Wordless Integer and Floating-Point Computing

Henry Dietz[0000-0002-5878-881X]

University of Kentucky, Lexington KY 40506, USA
hankd@engr.uky.edu

Abstract. In most programming languages, data is organized in structures that are explicitly mapped to machine words each containing a fixed number of bits. For example, a C variable declared as an `int` might be specified to be represented by a 32-bit word. Given computer hardware in which data are organized as fixed-size words, this seems intuitive and efficient. However, if the integer is known to always have a value between 0 and 100, at most only seven of those 32 bits are needed; the other 25 bits are always 0. Programming languages like C allow integer variables to be declared as having any of several bit precisions, so declaring the variable as `uint8_t` could reduce waste to just one bit. The catch is that the index is really only seven bits long when it holds a value greater than 63. Operating on more bits than necessary dramatically increases both the storage space and the number of gate operations needed to perform operations like addition or multiplication. The solution proposed here is to implement a programming model in which integer and floating-point variables are represented by just enough bits to represent the values they contain at that moment in execution. The overhead involved in dynamically adjusting precision is significant, thus it is only used for SIMD-parallel variables implemented using the PBP execution model.

Keywords: Parallel Bit Pattern computing, SIMD, bit slice, bit-serial arithmetic, variable precision arithmetic, integer arithmetic, floating point.

1 Introduction

Through decades of exponential growth in the number of gates that can be cost-effectively put on a chip, the best way to make programs execute faster was to use increasing amounts of parallelism. Unfortunately, the happy prophecy of Moore's Law is no longer being met; as the rate of increase is decaying, it has become prudent to seek ways to increase compute power without using more hardware parallelism. This concern is amplified by the fact that power consumed per gate action has not been dropping even as fast as the number of gates that can be placed on a chip has grown. There are many ways that the power consumed per computation might be reduced, ranging from adiabatic logic to quantum computing, but one of the most immediately practical methods is to simply avoid performing gate-level operations that do not produce a useful result.

One of the most fundamental concepts in optimizing compilers is the elimination of unnecessary operations. Optimizations like common subexpression elimination, which removes repeated computations of the same value, were well known by 1970[1] and are implemented in nearly all modern compilers. These optimizations are quite effective in reducing the amount of work that must be performed to execute a program, but they generally are performed at the level of operations on values sized as machine words. In a 2017 paper[2], it was suggested that the key to dramatically reducing power consumed per computation is to instead focus on performing similar optimizations at the level of individual gate operations on bits. The methods recommended for minimizing the number of active gates in performing a word-level computation can be broadly divided into two categories: minimizing the number of bits that must be processed and minimizing the number of gate-level operations that must be executed for a given bit-level computation.

1.1 Minimizing The Number Of Bits

Several techniques have been suggested toward minimizing the number of bits that must be stored and processed for each value. The most obvious is that choices between data types should be made more carefully; although C/C++ programs commonly declare most integer variables as `int`, variables that do not need that large a value range, or that are never negative, should be declared using types that instantiate fewer bits, such as `uint8_t`. This type of transformation also can be automated by the compiler performing integer range analysis. In fact, the concept of using compile-time range analysis to infer variable types dates from the mid-1960s[3]. The 2017 work also suggested that precision of floating-point values should be a function of the accuracy required for the result, and that accuracy requirements for operations should be specified rather than precision of variables. It was noted that such accuracy requirements would even allow dynamic choices between float and double representations, or even the use of alternative approximate real-number formats such as LNS (log number systems). Finally, it was noted that smaller representations can be packed to hold more values in a fixed number of memory locations or registers, thus reducing the power associated with storing or transmitting each useful bit.

In the current work, the approach taken is to be able to treat any integer or floating-point variable as inherently variable precision, with the number of bits dynamically varying as the value is changed. Excess bits are dynamically trimmed as new values are generated. Even the signedness of an integer is treated as a dynamic property of the current value. For example, a variable with the value 4 would be represented as an unsigned 3-bit integer, and decrementing it to 3 would change the type to unsigned 2-bit integer. If that value of 3 was then negated, the type of that variable would change to signed 3-bit integer. Operations on integer values thus effectively eliminate redundant bit positions from the most significant bit (MSB) position downward. In effect, the normalization of ordinary floating-point values similarly removes redundant bit values from the MSB downward. It is possible to extend this notion further so that, using a different normalization rule, redundant bits are also removed from the least significant bit (LSB) upward. For example, while the fractional part of the representation of the floating-point value 3.0 would require two bits, the fractional part of 256.0 can be just a single bit long. The exponent also can dynamically change in size. These methods are discussed in Section 2.

