Journal of Instruction-Level Parallelism 1 (2006) 1-15 Submitted 2/06; published

Benefits of Register-Level Lookup for a CELL SPU Math Library

Christopher Kumar Anand ANANDC@MCMASTER.CA
Wei Li
Anuroop Sharma
Sanvesh Srivastava
McMaster University, Computing and Software, ITB-201
1280 Main Street West, Hamilton, ON L8S 4K1 Canada

Abstract

This paper demonstrates techniques for increasing instruction-level parallelism via register-level lookups as an alternative to predication, using examples from a library of elementary math functions targeting CELL SPUs. Comparing the performance of our library with functions released in the CELL SDK 1.1 which do not use register-level lookup to improve parallelism, we measure a performance advantage of 40 percent on average. Since some functions are too simple to benefit, the average masks much larger improvements for more complicated functions. We developed this library using a declarative assembly language and a SPU simulator written in Haskell. Through the examples, we show how this environment supports the rapid prototyping of compiler-optimization such as register-level lookups and the incorporation of code generation from mathematical models. The example code is documented in detail and should be a good introduction to these special features of the SPU instruction set architecture for both compiler writers and software developers.

1. Introduction

In this paper, we summarize our experience developing basic mathematical subroutines for the novel Synergistic Processing Units (SPUs) within the first implementation of the CELL broadband engine.

For many applications heavy in single-precision floating point arithmetic, the first CELL implementation with throughput of 64 flops per cycle promises significant advances in performance per watt, and we expect performance per dollar. But to get there will require an investment in better tools and practices. We hope to eventually share our tools and immediately inspire other tool builders, but we also hope to influence best programming practices by including non-trivial examples with detailed explanations.

We found that for elementary function calculation, register-level lookup table implementations using byte permutation and taking advantage of the zero-cost in mixing integer, logical and floating point operations (because of the unified register file) produce significant performance improvements over a more portable SIMD implementation.

We have implemented all of the single-precision functions provided by IBM's Math Acceleration SubSystem (MASS) library, using the techniques that we advocate. Performance comparison with the functions available in the Sony-Toshiba-IBM-supplied SDK [1] and pre-release hyperbolic functions, show that using these techniques improve throughput from a few percent to a factor of ten, with lookup-based implementations being twice as fast on average, for routines involving polynomial approximations (which excludes the short routines based on hardware interpolation like square root and divide). These are significant gains given that the existing SDK sample code is already branch-free and coded to take advantage of the basic 4X SIMD speedup.

Since we had originally planned to target VMX/Altivec with this work, we were curious as to the potential for additional parallel execution were the simple integer and logical instructions executable in parallel with floating point operations, as exists in some VMX/Altivec implementations. Fig. 4 shows quite a favorable instruction mix.

Although we do not explore such issues in this paper, this work is part of a larger project which seeks to exploit both coarse and fine levels of parallelism. Our approach involves breaking the current model for code generation, giving users greater access to the compiler internals, and mixing together high-level mathematical models, novel control-flow and declarative assembly language. In a separate paper we will describe Explicitly Staged Software Pipelining (ExSSP), a new approach to scheduling which, for the math functions described here, beats conventional unrolling and modulo scheduling of equivalent C code by 21 percent [2]. This multiplies the benefits of using register-level lookups described here. The bigger picture, including both measured additional acceleration and expected improvements in code transparency, justifies the decision not to use existing languages to describe code fragments in this paper, but to use our declarative assembly language embedded in the functional language Haskell. This is a literate language, meaning the source latex document can also be compiled by the Haskell compiler `ghc` and the resulting object can be used to generate C code, codegraphs for ExSSP, or to interpret the codegraphs, for example as part of automated unit testing. We intend to license both the tools and the exported C code under different licenses, although most users will probably use our functions as part of an IBM compiled library, the details of which are not yet known.

2. Declarative Assembler

As intrinsic functions used by many compilers to give high-level language programmers access to specific machine instructions, our declarative assembler is embedded into the language Haskell as a set of functions. For example

```
    vPositive = andc v signBit
```

defines a variable `vPositive` to be the result of applying the *and with complement* instruction to two register values `v` and signBit. Note that in Haskell, function application does not use parentheses. Unlike imperative languages, each variable can only be assigned once within its scope, and the order of assignment is not important. This is why it is referred to as *declarative*. This improves readability, and is somewhat closer to the data flow graphs used in our and other recent instruction schedulers. Scope can be determined by indentation, and `where` clauses. For example

```
fcoshAlt v = result
    where
        vMinusLog2 = fs v (unfloats4 $ log 2)
        minusVLog2 = fms v (unfloats4 $ (-1))
                           (unfloats4 $ log 2)
        result = fa (exp4 vMinusLog2) (exp4 minusVLog2)
```

is a complete function. The operator `$` controls evaluation order, and has the same effect as putting parentheses around the right hand side of the expression. In addition to functions for all of the SPU instructions we have implemented, there are several helper functions `bytes`, `unbytes`, `floats`, *etc.* which convert between register values and native Haskell values. The SPU instructions are actually implemented twice, one implementation simulates computation on the SPU with abstract register values (not assigned to registers), and the second implementation generates data flow graphs (which we refer to as codegraphs) for use by the instruction scheduler. We use this environment to interactively develop new instruction sequences and functions to construct declarative assembly functions

```
$ ghci `make` TestUtils InverseTrignometric
<snip>
Compiling TestUtils        ( Tricks/TestUtils.lhs, interpreted )
*TestUtils> let f x = (fma (unwrds4 0x3e7fffff) x (unfloats4 1))
*TestUtils> hexval $ f (unfloats [0,1/2,1-2**(-23),1]::SPUSim.Val)
"e000000feffffcfffffff8fffffffcf"
```

and to interactively test (and debug) functions

```
*TestUtils> TestUtils.fr InvTrig.arcsin asin (1/2) [i/111|i<-[-111..111]]
   Input Argument     Relative Error           Haskell                SPU
-----------------------------------------------------------------------
 -0.9099099099099   -0.5383666320527   -1.1430668239669   -1.1430668830871
 -0.9009009009009   -0.6136196790159   -1.1218407537931   -1.1218408346176
 -0.8828828828828   -0.5257811983413   -1.0819663183234   -1.0819662809371
<snip>
  0.9009009009009    0.6136196790159    1.1218407537931    1.1218408346176
  0.9099099099099    0.5383666320527    1.1430668239669    1.1430668830871

Values failed: 106
Values tested: 223

Percent goodness: 52.46636771300448%
```

We have found that good undergraduates can be productive in this environment within a few days.

