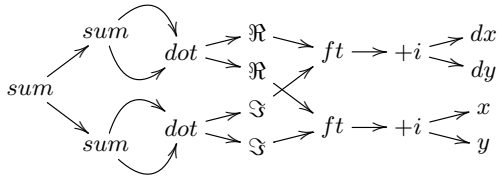
tive with respect to the real variable X using a simple recursion through the DAG as

```
>> diff (mp ["X"]) (norm2 (ft (x +: y)))
((((Re(FT((d(X[16][16][16])+:0.0[16,16,16])))).(Re(FT((X[16][16][16]+:Y[16][16][16])))))
+ ((Re(FT((d(X[16][16][16])+:0.0[16,16,16])))).(Re(FT((X[16][16][16]+:Y[16][16][16]))))))
+ (((Im(FT((d(X[16][16][16])+:0.0[16,16,16])))).(Im(FT((X[16][16][16]+:Y[16][16][16])))))
+ ((Im(FT((d(X[16][16][16])+:0.0[16,16,16])))).(Im(FT((X[16][16][16]+:Y[16][16][16]))))))
```

but this flat representation is deceptively long as it hides the redundancy tracked by CEL internally. Differentiating with respect to both real variables

$$\mathrm{d}\left\|\mathrm{ft}(x+iy)\right\|_2.$$

would double the above length, but not with sharing:



(5)

which maintains the structure of the efficient computation graph, but does not explicitly contain the gradient.

To extract the gradient, we need to put the exterior derivative into a normal form, which we derive by considering the derivative of the Taylor series of a function $f(X,Y)$ at (X_0,Y_0):

$$f(X,Y) = f(X_0,Y_0)+(X-X_0)\cdot\nabla_X f+(Y-Y_0)\cdot\nabla_Y f+(\text{higher order}), \quad (6)$$

whose derivative is

$$\mathrm{d}f(X,Y) = 0+\mathrm{d}X\cdot\nabla_X f+\mathrm{d}Y\cdot\nabla_Y f+(\text{higher order}). \quad (7)$$

If the derivative exists and we can transform the expression for the exterior derivative into a sum of dot products—with the left-hand arguments of the dot products being independent exterior differentials, plus higher-order terms, then the right-hand arguments of the dot products are the desired gradients.

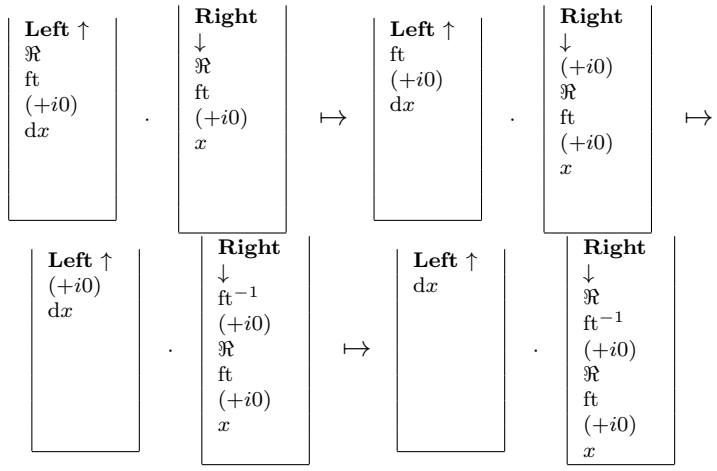
To produce this normal form, we apply simplification rules which

- transform a dot product containing a sum into a sum of dots
- swap arguments of a dot product with a differential in the right argument

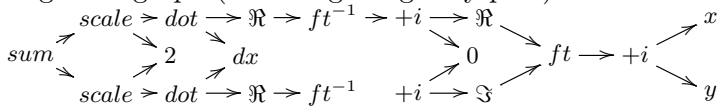
- transform a dot product whose left argument is a linear function applied to a term containing a differential, by popping the linear function from the left argument and applying its adjoint to the right argument

and apply additional transformations which are tied to the specific representation of special operations, and are beyond the scope of this paper. The key point is that we can view the arguments of the dot product as two stacks and we keep popping linear operations off one and pushing their adjoints onto the other. Because some operations are defined on complex values while the dot product is only defined for real values, some linear operations do not have simple adjoints and must be replaced by sums of real and imaginary parts, etc. In the following example, we look at the pop/push operations after the other simplifications have finished, and use the operator $(+i0)$ to mean the embedding of a real value into a complex value by adding a zero imaginary part. This is a simplification of the actual procedure, which interleaves the application of different rules, and also recombines the real and imaginary parts after the pop/push operations to avoid multiple applications of the Fourier Transform.

