A Domain-Specific Language for the Generation of Optimized SIMD-Parallel Assembly Code

Christopher Kumar Anand* Wolfram Kahl*

anandc@mcmaster.ca kahl@cas.mcmaster.ca

Software Quality Research Laboratory
Department of Computing and Software, McMaster University
Hamilton, Ontario, Canada L8S 4K1

May 2007

Abstract

We present a domain-specific language (DSL) embedded into Haskell that allows mathematicians to formulate novel high-performance SIMD-parallel algorithms for the evaluation of special functions.

Developing such functions involves explorations both of mathematical properties of the functions which lead to effective (rational) polynomial approximations, and of specific properties of the binary representation of floating point numbers.

Our framework includes support for estimating the effectiveness of different approximation schemes in Maple. Once a scheme is chosen, the Maple-generated component is integrated into the code generation setup. Numerical experimentation can then be performed interactively, with support functions for running standard tests and tabulating results. Once a satisfactory formulation is achieved, a code graph representation of the algorithm can be passed to other components which produce C function bodies, or to a state-of-the-art scheduler which produces optimal or near-optimal schedules, currently targeting the "Cell Broadband Engine" processor.

Encapsulating a considerable amount of knowledge about specific "tricks" in DSL constructs allows us to produce algorithm specifications that are precise, readable, and compile to optimal-quality assembly code, while formulations of the equivalent algorithms in C would be almost impossible to understand and maintain.

Keywords: Domain-specific languages, code generation for SIMD-parallelism, high-performance floating-point function evaluation, special functions.

Software Quality Research Laboratory

McMaster University

SQRL Report No. 43

*The authors thank CFI, OIT, NSERC and IBM Canada for financial support.

Contents

1 Introduction

In this paper we describe the interface to a development environment for high performance scientific kernels with rich support for SIMD parallelism. The development environment called Coconut (COde CONstructing User Tool) is a platform for experimenting with novel ideas in reliable and high performance code generation.

The aim of the present work is to provide an interface for experts in high performance numerical software to provide processor-specific implementations of mathematical functions, with the following goals:

1. The language supports safety, and is familiar to mathematicians.
2. Common code construction tasks are simplified.
3. The user is insulated from details of the target intermediate language not relevant to their task.
4. The environment supports exploration and rapid prototyping.
5. Alternatives for instruction selection can be provided by the user.

As a practical goal, we also wanted new students working on Coconut to be able to quickly learn how to leverage SIMD parallelism for whatever instruction set architecture (ISA) we choose to target.

We have met these goals, to the point that undergraduate mathematics students were able to contribute meaningfully to a production-quality library of twenty-six special functions, which outperform conventional implementations by a wide margin as a direct result of patterns encapsulated in our domain-specific language.

1.1 Embedding into Haskell

The main vehicle of interaction with the Coconut development environment is a domain-specific language (DSL) for program development with full control over assembly code generation. This DSL provides controlled access to both syntactic machine instruction constructs, and to their semantics. The sets of machine instructions are complemented with a rich, extensible vocabulary at a higher level of abstraction.

For domain experts to be productive, the DSL should "feel natural", *i.e.*, conform to informal function/variable semantics. This entails constraints both on syntax and on semantics. It also implies a strong requirement for abstraction capabilities, that is most easily met by embedding our DSL into the purely functional general-purpose programming language Haskell [Peyton Jones+ 2003]. As extensively discussed in the literature [Hudak 1996; Hudak 1998], Haskell is a particularly good host language for embedded DSLs because of its semantic purity and rich abstraction capabilities. Its strong "mathematical flavour" makes it easy for mathematicians to learn.

At the core of the structured vocabularies of our DSL are the pure data flow machine instructions to which we add instructions which involve, on the one hand, more complicated state and control flow, and on the other hand, higher-level patterns. By implementing the semantics of the core assembly language, we allow the developer to perform rapid exploration and development inside a Haskell interpreter. This also makes it easy to turn exploratory tests into compile-time assertions executed in the interpretive component of our system. The intermediate language we use for code generation is based in *code graphs*, a kind of hypergraphs that at the data-flow level include the possibility of expressing explicit alternatives [Kahl, Anand+ 2006]. Alternatives can be used for instruction selection by the back end.

By encapsulating common patterns into the DSL, the resulting implementations are short, which improves readability, while taking advantage of optimization within the patterns to achieve peak performance.

We designed our DSL to preserve the declarative flavour of programming in Haskell, and we exclusively use the literate programming [Knuth 1984; Knuth 1992] style with LaTeX embedding that is supported by the Haskell standard; this also improves readability and hence quality.

1.2 Related Work

There is a lot of work using generative techniques to capture high-level patterns in high-performance computing, whether they are called generative programming, template programming or simply use generative ideas prior to their widespread recognition. Our DSL could be used in conjunction with such systems as a specification language for low-level constructs which are targets for transformations in the higher-level generative system. SAC is a good example of such a high-level system [Grelck, Scholz 2006].

CorePy is another tool/research project which makes access to hardware instructions a central feature [Mueller, Lumsdaine 2006a; Mueller, Lumsdaine 2006b]. CorePy also targets VMX/Altivec and the SPU ISA. They embed their environment in Python, which gives them some of the benefits we derive from Haskell (sparse syntax, interpreted environment).

Research on the applicability of SIMD parallelism is quite advanced, as evidenced by support in commercial compilers. In the direction of elementary function evaluation, [Bandera, Gonzalez$^+$ 2004] use ideas from [Merrheim, Muller$^+$ 1993] to evaluate single polynomials both with existing SIMD operations, and using two proposed instructions. Interestingly, one of those instructions (variable bit shift) is now available in the SPU ISA, and we do make extensive use of it, although not in the way proposed, because we are interested in evaluating elementary functions multiple times in parallel, whereas they used parallelism to accelerate the evaluation of a single polynomial.

Using permute instructions to do lookups has been discovered a few times, for example, [Dubey, Kaplan$^+$ 2001; Shi, Lee 2000; Sazegari 2002], but there is no record of its use in piecewise polynomial evaluation.

1.3 Overview

We start with a motivating example: literate code for hyperbolic tangent, in which we explain the issues in developing efficient SIMD implementations of special functions, the interactive nature of the development using both Haskell support functions and Maple. In Sect. 2 we present the most important details of SIMD parallelism in general, and the specific features which play a role in our work. We follow this with a description of the syntax of the DSL and general features in Sect. 4, and in Sect. 5 we present some of the patterns we abstracted from the development. Then in Sect. 6 we apply these patterns to another example, cube root, which incorporates a known dangerous optimization, for whose validity we include a compile-time assertion. The next section is a report on related applications of the DSL. Finally, we relate our positive experience with both expert and novice users, and in Sect. 8 we explain some of the advantages of this generative approach and things we hope to do in the future which would be impossible without it.

2 Overview of SIMD and the SPU ISA

Single Instruction Multiple Data (SIMD) instructions operate on more than one data element in parallel. The first common SIMD architectures were vector processors performing operations common in linear algebra on arrays of floating point numbers. Not surprisingly, they were good at large dense linear algebra, but difficult to use for other applications. The second generation of SIMD architectures was aimed at so-called multi-media operations, operations on pixel arrays, and three-dimensional vectors in computer graphics. These *instruction set architectures* (ISAs) are more diverse, with VMX/Altivec [IBM 2005] on Power and SSE [Ramanathan 2006] on x87 being the best-known.

2.1 Data Flow

The SPU ISA [IBM 2006] uses 128-bit operands, in common with VMX and SSE. It contains a rich set of operations formed by dividing the 128-bit operands into 8-, 16-, 32- or 64-bit quantities and performing the usual scalar operations independently on each. See Fig. 1 for an example 32-bit add instruction operating on four elements. This results in a useful level of parallelism, but introduces

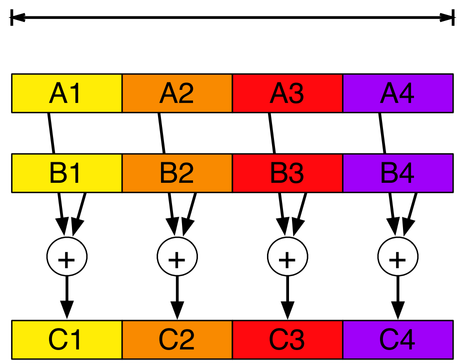

Figure 1: fma, a 32-bit add operating on two 128-bit wide operands.

alignment issues in data. To be able to handle arbitrarily-structured data (necessary if you want to be able to insert SIMD-optimized functions into existing applications) all SIMD instructions have some instructions to rearrange data. Two approaches are possible, a large set of instructions with specific functions (e.g. unpacking pixel data into vectors by component), or a small set of software-controllable instructions. All ISAs follow a middle path, with VMX/SPU ISA being more generic. The instructions of most use in synthesizing loop overhead are the byte permute instruction **shufb** (analogous to VMX's *vperm*), shown in Fig. 2, and quadword rotate instructions, including **rotqby** shown in Fig. 3. As shown in the figures, SPU byte permutation can be used to move 32-bit components from one slot to another one (useful for transposing single-precision floating point matrices, for example), or duplicate bytes. It can rotate bytes through cycles, which can be used to count through loop induction variables

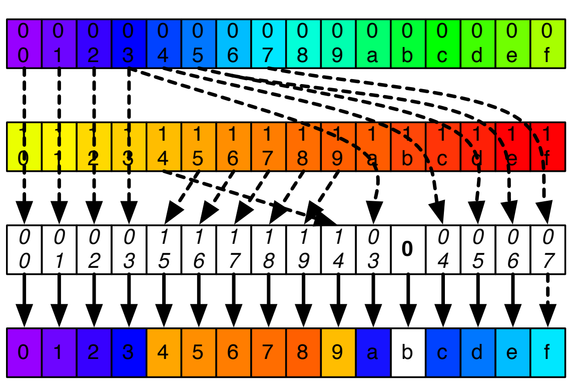

Figure 2: shufb byte permutation taking two source operands (coloured) and an operand of byte indices.