1.2 Minimizing The Number Of Gate-Level Operations

Minimizing the number of gate-level operations that must be executed for a given bit-level computation seems impossible for computers that inherently operate on a machine word at a time. However, it was noted that bit-slice hardware, in which word-level operations were literally performed one bit at a time, was once extremely common – and perhaps it is time to revive that model. Bit-serial processing of values was particularly common in SIMD supercomputers, and was used in the ICL Distributed Array Processor (DAP)[4], STARAN[5], Goodyear Massively Parallel Processor (MPP)[6], Thinking Machines CM and CM2[7], and NCR GAPP[8]. The key benefit in using SIMD-parallel execution of bit-serial operations is that it allowed simpler hardware to execute with a faster clock. For example, a throughput of one 32-bit addition per clock cycle can either be obtained by having a fast 32-bit addition circuit perform one addition in one clock cycle or by executing one-bit ripple-carry addition steps in a sequence of 32 clock cycles, but with a parallelism width of 32. Bit serial addition of 32 32-bit values in 32 clock cycles requires only 32 one-bit full adders, for a total of approximately 5×32 gates, yielding 32 results after a total of roughly $5 \times 32 \times 32 = 5120$ gate actions. In contrast, a single fast 32-bit adder built using carry lookahead will require at least twice as many gates, doubling the number of gate actions and hence power consumed for computing the same 32 results; each clock cycle also will be at least an order of magnitude longer, because the longest path through the 32-bit carry lookahead is more than ten times the delay of a one-bit full adder. The bit-serial SIMD machines leveraged this benefit, but generally used a fixed, microcoded, sequence of bit-level operations for each word-level operation; by optimizing at the bit level across multiple word-level operations, as well as performing constant folding where bit values are known, the number of bit-level gate operations can be reduced even more dramatically. It also was suggested that such analysis could target implementation using a quantum computer, potentially leveraging the ability of such systems to have a individual gate-level operation applied to exponentially many superposed bit values with unit cost.

The approach discussed in the current work is perhaps best described as a layered application of symbolic execution. In the top layer, each operation on a dynamic-precision variable can be translated into the simplest possible equivalent set of bit-level operations, and the bookkeeping necessary to adjust precision is performed. This symbolic manipulation is relatively expensive, so it generally is not used for scalar variables. Variables that are massively-parallel SIMD data structures multiply the operation cost without incurring additional bookkeeping, so the benefit in performing the symbolic manipulation can far outweigh the overhead. The layers below that are logically performing SIMD-parallel operations on huge bit vectors distributed across single-bit processing elements. However, the implementation actually recognizes and removes redundancies in four lower layers leveraging the new, quantum-inspired, parallel bit pattern (PBP)[9] model of computation. In PBP, the value of an E-way entangled superposition is represented as a SIMD-parallel 2^E-bit value, and the current work treats it as precisely that: a collective reference to corresponding bits in 2^E SIMD processing elements (PEs). The dynamically-optimized execution of the bit-level SIMD operations is detailed in Section 3.

Section 4 summarizes a few preliminary performance results obtained by executing a prototype C++ implementation on a conventional processor using only SIMD hardware parallelism within a 32-bit or 64-bit word. Although use of a conventional processor prevents the power savings from being realized, the system does allow precise counting of gate-level operations needed for conventional word-level SIMD execution, bit-level optimized SIMD execution, and the execution of bit-level SIMD operations using PBP entangled superpositions.

The contributions of this work are summarized in Section 5, along with directions for future work.

2 Wordless Integer And Floating-Point Variables

The primary contribution of the current work is the concept and implementation of an efficient mechanism for wordless variables: variables for which the number of bits used to represent a value dynamically is adjusted at runtime to minimally cover the specific values being represented.