For the real/x part of the term graph (5), the pop/push proceeds as



The resulting term graph (including imaginary part) is



(8)

Being a simple example, the full set of transformations would also identify the adjoints ℜ and $(+i0)$, as well as ft and ft^{-1} as inverses and collapse the right-hand side to the single term x. This does not happen for interesting models, *e.g.*, including under sampling.

4 Runtime & Verification

So far, we have looked at high-level specifications and model-level transformations. Back-end transformations, including parallelization, cannot be understood without understanding the target. We currently support Single Instruction Multiple Data (SIMD), distributed (cores have private memory) and multi-core (cores access shared memory) parallelism, and are working towards supporting heterogeneous systems. We ensure the correctness of SIMD parallelism through testing, because it has proven adequate, while distributed and multi-core parallelism introduce hazards ("heisenbugs") which are very difficult to discover through testing. To support compile-time verification, these levels of parallelism are factored through a virtual machine which executes Atomic Verifiable Operations (AVOps)—indivisible components of pure computation or communication, including synchronization via signalling. Problem-specific pure computations are extracted from the model by the compiler. A single sequence of AVOps of both types defines the comptuation, although computation may occur out of order, as in an out-of-order processor. Unlike most CPU instructions, the AVOp runtime does not manage out-of-order execution and buffer renaming to preserve serial semantics. Instead, characteristics of the run-time are used by a linear-time verifier run at compile time.

In previous implementations [3], signalling was the central research focus, with the assumption that, as on the Cell/B.E., signalling has hardware support, but many multi-core systems lack hardware signals (and private memory). As such, we implemented an alternative run-time system which does not support signals, but in which synchronization is guaranteed by finite-sized ring buffers and the insertion of no-ops to guarantee the completion of previously dispatched computations prior to the commencement of the preceding computations. No-ops are used where other parallelization frameworks would require a barrier.

4.0.3 Multi-Core Runtime

The basic idea behind the runtime system is to have the main thread, known as the dispatcher, dedicated entirely to offloading work to a set of worker threads. Each of the worker threads operates independently and continuously to execute the AVOps placed by the distributor in the worker-specific ring buffer, as in Figure 1. The AVOps can be thought of as instructions on a virtual machine, with pointers corresponding to registers. As the worker thread finishes operations, it marks the corresponding AVOp

as completed. Each worker executes as a pthread, using the non-portable function `pthread_setaffinity_np` to tie each worker thread to a core to keep threads close to their data in cache.

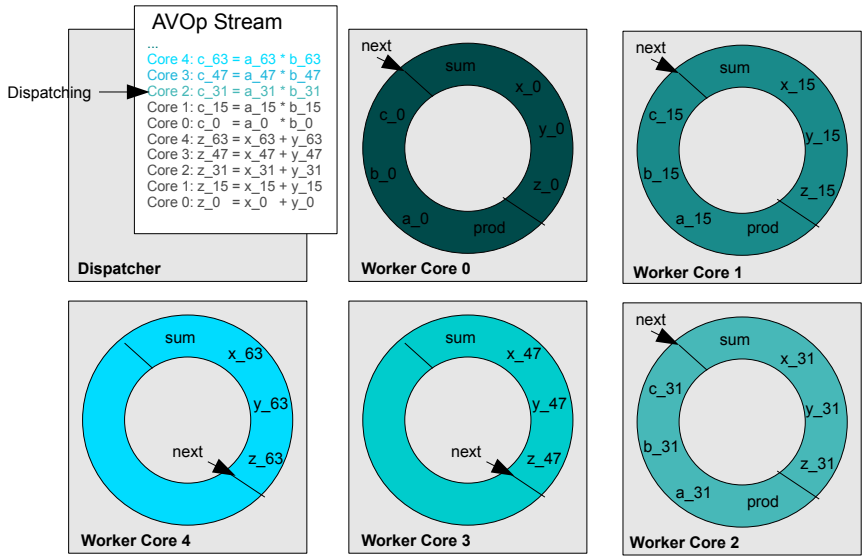


Figure 1: AVOps distributed onto worker core ring buffers

At `dispatch`, the dispatcher thread receives a stream of AVOps to allocate. Each AVOp has been marked with a designated core. Before providing each core with the next AVOp, the dispatcher first checks that there is sufficient room on the corresponding worker thread's ring buffer. If there is insufficient room, the dispatcher waits. The ring buffer implementations are lock-free, with only one thread having write permission to a given buffer element at a time. Because writes and reads can be reordered, vendor-provided memory fence instructions (`_mm_lfence` and `_mm_sfence`) are inserted using processor intrinsics in C to insure that both the dispatcher and worker have a consistent view of the ring buffer.