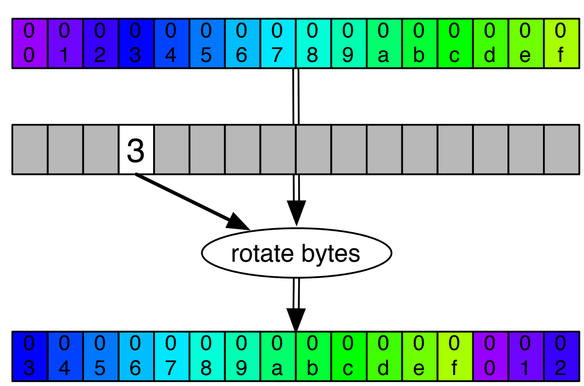

Figure 3: rotqby, a quadword byte rotate, using a run-time value from a second operand.

when the loop sizes are known at compile time. Unlike the *vperm* instruction of VMX, **shufb** can also insert three special byte values (zero, minus one, and high-bit set) if it encounters special byte values in the index quadword, see byte `b` in the figure. Byte and bit rotate instructions (like **rotqby** in Fig. 3) take both immediate counts and counts from operand registers.

2.2 Storage Model

There are different ways designers have adapted to increasing memory access latencies. Cache memory is used in all desktop and server computing. Cache memory may include software-directed or hardware-predictive streaming, in which regularly strided patterns of memory access are used to fetch data from main memory into cache before it is requested. Such prefetching is a type of stream processing, which is also central to the way in which graphics processors efficiently access memory.

Distributed computation, on the other hand, involves separate memory spaces with explicit data transfer operations to access non-local memory. In the case of GPUs and accelerator cards, this could be accomplished with direct memory access (DMA) engines, running asynchronously with computation, but with latency similar to cached memory access.

As we move to higher levels of parallelism on a chip, which Cell is pioneering, the overhead associated with a shared memory space (page tables, cache coherency, ...) will increase superlinearly. To avoid this, the Cell processor's SPU compute engines have their own local stores (LSs), and use DMA to transfer data to and from system memory. This has the positive effect that access to local memory can be as predictable as accesses on a normal processor which hit in L1 cache. This introduces a difference between local and non-local memory access which is reflected in the structuring of our DSL.

2.3 Control Flow

Instructions for computation and local data movement, which exist in various permutations in all SIMD ISAs, effect register values, and hence data flow. The SPU ISA also contains interesting control-flow instructions which effect state, including signaling, messaging and atomic read/write instructions, and branch hinting instructions required for efficient loop implementations.

We group these instructions separately in the DSL in a way that prevents them to be accidentally mixed into pure data-flow calculations, because for most users they will only be used via the higher-level patterns which contain them.

2.4 Instruction Summary

To make the paper self-contained, we include in Table 1 a description of all the machine instructions we use in the paper. Machine instructions are set in **bold**. They operate on 128-bit vectors, and all of the instructions in the examples can be interpreted to be operating on four 32-bit values in parallel. Functions which translate between Haskell values and DSL register values are set in *italics*, and words from the DSL vocabulary are <u>underlined</u>.

3 Motivating Example: Hyperbolic Tangent

Before going into the details of the DSL, we present an example of how our system insulates the domain expert from many of the more difficult aspects of writing assembly code, and provides necessary input to exploration of different approximation strategies in Maple.

We develop the code for hyperbolic tangent as an example of the workflow we are facilitating with our DSL. This includes (1) the definition of the intervals used in the piecewise polynomial, (2) the Maple code which calculates the piecewise polynomials, and (3) the declarative assembly code, which

Instruction	*Description*
selb	bits in third argument select corresponding bits in first or second argument
shufb	bytes in third argument index bytes to collect from first two arguments
fma, **fs**	32-bit floating point fused multiply-add, subtract
andc	and first argument with complement of the second argument
mpya, **mpy**	16-bit integer multiply fused with 32-bit add, just multiply
mpyui, **mpyi**	integer multiply with immediate arguments
roti	rotate each 32-bit word by an immediate constant number of bits
rotqbii	rotate whole quadword up to 7 bits left, number given by immediate
shli	shift each 32-bit word left by an immediate constant number of bits
ceqi	set each output word to 1 if input word matches immediate constant, to 0 otherwise

Table 1: Summary of SPU instructions used in this paper.

for ftanh is eight lines of Haskell, which generate 48 machine instructions and 34 128-bit constants, which is about half as many machine instructions as required by a standard approach.

3.1 Polynomial Approximation

Hyperbolic tangent, see figure 4, is defined by

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \tag{1}$$

but using this definition for computation would be difficult because we would run into problems with subtraction of similar numbers, and division of similar large and small numbers, all of which introduce additional error. Fortunately, hyperbolic tangent rises very quickly to 1.[1]

```
tanhSaturate :: Double
tanhSaturate = roundToRepresentable (atanh (1 - 2 ** (-25)))   -- = 8.664339742098155
```

Therefore, any number larger than *tanhSaturate* in magnitude should round to ± 1.

The standard Remez Exchange algorithm [Golub, Smith 1971] can find best approximating polynomials to a differentiable function on an interval, but in this case a single approximating polynomial will either have very high degree or be very inaccurate. A standard approach is to look for a piecewise polynomial approximation, and we have standard tools to help in the search. Unfortunately, the search space is large, and some domain knowledge is required to narrow the search. We use a Maple library function to perform the search for a given polynomial. The Maple procedure can fail for several reasons, including ill conditioning within the algorithm. Even if the search succeeds, the reported quality (maximum error) of the solution may not meet the requirements for a particular function. Increasing precision requires either higher-order polynomial approximations, or a different strategy for assigning intervals. This has to be decided by the domain expert on a case-by-case basis, after a lot of experimentation.

Another problem is that the Maple function will report the error assuming infinite precision arithmetic, so allowances have to be made for the actual precision. An expert will recognize polynomials which are

[1] $1 - 2^{-25}$ is the smallest number which when rounded to a representable number is 1.

likely to introduce unacceptable rounding errors (e.g. the coefficients grow quickly relative to the size of the domain of the polynomial). Finally, the reported error is an absolute error. Since floating point numbers, by definition, have a level of precision inversely proportional to the number represented, this is usually not an acceptable measure. So weighting of the approximation process may need to be added by hand. Even this, however, will not lead to satisfactory answers where the function being approximated crosses zero, or has a singularity or asymptote. In such cases, the domain expert will use different strategies to identify appropriate piecewise polynomial approximations fitting into interval organizations that allow efficient coding.

Our example has no singularities or vertical asymptotes, but it does have a zero, and this affects the quality of polynomial approximations of a given order. We can get around this by varying the size of the intervals. In general, this will greatly increase the complexity of the interval identification, but we have identified one pattern which is very efficient: logarithmically stepped intervals, as shown by alternating colours in figure 4.

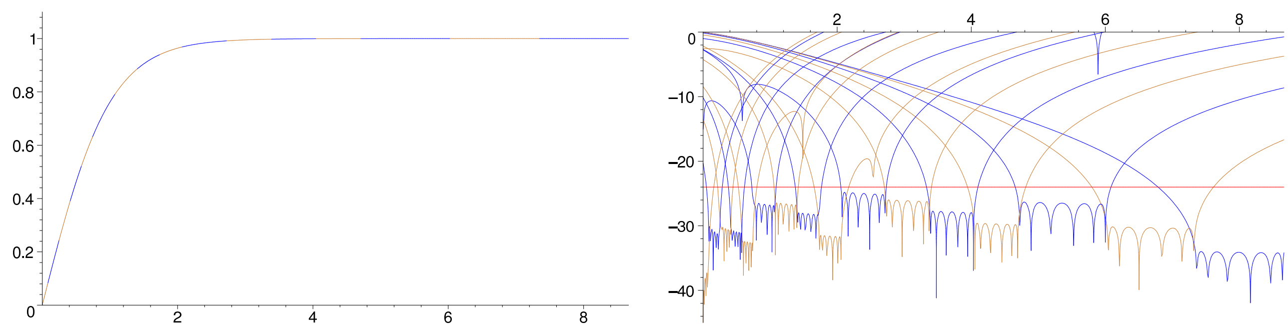

Figure 4: Sixteen approximating polynomial segments for tanh, *left*, and the corresponding error in bits, *right*.

3.2 Calculation of the Intervals

The efficient implementation of this approach relies on the fact that such logarithmically stepped interval patterns can be specified by using combinations of bits from both mantissa and exponent of the floating-point representation as interval identifiers; different such combinations are possible, and to specify such an interval pattern we furthermore need the length of the range to be covered by all intervals (*rangeEnd*), and we found cases where it is beneficial to skip some of the smallest intervals (*skipIntervals*) and add an equal number of large intervals at the end. For tanh, we choose the following interval pattern specification:

```
tanhLookup = RegLookupSpec
  { mantissaBits = 2
  , exponentBits = 2
  , skipIntervals = 3
  , rangeEnd      = tanhSaturate
  }
```

With this definition, evaluating

```
lookupBreaks tanhLookup
```

calculates the intervals in a form that can easily be transferred to Maple.

