

Vectorization of Multibyte Floating Point Data Formats

Andrew Anderson
Lero, Trinity College Dublin
aanderso@cs.tcd.ie

David Gregg
Lero, Trinity College Dublin
dgregg@cs.tcd.ie

ABSTRACT

We propose a scheme for reduced-precision representation of floating point data on a continuum between IEEE-754 floating point types. Our scheme enables the use of lower precision formats for a reduction in storage space requirements and data transfer volume. We describe how our scheme can be accelerated using existing hardware vector units on a general-purpose processor (GPP). Exploiting native vector hardware allows us to support reduced precision floating point with low overhead. We demonstrate that supporting reduced precision in the compiler as opposed to using a library approach can yield a low overhead solution for GPPs.

CCS Concepts

•Computer systems organization → Single instruction, multiple data; •Software and its engineering → Software performance; Data types and structures; •Theory of computation → Vector / streaming algorithms;

Keywords

Approximate Computing; Floating Point; Multiple Precision; SIMD; Vector Architecture

1. MOTIVATION

It has long been recognized that different applications and algorithms need different amounts of floating point precision to achieve accurate results [3, 13, 16]. For example, 64-bit double precision floating point is often needed for scientific applications, whereas 32-bit single precision is sufficient for most graphics and computer vision applications.

Modern GPUs and embedded processors increasingly support 16-bit floating point for applications that are highly tolerant of approximation such as speech recognition [6].

In some contexts it is possible to customize the level of precision precisely to the application. For example, field-programmable gate arrays (FPGAs) can be used to imple-

ment customized floating point with bit-level control over the size of the mantissa and exponent [5, 16].

More recently, Schkufza et al. [14] have shown how super-optimization can be used to generate iterative floating point algorithms that guarantee a given level of accuracy. For example, an exponential function may return a 64-bit floating point value in which at least the first, say, 40 bits of the mantissa are guaranteed to be accurate.

A problem with customizing floating point precision on general-purpose processors (GPPs) is that most support only two standard types: IEEE-754 single precision (or `binary32`) and double precision (`binary64`). If an operation needs more precision than `binary32` and less than `binary64`, the developer has no choice but to use `binary64`.

In this paper we propose a compiler-based mechanism for supporting several non-standard *multibyte* floating point memory formats, such as 24-, 40-, or 48-bit floating point. By *multibyte* we mean that these formats differ in length from standard IEEE-754 formats by multiples of 8 bits.

By using these types, a developer can reduce the precision of floating point data in memory, resulting in reduced storage requirements for their data.

An increasingly important factor in the design of computing systems is energy consumption. In embedded computing systems the energy consumed by data movement can be much greater than the energy used to perform arithmetic [4].

According to Gustafson [7], a typical 64-bit multiply-add operation costs around 200pJ, while reading 64 bits from cache costs 800pJ and a 64-bit main memory read costs 12000pJ. Reducing the amount of data movement is therefore essential to reducing energy consumed by an application, and in particular, reducing the number of costly last-level cache misses.

Customizing precision of floating point data to the needs of the application is one way to reduce data movement. It is straightforward to implement a C/C++ data type representing, for example, 24-bit floating point values (Figure 1).

However, we have found that the performance of such code can be extraordinarily poor when operating on arrays of data. We therefore propose a technique to generate vectorized code that performs a number of adjacent reads or writes together, along with the required packing or unpacking.

Contributions

- We present a practical representation of multibyte floating point values that can easily be converted to and from the next largest natively-supported type.

- We propose a compiler vectorization scheme for packing and unpacking our floating point types, enabling their use in vectorized floating point computations.
- We demonstrate experimentally that our techniques provide a low-overhead way to support customized floating point types on general-purpose processors.

2. CUSTOMIZING FLOATING POINT

The IEEE-754 2008 standard [17] defines a number of finite binary representations of real numbers at different resolutions (16, 32, and 64 bits, among others). Each format encodes floating-point values using three binary integers: *sign*, *exponent*, and *mantissa*, which specify a point on the real number line following a general formula:

$$v = (-1)^s \times (1 + \sum_{i=1}^M (m_i 2^{-i})) \times 2^{e-bias}$$

v is the real number obtained by the evaluation of the formula for a floating-point value with sign s , mantissa m of length M bits, and exponent e . The value $bias$ is an integer constant which differs for each format. Different formats (**binary32**, **binary64**, etc.) use different numbers of bits for the exponent and mantissa components.

The exponent determines a power of two by which the rest of the number is multiplied, while the mantissa represents a fractional quantity in the interval $[1, 2)$ obtained by summing successively smaller powers of two, starting from 2^{-1} and continuing up to the length of the mantissa. If the i th mantissa bit is set, the corresponding power of two is present in the sum. For normalized numbers (those with a nonzero exponent), the leading 1 is not explicitly stored in the mantissa, but is implied.

The structure of the IEEE-754 binary encoding means that a change in an exponent bit strongly influences the resulting value, while a change in a mantissa bit has less influence. Furthermore, a change in any bit of the mantissa has exponentially greater effect on the resulting value than a change in the next-least-significant bit.

These observations lead naturally to a scheme for representing values at precisions between those specified by the IEEE-754 standard: varying the number of low-order mantissa bits. Previous proposals based around this concept have typically made use of either customized floating point hardware support via FPGA [5] or high-level data reorganization in massively parallel systems [9].

In this paper, we demonstrate how reduced precision floating point representations can be used on general-purpose processors without requiring any special hardware support.

The rest of our paper is organized as follows: first, in Section 3, we discuss our scheme for representing floating point numbers with reduced precision in memory, and a straightforward implementation of the scheme in scalar code using a library of datatypes implementing different memory representations of floating point numbers with different precision.

Next, in Section 4, we discuss aspects of contemporary GPP architecture which confound the straightforward library approach, and propose a vectorized, compiler-accelerated approach. Section 5 discusses *rounding*, an important aspect of implementing floating point. Section 6 presents an experimental evaluation of both library and compiler-accelerated

schemes on a recent general-purpose processor. Section 7 discusses related work, and Section 8 concludes.