3. SPU Instruction Set Architecture

For a list of instructions and discussion of their use, readers should consult the on-line documentation made available by IBM, most of which is also included in the SDK [1, 3]. We would like to draw attention to the unusual features of the SPU ISA on which register-level lookup depends.

Perhaps we should explain why conventional lookup techniques will not work and what automatic vectorization would produce. Any piecewise-continuous function can be approximated arbitrarily by linear segments, if the number of segments is not limited. For fixed machine precision, the linear segments can be chosen to produce correctly rounded results. As switching speed has outpaced bus transmission, large lookup tables have become relatively more expensive compared to polynomial approximations with smaller numbers of segments. Processing multiple inputs in parallel, SIMD complicates this picture. C code of the following form

```
   index = (int) (ceil(x));
   result = a0[index] + x * (a1[index] + x * (a2[index] + x * a[3]);
```

does not have an instruction-for-instruction SIMD translation in general, because four inputs will produce four different indices, which will produce four different addresses. So the loads still require one instruction per input, and even worse, on the SPU addresses only come from the first word, so the inputs must be shuffled before the load, and the loaded values must be merged, requiring several more instructions.

Fortunately, this is not the end of the story for lookups on the SPU. Similar to the `vperm` instructions in VMX/Altivec, the SPU has an instruction

```
   v = shufb a0to15 a16to31 byteIndices
```

in which `v` is formed from the bytes $[a_{b_0}, a_{b_1}, ..., a_{b_{15}}]$, where b_i is the ith byte of `byteIndices` and a_j is the jth byte of the concatenation of `a0to15` and `a16to31`. Depending on the contents of `byteIndices`, this instruction can be used to do 16 byte lookups in parallel, from a list of 32 bytes, 8 halfword lookups from 16, 4 word lookups from 8, 2 doubleword lookups from 4 or 1 quadword lookup from a list of 2 quadwords. The last is just a select statement which chooses the first or second register value. Since the indices are in a register, they can be constructed at compile time or at run time. In contrast to the VMX implementation, `shufb` also inserts three special byte values when the corresponding index byte is one of three special values having the high bit set. This is hard to use with run-time generate lookup maps, but is easy to use for compile-time lookup. We have used it to generate 9-way lookups, where one of the entries in the lookup is zero.

4. Register-level Lookups

To make register-level lookups practical, we found that some language support is required. Although it would be possible for a conventional compiler to look for patterns of conditionals and construct branch-free implementations using lookups, this is beyond the scope of our project, and for function approximation applications it makes more sense to build lookup into the symbolic computation of the spline approximation. For these reasons, we have chosen to implement different lookup constructors, as required. Parallel lookup of four words from tables of eight words (two registers) is the most common, with 10 variants being potentially useful, where each variant depends on the positions of the three bits which form the index into the table of eight. We have only used the variants with three contiguous bits in the high or low order bits, which correspond to words arranged contiguously and bytes of words maximally interleaved, respectively. For reasons of space, we will not present such lookups here, since they occur in a more exotic form in the arcsine example to follow. We will present complete details for log-linear lookups in length 16 tables. To be optimally efficient, such lookups must be done 8 at a time, so for operations on word values, two registers should be processed at a time.

4.1 Lookup in 16-word table

Make lists of floating-point numbers into registers for 16-way lookups. The function `eightX2Table` maps this over a list of such lists, as occur when looking up coefficients for a list of 16 polynomial segments.

The lookup of 8 values requires 4 `shufb`s, 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.

```
eightX2Pair :: SPUType val inject => [Double] -> ((val,val),(val,val))
eightX2Pair fs = if length fs == 16 then ((high0,high1),(low0,low1))
                                    else error "MathUtils.eightX2Pair"
  where
```

Convert Haskell double-precision coefficients into register values and divide them into lists of high- and low-order halfwords.

```
    (high,low) = unInterleave $ concat $ map (shorts.(`asTypeOf` zero).unfloats) fsIn4s
    fsIn4s = in4s fs
```

where zero is a constant needed here to force compile time evaluation of the lookup table, and `in4s` puts the list of doubles into lists of four. Now put the high and low shorts together

```
    (highS0,highS1) = splitAt 8 high
    (lowS0,lowS1) = splitAt 8 low
    [high0,high1,low0,low1] = map unshorts [highS0,highS1,lowS0,lowS1]
```

Looking up values in tables as constructed above 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, involves four instructions

```
eX2l :: SPUType val inject => ((val,val),(val,val)) -> val -> (val,val)
eX2l ((h0,h1),(l0,l1)) key = (v1,v2)
  where
    [v1,v2] = [shufb high low fstInterleave
              ,shufb high low sndInterleave
              ]
    high = shufb h0 h1 key
    low  = shufb l0 l1 key
```

and requires two auxilliary lookup constants:

```
    fstInterleave = unbytes [ 0, 1, 16,17,  2, 3, 18,19
                            , 4, 5, 20,21,  6, 7, 22,23
                            ]
    sndInterleave = unbytes [ 8, 9, 24,25, 10,11, 26,27
                            ,12,13, 28,29, 14,15, 30,31
                            ]
```

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

4.2 Mixed log/linear intervals

We can do lookups based on intervals which mix logarithmic and linear scales by using some bits from the floating-point exponent and some from the mantissa. This is an efficient way to construct lookup keys corresponding to a contiguous set of intervals, including the case of varying interval sizes. Varying the size of the intervals can be used to reduce the complexity of the approximation, for functions with singularities or zero crossings.

To simplify lookups for the user, we encapsulate the defining properties of the particular lookup in a data structure `LookupSpec`, which contains the numbers of bits from the mantissa and the exponent to be used in the lookup, thereby defining the relative sizes of the intervals. For example, with one exponent bit and three mantissa bits, eight intervals of size a will be followed by eight of size $2a$. For added flexibility, we allow the user to specify a number of interval sizes to skip, so in the above, skipping one interval would result in seven size a, eight size $2a$ and one size $4a$ intervals.

```
calcBreaks :: Int -> Int -> Int -> Double -> LookupSpec
calcBreaks mant exp skip endPoint
      = if totalBits /= 4 || skip >= 2^mant || skip < 0 || mant < 0 || exp < 0
        then error "calcBreaks only implemented for 16-way lookup"
        else LookupSpec mant exp skip totalBits endPoint b'
  where
    totalBits = mant + exp
    b = [sum $ take n $ drop skip $ concat [replicate (2^mant) $ 2**(fromIntegral i-3)
                                            |i<-[0..]]
        | n<-[0..16]]  :: [Double]
```

to save a conversion at run-time, we allow the user to specify the total width of all the intervals, which is used to scale the break points between intervals.

```
    b' = map (/scale) b
    scale = (b !! 16)/endPoint
```