In Maple, we used the following code to find approximations, with the interval break points stored in the array `breaks`:

```
i:=0;                                            # line 1

ax:=numapprox[minimax]                           #      2
      (x->limit((tanh(y)/y-1)/y, y=x)            #      3
      ,(breaks[i+1]) .. (breaks[i+2])            #      4
      ,[polyOrd-2,0]                             #      5
      ,x->x                                      #      6
      ,'da[i]');                                 #      7

aa[0]:=x->x*(1+x*ax(x));                         #      8

for i from 1 to 15 do                            #      9
  aa[i]:=numapprox[minimax]                      #     10
      (x->tanh(x)                                #     11
      ,(breaks[i+1]) .. (breaks[i+2])            #     12
      ,[polyOrd,0]                               #     13
      ,x->x                                      #     14
      ,'da[i]');                                 #     15
od;
```

The first polynomial is calculated in lines 2–7 separately from the rest (lines 9–15) because it contains a zero crossing. The arguments to the approximation generation function are the function to be approximated (line 3), the domain interval of the approximation (line 4), the order of the rational polynomial (numerator,denominator) (line 5), the weighting (identity) (line 6) and the variable in which to store the estimate for the maximum error (line 7).

Building on our accumulating experience with variations on these Maple patterns we are considering automatic generation of such Maple code blocks, but up to now have not yet seen a real need for this.

3.3 Using an In-Register Lookup Table

Implementing tanh with the above interval pattern, which uses four-bit interval identifiers, requires lookup in 16-element arrays; with the large register file of the SPU, we can easily afford to store these $6 * 16$ floating-point numbers in 24 registers.

This 16-way register lookup can be performed for two keys at a time without performance loss, so we use a two-way parallel "unrolling" of the tanh function. The lookup implementation relies on the specification *tanhLookup* shown above, and uses the coefficient array *tanhC* produced using Maple.

ftanh = use16X2lookup *tanhLookup* *tanhC* *tanhKeyResult*

The higher-order function use16X2lookup, see Sect. 5.5.2, implements this pattern, generates vector constants containing byte-scrambled versions of the binary representation of the Maple-generated floats *tanhC* arranged to ease the lookup, constructs the 16-way lookup, and connects it with two instances of *tanhKeyResult*, a function that accepts as arguments an original argument *v* (one of the two parallel arguments) and the coefficients *coeffs* resulting from the lookup, and produces as its results a *key* to be used for the lookup and the final result of the calculation. This kind of "tying the knot" is only possible in non-strict (lazy) programming languages — the *key* of course must not depend on the *coeffs*.

$\mathit{tanhKeyResult}\ \mathit{coeffs}\ v = (\mathit{key}, \mathit{result})$
where

We will use the absolute value to create the lookup *key* and as starting point for the remaining computations, and obtain it by masking out the bit pattern signBit covering exactly the sign bit (of each vector element):

$$\mathit{key} = \mathbf{andc}\ v\ \underline{\mathrm{signBit}}$$

This key is used by use16X2lookup to look-up *coeffs*, and also together with these coefficients to evaluate the resulting polynomials using Horner's rule:

$$\mathit{polyVal} = \underline{\mathrm{hornerV}}\ \mathit{coeffs}\ \mathit{key}$$

We also compare (using the floating-point "greater-than" comparison instruction **fcmgt**) the *key* to $\operatorname{arctanh}(1 - 2^{-24})$, because this is the smallest number which rounds to 1 at 24-bit precision — all higher numbers round to 1. This comparison produces a select mask:

$$\mathit{isBig} = \mathbf{fcmgt}\ \mathit{key}\ (\mathit{unfloats4}\ \mathit{tanhSaturate})$$

This mask can be used by the bit-select instruction **selb**; we apply this to the result of polynomial evaluation, to force evaluations at large numbers to 1.

$$\mathit{resultOrOne} = \mathbf{selb}\ \mathit{polyVal}\ (\mathit{unfloats4}\ 1)\ \mathit{isBig}$$

Finally, we obtain the *result* by restoring the sign bit from the argument *v*:

$$\mathit{result} = \mathbf{selb}\ \mathit{resultOrOne}\ v\ \underline{\mathrm{signBit}}$$

3.4 Using this Definition

As we will explain in Sect. 4, we can now use ftanh to calculate the application of tanh to each floating-point number in two four-tuples; we use lists at the interface, need to make the desired type explicit, and need to pack and unpack between *Float* lists and vectors:

$$(\mathit{floats}\ \text{`}\mathit{prod}\text{`}\ \mathit{floats})\ (\underline{\mathrm{ftanh}}\ (\mathit{unfloats}\ [0.1, 0.2, 0.3, 0.4], \mathit{unfloats}\ [1\,..\,4] :: \mathit{SpuSimValue}\ \mathit{REG}))$$

This capability is extremely convenient for testing the numeric properties of a function definition like that of ftanh; for the direct test above, one would compare the resulting numbers with

$$\mathit{map}\ \mathit{tanh}\ ([0.1, 0.2, 0.3, 0.4] +\!\!+ [1\,..\,4 :: \mathit{Float}])$$

We also have special testing wrappers for such functions which eliminate the explicit interaction with the type system at the interface, and are therefore the preferred means to conduct such testing.

For SPU assembly code generation, at type *SpuMNode REG* instead of *SpuSimValue REG*, or function ftanh generates a code graph in the sense of [Kahl, Anand+ 2006], which is then wrapped into additional loop overhead (see Sect. 7). An automatically-generated rendering of the resulting code graph, is shown in Fig. 5.

This code graph is then split for software pipelining [Thaller 2006] and scheduled to produce a very efficient tanh kernel.

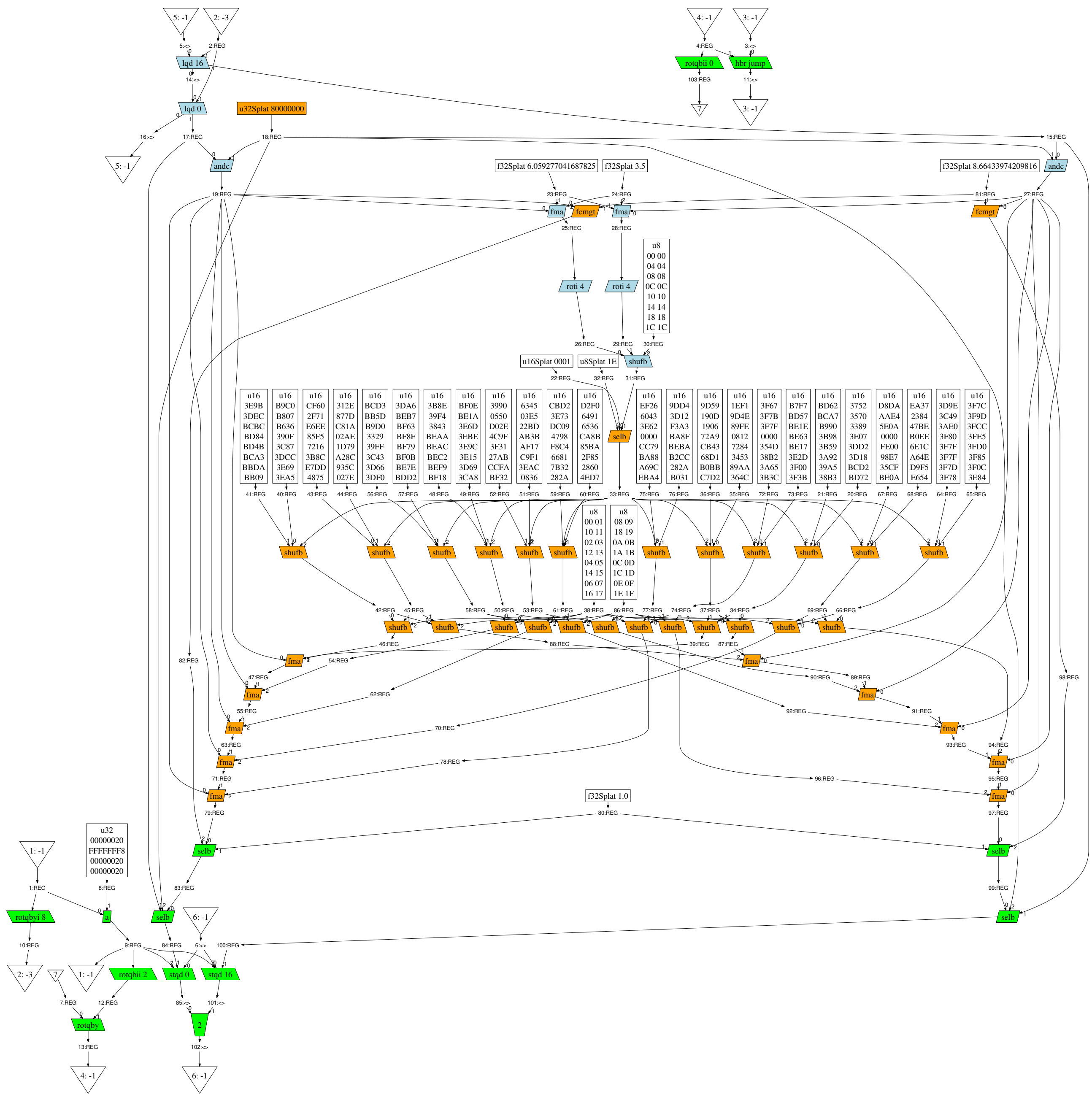

Figure 5: Assembly code graph for tanh

4 Language Organization

The embedded domain-specific language we used above for the creation of a high-performance implementation of tanh consists of several sub-languages, which we summarize now, before we turn to the technically non-trivial aspect of how to provide, in the Haskell embedding, an interface that treats symbolic entities “naturally” as functions. This is a major usability criterion for domain experts in this field, since they typically have a mathematical background.