Although there are languages in which arbitrary numbers of bits may be used to represent a variable, such as Verilog[10] and VHDL[11], bit precision generally is fixed at compile time. This is not surprising in that most such languages are intended to be used to specify hardware designs, and bit precision thus corresponds to the physical number of wires in a datapath. In allowing runtime adjustable precision for a variable, the most similar prior work is not minimizing the number of bits in a representation, but facilitating computations upon multi-word “Big Numbers.” There are many libraries providing variable-length multi-word value manipulation, including The GNU Multiple Precision Arithmetic Library (GMP)[12], BigDigits[13], and ArPALib[14]. Such libraries often use extremely clever implementations of operations in order to improve speed when operating on high-precision values; for example, multiply is often implemented by algorithms other than the usual shift-and-add sequence. In contrast, most values used in programs fit within an individual machine word with space to spare, and none of the above libraries attempts to avoid storing or operating on those unnecessary bits within a word.

2.1 Wordless Integers

In most computers, an unsigned integer is represented as an ordered set of k bits, b_{k-1}, b_{k-2}, …, b_1, b_0, such that the value of bit b_i is $b_i \times 2^i$. That is essentially the representation proposed here. The main difference is that the value of a bit position b_i is not a single bit, but an entire vector of bits distributed one per PE across the *nproc* PEs of a SIMD machine. Thus, rather than representing an integer as an ordered set of bits, it would be more accurate to say an integer is an ordered set of bit-index values, x_i, such that the corresponding bit value in each PE *iproc* is PE[*iproc*].mem[x_i]. The value of k, the number of bits in the representation, is variable and therefore must be recorded as part of the integer data structure.

Removal Of Redundant Leading Bits. An unsigned integer's value is not affected by any bit position holding zero. Thus, it would be possible to record only the positions of potentially non-zero bit values: that is, only values of x_i such that there exists at least one PE *iproc* where PE[*iproc*].mem[x_i]$\neq 0$. In practice, unsigned integer values close to zero are used much more frequently than larger values, thus the probability of a potentially non-zero bit value in position i dramatically decreases as i increases. This suggests that general-purpose methods for encoding sparse data are not needed; it is typically sufficient to truncate any leading bit positions that are zero and keep a count of the number of bit positions retained.

Signed integers, represented as 2's-complement values, present a significantly different encoding problem. Leading zero bits still have no effect on the value because, in effect, a positive value represented in 2's-complement uses the exact same encoding as that value would have as an unsigned quantity. However, negative values essentially treat leading bits in the inverted sense: leading one bits have no effect on the value of a negative number. In other words, the usual gate-level description of sign extension, converting a signed integer value to a larger number of bits, involves filling the additional leading bits with copies of the originally most significant bit. Thus, to reduce the number of bits in a signed integer while maintaining the value, one repeatedly removes the most significant bit, b_{k-1}, until either $b_{k-1} \neq b_{k-2}$ or k=1. However, if the value is positive, this bit-removal process will stop with one more bit in the representation than the same value would have if considered unsigned. For example, the value 3 as an 8-bit signed integer is `00000011`, and as an unsigned integer it can be reduced to `11`, but as a signed integer the simplest representation would be `011` because `11` would be interpreted as the value -1.

At this point it is useful to recall that a bit position does not hold just one bit in our system, but a vector of bit values spread across the SIMD PEs. Thus, the leading bits do not need to be all zero across the machine nor all one in order for precision to be reduced. Consider a 2 PE system representing the 8-bit value `00000011` (3) in PE 0 and the value `11111011` (-5) in PE 1. Across the machine, the bit-level representation could be summarized as {{0,1}, {0,1}, {0,1}, {0,1}, {0,1}, {0,0}, {1,1}, {1,1}}. The rule is simply that as long as the same ordered set of bit values that occurs in the most significant position is repeated below it, the leading bit position may be removed. Thus, the bit-level representation here reduces to {{0,1}, {0,0}, {1,1}, {1,1}}. In classical SIMD terminology, the most significant bit, b_{k-1}, can be removed *iff* PE[*iproc*].mem[x_{k-1}]=PE[*iproc*].mem[x_{k-2}] *for all* PEs, which would seem to require a comparison operation within each PE followed by an ALL reduction.