The calculated break points are then used both to construct the approximations (using Maple in our case), and to generate the code to construct the lookup key at run-time:

```
coeffs :: SPUType val inject => LookupSpec -> [[Double]] -> (val,val) -> [(val,val)]
coeffs spec rawCoeffs (v1,v2) =  eightX2Lookup (eightX2Table $ rotLists rawCoeffs) lookup
  where
```

to get the exponent bits to end with 4 zero bits for the beginning of the first interval, we position the beginning of the interval at 2. The first $2^{\mathrm{mant}} - \mathrm{skip}$ break points are in the first exponent level $[2, 4)$, so we need to start at $2 + \mathrm{skip}\, 2^{1-\mathrm{mant}}$. The first sup-interval divides into $[2, 4]$ 2^{mant} times, so the scale is calculable as $2^{1-\mathrm{mant}}/(b_1 - b_0)$

```
    offset = unfloats4 $ 2 + (fromIntegral $ skip spec) * 2**(fromIntegral $ 1-(mant spec))
    scale  = unfloats4 $ 2**(1 - (fromIntegral $ mant spec))
                         / (((breaks spec) !! 1) - ((breaks spec) !! 0))
```

the bits we want for the lookup are $(8-\mathrm{exp})...(9+\mathrm{mant})$, which crosses a byte boundary if $\mathrm{mant} \neq 0$, so we need to use a rotate

```
    rightBits v = join [reven,rodd]
      where
        rodd = rotqbii vso bitShift
        reven = roti vso bitShift
        vso = (fma v scale offset)
        bitShift = (fromIntegral $ (mant spec)+2)
```

put the bits from word 0 in bytes 0 and 1, etc., this is why we need to unroll

```
    shuffled = shufb (rightBits v1)
                     (rightBits v2)
                     (unbytes [0,0, 4,4, 8,8, 12,12, 16,16, 20,20, 24,24, 28,28])
```

which is where we would waste an instruction by doing only one lookup in parallel. Now merge these bits to get lookups from two positive inputs

```
    lookup = selb (unshorts8 0x0001) shuffled (unbytes16 $ 32-2)
```

To provide a nicer interface, we generate the register constants for the tables here.

```
    rotLists = map ((\(x,y)->y++x).(splitAt (2^(totalBits spec) - (skip spec))))
```

5. Examples

5.1 Arcsine

Arcsine is an odd function defined on the interval $[-1, 1]$. To improve rounding and reduce the intervals on which polynomial approximations must be defined, we evaluate it on the absolute value and copy the sign to the result.

```
arcsin x = selb yPositive x signBit
  where
    xPositive = andc x signBit
```

Because sine has zero slope at 1, arcsine has infinite slope, and has a singularity of type $\pi/2 - \sqrt{x-1}$, as seen in Fig. 1. Since arcsine has a zero crossing at zero with slope one, the polynomial approximation at zero must have zero constant term and one first-order coefficient. This forces at least two different types of approximations. We will use

$$p_0 = x\left(1 + \left(-3.8452 \cdot 10^{-7} + (0.1666 + (-0.0004 + 0.0779x)\,x)\,x\right)x\right) \tag{1}$$

$$p_1 = -2.5254 \cdot 10^{-6} + (1 + (-0.0010 + (0.1739 + (-0.0255 + 0.1147x)\,x)\,x)\,x)\,x \tag{2}$$

$$p_2 = -1.4452 \cdot 10^{-4} + (1.0026 + (-0.0203 + (0.2458 + (-0.1614 + 0.2186x)\,x)\,x)\,x)\,x \tag{3}$$

$$p_3 = -0.0029 + (1.0381 + (-0.2005 + (0.7060 + (-0.7520 + 0.5235x)\,x)\,x)\,x)\,x \tag{4}$$

on segments $[0, 1/8)$, $[1/8, 2/8)$, $[2/8, 3/8)$, $[3/8, 4/8)$, and

$$p_4 = -1/2\pi + \sqrt{2.4667 + (-3.1340 + (0.9628 + (-0.4209 + (0.1565 - 0.0312x)\,x)\,x)\,x)\,x} \tag{5}$$

$$p_5 = -1/2\pi + \sqrt{2.4658 + (-3.1271 + (0.9406 + (-0.3854 + (0.1279 - 0.0220x)\,x)\,x)\,x)\,x} \tag{6}$$

$$p_6 = -1/2\pi + \sqrt{2.4644 + (-3.1175 + (0.9152 + (-0.3513 + (0.1052 - 0.0159x)\,x)\,x)\,x)\,x} \tag{7}$$

$$p_7 = -1/2\,\pi + \sqrt{(-2.4621 + (0.6422 + (-0.2429 + (0.0742 - 0.0114\,x)\,x)\,x)\,x)\,(x - 1.0)} \tag{8}$$

on $[4, 5/8)$, $[5/8, 6/8)$, $[6/8, 7/8)$, $[7/8, 8/8)$, using the Maple code

```
i:=0;
axDivx:=numapprox[minimax](x->limit(arcsin(y)/(y),y=x),i/8..(i+1)/8,
                           [polyOrd-1,0],1,'d2[0]');
ax[0]:=x->x*axDivx(x);
for i from 1 to 3 do
  ax[i]:=numapprox[minimax](x->((arcsin(x))),i/8..(i+1)/8,
                            [polyOrd,0],x->x,'d2[i]');
od;
```

and

```
for i from 4 to 6 do
  asqrt[i]:=numapprox[minimax](x->((arcsin(x)-evalf(Pi/2)))^2,i/8..(i+1)/8,
                               [polyOrd,0],1,'d3[i]');
od;
i:=7;
asqrtDivx1:=numapprox[minimax](x->limit(((arcsin(y)-Pi/2))^2/(y-1),y=x),i/8..(i+1)/8,
                               [polyOrd-1,0],1,'d3[i]');
asqrt[7]:=x->asqrtDivx1(x)*(x-1);
```