4.1 Domain-Specific Sub-Languages

Single-instruction register computations are conceptually the simplest and form the core sub-language. For some purposes it makes sense to sub-divide these further into, "true SIMD" instructions like **fma** that perform a single operation multiple times in parallel, and vector-spanning operations like **shufb** that affect the whole width of the vector in a more complex way.

The former are conceptually close to the corresponding mathematical operations, which influences the style in which program authors think about them — this is visible in the implementation of *tanhKeyResult* above, which relied exclusively on such instructions; it is probably even more obvious in the cube root example in Sect. 6 below. This style of "mathematician-friendly" code documentation allows mathematicians to produce code using such instructions with high confidence, even if they are not familiar with all the details of the underlying assembly language.

As we have seen above in the tanh example, "tricky" uses of **shufb** instructions crossing the SIMD borders can be entirely encapsulated in "pattern" combinators like use16X2lookup, which combine several such instructions in a way that provides useful services to pure SIMD programs.

Interaction with memory, *i.e.*, load and store, is worth treating separately, mainly because different patterns of reasoning are necessary to establish the necessary independence and dependency relations between different occurrences of these instructions.

Fig. 6 sketches the relations between these and the other main sub-vocabularies of our DSL; control and synchronization instructions require yet more complicated reasoning tools to be able to justify the convoluted usage patterns necessary for high-performance computation.

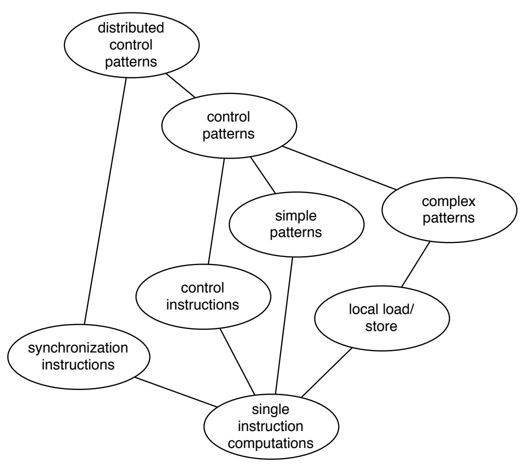

Figure 6: Inclusion relation for language elements.

On top of these basic instruction languages, we offer "pattern" vocabularies, including under "simple patterns" the functions use16X2lookup and lookupBreaks we used in our tanh implementation.

4.2 Pseudo-Overloading of Function Types

For mathematicians, it is perfectly natural to think of instructions like **fm** (component-wise floating-point multiply of four-element vectors) as "perhaps somewhat distorted" cousins of familiar operations like real-number multiplication, but definitely as *functions*.

In the context of embedding these instructions into Haskell, a programming language that mathematicians typically relate to with relative ease, this means that the Haskell incarnation of **fm** is expected to be a proper Haskell *function* again, to be applied to its arguments via *normal function application*.

If such a Haskell incarnation is expected only to embody the *semantics* of **fm**, then there are no problems with this — **fm** would then map an operation quite similar to Haskell *Float* multiplication over quadruples of *Float*s.

However, for *code generation* we need **fm** to embody a *syntactic* representation of the corresponding assembly instruction, and this is not easily cast into the shape of a pure Haskell function.

The problem is that sharing needs to be preserved; consider the following code for calculating eighth powers:

$$\begin{array}{l} \mathit{pow8}\ x = \mathsf{let} \\ \quad \mathit{x2} = \mathbf{fm}\ x\ x \\ \quad \mathit{x4} = \mathbf{fm}\ \mathit{x2}\ \mathit{x2} \\ \quad \mathsf{in}\ \mathbf{fm}\ \mathit{x4}\ \mathit{x4} \end{array}$$

If x is a register reference, then, according to the Haskell semantics of function application, the equality

$$\mathit{pow8}\ x = \mathbf{fm}\ (\mathbf{fm}\ (\mathbf{fm}\ x\ x)\ (\mathbf{fm}\ x\ x))\ (\mathbf{fm}\ (\mathbf{fm}\ x\ x)\ (\mathbf{fm}\ x\ x))$$

has to hold, and in naïve implementations of **fm** as instruction representation generator this would produce seven multiplication instructions, instead of the expected three.

For the seasoned Haskell programmer, the obvious solution would be to change the interface, and make **fm** a monadic action in some code generation monad.[2] Then, one would have to write:

$$\begin{array}{l} \mathit{pow8}\ x = \mathsf{do} \\ \quad \mathit{x2} \leftarrow \mathbf{fm}\ x\ x \\ \quad \mathit{x4} \leftarrow \mathbf{fm}\ \mathit{x2}\ \mathit{x2} \\ \quad \mathbf{fm}\ \mathit{x4}\ \mathit{x4} \end{array}$$

and only three instructions would be generated.

However, with the monadic set-up, straight-forward substitution does not work — instead of writing $\mathbf{fm}\ (\mathbf{fm}\ x\ y)\ z$ with **fm** as a normal function, in the monadic variant one would have to write something like $\mathbf{fm}\ x\ y \ggg\!= (\text{`}\mathbf{fm}\text{`}z)$, or use the do-notation again:

$$\begin{array}{l} \mathsf{do}\ u \leftarrow \mathbf{fm}\ x\ y \\ \quad\ \ \mathbf{fm}\ u\ z \end{array}$$

We found the necessary syntactic overhead for making functions like **fm** monadically typed is a serious impediment to using the language as a "natural medium of expression" for a domain expert in this field.

Therefore, **fm** needs to be a proper function, and will have the type signature

$$\mathbf{fm} :: \mathit{SpuType}\ f \Rightarrow f \rightarrow f \rightarrow f$$

of a binary operation on any type f that can be considered as somehow standing for a vector of SPU *float* values.

This leaves only the type f that **fm** operates on open for adaptation.

[2] See, *e.g.*, `http://haskell.org/haskellwiki/Monad` for links to introductory material about the use of monads to structure computation in Haskell; this was originally inspired by Moggi's use of monads to structure denotational semantics of imperative programming languages [Moggi 1991b; Moggi 1991a] and popularized in the functional programming community by Wadler [Wadler 1990; Wadler 1992].

The option to use **fm** on the semantic level is attractive, useful for rapid prototyping, and occasionally also useful for compile-time decisions, so needs to be made available.

The Haskell type class system provides the necessary overloading to have the "semantic implementation", i.e., interpretation, as one instance, and a syntactic representation as a different instance. The problem is only the desired sharing, and we note that in our context, actually *all possible* sharing is desired, implementing the common compiler technique of common sub-expression elimination.

This allows us to provide as operand type "monadic nodes", *MNode*s, defined as monadic actions that construct code graph nodes in a state monad with a code graph as state, with the twist that new nodes are only constructed when necessary, and re-usable nodes are located using a dictionary also kept in the state. These monadic nodes achieve convenience of expression at the expense of significantly higher cost of construction of the syntactic representation, but we do not foresee this to become a bottleneck soon.

5 Selected Code Patterns

We now present a selection of code patterns incorporated into our DSL vocabulary, several of which were used in Sect. 3 to implement tanh.

5.1 Horner Evaluation

Horner's rule:

$$\sum_{i=0}^{n} a_i x = a_0 + x\,(a_1 + x\,(a_2 + ...xa_n)\,...)$$

can be used to evaluate a polynomial in n fused multiply-adds. With SIMD instructions one can compute four single-precision values in parallel using a series of **fma**s. The function hornerV does this, and works for any order polynomial. The order of the polynomial is one minus the length of the list of coefficients. This is the highest-throughput way of evaluating polynomials. Other methods can reduce latency by increasing parallelism, but this is wasteful for deeply software-pipelined loops.

```
hornerV :: SPUType val ⇒ [val] → val → val
hornerV (a1 : a2 : as) v = fma v (hornerV (a2 : as) v) a1
hornerV [a1] _ = a1
hornerV [] _ = error "hornerV: need a coeff"
```

5.2 Mantissa Extraction.

An IEEE floating point number is stored as packed bit fields: 1 sign bit, 8 exponent bits and 23 mantissa bits (for single precision). The SPU ISA uses the same format, although exceptional values are not recognized. So on the SPU all values of the exponent and mantissa bit fields represent the floating point number

$$x = (-1)^{\text{sign}} 2^{\text{exponent}-127} 1.\text{mantissa}. \tag{2}$$

Common patterns in special function evaluation include extraction of the exponent and mantissa fields. Since the exponent bits are often used in different bit positions, we have higher-level patterns which construct declarative assembly and the various constants required to put the exponential bits into any position in the word.