3. REDUCED PRECISION ON GPPS

It is possible to *emulate* floating point operations in software using integer instructions. However, each floating point operation requires many integer instructions so emulation is slow.

In particular, the IEEE-754 standard has several special values and ranges such as not-a-number (NaN), positive and negative zero, infinities, and sub-normal numbers, each of which adds to the complexity of software emulation [15].

Modern processors provide hardware floating point units which dramatically reduce the time and energy required for floating point computations. However, these units normally support just two floating point types, typically **binary32** (**float**) and **binary64** (**double**).

3.1 Our Approach

We propose a set of non-standard floating point multibyte types that we refer to as *flytes*. Rather than emulating computation on *flytes* in software, we convert them to/from the next largest natively supported type. Thus, a 24-bit *flyte* (**flyte24**) is converted to **binary32** before computation, and the **binary32** result is converted back to **flyte24** afterwards.

We need to solve two problems: (1) efficiently loading and storing non-standard data sizes, such as 24-bit data, where there is no hardware support for such operations, and (2) quickly converting between built-in floating point types and our non-standard types.

In general, converting between floating point formats has many special cases. In particular, converting between formats with different-sized exponents may cause numbers to overflow to infinity or underflow to/from sub-normal numbers. Dealing correctly with these cases in software is complicated and slow.

Our solution to the problem is that in all our *flyte* types, the size of the exponent is equal to the size of the exponent of the **next largest built-in type**. For example, in our **flyte16** and **flyte24** types, the exponent has eight bits, just like **binary32**. This dramatically reduces the complexity of conversions.

Efficiently supporting non-standard floating point types using this approach creates two types of problems. The first is supporting the loading, storing, and conversion of reduced precision types with acceptably low overhead. This is the topic of the majority of this paper.

The second type of problem is that performing computation in one floating point type and storing values in a less precise type introduces issues, such as double-rounding, that complicate numerical analysis. We make every effort to be clear about this latter group of problems but in most cases we do not have comprehensive solutions. Double rounding in particular is a topic of extensive study, and we refer the reader to the work of Boldo and Melquiond [2] for an in-depth discussion.

Our techniques are aimed squarely at problems where some approximation is acceptable and the developer has a good understanding of exactly how much precision is required. Our main contribution is to show *how* to implement multibyte floating point formats efficiently; the question of *whether* to use them in any particular algorithm depends on the numerical properties of the algorithm.

3.2 Simple Scalar Code Approach

Figure 1 shows a simple implementation of the `flyte24` type in C++. It relies on the *bit field* facility in C/C++ to specify that the `num` field contains a 24-bit integer. It also uses the GCC `packed` attribute to indicate to the compiler that arrays of the type should be packed to exactly 24 bits, rather than padded to 32 bits. Figure 1 also shows a routine for converting from `flyte24` to `float` (i.e. `binary32`). The 24-bit pattern stored in the `flyte24` variable is scaled to 32 bits and padded with zeros. The resulting 32-bit pattern is returned as a `float`.

```
class flyte24 {
private:
    unsigned num:24;
public:
    operator float() {
        u32 temp = num << 8;
        return(cast_u32_to_f32(temp));
    };
    ...
} __attribute__((packed));
```

Figure 1: Simple implementation of `flyte24` in C++

The code that is sketched in Fig. 1 can be used to implement programs with arrays of `flyte24` values, but it is very slow. Figure 6a shows a comparison of the execution time of several BLAS kernels using `flyte24` and other `flyte` and IEEE-754 types.

The order of magnitude difference in execution time is the result of (1) the cost of converting between `flyte24` and `binary32` before and after each computation; and (2) the cost of loading and storing individual 24-bit values.

In particular, storing data to successive elements of packed `flyte24` arrays can result in sequences of overlapping 3-byte aligned stores. Load/store hardware in GPPs is not designed to deal with such operations, which results in extraordinarily slow execution.

Table 1 summarizes our proposed set of *multibyte* formats for floating-point values which preserve the sign and exponent fields of the corresponding IEEE-754 representations.

Table 1: *flyte* storage formats for IEEE-754 types.

IEEE-754 type	<i>flyte</i> format	<i>flyte</i> layout (bits)		
		Sign	Exp.	Mant.
<code>binary32</code>	16-bit	1	8	7
<code>binary32</code>	24-bit	1	8	15
<code>binary32</code>	32-bit	1	8	23
<code>binary64</code>	40-bit	1	11	28
<code>binary64</code>	48-bit	1	11	36
<code>binary64</code>	56-bit	1	11	44
<code>binary64</code>	64-bit	1	11	52

4. ACCESS IN REDUCED PRECISION

Reading and writing reduced precision representations might be expected to incur a significant performance penalty due to the overheads outlined in Section 3.2. Particular concerns are (1) the overhead of conversion between data formats, in addition to (2) the overheads of memory access to

arrays of datatypes that may have non-power-of-two byte width, where memory movements may be overlapping and misaligned. Although these overheads are encountered both when reading and when writing reduced-precision representations, there are important differences between the two cases.

4.1 Reading In Reduced Precision

Modern instruction set architectures typically have native support for data movement using types with power-of-two byte widths – usually 1, 2, 4, and 8 bytes. Since our *flyte* types differ in width from standard IEEE-754 types by multiples of 8 bits, this means we can always fetch a *flyte* with a single read using a wider native type (e.g. a 4-byte read for a `flyte24`).

Since we propose to store *flytes* packed consecutively in arrays without padding, the majority of such accesses will be misaligned. Specifically, a consecutive *flyte* array access will only be aligned to the next largest native type once every $lcm(\text{native}_{bits}, \text{flyte}_{bits}) / \text{flyte}_{bits}$ array elements.