The apparent complexity of this precision minimization is, however, misleading. Using the PBP model for our SIMD execution, each of the bit vectors stored across the SIMD PEs is implemented by a pbit (pattern bit)[9]. Each pbit value is identified by a pattern register number, essentially the x_i value described above. However, these pbit values are assigned register numbers based on uniqueness: whenever a pbit value is created, it is hashed to determine if that same pbit value has appeared before. If it has, the system ensures that the same register number is used to identify the result; otherwise, it allocates a new register number for the result. Thus, the comparison PE[*iproc*].mem[x_{k-1}]=PE[*iproc*].mem[x_{k-2}] for ALL PEs is implementable as simply x_{k-1}=x_{k-2}, and the actual bit vectors are never accessed. Once completing that minimization, if the most significant remaining bit of a signed integer references the register that holds all zeros, then all values are positive, and the value can be treated as unsigned with that leading zero bit removed.

As a result, the data structure used to represent a variable-precision integer, henceforth called a pint (pattern integer), is:

```
bool has_sign;        // has a sign bit?
uint8_t prec;         // current number of active pbits
pbit bit[PINTBITS];   // pbit register numbers
```

2.2 Manipulation Of Integer Values

It would be valid to consider a pint to have a “normal form” that is minimized as described above. However, given that two different pint values may have different precisions, and perhaps even different signedness, there are a few library-internal routines needed to manipulate these properties so word-like operations can deal with arguments consistently.

Minimize. The **`pint Minimize() const;`** operation simply returns the normal-form version of a pint value. This is done at the end of every library operation that might otherwise result in an unnormalized result.

Extend. The pint **`pint Extend(const int p) const;`** operation returns the “denormalized” version of a pint value with exactly **`p`** pbit precision. This may be used to add extra leading bits or to truncate a value by clipping leading pbits. Note that, in an implementation using lazy evaluation, clipping leading pbits could cause the entire computations that would have created those pbits to be removed.

Promote. The pint **`pint Promote(const pint& b) const;`** operation returns the unnormalized version of a pint value promoted to the smallest precision and signedness that can represent both its value and the value **`b`**. For example, this operation is a necessary precursor to bitwise operations like AND, OR, and XOR.

2.3 Wordless Floating Point

The IEEE standard for floating-point arithmetic defines the internal structure of a word to be used to represent the approximate value of a real number, as well as various accuracy and other constraints on operations.

IEEE 754[15] specifies that a single-precision floating-point value, a `float`, is packed into a 32-bit word. The most significant bit is the sign of the fraction, 0 for zero or positive, 1 for negative. The 23 least-significant bits are the magnitude of the fractional part of the value, with an implicit 24th bit which is treated as a leading one for normalized values. The remaining eight bits are the exponent, which is a power-of-two multiplier for the fractional part. The exponent is encoded as a 2's-complement integer, but a bias of 128 is added so that the minimum value presents as 0 rather than -128.

There are many details that are specified by the IEEE standard. For example, zero is not representable as a normalized number because normalization specifies that the (not stored) most significant bit of the fractional part is a 1. Values with the minimum allowed exponent are treated specially – as *denormals*, in which the implicit most significant bit of the fraction is essentially ignored and normalization is not performed. For example, this allows representing zero as a fractional part that is 0 and an exponent that is the minimum value; conveniently making float zero have the same bit-level representation as the 32-bit integer value zero. Similarly, the fact that the exponent bits reside above the fraction bits means that floating-point comparisons for less than and greater than can use the same logic employed for integer comparisons to compare the absolute values of floats. The IEEE standard also provides for direct representation of +/- infinity and NaN (not-a-number) values, and further specifies rounding modes and accuracy requirements for operations.

The wordless floating-point representation for a `pfloat` is based loosely on the IEEE specification, but differs in a variety of important ways. A single pfloat value represents not just one float value, but one float value per virtual SIMD processing element. The component fields within a pfloat are functionally much like the sign, exponent, and mantissa components of an IEEE float but, as is described below, they are represented and manipulated somewhat differently. Similarly, the normal form for a pfloat, and normalization algorithm, is quite different from that of an IEEE float.

Sign. The pfloat representation of the value sign uses a pint that contains a single pbit. That pbit normally has the exact same meaning as the sign bit in the IEEE standard format: 1 is negative and 0 is non-negative (positive or zero). However, because a pfloat value of 0 is always given a 0 sign bit, the encoding that would represent -0 is instead available for other use, such as representation of NaN.

Exponent. The exponent is stored as a pint specifying a power-of-two multiplier. This differs from the standard in that IEEE 754 uses the minimum possible exponent value to indicate that the value is a denormal, and further requires adding a bias factor to make the minimum exponent value be stored as a field full of 0s.