The fractional part represented by the mantissa is most often used as a floating point number, so we have a pattern to form the number 1.mantissa$\in [1, 2)$, by merging the mantissa bits with the bit pattern for 1.0.

$$
\begin{aligned}
&\underline{\text{onePlusMant}} :: \mathit{SPUType\ val} \Rightarrow \mathit{val} \rightarrow \mathit{val}\\
&\underline{\text{onePlusMant}}\ v = \mathbf{selb}\ \mathit{floatOne}\ v\ \mathit{mantissaBits}\\
&\quad \text{where}\\
&\qquad \mathit{floatOne} = \mathit{unwrds4}\ 0\ \mathit{x3f800000}\\
&\qquad \mathit{mantissaBits} = \mathit{unwrds4}\ 0\ \mathit{x007fffff}
\end{aligned}
$$

5.3 Exponent Extraction

Depending on the use, extracting the exponent can be combined with other operations, but we provide an odd-pipeline method of putting the exponent byte into any aligned byte surrounded by zeros. This works by rotating the whole quadword left one bit, to rotate out the mantissa's sign bit and put the exponent byte into the first byte slot of each word. Then use **shufb** to insert that byte into the desired position slot of the word which is otherwise filled with zeros.

$$
\begin{aligned}
&\underline{\text{extractExp}} :: \mathit{SPUType\ val} \Rightarrow \mathit{Integer} \rightarrow \mathit{val} \rightarrow \mathit{val}\\
&\underline{\text{extractExp}}\ \mathit{destSlot}\ v = \mathbf{shufb}\ \mathit{vShifted}\ \mathit{vShifted}\ \$\ \mathit{unbytes}\ \$\ \mathit{mapWord}\ \mathit{destSlot} =\!<\!<\ [0\,..\,3]\\
&\quad \text{where}\\
&\qquad \mathit{vShifted} = \mathbf{rotqbii}\ v\ 1\\
&\mathit{mapWord}\ 0\ i = [0 + 4 * i, \mathit{shufb0x00}, \mathit{shufb0x00}, \mathit{shufb0x00}]\\
&\mathit{mapWord}\ 1\ i = [\mathit{shufb0x00}, 0 + 4 * i, \mathit{shufb0x00}, \mathit{shufb0x00}]\\
&\mathit{mapWord}\ 2\ i = [\mathit{shufb0x00}, \mathit{shufb0x00}, 0 + 4 * i, \mathit{shufb0x00}]\\
&\mathit{mapWord}\ 3\ i = [\mathit{shufb0x00}, \mathit{shufb0x00}, \mathit{shufb0x00}, 0 + 4 * i]\\
&\mathit{mapWord}\ \_\ \_ = \mathit{error}\ \texttt{"extractExp:mapWord not a slot"}
\end{aligned}
$$

5.4 Calculating a Bit-Shifted Division

In integer arithmetic, we can save considerable numbers of instructions by using *approximate inverses* instead of converting numbers to floating point, doing expensive arithmetic, and converting back.

Some of the most common uses are captured by a pattern that approximates the calculation of the result of the division by q of the integer-coefficient linear form $p \cdot x + s$ in such a way that the integral part of the quotient $\frac{p \cdot x + s}{q}$ is returned in the left $(32 - n)$ bits, and the fractional part in the right n bits.

Given a precision n, this calculates for each word element v_i of the vector v, the approximate linear function

$$
\frac{pv_i + s}{q} \approx_{\text{fixed point}(n)} \left\lfloor 2^n \left(\frac{p}{q} \cdot v_i + \frac{s}{q} \right) \right\rfloor \quad .
$$

Since p, q, and s are compile-time integer constants, we can approximate this using a linear form with coefficients calculated at compile-time. As a result of this approach, the result has only up to 16 correct digits corresponding to the integer part (starting at the 2^n bit) and a number of correct fractional digits to the right of this point. The number of correct digits decreases with the size of the inputs.

If fractional digits are needed, it is easier to enumerate all possible inputs than to formally reason about the achieved precision; we show an example of this in our cube root implementation in Sect. 6.

In the implementation, we have to check to see whether a nonzero offset is required. If it is, we need to use a fused multiply add, **mpya**, and two register constants; if not, we can use the immediate form if the multiplicand is in the limited range for signed or unsigned immediates, otherwise we use the register form with a single register constant.

$$
\begin{array}{l}
\mathit{divShiftMA} :: \mathit{SPUType\ val} \Rightarrow \mathit{Integer} \rightarrow \mathit{Integer} \rightarrow \mathit{Integer} \rightarrow \mathit{Integer} \rightarrow \mathit{val} \rightarrow \mathit{val} \\
\mathit{divShiftMA}\ p\ q\ s\ n\ v \\
\quad \begin{array}{ll}
\mid s \not\equiv 0 & = \mathbf{mpya}\ m\ v\ b \\
\mid m' < 2 \uparrow 10 \wedge m' > 0 & = \mathbf{mpyui}\ v\ m' \\
\mid m' < 2 \uparrow 9 \ \ \wedge m' \geqslant (-2 \uparrow 9) & = \mathbf{mpyi}\ v\ m' \\
\mid \mathit{otherwise} & = \mathbf{mpy}\ v\ m
\end{array} \\
\quad \text{where} \\
\qquad m' = (p * 2 \uparrow n + (q-1))\ `\mathit{div}`\ q \quad \text{-- integer exponent and division} \\
\qquad m = \mathit{unwrds4}\ m' \\
\qquad b = \mathit{unwrds4}\ \$\ (s * 2 \uparrow n)\ `\mathit{div}`\ q
\end{array}
$$

By encapsulating all these relatively tricky details into a DSL function, we free the domain expert from doing all this low-level arithmetic and allow them to work on a higher level of abstraction, as we will show in Sect. 6.

5.5 Mixed log/linear intervals

We can do lookups based on intervals which mix logarithmic and linear scales by forming an index from a combination of bits from the exponent field and from the mantissa. This is an efficient way to construct lookup keys corresponding to a contiguous set of intervals, with either equal-sized intervals or intervals whose size doubles regularly from one end to the other. Varying the size of the intervals may result in an approximation with lower order but the same maximum error, especially for functions with singularities or zero crossings.

However we choose the interval domains of the polynomial approximations, we have to look up the correct set of coefficients at run time. On SIMD architectures with byte permutation, like VMX and the SPU ISA, this can be done efficiently for some sizes of tables entirely in registers without accessing memory.

We now present two patterns for lookup. In the first case, we start with a field of bits and look up based on that, without making any assumptions about the meaning of the bits. In some functions, the bit keys are constructed by concatenating bits containing heterogeneous information, *e.g.* in difficult cases where polynomial approximation works in one part of the domain but not in another.

In the second case, log-linear intervals, we have a pattern integrating key construction and lookup, because correctly constructing keys to match log-linear intervals is error-prone.

5.5.1 Register Lookup in 8-Word Tables

In this section we look at patterns for lookups in tables of 8 words. In most applications, the words are 32-bit floating-point polynomial coefficients, but not always; for example we also have one application where the lookup retrieves bit masks. Therefore we strive to keep the exposed components of our tool set for this kind of pattern as general and modular as possible, to allow the domain expert to replace components of our high-level patterns, or modify components when inter-component optimizations are possible.

For 8-way lookup, the pattern has two main parts: (1) constructing keys (maps of byte indices) and instructions to perform the lookup, and (2) constructing the lookup tables, which are lists of register constants packed with single-precision floating point values.

In the current pattern, a table contains eight words, and a single lookup selects one of these. Since on the SPU, the main instruction supporting such lookups, **shufb**, does selection of Bytes, not words, our lookup needs to locate four out of 32 bytes to assemble one word, and a single **shufb** instruction allows exactly selection out of the 32 bytes contained in its first two register arguments.

Therefore we need 5 bits for lookup keys for the the individual byte, and three of these, "`kkk`" will be the high-level key for selection of one of the eight original alternative words.

Since application may produce these three-bit word-keys in many different ways, we must not make any assumptions about their original alignment. If the original alignment of the three key bits falls within the five bits used by **shufb**, i.e., in one of the three byte patterns `***kkk**`, `****kkk*`, and `*****kkk`, then we can directly use that key alignment; otherwise we have to rotate the word-key into one of these three positions, and we choose the last.

For the byte lookups, the word-key "`kkk`" needs to be replicated four times, and for each of the three possible word-key positions, it is completed in a different way to four different five-bit byte-keys (*i.e.*, keys for byte selection), resulting in **shufb** argument vectors of the following shape:

```
***kkk00 ***kkk01 ***kkk10 ***kkk11    or
***0kkk0 ***0kkk1 ***1kkk0 ***1kkk1    or
***00kkk ***01kkk ***10kkk ***11kkk .
```

The function *key8Word* generates these **shufb** maps from keys in different bit positions. The bit positions are given as $\log_2$ of the positional value, *i.e.*, little-endian bit number, with the right-most bit of the 32-bit word being considered as at position 0.

Most of the complication arises from generating a *join*ed code graph with two paths, each using instructions in different pipelines, so the scheduler can make the choice; this increases superscalar dispatch.

In detail, one alternative does rotation within the word, **roti**, while the other does it within the quadword, **rotqbii**. The second instruction only supports rotations of less than eight bits, so the relevant bits may leave the component word in the second case. As a result, the modulo arithmetic has to be done differently in the two cases.

Since we can only join register values which will become nodes in the code graph, and not arbitrary Haskell data types, we have to encapsulate the variation within an auxiliary function (*splat*).