There is no communication between asynchronous worker threads, and parallelization hazards are avoided by spacing potentially conflicting AVOps in the input stream so that the first AVOp must have been completed and retired from it's ring buffer before the second AVOp is dispatched.

4.0.4 Verification

While a majority of the computation for the target applications is, by nature, hazard free, there are still potential hazards. For example, several easy-to-parallelize linear operations on large vectors are followed by a dot product requiring a sum across cores. By design, AVOps do not contain standard locking mechanisms (such as monitors, semaphores and locks) to guard memory and avoid corruption. We, therefore, implement a verification system (in C) to check that a stream of AVOps is safe, based on three rules:

1. Reads and writes on the same core are safe.
2. Reads on different cores are safe.
3. No other core can read or write to a core that is writing.

These rules are enforced by running the stream of AVOps to be verified through ring buffers of the predetermined size for the specified number of cores. Before each dispatch, it checks that all inputs and outputs for the new AVOp and all AVOps currently in ring buffers obey the three rules. If it is safe to add that AVOp, it is added to the appropriate ring buffer (overwrites one or more old AVOps as it loops around the ring buffer). If the AVOp is not safe, the verifier stops and reports the hazard.

4.0.5 Performance

Although our synchronization is lock-free, and designed for the type of weak memory model multi-core CPUs use to achieve high performance, the spins used to achieve this utilize clock cycles without progress, which will somewhat reduce efficiency. Based on the total CPU time reported by unix `time`, we found overhead under 5% when scaling up to 10 cores, [9].

5 Parallelization and Supporting Transformations

Now we turn the term graph derived from a mathematical model into a sequence of AVOps to execute with our run-time system. As part of this process we also define new pure-computation AVOps.

Efficient parallel performance is about putting data close to computation, whether on the same node, same core or, in the case of SIMD, same slot. Some tasks are embarrassingly parallelizable, while others are serial and most are somewhere in between. The main advantage to having a term

graph with high-level nodes is that we can analyze both the global and the local structure of the computation. Some nodes can perform operations in place, which is an advantage if the global computation can be structured so that all other consumers of the input have finished execution or have their own copies of the data.

Different nodes put other constraints on memory allocation: A single FT node requires that all input data be contained in a single memory space, whereas a vector addition requires that corresponding elements of both vectors be in the same memory space, but otherwise memory can be arbitrarily distributed across the system. As a prelude to parallelization, we make transformations which reduce the constraints on memory distribution and transform constraints to make adjacent constraints compatible, such as breaking a 2D FT node into a composition of two 1D FTs which only require that each row or column be contained in the same memory.

Specific hardware favours some implementations over others. On cached architectures, chunks of data fitting in cache lines will be processed faster, so computation should be subdivided along cache lines, and the complete cache line should be processed together, and as many operations performed before storing back to main memory. In-register SIMD favours operations which combine elements stored at address equivalent modulo the SIMD-size. This means that, in the case of the Fourier transform, for instance, multiple row FTs of interleaved rows can be much faster than row FTs of non-interleaved data.

Knowing these effects on parallel performance, we annotate the computation graph with parallelization instructions one node at a time, starting with the nodes with the strictest constraints on memory distribution and extending to up- and down-stream nodes which have strictly weaker memory constraints and which can be performed in-place. Some operations cannot be performed in-place, or have constraints which are incompatible with their neighbours, and this is ok. So, for example, our column FT nodes are marked as in-place operations with computation subdivided into a multiple of the SIMD-size columns.

Note that for our target domains, this greedy process works because the Fourier Transform is both the most constrained and the most computationally intensive. Some loss of efficiency in other computations will be overcome by improvements in the efficiency of the FTs.

In the next phase, we create a memory map, which, for non-streaming applications, works as follows: Memory is assigned to inputs and outputs of the global computation. In our case, the inputs are model variables and parameters, and the outputs are function values and gradients. Next,

the remaining nodes are topologically sorted and unassigned outputs are assigned to memory. A node which was identified as in-place in the first step has the memory assignment copied from the overwritten output. A node which is not in-place but is executed interleaved with its neighbours has memory allocated for a number of chunks of data equal to workers×pipeline-depth. The pipeline-depth is the number of operations which can fit in the ring buffers for each worker and potentially interfere with each other. This is the most common mechanism for ensuring parallel safety, and it involves no loss in throughput. Finally, nodes whose consumers use a different memory division (*i.e.,* row and column FTs) have their total memory requirement allocated, because generally no downstream computations can start until this node fully completes.