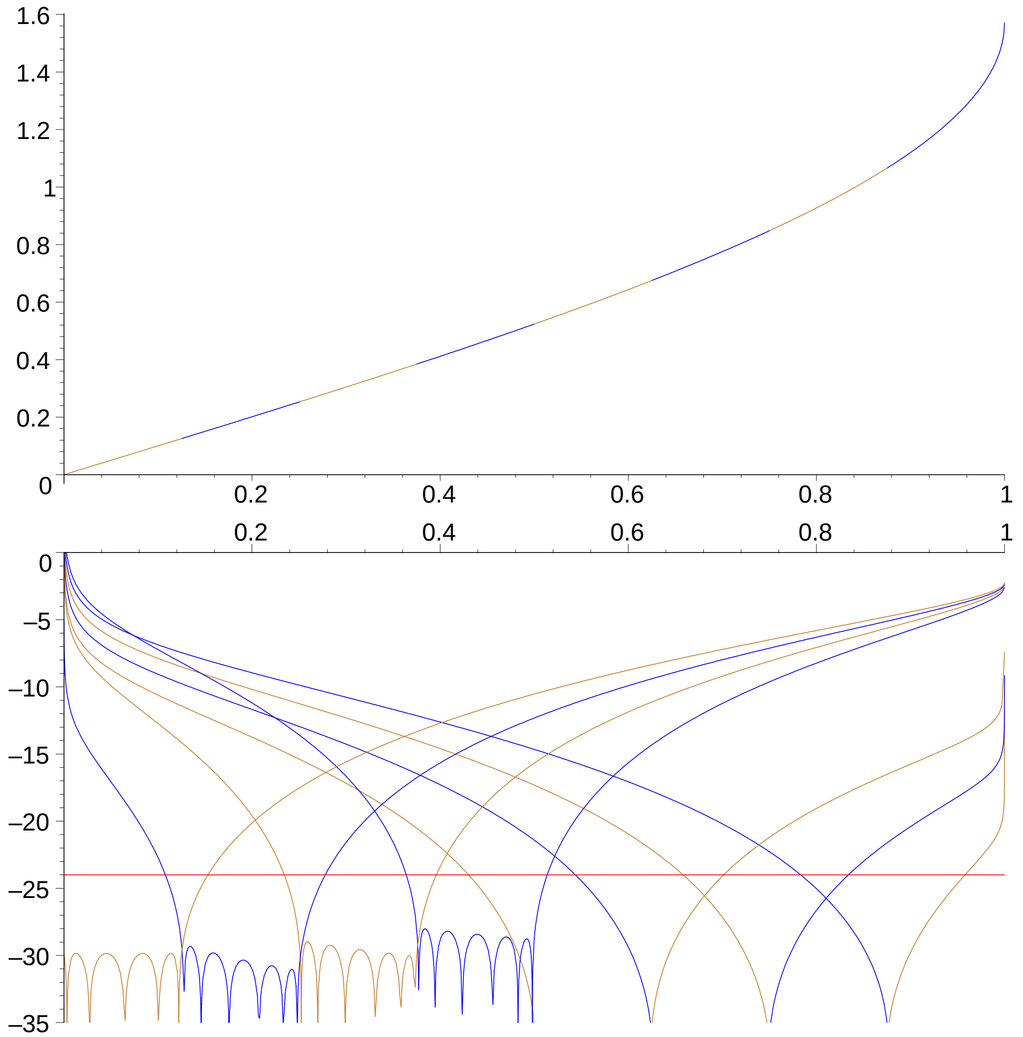

Figure 1: Eight polynomial segments of two types, *above*, and the corresponding errors in bits, *below*.

Since the intervals are all of length 1/8, the lookup can be generated from the three significant bits of the mantissa of a floating point number, or from three bits of an integer after conversion t fixed point. The only tricky point is insuring that 1 is in the last interval, and not the first. To achieve this we multiply by one ulp less than a quarter and add one to put the three bits which select for the interval in bits 3-5 of byte 1 of the appropriate word.

```
look1 = fma (unwrds4 0x03e7fffff) xPositive (unfloats4 1)
```

These bytes are then replicated to all the bytes in the word, and the words are merged with the appropriate low-order bits.

```
look2 = shufb look1  look1
        $ unbytes [1,1,1,1, 5,5,5,5, 9,9,9,9, 13,13,13,13]
look3 = selb (unwrds4 0x00010203) look2  (unwrds4 0x1c1c1c1c)
```

To improve the accuracy of the polynomial approximations, we found it necessary to evaluate the polynomials

```
poly = hornerV (contigLookup arcsinTable look3) xCentred
```

using interval-centered coordinates

```
xCentred = fs xPositive offset
```

where it is very important that the same lookup key is used to look up the offset to centre the input, otherwise boundary cases could produce arbitrary errors.

```
[offset] = contigLookup (contigTable offsets) look3
```

For the first intervals we now have the final answer, but for the second intervals we need to apply the square root and subtract from $\pi/2$.

```
piOver2sqrtP = fs piOver2 (sqrtSPU poly)
```

The appropriate final result is chosen with a select mask

```
yPositive = selb poly piOver2sqrtP switch
```

which must again be looked up using the same key, to prevent problems with edge cases.

```
[switch] = contigLookup (contigTableWord switches) look3
```

To ensure synchronization, the following constants are printed from Maple to Haskell:

```
switches = [[0, 0, 0, 0, -1, -1, -1, -1]]
offsets = [[0, 0.1875, 0.3125, 0.4375, 0.5625, 0.6875, 0.8125, 1]]
```

5.2 Hyperbolic Tangent

Hyperbolic tangent is defined by

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \tag{9}$$

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, $\operatorname{arctanh}(1 - 2^{-24}) = 8.6643397420981601947$, and can be approximated by polynomials in the range $[-8.664339, 8.664339]$. So any number larger than this in magnitude should round to ± 1.

The function is unrolled to process two inputs at once because the 16-way lookup can be better amortized this way.

For the final step, we put the sign back:

```
ftanh (v1,v2) = (selb result1OrOne v1 signBit, selb result2OrOne v2 signBit)
where
```

take the positive part

```
        v1Positive = andc v1 signBit
        v2Positive = andc v2 signBit
```

compare to $\operatorname{arctanh}(1 - 2^{-24})$ because this is the smallest number which rounds to 1, all higher numbers round to 1, and form a select mask

```
        isBig1 = fcmgt v1Positive (unfloats4 $ 8.6643397420981601947)
        isBig2 = fcmgt v2Positive (unfloats4 $ 8.6643397420981601947)
```

which is applied to the final result:

```
        result1OrOne = selb result1 (unfloats4 1) isBig1
        result2OrOne = selb result2 (unfloats4 1) isBig2
```

do parallel lookup for two vector inputs (8 floats) of polynomial coefficients, generated by Maple:

```
        cs = (coeffs tanhLkup tanhC (v1Positive,v2Positive))
```

evaluate polynomials using Horner's rule:

```
        result1 = hornerV (map fst cs) v1Positive
        result2 = hornerV (map snd cs) v2Positive
```

where the instructions to form the lookup key and the break points between intervals are caluculated using the utility function

```
tanhLkup = calcBreaks 2 2 3 8.6644
```