In the final step, the "select bytes" instruction **selb** uses the constant *mask* for selecting the key bits `kkk` and the constant *c0123* for inserting the `00`, `01`, `10`, `11` around the key to generate a 5-bit lookup index to be used by the **shufb** in lookup8Word below for look-up into 32-byte tables (in the two registers addressable by a single **shufb**).

$$\mathit{key8Word} :: \mathit{forall\ val} \circ (\mathit{SPUType\ val}, \mathit{HasJoin\ val}) \Rightarrow \mathit{Integer} \rightarrow \mathit{val} \rightarrow \mathit{val}$$
$$\mathit{key8Word\ low\ v} = \mathbf{selb}\ \mathit{c0123\ look2\ mask}$$
$$\quad \mathsf{where}$$

To provide a simpler interface to the domain expert, we use the bit position of the 3-bit key (passed in as the exponent of the place value of the *low* bit) to determine the minimum number of instructions to generate the key. If the key bits are within the five low-order bits of any byte, we do not need a rotate. In all cases we need to know the byte position of the bits (after rotation) so that byte can be replicated to all four bytes corresponding to each word being looked up.

$$\mathit{byte}, \mathit{bit} :: \mathit{Integer}$$
$$\mathit{look2} :: \mathit{val}$$

```
((byte, bit), look2) = if lowBit ∈ [0, 1, 2]
  then (low8, splat 16 v)   -- no rotation needed
  else ((1 + lowByte, 0), join [splat 4 $ roti v distance, splat 16 $ rotqbii v distance]
       )
```

From the bit position, we look up the correct *mask* to use to insert the appropriate additional bits *c0123*.

```
c0123, mask :: val
(c0123, mask) = case bit of
  0 → (unwrds4 0 x00081018, unwrds4 0 x07070707)
  1 → (unwrds4 0 x00011011, unwrds4 0 x0e0e0e0e)
  2 → (unwrds4 0 x00010203, unwrds4 0 x1c1c1c1c)
  _ → error "key8Word: impossible"
```

These masks depend on the (rotated) *bit* index of the lowest bit of the `kkk` key as it is located within the five-bit keys; the basis for this is the position *low* of the key in the function argument *v*, for which we calculate the byte-coordinates, and the number of bits we would have to rotate left to get *low* aligned on the lowest bit of a byte:

```
low8@(lowByte, lowBit) = low `divMod` 8
distance = (8 − lowBit) `mod` 8
```

The auxiliary function *splat* has as its main task to produce a **shufb** instruction that achieves replication of the three key bits “`kkk`” over all four bytes of the respective word; the replication indices are calculated taking into account the possibility that **rotqbii** might have shifted the key over a word boundary, and around the quadword boundary (16 bytes). For **roti**, the **shufb**-index needs to point within the same word, so *rotWidth* is instantiated to 4 in that case. (The generated indices all refer to the first argument of the **shufb** instruction, so the second argument is arbitrary.)

```
splat :: Integer → val → val
splat rotWidth x = shufb x x $ unbytes $ map (`mod`16)
  $ map (((3 − byte) `mod` rotWidth)+) $ replicate 4 =<< [0, 4, 8, 12]
```

Corresponding to different positions for the index bits, `kkk`, are three different ways of arranging the bytes in the 16-byte constants:

```
mk8WordTbl :: (SPUType val) ⇒ Integer → [[Double]] → [(val, val)]
mk8WordTbl low xs = map (mk8WordPair low) xs

mk8WordPair :: SPUType val ⇒ Integer → [Double] → (val, val)
mk8WordPair low xs = unbytes `prod` unbytes $ splitAt 16 abnormalBytes
  where
    (x1s, x2s)    = splitAt 4 xs
    normalBytes   = concatMap (bytes ∘ idSim ∘ unfloats) [x1s, x2s]
    abnormalBytes = map snd $ List.sort $ zip lexOrd normalBytes
    lexOrd = case (low `mod` 8) of
      1 → [(i, j, k) | j ← [0 .. 7], i ← [0 .. 1], k ← [0 .. 1]]
      2 → [(j, i, 0) | j ← [0 .. 7], i ← [0 .. 3]]
      _ → [(i, j, 0) | j ← [0 .. 7], i ← [0 .. 3]]
```

To insure that the lookup key is compatible with the format of the lookup table, we encapsulate both parts in one pattern:

$$\underline{\text{lookup8Word}} :: (\mathit{SPUType\ val}, \mathit{HasJoin\ val}) \Rightarrow (\mathit{Integer}, \mathit{Integer}) \to [[\mathit{Double}]] \to \mathit{val} \to [\mathit{val}]$$
$$\underline{\text{lookup8Word}}\ (\mathit{high}, \mathit{low})\ \mathit{tbl}\ v = \mathit{bitWidthErr}\ \texttt{"lookup8Word"}\ \mathit{high}\ \mathit{low}\ \$\ \mathit{map}\ \mathit{index}\ \mathit{regTbl}$$
$$\quad \text{where}$$
$$\qquad \mathit{regTbl} = \mathit{mk8WordTbl}\ \mathit{low}\ \mathit{tbl}$$
$$\qquad \mathit{index}\ (\mathit{v1}, \mathit{v2}) = \mathbf{shufb}\ \mathit{v1}\ \mathit{v2}\ \$\ \mathit{key8Word}\ \mathit{low}\ v$$

This pattern is typical in that a small number of instructions are generated following a lot of case checking to test different preconditions for different combinations of instructions. Many possible illegal parameters are filtered out, and we try to return meaningful errors to the domain expert rather than letting the Haskell run-time system trap an irrefutable pattern.

5.5.2 Lookup in 16-Word Table — Lazy Higher-Order Code Generation

Similar to the 8-way lookup in Sect. 5.5.1, we now provide functions to allocate lists of floating-point numbers into registers for 16-way lookups. The function *mk16WordTbl* maps this over a list of such lists; this is used when looking up coefficients for a list of 16 polynomial segments.

The lookup of 16 values requires 3 **shufb**s to look up four words or 4 **shufb**s to look up 8: two to look up the high- and low-order halfwords, and two to separate the first and second sets of four halfwords, and interleave the high- and low-order parts.

We look up values in such tables from single keys with the bit patterns `000kkkk0 000kkkk1` where `kkkk` is the 16-way lookup bit pattern, and this halfword pattern is repeated for 16 keys. This involves four instructions and requires two auxiliary lookup constants:

$$(\mathit{fstInterleave}, \mathit{sndInterleave}) = \mathit{unbytes}\ `\mathit{prod}`\ \mathit{unbytes}\ \$\ \mathit{splitAt}\ \mathit{bytes}\ \$\ \mathit{concat}$$
$$\quad \$\ \mathit{zipWith}\ (+\!\!+)\ (\mathit{chunks}\ 2\ [0\,..\,\mathit{bytes}])\ (\mathit{chunks}\ 2\ [\mathit{bytes}+1\,..])$$
$$\quad \text{where}\ \mathit{bytes} = 16$$

Keys can be constructed in different ways depending on the application, but some constructions would be hard to get right without language support.

The calculated break points are used both to construct the approximations (using Maple in our case), and to generate the code to construct the lookup key at run-time, in a function that accepts, besides the lookup specification and the list of coefficient lists, a pair of two arguments for which the lookup is performed in parallel, returning a pair of retrieved lists of coefficients:

$$\mathit{lookup16X2Coeffs} :: (\mathit{SPUType\ val}) \Rightarrow \mathit{LookupSpec} \to [[\mathit{Double}]] \to (\mathit{val}, \mathit{val}) \to ([\mathit{val}], [\mathit{val}])$$

We do not go into further detail concerning the construction of these lookup tables and of the machine instructions for performing the lookup, since these are similar in spirit to those of Sect. 5.5.1. Instead, we explain in more detail a non-trivial application of characteristics of the host programming language Haskell.

Since *lookup16X2Coeffs* allows parallel lookup of two keys, we offer a wrapper function (which we used in Sect. 3) that “unrolls” an argument function twice to make use of this.

$$\underline{\text{use16X2lookup}} :: (\mathit{SPUType\ val}) \Rightarrow$$
$$\quad \mathit{RegLookupSpec} \to [[\mathit{Double}]] \to ([\mathit{val}] \to \mathit{arg} \to (\mathit{val}, \mathit{result})) \to (\mathit{arg}, \mathit{arg}) \to (\mathit{result}, \mathit{result})$$

The implementation of this function “ties the knot” by creating the kind of apparently recursive data dependencies that can only be resolved in a non-strict programming language like Haskell, when an appropriately non-strict argument function *mkKeyResult* is supplied for which the *key* result does not depend on the *coeffs* argument.

<u>use16X2lookup</u> *spec rawCoeffs mkKeyResult* (*v1*, *v2*) = let
  (*coeffs1*, *coeffs2*) = *lookup16X2Coeffs* (*lookupMemo spec*) *rawCoeffs* (*key1*, *key2*)
  (*key1*, *result1*) = *mkKeyResult coeffs1 v1*
  (*key2*, *result2*) = *mkKeyResult coeffs2 v2*
  in (*result1*, *result2*)

In Fig. 7, we show a direct code graph representation of this function: For each of the two calls to *mkKeyResult*, wich are drawn with arrow from argument nodes to the *mkKeyResult* hyperedge, and arrows from the hyperedge to its results, there appears to be a cycle around *key* and *coeffs*.