There would be no significant benefit in adding a bias to the pfloat exponent pint value. There also is no well-defined minimum possible exponent value for a pfloat because the pint exponent field has runtime-variable precision; thus, picking a bias value would artificially impose a minimum bound on the exponent value. It is important to remember that a single pfloat can represent an exponential number of float values, so the overhead of maintaining a scalar variable holding the current precision of the exponent is negligible in comparison to the amount of float data being represented.

There are two motivations for denormals in the IEEE standard. The first is the need to be able to represent the value zero, which is literally impossible to represent as a normal value – and a pfloat cannot circumvent this issue. The second motivation is to allow values between the smallest representable normal number and zero. Without denormals, the difference between the second smallest and smallest normal values would be much less than the difference between the smallest normal value and zero. However, non-zero denormal values could be represented as normals if the exponent field had a larger range. The variable precision of the pint exponent field of a pfloat means that expanding the exponent range naturally occurs as needed – and the number of pbits used to represent the exponent is not artificially increased to cover representation of a fixed minimum value. Thus, the only ordinary pfloat value that is denormal is zero.

Mantissa. In the IEEE standard, the exponent value distinguishes between normal and denormal values, and the mantissa of a normal number has an implicit leading 1 bit, whereas a denormal has an implicit leading 0. In effect, the implicit leading bit is the value of (exponent!=*minimum*), a test that is nonsensical for a pfloat because there is no fixed *minimum* exponent value. Instead, the single denormal value, zero, is represented by the mantissa pint having the value 0. This implies that the exponent field is essentially meaningless when the mantissa is 0. The smallest pint representation occurs for the values 0, 1, or -1 all of which are representable using a single pbit. Giving the value zero an exponent of 0 seems the obvious choice. We further suggest that a mantissa of zero with an exponent of 1 represent infinity. Thus, NaN, zero, infinity, and negative infinity – all values distinguished by having the mantissa field be zero – are not subject to normalization. These pfloat values not subject to normalization are given in Table 1; each is represented using only 3 pbits because the normal pint handling removes leading 0 pbits.

Table 1. The pfloat value representations not subject to normalization.

Decimal Value	Sign	Exponent	Mantissa (8 bit precision)
0.0	0	0	0
NaN	1	0	0
Infinity	0	1	0
Negative Infinity	1	1	0

Normalization. A normal form for floating-point numbers provides a unique representation for each possible value. Without normalization, each number would have many different representations thus wasting multiple bit patterns on encoding a single value, just as the decimal float value 42.0×10^0 is equivalent to 4.2×10^1 and also 420.0×10^{-1}. For an IEEE float, the normal form places the most-significant non-zero bit in the mantissa one bit to the left of the mantissa bits stored. That bit value does not need physical storage because 1 is the only non-zero bit value. However, for a pfloat, the mantissa can be variable size, so which pbit position would correspond to the most-significant bit position? There are two choices; one obvious, the other not.

The obvious normal form is derived by modeling the normal form used by traditional float values. Rather than letting the mantissa of a pfloat vary in size completely dynamically, suppose that a particular mantissa precision is selected. Normalization can be performed by simply adjusting the pfloat so that the most significant bit of the mantissa is 1 for all the values within that pfloat. Table 2 gives some examples of the number of pbits used to encode various decimal values.

Table 2. Some pfloat value representations, MSB normalized.

Decimal Value	Sign	Exponent	Mantissa (8 bit precision)
1.0	0	0	10000000
2.0	0	1	10000000
5.0	0	10	10100000
0.5	0	-1	10000000
-42.0	1	101	10101000

This type of floating-point normalization was often implemented by an expensive process in bitwise SIMD computers: one-bit-position-at-a-time disabling of processors with values already in normal form, shifting the selected mantissas one bit position, and decrementing their exponents. However, bitwise normalization can be performed in log(precision) steps. For example, with 8-bit mantissa precision, the checks would be for top 4 bits all 0, then top 2 bits, and finally top bit, completing in just three steps rather than eight. The problem with this relatively conventional MSB normalization is that the pint mantissa fields naturally trim leading zeros, but not trailing zeros. Thus, the number of apparently active bits can be somewhat inflated.