These break points are then copied into the Maple code to compute and package the coefficients for the polynomials:

```
i:=0;ax:=numapprox[minimax](x->limit((tanh(y)/y-1)/y,y=x),(breaks[i+1])..(breaks[i+2])
                          ,[polyOrd-2,0],x->x,'da[i]');
aa[0]:=x->x*(1+x*ax(x));
for i from 1 to 15 do
  aa[i]:=numapprox[minimax](x->tanh(x),(breaks[i+1])..(breaks[i+2]),[polyOrd,0],x->x,'da[i]');
od;
```

5.3 Hyperbolic Sine

Hyperbolic sine is defined by

$$\sinh x = \frac{e^x - e^{-x}}{2}. \tag{10}$$

It is difficult to approximate by polynomials over large ranges, because it grows exponentially. Therfore, for large values we use (10), but for small values of x, such that e^x and e^{-x} are close in value:

(i) precision loss grows as n where $x = 2^{-n}$, because of similarity, and

(ii) errors in e^x have the opposite sign as errors in e^{-x}, so they reinforce each other.

The only way to get accurate results near 0 is to use polynomial approximations. The Taylor series for sinh at 0 is odd, near 0. Odd polynomials with the same number of coefficients are better at approximating sinh than general polynomials. In branchless implementations, the easiest way to use different methods to calculate sinh differently on two intervals is to calculate both approximations, and use `selb` to select the correct result. Since both methods involve polynomials approximations, over subintervals, we can try to share the execution of polynomial execution and/or coefficient lookup. Calculating a key to look up in multiple intervals efficiently depends on the structure of the problem. This is easier to do with the hyperbolic sine intervals near zero, because the exponential intervals repeat periodically.

The following Maple code calculates minimax polynomials for $\sinh(\sqrt{x})/\sqrt{x}$, and uses them to construct odd approximating polynomials for $\sinh(x)$ over the four intervals $[0, 1)$, $[1, \sqrt{2})$, $[\sqrt{2}, \sqrt{3})$, and $[\sqrt{3}, 2)$.

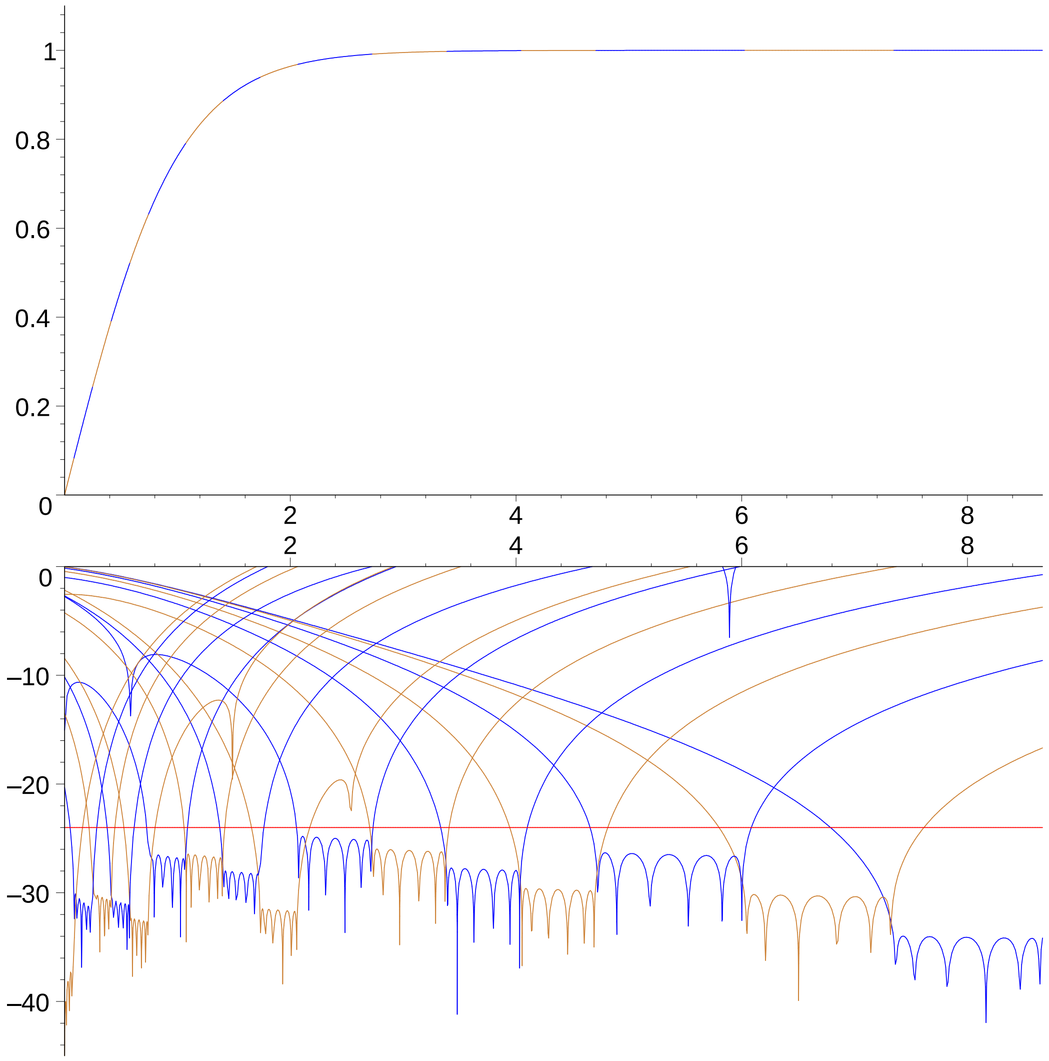

Figure 2: Sixteen approximating polynomial segments, *above*, and the corresponding error in bits, *below*.

```
polyOrd:=4;
i:=0;
ax:=numapprox[minimax](x->limit((sinh(y)/y-1)/y^2,y=sqrt(x)),0..1,[polyOrd-2,0],x->x,'d2[i]');
aa[i]:=x*(1+x*ax(x));plt[i]:=x*(1+x^2*ax(x^2));
for i from 1 to 3 do
  ax:=numapprox[minimax](x->limit(((sinh(y))/y),y=sqrt(x)),i..i+1,[polyOrd-1,0],x->x,'d2[i]');
  aa[i]:=x*(ax(x));plt[i]:=x*(ax(x^2));
od;
```

Note that the first coefficient of the first polynomial is forced to be 1. This is necessary to ensure relative accuracy for numbers near zero, which comprise half the representable floating point numbers. These intervals are chosen for three reasons:

1. the minimum ratio between the positive and negative terms in (10) is $\log_2(e^2/e^{-2}) = 5.77$, so approximation errors in the second term must be greater than 32 ulps before they can effect the result;
2. the minimax polynomials are calculated using $\sqrt{x}$, so accuracy of the approximations could be expected to depend on the width of the intervals measured in this coordinate;
3. a single computation can produce both two bits to choose the interval and one bit to select between the two calculation types.