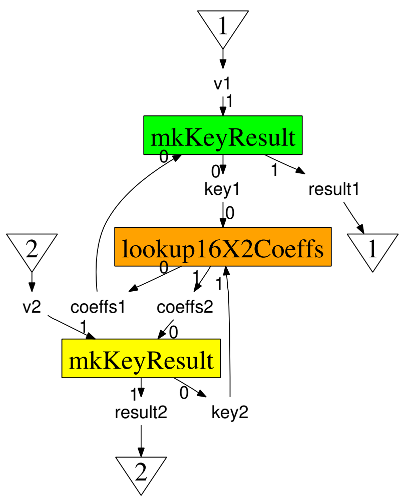

Figure 7: <u>use16X2lookup</u> definition

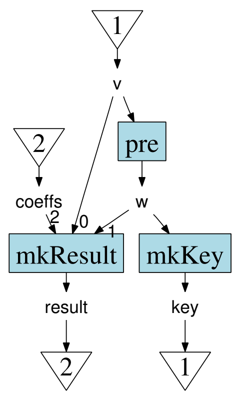

Figure 8: *mkKeyResult* example

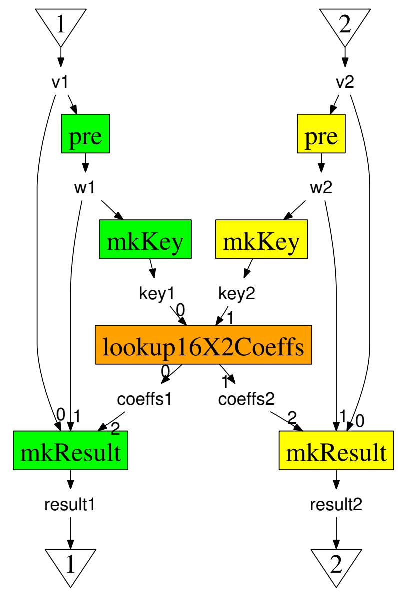

Figure 9: <u>use16X2lookup</u> application

We can make it explicit that *key* does not depend on *coeffs*, for example by specialising for *mkKeyResult* functions of the following shape:

mkKeyResult coeffs v = let *w* = *pre v* in (*mkKey w*, *mkResult v w coeffs*)

This definition is depicted in Fig. 8; since there is no path in this from the *coeffs* input to the *key* output, application of <u>use16X2lookup</u> to a function of this shape, depicted in Fig. 9, is cycle-free, and therefore can be used for code generation. Since our <u>ftanh</u> function from Sect. 3 is built using *use16X2lookup1*, it is no pure coincidence that the resulting code graph in Fig. 5 exhibits the same symmetries as Fig. 9.

Although this pattern may at first sight appear to be somewhat convoluted, it has the advantage that is guarantees that the argument function *mkKeyResult* can use only one lookup; the two lookups required by the two calls to *mkKeyResult* are then parallelised in *lookup16X2Coeffs*. The "obvious" reformulation with a second-order function of type $((val \to [val]) \to arg \to result)$ as third argument fails to establish this guarantee, and therefore would not enable this parallelisation.

6 Another Example: Cube Root

Cube Root is defined to be the unique real cube root with the same sign as the input. We calculate it using

$$(-1)^{\text{sign}}\ 2^e\ (1+\text{frac}) \mapsto (-1)^{\text{sign}}\ 2^q\ 2^{r/3}\ f(1+\text{frac}) \tag{3}$$

where q and r are integers such that

$$e = 3 * q + r, \qquad 0 \leq r < 3, \tag{4}$$

and $f(x)$ is a piecewise order-three polynomial minimax approximation of $(x)^{1/3}$ on the interval $[1, 2)$.

As mentioned in Sect. 5.4, given the small possible number of inputs to the *divShiftMA* call below, the easiest way to verify correctness of the bit manipulations involved in the calculation of the fractional digits is performing an exhaustive test; we embody this in an assertion *cbrtAssert* defined at the end of this section.

$\underline{\text{cbrt}}\ \mathit{v} = \mathit{assert\ cbrtAssert}$ `"cbrt"` result
  where

Since we process the input in components, we cannot rely on hardware to round denormals to zero, and must detect it ourselves by comparing the biased exponent with zero:

$$\mathit{denormal} = \mathbf{ceqi}\ \mathit{exponent}\ 0$$

and returning zero in that case

$$\mathit{result} = \mathbf{selb}\ \mathit{unsigned}\ (\mathit{unwrds4}\ 0)\ \mathit{denormal}$$

We calculate the exponent and polynomial parts separately, and combine them using floating-point multiplication,

$$\mathit{unsigned} = \mathbf{fm}\ \mathit{signCbrtExp}\ \mathit{evalPoly}$$

Insert the exponent divided by three into the sign and mantissa of the cube root of the remainder of the exponent division.

$$\begin{aligned}\mathit{signCbrtExp} = \mathbf{selb}\ \mathit{signMant}\ (\mathit{join}\ \$\ \mathit{map}\ (\$\mathit{expDiv3shift16}\ 7)\ [\mathbf{shli}, \mathbf{rotqbii}])\\ (\mathit{unwrds4}\ \$\ 2 \uparrow 31 - 2 \uparrow 23)\end{aligned}$$

We use the function $\underline{\text{extractExp}}$ from Sect. 5.3 to extract the exponent bits, dropping the sign bit, into the third byte:

$$\mathit{exponent} = \underline{\text{extractExp}}\ 3\ \mathit{v}$$

$$\mathit{expDiv3shift16} = \mathit{approxDiv3}\ \mathit{exponent}$$

Put the high two bits of the remainder, which are accurate into the low-order byte of each word, and set all other bytes to zero.

$$\begin{aligned}\mathit{remainder} = \mathbf{shufb}\ \mathit{expDiv3shift16}\ \mathit{expDiv3shift16}\\ \$\ \mathit{unbytes}\ \$\ (\mathit{padLeftTo}\ 4\ \mathit{shufb0x00} \circ (:[\,])) =<< [2, 6\,..]\end{aligned}$$

By comparing remainder with $0 \cdot 64, 1 \cdot 64, 2 \cdot 64$ we can form masks and use them to select 2^r from pre-calculated values $2^0, 2^{1/3}, 2^{2/3}$.

$$\begin{aligned}\mathit{oneOrCbrt2} = \mathbf{selb}\ &(\mathit{unfloats4}\ 1)\\ &(\mathit{unfloats4}\ \$\ (1 + 2 ** (-24)) * 2 ** (1\ /\ 3))\\ &(\mathbf{cgti}\ \mathit{remainder}\ (2 \uparrow 6))\end{aligned}$$

$$\mathit{cbrtRem} = \mathbf{selb}\ \mathit{oneOrCbrt2}$$

$$(\mathit{unfloats4}\ \$\ (1 + 2 ** (-24)) * 2 ** (2\ /\ 3))$$
$$(\mathbf{cgti}\ \mathsf{remainder}\ (2 \uparrow 7))$$

Combine the byte with the sign bit with the bytes with the mantissa of $1, 2^{1/3}, 2^{2/3}$.

$$\mathsf{signMant} = \mathbf{shufb}\ \mathsf{v}\ \mathsf{cbrtRem}\ \$\ \mathit{unbytes}$$
$$[0, 17, 18, 19, 4, 21, 22, 23, 8, 25, 26, 27, 12, 29, 30, 31]$$

Merge the mantissa bits with a constant 1.0 to form 1.mantissa.

$$\mathsf{frac} = \underline{\text{onePlusMant}}\ \mathsf{v}$$

Using either the argument or the fractional bits which have been extracted, take the bits with values $2^{22}, 2^{21}, 2^{20}$ and form a lookup key, then use it to look up *length expCoeffs24bits* coefficients from register values constructed using the polynomial coefficients *expCoeffs24bits*.

$$\mathsf{coeffs} = \underline{\text{lookup8Word}}\ (22, 20)\ \mathsf{expCoeffs24bits}\ \$\ \mathsf{join}\ [\mathsf{v}, \mathsf{frac}]$$

Evaluate the polynomial on the fractional part.

$$\mathsf{evalPoly} = \underline{\text{hornerV}}\ \mathsf{coeffs}\ \mathsf{frac}$$

One of the patterns we use is only calculates an accurate value under a complicated set of preconditions, so we define the function at the top level

$$\mathsf{expBias} = 127$$
$$\mathsf{approxDiv3} = \mathsf{divShiftMA}\ 1\ 3\ (2 * \mathsf{expBias})\ 16$$

and test that for all input values we are going to use, the precondition (that the first two fractional bits in the approximate division by three are correct).

```
cbrtAssert = List.and [(divMod i 3) ≡ (extractDivMod $ approxDiv3 $ (unwrds4 $ i + expBias :: Val))
                       | i ← [expBias − 255 .. expBias]]
  where
    extractDivMod w = case bytes w of
      _ : v1 : v2 : _ → (v1 − expBias, div v2 64)
      _               → error "impossible"
```

7 Other Features

In addition to special function support, we exploit code generation in several other aspects of this project: support for iteration, support for linear algebra, and support for interpolation. We now describe the first two informally in the remainder of this section; the technical details are outside the scope of the current report.

7.1 Iteration Patterns: "Tickers"

We separate control flow and data flow into a nested code graph representation called *MultiLoop*, since generated code will typically be structured as a loop with versioned loop bodies. We allow constraint annotations for software pipelining; and allow users or high-level code generators to supply declarative code which makes one step in the iteration through the list of addresses of scheduled versions.

To specify correct iterator code, it is necessary to understand software pipelining of the MultiLoop and keep track of several details. To insulate the domain expert from these intricacies, several code generators, called *tickers*, generate the required instructions for common cases.

Since computation (integer, logical and floating-point) dominates most of the code bodies we have considered, the design goal of most tickers is to replace as much of the loop overhead with data movement instructions. In all implemented cases, we are able to limit the number of arithmetic instructions to one add. This is possible where at most seven pointers and one counter need to be updated, and sixteen-bit integers are sufficient. Because the Cell SPUs have limited addressable local memory, this is a reasonable assumption.

By their nature, ticker generators are not very portable, because their efficiency derives from taking advantage of many architecture and implementation details at the same time.

At the present, ticker generators exist for

- so-called vector math functions, (compositions of *map*, *fold*, *zip* and *unzip* over linear arrays),
- the pattern for fixed-size matrix-matrix (block) multiplication,
- fixed-size three-dimensional (partial) separable transformations (Fourier, wavelet, *etc.*.).