Unaligned access can cause extra cache misses versus aligned access due to the possibility that the accessed item spans a cache line boundary. The strategy of using overlapping accesses at the next largest native type allows us to utilize the vectorization approach of Anderson et al. [1]. (Section 4.3) for our vector memory accesses. Also, the conversion process when reading *flytes* is relatively simple, requiring only that the read data be shifted and padded with zero bits.

4.2 Writing In Reduced Precision

Writing data to a reduced-precision format is more complex than reading, both in terms of the memory movement (since memory locations are modified), and due to the fact that when writing, the number format narrows, and precision is lost.

Loss of precision is a natural consequence of working with floating point numbers. The precise result of a floating point computation can be a real number which cannot be exactly encoded in a finite representation (for example, it might require infinitely many digits). In these cases, loss of precision is necessary to make the result representable.

The IEEE-754 standard specifies several methods which are used to *round* numbers to a nearby representable value. One straightforward way to perform rounding is to simply truncate extra digits (i.e. bits, in a binary representation). This is the standard round-to-zero rounding mode [17]. Other rounding modes are specified by the IEEE-754 standard, including round-to-nearest-even and round-to-infinity (positive or negative).

4.3 Vectorized Reading and Writing

We propose a compiler-based approach which can greatly reduce the overhead of operating on *flyte* arrays using automatic vectorization. Our approach uses vector instructions to load, convert, and store *flytes*. We generate vectorized code to load a number of *flyte* values at once, and unpack those values in parallel to the next largest IEEE type using vector shuffle and blend instructions. We use vectorization not only to amortize the cost of converting each element (as it does for other operations), but also to help overcome penalties associated with *flyte* types’ unnaturally aligned memory accesses.

By restricting the size of a *flyte* to be a multiple of 8 bits, we ensure that widely available fast byte-level vector reorganization instructions can be used. Finally, when computation is complete, we again use vector instructions to convert back to *flyte* types. This may involve a rounding step when reducing the precision of a standard IEEE-754 format to a *flyte* type, followed by packing the data in vector registers before the results are written to memory.

Vectorized loading and storing of packed data elements that do not correspond to any native machine type presents additional challenges over scalar memory movement. Since vector lengths on modern GPPs are usually a power-of-two bytes, vectorized access to *flyte* arrays often leads to the splitting of data elements across vector memory operations, where the leading bytes of an element are accessed by one operation, and the trailing bytes by the next. Figure 2 displays one such scenario: storing in **flyte24** format computational results produced in **binary32**.

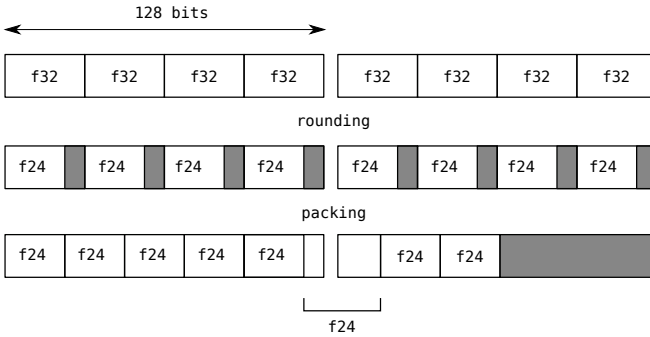


Figure 2: Layout of data in 128-bit (16-byte) vector registers. (top) before format conversion, (center) after rounding to the desired size, and (bottom) the desired layout in memory. Note that the desired memory layout requires data elements to straddle the boundary between vector registers.

While vectorized reading is not significantly more complicated than scalar reading, vectorized writing has additional issues. A straightforward approach to vectorized writing of unpadding *flyte* data could pack as many consecutive *flyte* elements in a vector register as would fit, and perform a store with the trailing unused byte positions predicated off.

Subsequent stores, if there are more than one, could overlap prior stores in these unused positions, so that the data is consecutive in memory. Due to the structure of load/store hardware in GPPs, this approach is likely to be extremely inefficient.

Our vectorized approach to storing values in reduced precision format works by packing all the rounded data elements to be stored consecutively in a set of vector registers, which are mapped to a set of consecutive, **non-overlapping** vector memory movements (shown in Figure 2).

We use a two-phase permute and blend approach. Vector permute instructions are initially used to compact the rounded data in each register into memory order, and align the contents of some registers so that the leading data element in each register is located at the position of the first unused byte in the previous register. Next, the compacted vector registers are combined together using vector blend instructions until a number of fully packed vector registers result.

The resulting registers can be stored without overlap, and data elements are correctly split across register boundaries. If the data written cannot be packed perfectly into full vector registers, some vector stores may be partial stores with some additional implementation concerns. These are described in detail in the vectorization approach of Anderson et al. [1].

4.4 Controlling Format Conversion

Programming language designers have three conflicting goals when deciding the rules for evaluating floating point expressions. Evaluating expressions at higher precision may result in more accurate answers, which suggests that higher precision should be used if it is available.

On the other hand, programmers like to get bit-exact identical results from their program regardless of which compiler is used, which suggests that the language should strictly define the precise precision and each floating point operation.

The IEEE-754 standard stipulates [17, §11] that conforming languages *should* support reproducible programming, and defines circumstances when programs should have numerically identical results across compliant platforms. Finally, giving the compiler the freedom to choose precision may result in more efficient code.

Controlling exactly when format conversion occurs is an important part of using non-standard formats such as our *flytes*. Languages such as C99 [8] provides features aimed at providing consistency in the output of the same program on different platforms.

In particular, the programmer can choose to specify the rules determining the precision of intermediate values in expressions using the `FLT_EVAL_METHOD` facility.

The programmer can also use the pragma directive `STDC FP_CONTRACT` to enable or disable *contraction* – atomic evaluation of floating point expressions which can use higher precision and omit rounding errors implied by the source code and `FLT_EVAL_METHOD` [8, §6.5]. Both facilities modify behaviour at the expression level.

For example, the programmer can choose to have all subexpressions within an expression tree evaluated at the exact width of the widest operand to each operator (by setting `FLT_EVAL_METHOD = 0`).