The resulting polynomials

$$p_0 = x\left(1 + x^2\left(0.166667 + \left(0.008330 + 0.000203x^2\right)x^2\right)\right) \tag{11}$$
$$p_1 = x\left(0.999986 + \left(0.166705 + \left(0.008293 + 0.000215x^2\right)x^2\right)x^2\right) \tag{12}$$
$$p_2 = x\left(0.999884 + \left(0.166853 + \left(0.008221 + 0.000227x^2\right)x^2\right)x^2\right) \tag{13}$$
$$p_3 = x\left(0.999535 + \left(0.167197 + \left(0.008107 + 0.000240x^2\right)x^2\right)x^2\right) \tag{14}$$

all have better than 23 bits of relative accuracy, as shown in Figure 3.

We calculate $e^x = 2^{x/\log 2}$, by separating the integral part of $x/\log 2$ and placing it in the exponent bit field, using the two most significant fractional bits to look up an interval and the remaining bits to evaluate a polynomial approximation. The following Maple code calculates minimax polynomials for the final step:

```
for i from 0 to 3 do
  axx:=numapprox[minimax](x->evalf((2^(x+i/4))),1..1+(1)/4,[polyOrd,0],1,'d2[i+4]');
  aa[i+4]:=axx(x);plt[i+4]:=axx(x);
od;
```

$$p_4 = 1.00456 + (0.673227 + (0.274550 + (0.026672 + 0.020986x)\,x)\,x)\,x \tag{15}$$
$$p_5 = 1.194632 + (0.800607 + (0.326497 + (0.031719 + 0.024957x)\,x)\,x)\,x \tag{16}$$
$$p_6 = 1.42066 + (0.952087 + (0.388272 + (0.037721 + 0.029679x)\,x)\,x)\,x \tag{17}$$
$$p_7 = 1.68946 + (1.13222 + (0.461736 + (0.044858 + 0.035294x)\,x)\,x)\,x. \tag{18}$$

This time using arbitrary polynomials. In infinite precision, the four polynomials are similar, as seen in Figure 3, but in finite precision they may have different rounding. This is something we want to take advantage of in a later stage to improve accuracy.

To share computation for all of these paths, we need to explicitly add 0 to the first four polynomials so they all have the same number of coefficients. We also need to switch the variable in the polynomial evaluation between x and x^2. The switches between one version for the interval $[0, 2)$ and another for the outer interval could be made with `selb`, but we can also use `shufb` instructions in the odd pipeline for better balance.

We now go through the implementation to explain how the keys to select execution paths are generated unconditionally.

```
fsinh v = signedResult
  where
```

Evaluate on the absolute value, so truncation always goes the same way,

```
    vPositive = andc v signBit
```

since sinh the signed result is obtained by copying the sign from the input.

```
    signedResult = selb resultOrMax v signBit
```

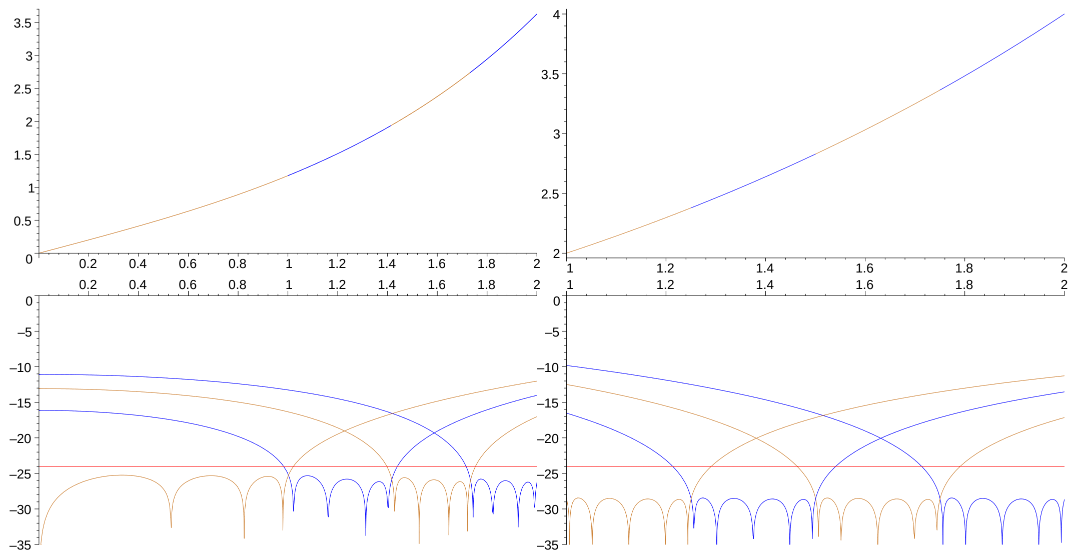

Figure 3: *Left:* approximating sinh on the interval $[0, 2)$ with polynomials, *above*, and the corresponding error in bits, *below*. *Right:* approximating exp on the interval $[1, 2)$ with polynomials, *above*, and the corresponding error in bits, *below*.

Detect inputs which will saturate on the output and generate a mask

```
isReallyBig = fcmgt vPositive (unfloats $ map asinh $ floats $ (maxFloat::SPUSim.Val))
```

which needs to be to saturate results before the sign is copied.

```
resultOrMax = selb posResult maxFloat isReallyBig
```

A single fused multiply-add with suitably chosen constants generates two switches: Bit 3 switches between the two approximations, and bits 10 and 11 select the interval:

v	bitSwitch
0	0x0ff00000
1	0x0ff40000
$\sqrt{2}$	0x0ff7ffff
$\sqrt{2} + \text{ulp}$	0x0ff80000
$\sqrt{3}$	0x0ffbffff
$\sqrt{3} + \text{ulp}$	0x0ffc0000
$2 - \text{ulp}$	0x0fffffff
2	0x10000000
$2^{19} - 2^{-5}$	0x1ffffffe
2^{19}	0x20000000

```
v2 = fm v v
bitswitch = fma v2 (unwrds4 0x0d000000) (unwrds4 0x0ff00000)
```