We can infer that memory requirements and throughput strongly depend on the exact presentation of the computation graph. We mentioned that row-FTs are less efficient than interleaved row-FTs (or column-FTs), and this is the basis of one of our pre-parallelization transformations: Given a 2d FT followed by a projection, π, onto a subspace which is separable (*i.e.,* is the composition of a projection onto rows and another onto columns)

$$\pi \longrightarrow \mathrm{ft} \longrightarrow +: \begin{array}{l} \nearrow X \\ \searrow Y \end{array} \tag{9}$$

we can separate both transformations and interleave them, thereby not computing FTs of rows which would be projected out:

$$\pi_{\mathrm{row}} \longrightarrow \mathrm{ft}_{\mathrm{row}} \longrightarrow \pi_{\mathrm{col}} \longrightarrow \mathrm{ft}_{\mathrm{col}} \longrightarrow +: \begin{array}{l} \nearrow X \\ \searrow Y \end{array} \tag{10}$$

and the remaining computations will go even faster by switching to column-FTs (and transposes):

$$\Big(\mathrm{tr} \longrightarrow \pi_{\mathrm{col}}\Big) \longrightarrow \mathrm{ft}_{\mathrm{col}} \longrightarrow \Big(\mathrm{tr} \longrightarrow \pi_{\mathrm{col}}\Big) \longrightarrow \mathrm{ft}_{\mathrm{col}} \longrightarrow +: \begin{array}{l} \nearrow X \\ \searrow Y \end{array} \tag{11}$$

where we have also used the fact that $\mathrm{tr} \circ \pi_{\mathrm{col}} = \pi_{\mathrm{row}} \circ \mathrm{tr}$, and parentheses mark operations fused into a single AVOp.

6 Conclusion and Work in Progress

We have presented some highlights of recent work on CEL, giving the flavour of the modelling DSL for mathematicians and physicists and the term graph

rewriting DSL used to specify rule-based optimizations. To handle billion-variable problems, we needed to treat vectors, linear transformations and generalized stencil computations as first-class objects and not compositions of scalar values and operations, and this necessitated a novel method for calculating gradients.

This is more than a proof-of-concept: It has been used to produce a high-performance MRI reconstruction system for Alltech Medical Systems America; and we have met the enumerated design goals which close section §1. But a lot of work remains to encode, as term graph rewriting rules, all the techniques used by experts in high-performance signal processing and image reconstruction, and we see a lot of scope for using strong typing to further enhance software quality.

We thank NSERC, IBM Canada, MITACS and OCA for research assistance.

References

[1] Anand, C.K., Kahl, W.: An optimized Cell BE special function library generated by Coconut. IEEE Transactions on Computers (2009)

[2] Corpetti, T., Mémin, E., Pérez, P.: Dense Estimation of Flud Flows. IEEE Trans. on Pattern Analysis and Machine Intelligence 24(3) (March 2002)

[3] Dobrogost, M., Anand, C., Kahl, W.: Verified Multicore Parallelism using Atomic Verifiable Operations, pp. 3–45. CRC Press (2013)

[4] Ghosh, D.: Dsl for the uninitiated. Commun. ACM 54(7), 44–50 (Jul 2011), `http://doi.acm.org/10.1145/1965724.1965740`

[5] Héas, P., Mémin, E., Heitz, D.: Self-similar regularization of optic-flow for turbulent motion estimation. The 1st International Workshop on Machine Learning for Vision-based Motion Analysis, 1–12 (2008)

[6] Holland, D.J., Malioutov, D.M., Blake, A., Sederman, A.J., Gladden, L.F.: Reducing data acquisition times in phase-encoded velocity imaging using compressed sensing. J. Magn. Reson. 203, 236–246 (Apr 2010)

[7] Issa, B., Moore, R.J., *et al*: Quantification of blood velocity and flow rates in the uterine vessels using EPI at 0.5T. J Magn Reson Imaging 31, 921–927 (Apr 2010)

[8] Marshall, I.: Computational simulations and experimental studies of 3D phase-contrast imaging of fluid flow in carotid bifurcation geometries. J Magn Reson Imaging 31, 928–934 (Apr 2010)

[9] Pavlin, J.L.M.: Symbolic Generation of Parallel Solvers for Unconstrained Optimization. MSc Thesis, McMaster University (2012)

[10] Tikhonov, A.N., Arsenin, V.Y.: Solutions of Ill-Posed Problems. Winston (1977)