Such strict constraints on the types of all intermediate floating point values can result in very poor performance for *flytes*. For example, **flyte24** is stored in memory as a 24-bit value, but we must convert it to **binary32** before performing any operations. In a larger expression of **flyte24** values, the generated code is likely to be much faster if all intermediate values can be kept in **binary32** rather than being converted down to **flyte24** after every operation.

C99 also provides a feature that allows floating point operations to be performed at a higher precision than the source values or result: *contraction* of floating point expressions, which is controlled by the standard pragma directive `FP_CONTRACT`. In all our experiments, `FP_CONTRACT` is on.

One problem not addressed by C99’s floating point support is the issue of using higher precision values across *statements*. C99 requires that the value stored in a variable must be converted to the type of that variable at the point where it is written to storage.

We propose a type qualifier `AT_LEAST` which is used to tag a floating point type informing the compiler that it is free to use a higher precision to store results of that type, rather than converting strictly to the precision of the storage type.

This allows the use of higher-precision types in code like that in Figure 3 where accumulation into a variable would otherwise result in many lossy conversions.

```
flyte24 sum(flyte24 * a, int size)
{
    AT_LEAST flyte24 sum = 0.0;
    for(int i = 0; i < size; i++) {
        sum = sum + a[i];
        /*
         * without AT_LEAST, C99 will truncate
         * sum here in every loop iteration
         */
    }
    return sum;
}
```

Figure 3: Example of the use of the `AT_LEAST` qualifier to allow accumulation in a higher precision than the input/output.

The code in Figure 3 shows the variable `sum` of type `AT_LEAST flyte24` which the compiler can represent as any floating point type with at least the precision of a `flyte24`. In general, the next largest native type is a good choice for a use of the `AT_LEAST` qualifier.

5. ROUNDING

When numbers represented in IEEE-754 floating point format are used in computations (such as addition and subtraction), the natural result of computation is often a real number which is not exactly representable in the finite representation. In these cases, the standard specifies a way to round these numbers to a nearby representable value.

Computations on *flytes* are performed using the next largest standard floating point size. After each operation, the built-in floating point type performs its own rounding. However, a question arises when we convert from IEEE floating point types to *flytes*: should we round again during conversion?

Double rounding is discussed in considerable depth by Boldo and Melquiond [2], and we do not reproduce their arguments here. Note that our aim in this paper is simply to demonstrate that a low-overhead implementation of customized floating point types is feasible on general purpose processors.

Any compiler framework which might implement our proposed scheme should take the necessary steps to ensure that the transformations applied to convert between floating point representations are correct; in the remainder of this section, we describe some low-overhead mechanisms which can be used to implement them.

5.1 Round-towards-zero

The simplest approach is to round by truncating the lower mantissa bits, an option which is known as *round-towards-zero* in IEEE 754. Rounding towards zero is simple to implement, but IEEE floating point has a number of special cases that we must treat correctly.

In IEEE-754 a NaN value has an exponent consisting entirely of ones, and a mantissa value that is non-zero. If the non-zero part of the mantissa is in the lower bits, truncating those bits may cause the entire mantissa to have a zero

value. This would change the value from NaN to a value with all ones in the exponent and zero mantissa, which represents infinity in IEEE-754.

However, there are two types of NaNs: signalling NaNs, which cause a floating point exception, and quiet NaNs which indicate an invalid value with causing an exception. In IEEE-754 binary floating point formats, quiet NaNs are distinguished from signalling NaNs by the value of the *most significant bit* of the mantissa, which is preserved by truncation of up to $M - 1$ bits.

In IEEE-754, a subnormal number has a zero exponent, and the mantissa represents a very small fixed-point number. Truncating the final bits of a sub-normal number may cause its value to change to zero. This is correct behaviour, since the non-zero part of the number is too small to represent in our smaller *flyte* type, and zero is the closest representable value.

5.2 Round-to-nearest

Round-towards-zero is simple to implement, but it can result in large errors. For example, if round-towards-zero were applied in decimal, the number 9.9 would be rounded to 9, rather than 10. The maximum error can be reduced by rounding to the nearest representable number rather than simply truncating.

A special case is where the number to be rounded is exactly between two values. To break the tie, IEEE-754 specifies that exact ties should be rounded to the nearest even number.

Figure 4 shows a 32-bit floating point number being rounded to our 24-bit representation using round-to-nearest, where exact ties between two values are rounded to the nearest even value.

Rounding to nearest even is expressed in terms of the relationship between three bits (Guard, Round and Sticky) around the point where the number is rounded. There are no hardware instructions in conventional processors for rounding floating point values to *flytes*, and we must therefore round in software.

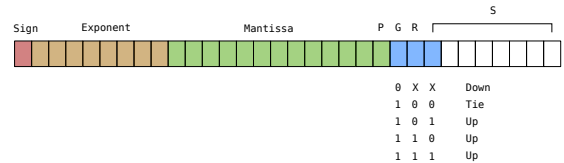


Figure 4: Rounding to nearest even. Marked bit positions correspond to **G**uard, **R**ound, and **S**ticky bits. To avoid overflow into the exponent when rounding to nearest even in software, the pre-guard bit (**P**) must also be inspected.

In the most straightforward case, rounding to nearest even simply involves computing the nearest representable number. An easy way to do this is shown in Figure 5.

The code in Figure 5 adds half of a unit of least precision (ULP) in the new smaller format to the value in the existing larger format.

As long as the number to be rounded is not exactly between two representable values in the new format, this will result in correct round-to-nearest-even. Implementing precise round-to-nearest-even requires checking for this tied special case, and rounding accordingly. In addition, there are

```

flyte24 round_to_nearest(float num)
{
    u32 temp = cast_f32_to_u32(num);
    // round by adding 0.5 ULP
    temp = temp + 128;
    // truncate last eight bits
    temp = temp >> 8;
    return cast_u32_to_flyte24(temp);
}

```

Figure 5: Heuristically rounding to nearest even by adding half of a unit of least precision (ULP).

numerous special cases which must be checked to correctly implement IEEE-754 mandated behaviour.