The less obvious option would be to normalize values not based on the position of their most-significant 1 bit, but based on the position of their least-significant 1 bit. Clearly, normalizing so that the least-significant 1 bit is in the least significant bit position is stripping trailing 0 bits, and combining that with the stripping of leading 0 bits inherently done by pint processing should result in the shortest possible mantissas. This can be seen in Table 3, which shows that the same values given in Table 2 with MSB normalization become significantly shorter with LSB normalization. However, LSB normalization increases overhead in operations like addition and tends to increase entropy, so the current system defaults to MSB normalization.

Table 3. Some pfloat value representations, LSB normalized.

Decimal Value	Sign	Exponent	Mantissa (8 bit maximum precision)
1.0	0	0	1
2.0	0	1	1
11.0	0	0	1011
0.5	0	−1	1
-42.0	1	1	10101

3 Runtime Optimizations

The key concept being leveraged in the current work is that the new Parallel Bit Pattern (PBP) computing model, which was inspired by quantum computing, also can be treated as an extremely efficient model for massively-parallel bit-serial SIMD computation. Instead of viewing the PBP implementation of an E-way entangled superposition as a quantum-like phenomenon, the current work treats it as 2^E virtual bit-serial SIMD processing elements (PEs): i.e., *nproc* is 2^E and *iproc* values range from 0 to 2^E-1. This enables two classes of work-reducing optimizations: compiler-like optimizations performed at the bit level at runtime and optimizations based on recognizing value patterns across groups of PEs.

3.1 Compiler-Like Optimization At Runtime

Compiler optimizations such as constant folding, recognition of algebraic simplifications, and common subexpression elimination are normally applied to word-level expressions at compile time. However, by applying these transformations to symbolic descriptions of massively-parallel bit-level operations, the number of actual massively-parallel bit-level operations that must be performed can be dramatically reduced.

In a traditional bit-serial SIMD computer, each gate-level operation would cause each processing element to produce a single-bit result in its own local memory or register file, and fixed gate-level sequences would be used to implement each word-level operation. For example, adding 4 to a 32-bit variable in each PE would typically invoke the standard 32-bit ripple-carry add sequence rather than taking advantage of the fact that adding 4 can be accomplished by a ripple-carry 30-bit increment sequence applied to the top 30 bits of the variable. In contrast, if the current value of that potentially 32-bit variable in each PE fit in just 12 bits, the methods used here would essentially recognize both that fact and the fact that adding 4 is equivalent to incrementing starting at bit position 2. Thus, the gate-level operation sequence used would be equivalent to a 10-bit ripple-carry incrementer – a much cheaper sequence in both execution time and total energy expended for the computation.

The recognition of such redundancies lies primarily in the concept of a pbit. A pbit logically represents a vector of *nproc* bits, but is actually a descriptor with the interesting property that any two equivalent bit vectors will always have the same descriptor value. This allows the system to dramatically reduce storage space by keeping only a single copy of each unique bit pattern, but also implies that comparing for equality is accomplished by simply comparing descriptors, and never requires examining the actual bits. As each pfloat or pint operation is lowered to pbit operations, the lowering is done by calling a function that not only is parameterized by the current precisions of the operands, but also applies standard compiler optimizations rather than simply generating a fixed sequence of operations.

Constant folding. At the bit level, there are only two constants: 0 and 1. In the current PBP implementation, these are represented by pbit descriptors with the corresponding values, 0 for a vector of all 0s and 1 for a vector of all 1s. When any gate-level operation on a pbit value is requested, the descriptors are first checked, and where all operands are constants the gate result is computed by literally performing the operation on the descriptors. For example, OR of descriptor 0 and descriptor 1 produces a result which is simply 0 OR 1 ⇒ descriptor 1, without accessing any actual bit vector.

Algebraic simplifications. Because the mapping between bit vector values and descriptors is 1:1 and *onto*, a wide range of algebraic optimizations can be applied without accessing any actual bit vector. For example, pbit 601 AND pbit 601 yields pbit 601. Similarly, pbit 601 AND pbit 1 yields pbit 601 and pbit 601 AND pbit 0 yields pbit 0.

Common subexpression elimination. The key bookkeeping problem in recognizing common subexpressions is the mapping to a "single assignment" form, but the pbit descriptors already have that property. Thus, if pbit 42 XOR pbit 601 was found to produce pbit 22, pbit 42 XOR pbit 601 will always produce pbit 22, and the operation does not need to be repeated.