The first nibble is replicated

```
zeroOneNibble = shufb bitswitch bitswitch
                    $ unbytes $ [0,0,0,0, 4,4,4,4, 8,8,8,8, 12,12,12,12]
```

and merged with the identity byte permutation to produce a **shufb** map which selects between corresponding words in the first and second register arguments according to whether the first or second approximation is used. This is the key step to balancing the odd and even pipelines, since this shuffle map is applied five times (and used once in the construction of the coefficient map). It is possible to construct all of these keys with odd pipeline instructions, rather than even ones, but this would introduce additional latency which would be harder to schedule, so we have chosen not to implement this.

```
sinhOrExp = selb (unbytes [0..15]) zeroOneNibble (unbytes16 16)
```

This map is first needed to select between the two segment bits for the sinh or exp lookups.

```
segBits = shufb bitswitch expSegBits sinhOrExp
```

To construct the coefficient lookup, we duplicate the segment bits into each byte of the respective word

```
segBytes = shufb segBits segBits
               $ unbytes $ [1,1,1,1, 5,5,5,5, 9,9,9,9, 13,13,13,13]
```

and insert the segment bits 10 and 11 into the in-order lookup key.

```
lookupKey = selb sinhOrExp segBytes (unbytes16 0x0c)
```

Calculate $x/log(2)$ for use in $e^x = 2^{x/log(2)}$:

```
vBylog2 = fma vPositive (unfloats4 $ 1/log(2)*(1+0*1*2**(-24))) (unfloats4 $ -2)
```

Convert the floating point number to an integer so it can be used as the exponent part of the floating point number. Multiply by 2^23 as part of the conversion so the integer part is put in the exponent bits (1-8), and the fractional part is put in the mantissa.

```
vBylog2AsInt = cflts vBylog2 23
```

Mask out the fractional bits and add 127×2^23 to get a correctly-biased exponent.

```
exponentBits = andc vBylog2AsInt (unwrds4 0x007fffff)
expPart = a exponentBits (unwrds4 0x3f800000)
```

Merge the lower 21 fractional bits from the integer representation of $|v|/\log 2$ with the exponent for 1. The result is a number in the interval $[1, 1 + 1/4)$.

```
frac = selb (unfloats [1,1,1,1]) vBylog2AsInt (unwrds4 0x001fffff)
```

The remaining two bits are used to construct a lookup key. This allows us to evaluate 2^x on the four subintervals $\cup\{[1, 1 + 1/4), [1 + 1/4, 1 + 1/2), [1 + 1/2, 1 + 3/4), [1 + 3/4, 2)\} = [1, 2)$.

```
expSegBits = rotqbyi (join [rotqbii vBylog2AsInt (-3), roti vBylog2AsInt (-3)]) 15
```

Use utility function to look up list of coefficients using **shufb** instructions. The first coefficient has to be treated separately, because there are really two types of polynomials being evaluated.

```
(c0:coeffs) = contigLookup (contigTable sinhAndExpC) lookupKey
```

For the fixed segments in $[0, 2)$, (11)-(14), the inner part of the polynomial is evaluated at x^2, while the outer linear factor is evaluated at x. For the four segments used to evaluate the remainder for the exponential computation, (15)-(18), both parts are evaluated on the fractional part in $[1, 2)$. We can use permutation maps to select between the two,

```
v2OrFrac = shufb v2 frac sinhOrExp
vOrFrac = shufb vPositive frac sinhOrExp
```

and evaluate the appropriate types of polynomials in parallel for each of the four inputs.

```
evalPoly = fma (hornerV coeffs v2OrFrac) vOrFrac roundingOrC0
```

Since the constant coefficient is zero for the first four segments, we can substitute a rounding term. The simplest rounding term, which is exactly correct near zero, is to add half an ulp of x.

```
roundingOrC0 = shufb (fm vPositive $ unfloats4 $ 1*2**(-24)) c0 sinhOrExp
```

For the outer intervals, to save an operation, we calculate

$$\sinh x = \left(\frac{e^x}{2}\right) - \frac{1}{4}\left(\frac{e^x}{2}\right)^{-1}. \tag{19}$$

```
halfExpV = fma expPart evalPoly oneUlp
invExp2v = frecip halfExpV
sinhFromExp = fnms (unfloats4 $ 1/4*(1+2**(-23))) invExp2v halfExpV
```

where, to get better rounding, we add an estimated ulp during the multiply-add which combines the exponentiations of the integral and fractional parts of x.

```
oneUlp = fm expPart (unfloats4 $ 2**(-21))
```

Finally, we select either the straight polynomial evaluation or the substitution of the polynomial into the

```
posResult = shufb evalPoly sinhFromExp sinhOrExp
```

6. Performance

SIMD implementations of many of the functions we implemented were released as part of the Sony-Toshiba-IBM SDK 1.1, [1]. Since these functions were written using predicated execution via selection and masking, but not using the permutation, they form a good basis for comparison in evaluating the effectiveness of permutations.

Overall, the use of lookups provides significant acceleration for math functions. In Figure 4, we show the speedup in the form of ratios of measured execution times, with both sets of functions in-lined into loops applying the function to 1000 single-precision floats. The resulting functions were instrumented using SPU hardware counters, and compiled with a pre-release version of XLC. We have not included short functions whose structure is determined by special instructions for reciprocal and square root estimates. We have included two pre-release functions for hyperbolic sine and tangent. The overall speedup from these functions is 89 percent, but there are significant differences in the amount of acceleration. Removing cube root, which shows the largest acceleration, and which is less accurate than the SDK function, the improvement is 38 percent. The logarithms are all slower using lookups, but these functions are also significantly more accurate. Using permutations introduces additional execution paths, with multiple polynomials evaluated over many smaller intervals. This multiplies the locations where truncating arithmetic in multiple instructions can cause significant rounding errors. We plan to develop semi-random searches through the space of perturbed constants (mostly rounded polynomial coefficients) to address this issue when we have hardware on site.

We have prepared another figure which may be of assistance to potential CELL adopters, and possibly CELL designers. On the left hand side of Figure 5, we show for each function we have implemented, the number of cycles to compute an element. The top (blue) bar shows cycles on Power 5/linux/MASS from [4]. The middle (orange) bar shows the results with our library, and the bottom (green) are the SDK 1.1 (plus pre-release `sinh` and `tanh`). This comparison suggests that SIMD is a better investment than multiple floating point units and out-of-order execution capabilities for this application.