5.3 Treatment of Special Values

The IEEE-754 floating point standard has special values and ranges as previously outlined in Section 2. Rounding interacts in different ways with these values and ranges, and behaviour which may be appropriate for some scenarios may not be for others.

The application programmer must make a choice of rounding approach based on the information available. We describe the behaviour of each of our proposed rounding approaches here with respect to IEEE-754 special values and ranges.

5.3.1 Normalized numbers

When a normalized number is being rounded, an infinity occurs when the rounded value is so large that the exponent is all ones after rounding (overflow). This is a natural consequence of conversion from a larger to a smaller finite representation.

However, when a normalized number is so small that its exponent is all zeros after rounding (underflow), it does not get rounded directly to zero, but instead to a subnormal number.

Underflow is *gradual*, and a number will underflow to zero only when it is so small that both exponent and mantissa are all zeros after rounding. Normalized numbers may therefore naturally be rounded to several different classes of value.

Very large positive or negative numbers may go to infinity when rounded, and very small numbers may become subnormal. This behaviour conforms to IEEE-754 semantics.

5.3.2 Subnormal numbers

Subnormal numbers are distinguished by a zero exponent, and lack the implied leading 1 in their mantissa. They represent numbers very close to zero. The expected behaviour of format conversion for subnormal numbers is slightly complicated due to the issues of overflow and underflow.

The closest value in the target representation for a very large subnormal number may be a very small normalized number (overflow), while the closest value for a very small subnormal number may be zero (underflow).

For subnormal numbers, rounding may validly cause the class of the value to change, either by underflow to zero, or by overflow to a small normalized number.

5.3.3 Infinities

Positive and negative infinities are encoded with an exponent which is all 1s and a zero mantissa. There are only two values in this class, which are distinguished from each other by their sign. The expected behaviour of format conversion for infinities is a correctly signed infinity in the target format.

5.3.4 NaN values

NaN values represent the result of expressions of *indeterminate form*, which cannot be computed, such as $\infty - \infty$. The expected behaviour of format conversion of a NaN value is a NaN value in the target format. However, NaN is not a singular value, but a value *range*. NaN values occupy a range of bit patterns distinguished by an exponent which is all ones and any nonzero value in the mantissa.

Since the mantissa is truncated by conversion to a shorter format, some NaN values cannot be represented after down-conversion, and indeed truncation may cause the non-zero part of the mantissa to be lost entirely.

However, as described in section 5.1, IEEE-754 non signalling (or *quiet*) NaNs always have a 1 in the *most significant bit* of their mantissa. Thus, although the exact mantissa value of a NaN may change after truncation, a quiet NaN cannot become a non-NaN value through truncation.

Signalling NaNs are different: the non-zero bits of a signalling NaN's mantissa may be entirely in the lower bits. A signalling NaN may be corrupted to become an infinity value by truncation. Thus, signalling NaNs should not be used with *flyte* values, unless the signal handler is modified to place a non-zero value in the higher-order bits of the NaN.

When using our add-half-an-ULP heuristic, a further problem can arise with non-signalling NaNs. If the mantissa value of the non-signalling NaN is the maximum representable value after truncation, then adding even half an ULP will cause an overflow from the mantissa to the exponent, and potentially into the sign bit, if the exponent is all ones.

Our slowest and most correct round-to-nearest mode checks for this case, and corrects the value if necessary. Our fast heuristic round to nearest approach (Figure 5) does not. However, despite extensive testing we have never seen a case where the floating point unit creates such a pathological NaN value as the result of arithmetic.

6. EXPERIMENTAL EVALUATION

We benchmarked the performance of our proposed scalar code implementation from Section 3.2 and vectorized implementation from Section 4.3. Experiments were run on 64-bit Linux with a 4.2 series kernel, using a machine with 16GB of RAM and an Intel Core i5-3450 (Ivy Bridge) processor.

We followed Intel's guidelines for benchmarking short programs on this architecture [12]. Figures 6 and 7 present the results of benchmarking. In our experiments, we use rounding towards zero, `FP_CONTRACT` is on, and variables which are used to accumulate are declared as `AT_LEAST` the target precision.

Problem size in experimental figures refers to the number of elements in arrays in BLAS operations - for vector-vector operations (BLAS Level 1) this is the number of elements in a vector, while for operations involving matrices (BLAS Level 2 and 3) this is the number of elements in a row or column of a square matrix, so that the total number of data elements is the square of the problem size.