The type of code generation varies a lot from one pattern to the next. In the simple map pattern, the complication comes from the need to both unroll and break a loop body into software staging. As a result, initial loads and final stores have pointer movement which is out of synchronization, depending on the number of stages in the scheduled loop. The map ticker pattern takes care of these computations, and packages the result plus the function body and load/store into a schedulable MultiLoop specification.

Operations which are essentially linear algebra have different requirements. Instead of simple unrolling, different types of blocking of operations takes place at a higher level. The resulting code graphs to be iteratively applied are larger and easier to schedule and the numbers of iterations are known at compile time, but the pointer arithmetic is more complicated.

As a result, these tickers are relatively simple compositions of basic code blocks (*e.g.*, load/store and matrix multiplication for a specific storage pattern) and iterators generated through a compile-time enumeration of the lifetime of the loop (counters and pointer) and a combinatorial optimization problem the aim of which is to efficiently implement a representation of an Abelian group on the byte values in a set of quadwords using machine instructions.

For example, the helper function

$$\underline{\text{buildRots}} :: \mathit{SPUType}\ \mathit{val} \Rightarrow [(\mathit{Int}, \mathit{Int}, [\mathit{Integer}])] \rightarrow ([\mathit{val}] \rightarrow [\mathit{val}], [\mathit{val}])$$

takes a list of tuples, each consisting of:

- the active position of the byte,
- the length of the cycle acting on this byte value, and
- the byte values which cycle through this position,

and generates a function composed of machine instructions that modify a set of register values, and a list of initial values for those registers. Each application of the function acts as the generator of the group, as shown in figure 10.

Other tickers are built with rotation instructions, and nested loops require that some of these structures index each other, giving the group the structure of a direct product.

In summary, a domain expert modifies control flow by modifying the iterator element, which corresponds to modifying the statements in a C *for* loop header. In this case, however, the iterator is written declaratively, using only data flow instructions, and our typing setup ensures that side effects

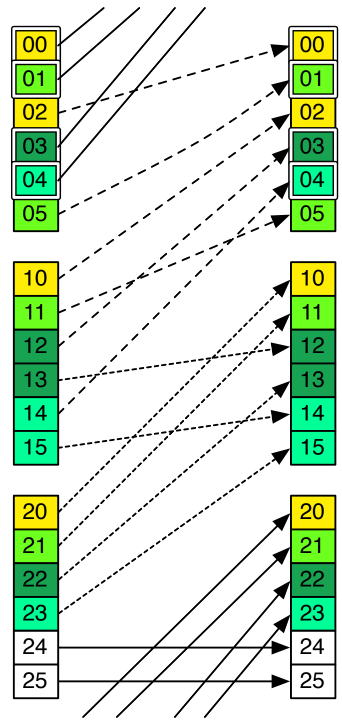

Figure 10: Simplified six-byte SIMD version of the generator of the Abelian group generated by buildRots. Byte values from each set cycle through the set of byte positions marked by colors. The positions of the active bytes have double outlines, and the three permute instructions required for the group generator are indicated by solid/dashed/dotted arrows.

are not possible. The efficiency of the iterators is entirely in the hands of the domain expert. Our experience is that coupled with a good scheduling algorithm [Thaller 2006], this approach results in near optimal code, so there is no performance advantage in directly writing assembly code.

7.2 Linear Algebra

In the linear algebra functions we have currently implemented, special-purpose code generators perform two types of functions: aggregation (*i.e.*, operations implemented using multi-level blocking), and providing optimized code for byte shuffling for operations like transpose, interleaving of real/imaginary parts, *etc.*.

Aggregation is almost equivalent to macro expansion, with simple tests to choose between multiplications and fused multiply-adds.

Optimization is more varied. For example, we have a specific pattern for an optimized outer product which, at code generation time, extracts results for part of the generated code from a symbolic computation in the "interpreted instance" of the DSL, and uses these results to guide the final arrangement of the generated code.

This *interpretation at compile time* could be called "way before interpretation", in analogy to "just-in-time compilation".

Additional computation is required in such cases where the required result is not the kind of simple map over parallel inputs shortly described in Sect. 7.1.

Currently, patterns for linear algebra generate code and make decisions about instruction selection and the number of some auxiliary register values, but they do not yet make high-level decisions about block size, *etc.*.

8 Conclusion and Outlook

We have used this DSL to construct twenty-six elementary math functions commonly found in optimized math libraries. Our single-precision implementations are significantly faster than the best alternatives. We attribute this to the support we have for patterns which are difficult to write by hand. Some of the work was carried out by undergraduate mathematics students who had little experience writing assembly language, but had seen functional programming before. They were able to be productive after very little instruction and able to use our DSL inside the Haskell interpreter GHCi to explore new ideas. Our DSL shifted the bottleneck from being productive using SIMD parallelism to being able to come up with efficient approximation schemes.

In the next phase, we will implement the same library in double precision. The parallelism issues are different in double precision, and the required accuracy is higher. We anticipate a smaller number of parameterized patterns to be efficient for double precision. Knowing this, and having the experience with single precision, we expect to get an even bigger boost in productivity from our DSL.

In parallel, we will work out details of higher level patterns to encapsulate inter-SPU synchronization.

References

[Bandera, Gonzalez+ 2004] G. Bandera, M. Gonzalez, J. Villalba, J. Hormigo, E. L. Zapata. *Evaluation of elementary functions using multimedia features.* In: Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04), 2004.

[Dubey, Kaplan+ 2001] P. K. Dubey, M. A. Kaplan, S. M. Joshi. *Efficient CRC generation utilizing parallel table lookup operations.* US Patent 6223320 , 2001.

[Golub, Smith 1971] G. H. Golub, L. B. Smith. *Algorithm 414: Chebyshev approximation of continuous functions by a Chebyshev system of functions.* Commun. ACM **14** 737–746, 1971.

[Grelck, Scholz 2006] C. Grelck, S.-B. Scholz. *SAC: a functional array language for efficient multi-threaded execution.* Int. J. Parallel Program. **34** 383–427, 2006.

[Hudak 1996] P. Hudak. *Building domain-specific embedded languages.* ACM Computing Surveys **28** 196–196, 1996.

[Hudak 1998] P. Hudak. *Modular Domain Specific Languages and Tools.* In P. Devanbu, J. Poulin, eds., Proceedings: Fifth International Conference on Software Reuse, pp. 134–142. IEEE Computer Society Press, 1998.

[IBM 2005] IBM. *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual.* IBM Systems and Technology Group, Hopewell Junction, NY, 2005.

[IBM 2006] IBM. *Synergistic Processor Unit Instruction Set Architecture.* IBM Systems and Technology Group, Hopewell Junction, NY, 2006.

[Kahl, Anand+ 2006] W. Kahl, C. K. Anand, J. Carette. *Control-Flow Semantics for Assembly-Level Data-Flow Graphs.* In W. McCaull et al., eds., 8th Intl. Seminar on Relational Methods in Computer Science, RelMiCS 2005, LNCS **3929**, pp. 147–160. Springer, 2006.

[Knuth 1984] D. E. Knuth. *Literate Programming.* The Computer Journal **27** 97–111, 1984.

[Knuth 1992] D. E. Knuth. *Literate Programming*, CSLI Lecture Notes **27**. Center for the Study of Language and Information, 1992.

[Merrheim, Muller+ 1993] X. Merrheim, J.-M. Muller, H.-J. Yeh. *Fast evaluation of polynomials and inverses of polynomials.* In: Proc. 11th Symp. Computer Arithmetic, 1993.

[Moggi 1991a] E. Moggi. *A Modular Approach to Denotational Semantics.* In D. H. Pitt et al., eds., Category Theory and Computer Science, LNCS **530**, pp. 138–139. Springer, 1991.

[Moggi 1991b] E. Moggi. *Notions of Computation and Monads.* Information and Computation **93** 55–92, 1991.

[Mueller, Lumsdaine 2006a] C. Mueller, A. Lumsdaine. *Expression and Loop Libraries for High-Performance Code Synthesis.* In: Proceedings of the 19th International Workshop on Languages and Compilers for Parallel Computing, 2006.

[Mueller, Lumsdaine 2006b] C. Mueller, A. Lumsdaine. *Runtime synthesis of high-performance code from scripting languages.* In: OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pp. 954–963, New York, NY, USA, 2006. ACM Press.

[Peyton Jones+ 2003] S. Peyton Jones et al. *The Revised Haskell 98 Report.* Cambridge University Press, 2003. Also on http://haskell.org/.

[Ramanathan 2006] R. Ramanathan. *Extending the World's Most Popular Processor Architecture, New innovations that improve the performance and energy efficiency of Intel architecture.* Intel Corporation, 2006.

[Sazegari 2002] A. Sazegari. *Vectorized Table Lookup.* US Patent 20020184480 , 2002.

[Shi, Lee 2000] Z. Shi, R. B. Lee. *Bit Permutation Instructions for Accelerating Software Cryptography.* asap **00** 138, 2000.

[Thaller 2006] W. Thaller. *Explicitly Staged Software Pipelining.* Master's thesis, McMaster University, Department of Computing and Software, 2006. http://sqrl.mcmaster.ca/~anand/papers/ThallerMScExSSP.pdf.

[Wadler 1990] P. Wadler. *Comprehending Monads.* In: Proc. 1990 ACM Conference on Lisp and Functional Programming, 1990.

[Wadler 1992] P. Wadler. *The Essence of Functional Programming.* In: 19th POPL, pp. 1–14, Albuquerque, New Mexico, 1992. acm press. invited talk.