On the right, we break down the instruction counts into odd and even pipelines (bottom and top open bars, respectively), which shows that even the most aggressive use of permutation for predicated execution and lookup is far from balancing the pipelines for library of functions. Each of the open bars is divided into closed bars corresponding to a hypothetical future refinement of the parallel execution capabilities of the SPU, to include not two but four pipelines, with the current even pipeline further divided into floating point and integer multiplication (orange) and simple integer and logical instructions (green), and the odd pipeline is divided into hypothetical permute/rotate (blue) and I/O+ units (red). On this code library, dividing the even (arithmetic) unit improves

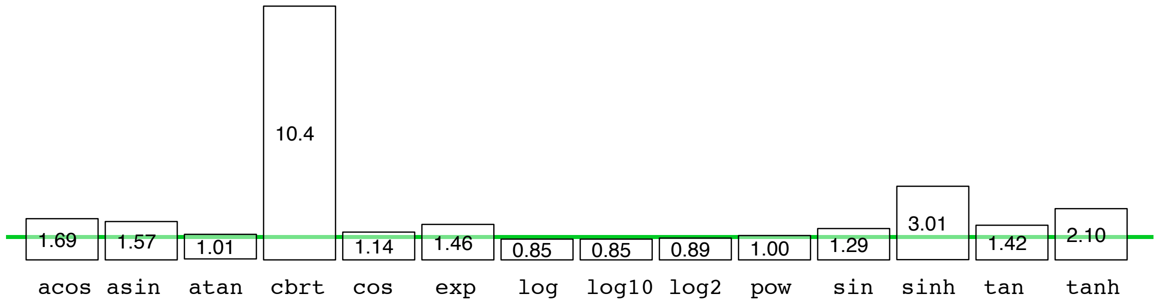

Figure 4: This table gives the ratio of execution times between lookup-based and conventional implementations. We have eliminated short functions dependent on special hardware support (division, square root,...). Average improvements are 95 percent including cube root, and 42 percent excluding it.

the possibilities for parallel execution significantly, but dividing the odd pipeline provides little additional benefit. Whether a scheduler could effectively use the increased parallelism, and whether the added complexity involved in another pipeline is worth the improved performance, are interesting questions.

7. Conclusion

We have quantified the performance improvements which can be expected by incorporating register-level lookups in mathematical function evaluation, showing that the increased instruction-level parallelism afforded by such patterns is important. We have demonstrated how such patterns can be supported by programming languages/compilers, using our development environment. And we have given detailed descriptions of three examples demonstrating the full range of the technique. In this way, we hope to inspire compiler-writers to incorporate support for register-level lookups, and provide a level of detail enabling anyone needing to implement efficient function evaluation on the SPU to do so.

Acknowledgment

The authors would like to thank Robert Enenkel and everyone at IBM who provided access to CELL hardware for testing, and Wolfgang Thaller, William Hua and Wolfram Kahl for their work on the Coconut development environment. This work was supported by NSERC, and IBM through a CAS Fellowship.

References

[1] `http://www.alphaworks.ibm.com/tech/cellsw/download`

[2] C. K. Anand, W. Kahl, W. Thaller, Explicitly Staged Software Pipelining, preprint, 2006. `http://www.cas.mcmaster.ca/ anand/papers/AnandKahlThaller2006.pdf`

[3] *Synergistic Processor Unit Instruction Set Architecture*, ver 1.1, Sony-Toshiba-IBM, Hopewell Junction, NY, January 30, 2006.

[4] `http://www-1.ibm.com/support/docview.wss?rs=2021&uid=swg27007063`

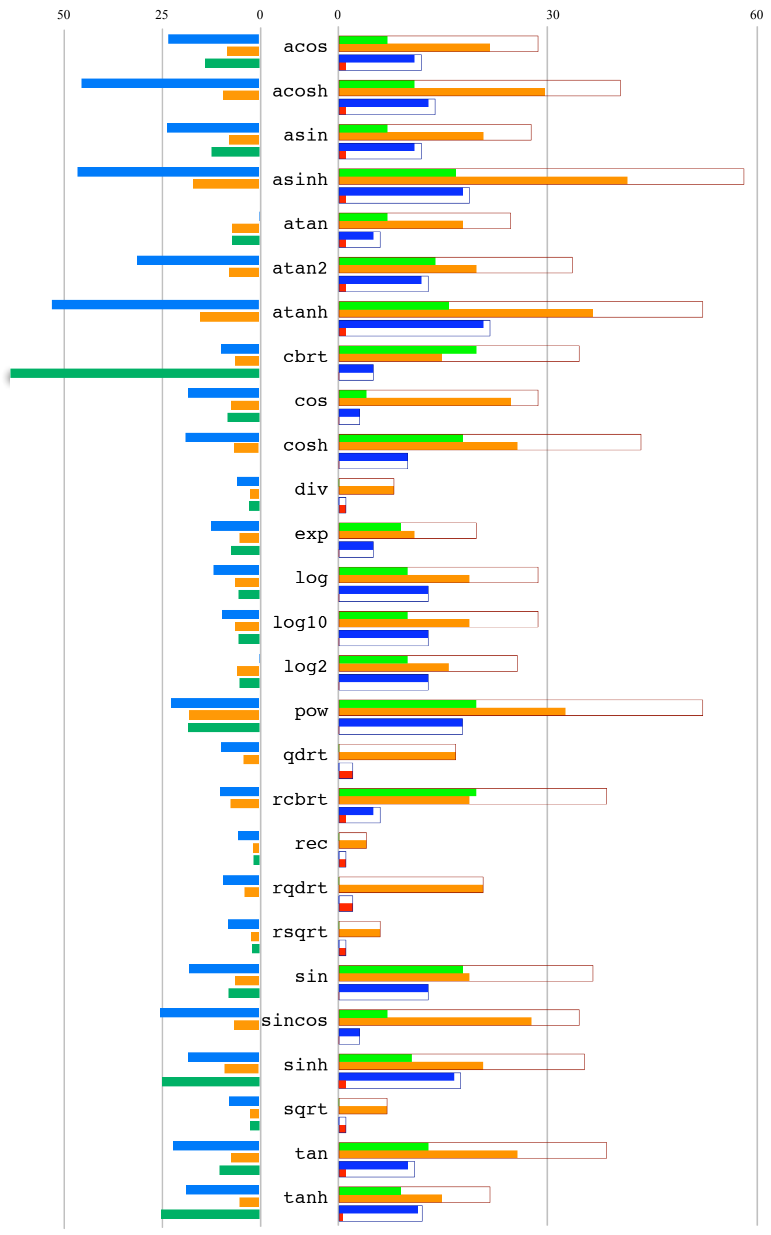

Figure 5: *Left:* measured cycle times per float when processing an array of floats, for scalar code (Power 5), SDK 1.1 SPU functions and our SPU functions. *Right:* instruction counts for even/odd pipelines (top/bottom open bars), further broken into four possible pipelines.