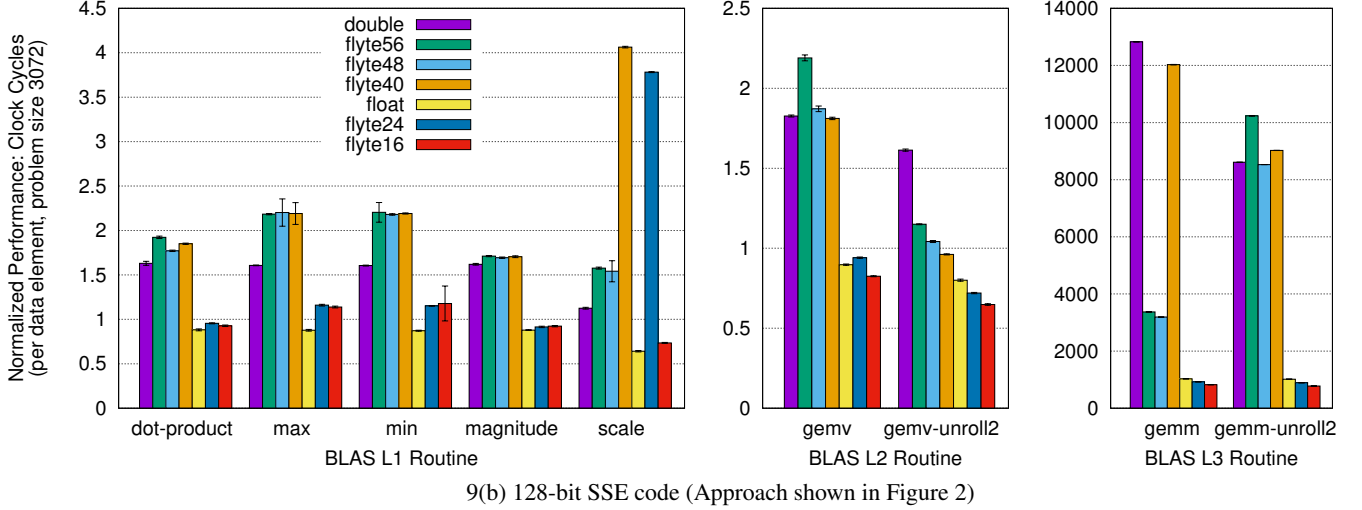
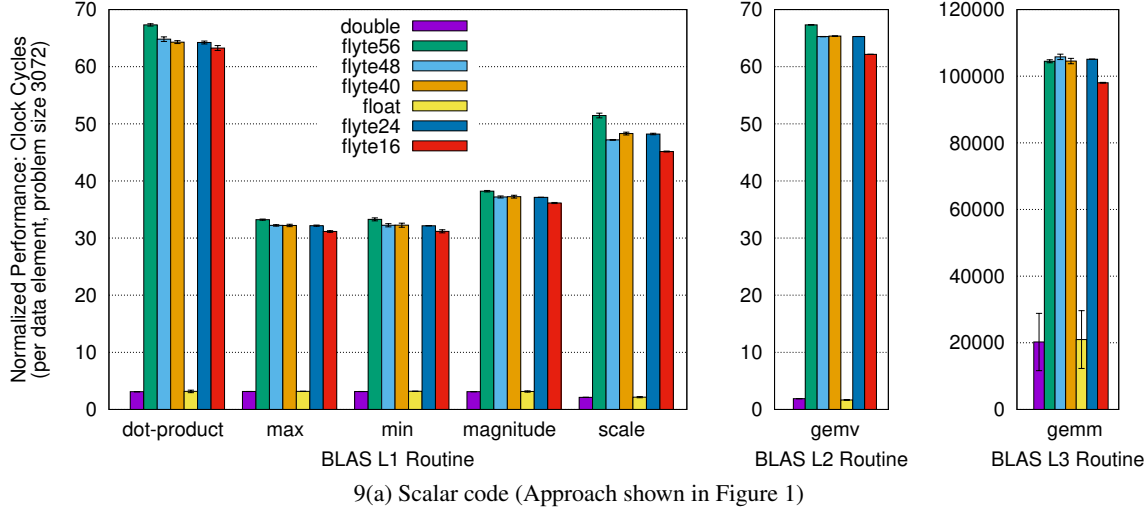


Figure 6: Variation of performance with precision of memory representation across a number of BLAS kernels. Performance displayed as normalized execution time (cycles per data element) – **lower is better**. Overhead versus native types can be read as the difference between any *flyte* type and the next largest native type.

6.1 Overheads: SIMD vs Non-SIMD

The large overhead of scalar access in Figure 6a is due to the design mismatch between our non-standard use-case and the typical structure of a GPP scalar datapath, discussed in more detail in Section 3.2. The GPP datapath is ill equipped to handle simultaneous misaligned accesses to data stored in packed non-power-of-two multibyte formats.

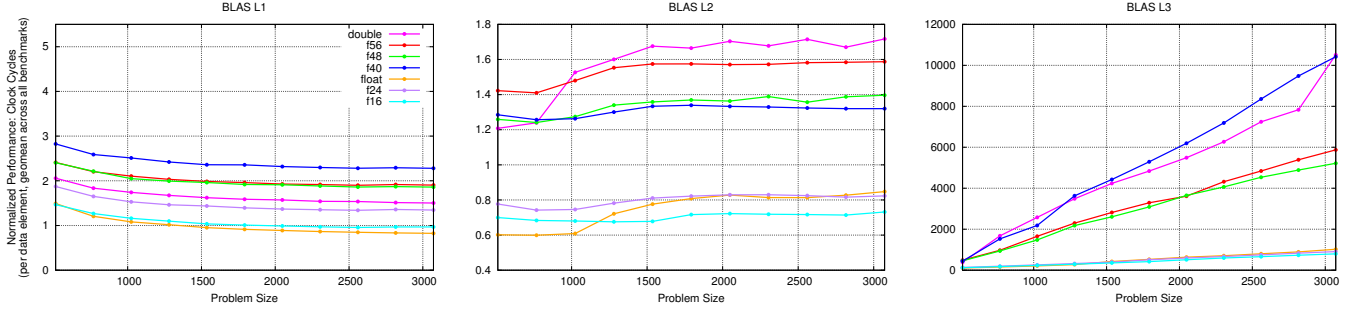
In contrast, our compiler-based vectorized approach (Figure 6b) marshalls and unmarshalls this data into consecutive, non-overlapping power-of-two length accesses which are the best-case for performance using the (SSE) vector datapath.

Overheads in the scalar implementation are in the mid to high tens of cycles per accessed element, while in the vectorized implementation, overheads versus native IEEE-754 types are on the order of a single cycle per data element. In computationally heavy programs like *magnitude*

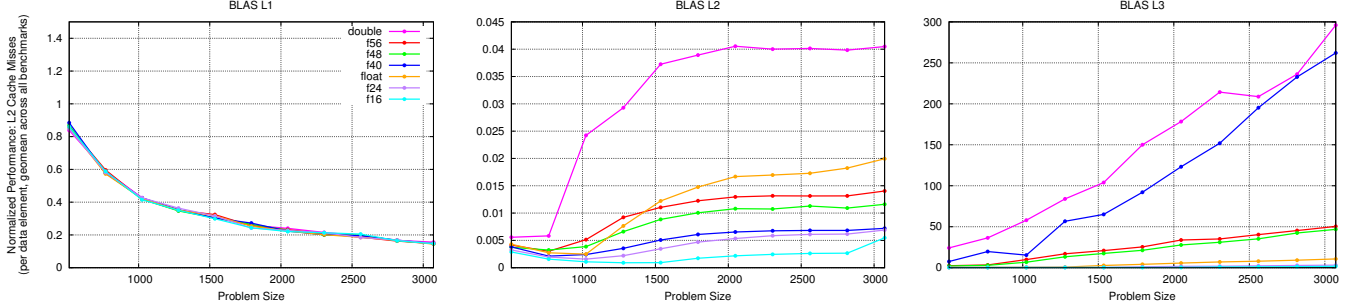
this overhead is effectively hidden by instruction-level parallelism (Figure 6b).

The relatively high overhead of *flyte40* and *flyte24* in the *scale* benchmark in Figure 6b is due to two factors: the alignment of the accesses is odd (5 and 3 bytes, respectively) meaning that data must be shuffled *between* vectors, rather than simply within vectors, as with other types. Furthermore, the benchmark performs an *in-place* update of the data, where reads and writes overlap, which reduces the available instruction-level parallelism.

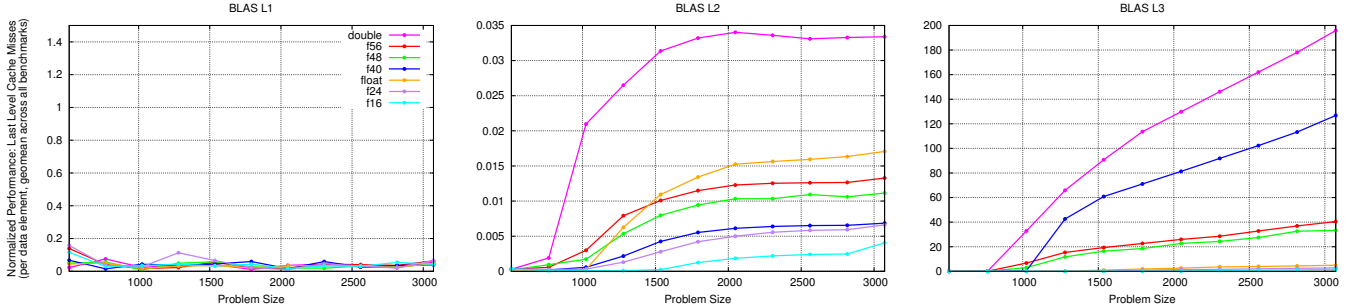
However, the overhead is still on the order of 3 cycles per data element in the worst case, which may be perfectly acceptable in many scenarios in return for a 37.5% reduction in memory traffic. Indeed, as can be seen from our results in Figure 7a, *flyte40* is only marginally slower overall considering BLAS Level 1 programs, and significantly reduces second-last and last-level cache misses for BLAS Level 2 and Level 3 programs.



9(a) Normalized variation in absolute performance (clock cycles per element processed) at each BLAS level as problem size increases. **Lower is better.**



9(b) Normalized variation in second-last level cache misses as problem size increases (misses per element processed). **Lower is better.**



9(c) Normalized variation in last-level cache misses as problem size increases (misses per element processed). **Lower is better.**

Figure 7: Summary of absolute performance (cycles per processed element) and cache behaviour (cache misses at L2 and L3) for our vectorized BLAS benchmark programs as problem size increases. Performance for each BLAS level is measured as the geometric mean performance across all the programs in that level. The programs in each BLAS level are shown in Figure 6b.

6.2 Effect of Unrolling Loops

The benchmark `gemv-unroll12` in Figure 6b is the BLAS Level 2 `GEMV` kernel unrolled twice to increase the amount of data movement per vectorized loop iteration. In this benchmark, data transfer accounts for a large portion of total execution time.

The benchmark demonstrates that a win-win is possible: the choice of a reduced-precision memory representation can actually *increase* overall performance versus the next largest IEEE-754 type, while also reducing memory requirements, even on a general purpose processor without special hardware support for non-standard floating point.

The benchmark `gemm-unroll12` in Figure 6b is the BLAS Level 3 `GEMM` kernel unrolled twice to increase the amount of data movement per vectorized loop iteration.

In effect, unrolling is equivalent to a simple 1-dimensional loop tiling with tile size $2 \times VF$ where VF is the vectorization factor. In this benchmark, data transfer accounts for a large portion of total execution time.

However, unlike `GEMV`, `GEMM` exhibits a slowdown when unrolled. We inspected the performance data and found that the number of last level cache misses was significantly elevated when `GEMM` was unrolled.

In this case, unrolling simply introduces too much data movement in the inner loop. It is likely that less naïve tilings could mitigate this effect, however, we aim only to show the effect of using reduced precision types.

6.3 Effect on Cache Behaviour

The primary effect of using shorter data representations is seen in the behaviour of the second-last and last-level caches. Using a shorter data representation means that each individual memory operation (i.e. cache line read or write from DRAM) stores or retrieves a larger proportion of data elements, resulting in fewer cache misses overall. Reducing the number of last-level cache misses in particular has a large effect on performance, and on energy efficiency [7].

6.3.1 BLAS Level 1

Figure 7 displays a summary of the absolute performance (cycles per data element) as well as the second-last and last-level cache behaviour for our benchmark programs. As in Figure 6, the programs are divided into three categories, one for each BLAS level.

For BLAS level 1 programs, we see that the variation in absolute performance remains small as problem size increases. BLAS level 1 programs are mostly compute-bound, so using smaller types does not significantly affect performance.

Figures 7b and 7c show that, for BLAS Level 1, there is very little difference in the number of second-last and last-level cache misses from using shorter types.

6.3.2 BLAS Level 2

For BLAS level 2 programs, we see that initially there is little variation in performance at small problem sizes. However, as the problem size increases, the effect of using smaller types becomes apparent. For BLAS Level 2 programs, we see a large variation in performance once the problem size exceeds the capacity of the L1 cache.

In Figure 7a, we see that for BLAS Level 2 programs, `float` and `double` initially outperform their reduced-precision representations, but once the problem size grows large enough, the situation is reversed.

Moreover, smaller representations outperform larger ones, in general. At the largest problem size in experiments, `GEMV` on `flyte40` outperforms `GEMV` on `double` by 23.5% (Figure 7a, center graph).

Figures 7b and 7c display the large reduction in second-last and last-level cache misses for BLAS Level 2 programs. In some cases, the reduction is as many as $6\times$ fewer last-level cache misses (compare `flyte40` and `double` in Figure 7c, center graph).

6.3.3 BLAS Level 3

For BLAS Level 3, we again see that using smaller types results in many fewer second-last and last-level cache misses, although the reduction is less pronounced for `flyte40`. 5-byte accesses are frequently split across cache lines, causing two misses, which offsets the reduction in misses from simply transferring less data overall.

However, we again see that performance closely tracks cache behaviour - our straightforward implementation of `GEMM` is heavily memory-bound, so this is not surprising. Overall, we see a significant reduction in cache misses for smaller types: as many as $4.5\times$ fewer last-level cache misses comparing `flyte56` and `double` (Figure 7c, rightmost graph).

7. RELATED WORK

Much prior work discusses reduced precision floating point [3, 10, 13]. Jenkins et al. [9] evaluate a reduced-precision scheme

using GPPs in an extreme-scale computing context. They do not utilize SIMD, address only reads, and convert in a pre-pass.

Many approaches use FPGAs or otherwise customized hardware; notably Tong et al. [16], who propose customizing ALUs to support short-mantissa representations. More recently, De Dinechin et al. [5] also propose custom hardware to support reduced precision. Ou et al. accelerate mixed-precision floating point using a vector processor with a customized datapath [11].

Our approach, since it targets GPPs, is necessarily less flexible than FPGA/custom hardware based approaches. However, it is precisely because GPPs are so widely deployed that reduced precision support on GPPs is attractive.

Prior work on loading only portions of floating point numbers by Jenkins et al. [9] does not perform an explicit rounding step, but directly truncates values. They perform a byte-level transpose on a matrix of floating point numbers stored in memory, chopping off some number of trailing bytes of each number.

This conversion approach is semantically equivalent to rounding to zero. Jenkins et al. evaluate the error introduced by rounding to zero and find it acceptable for their purposes.

In contrast, using our proposed techniques, the rounding step in Figure 2 can be implemented in any way which is suitable for the needs of the application, including rounding towards zero, rounding to nearest even, or rounding to odd as proposed by Boldo and Melquiond [2].

8. CONCLUSION

In this paper, we propose *flytes*; a scheme representing floating-point data in memory at precisions along a continuum between IEEE-754 types, converting to and from standard IEEE-754 types to perform computations.

We propose a method for converting between IEEE-754 floating point and *flytes*, and show how it can be accelerated using vectorization on general purpose processors, without requiring special hardware support.

Our proposed technique handles both reads and writes, and supports reduced precision floating point memory representations with very low overhead.

Our investigation shows that reducing the precision of floating point data in memory, and using SIMD operations as the basis of a compiler-accelerated scheme for performing conversions presents a low-overhead path to supporting customized floating point on commodity general purpose processors.

Acknowledgements

We would like to thank Ayal Zaks for many helpful comments on an initial draft of this paper. We would also like to thank the reviewers of PACT 2016 for their close attention which helped us greatly in improving the presentation.

This work was supported by Science Foundation Ireland grant 12/IA/1381, and also by Science Foundation Ireland grant 10/CE/I1855 to Lero — the Irish Software Research Centre (www.lero.ie).

9. REFERENCES

- [1] A. Anderson, A. Malik, and D. Gregg. Automatic vectorization of interleaved data revisited. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):50, 2015.
- [2] S. Boldo and G. Melquiond. When double rounding is odd. In *17th IMACS World Congress*, page 11, 2004.
- [3] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Exploiting mixed precision floating point hardware in scientific computations. In *High Performance Computing Workshop*, 2006.
- [4] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield. Efficient embedded computing. *IEEE Computer*, (7), 2008.
- [5] F. De Dinechin, C. Klein, and B. Pasca. Generating high-performance custom floating-point pipelines. In *International Conference on Field Programmable Logic and Applications*. IEEE, 2009.
- [6] P. R. Dixon, T. Oonishi, and S. Furui. Fast acoustic computations using graphics processors. In *International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2009.
- [7] J. Gustafson. Exascale: Power, cooling, reliability, and future arithmetic. *HPC User Forum Seattle*, September 2010.
- [8] B. S. Institution. *The C standard: incorporating Technical Corrigendum 1 : BS ISO/IEC 9899/1999*. John Wiley, 2003.
- [9] J. Jenkins, E. R. Schendel, S. Lakshminarasimhan, D. Boyuka, T. Rogers, S. Ethier, R. Ross, S. Klasky, N. F. Samatova, et al. Byte-precision level of detail processing for variable precision analytics. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2012.
- [10] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In *International Conference on Supercomputing*. ACM, 2013.
- [11] A. Ou, K. Asanovic, and V. Stojanovic. Mixed precision vector processors. *Technical Report No. UCB/EECS-2015-265*, 2015.
- [12] G. Paoloni. How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. *Intel Corporation*, 2010.
- [13] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2013.
- [14] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. *ACM SIGPLAN Notices*, 49(6), 2014.
- [15] N. Sidwell and J. Myers. Improving software floating point support. *Proceedings of GCC Developer’s Summit*, 2006.
- [16] J. Y. F. Tong, D. Nagle, R. Rutenbar, et al. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Transactions on Very Large Scale Integration Systems*, 8(3), 2000.
- [17] D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo, et al. IEEE Standard for Floating-point Arithmetic. *IEEE Std 754-2008*, 2008.