Applying these symbolic compiler optimizations at runtime implies significant overhead, but that overhead is independent of the value of *nproc*. Thus, as *nproc* is increased, the overhead quickly becomes negligible.

3.2 Optimizations Using Pattern Recognition Across PEs

Although classical SIMD models have the concept of values being spread across PEs, most do not provide a means for describing patterns across PEs. In contrast, using the new PBP model to implement bit-serial SIMD computations provides several layers of mechanisms for describing value patterns across PEs, and these can be used to dramatically increase the number of gate-level operations that can be recognized as redundant and avoided.

If the number of SIMD PEs is virtualized so that it may be different from the number of physical PEs, larger numbers of virtual PEs are classically simulated by multiple passes and excess physical PEs are disabled. For example, if there are 1024 physical PEs and 10000 virtual PEs are requested, each gate-level operation would be repeated 10 times and, in the last round, the last 240 physical PEs would be disabled. The hierarchical SIMD-like execution model employed by GPUs improves upon the classical virtualized SIMD model by fragmenting the PEs into SIMT warps[16], typically of 32 PEs each, which allows skipping execution of an entire warp if all the virtual PEs it contains are disabled. Continuing the example, those last 240 virtual PEs occupy the last 7.5 warps; thus, a GPU would skip the last 7 warps entirely and apply enable masking only for the half-enabled warp. The PBP model also fragments each 2^E-bit entangled superposition into smaller chunks, but allows use of far more sophisticated logic to determine when chunk computations can be skipped.

In the PBP model, the layer below pbit is RE, a layer in which each vector of bits is represented by a regular expression that would generate the bit vector. The regular expressions are not patterns of bits per se, but patterns of “chunks” that roughly correspond to the concept of warps in GPUs. Like pbits, both REs and chunks implement 1:1 and *onto* mappings between bit vectors and descriptors, and only a single copy of each unique chunk bit vector is stored. Thus, the same compiler optimizations that were discussed for pbit operations also can be applied for chunks. For example, in the current PBP system, there is an AC layer between REs and chunks that performs applicative caching – essentially implementing common subexpression elimination on chunks.

To appreciate the value of this chunk handling, it is useful to consider a simple example. For the example, suppose that the chunk size is (the ridiculously tiny) 8 bits and *nproc* is 32. As bit vectors with the PE0 bit in the rightmost position, in order of LSB to MSB, the value of *iproc* would look like:

```
10101010 10101010 10101010 10101010
11001100 11001100 11001100 11001100
11110000 11110000 11110000 11110000
11111111 00000000 11111111 00000000
11111111 11111111 00000000 00000000
```

However, the actual chunk pattern is:

```
chunk(2) chunk(2) chunk(2) chunk(2)
chunk(3) chunk(3) chunk(3) chunk(3)
chunk(4) chunk(4) chunk(4) chunk(4)
chunk(1) chunk(0) chunk(1) chunk(0)
chunk(1) chunk(1) chunk(0) chunk(0)
```

Thus, the total storage used for the above vectors is just 5 chunks, or $5 \times 8=40$ bits, not $5 \times 32=160$ bits. Low entropy of values across SIMD PEs is very common. Consider adding 1 to each value (incrementing each 5-bit value to produce a 6-bit result). The chunk pattern for 1 is:

```
chunk(1) chunk(1) chunk(1) chunk(1)
```

The LSB of the result should be four copies of `chunk(1)^chunk(2)`, which we will call `chunk(5)`, and the computation is performed once to produce the new bit vector chunk. The next three chunk operations would all be hits in the applicative cache. However, the RE layer does not necessarily need to even check the AC for this factoring, because it could represent the LSB as `chunk(2)`^{4} and 1 as `chunk(1)`^{4}, thus directly recognizing that there are three copies of the result from the first chunk operation.

In summary, whereas GPUs can improve performance over classical SIMD by skipping disabled warps, using the PBP model for bit-serial SIMD execution allows much more generalized skipping of chunk computations – as well as skipping of "disabled" chunks. It also has the significant benefit of potentially dramatically reducing storage space. However, the storage space reduction is somewhat compromised in the current system by the fact that once a unique chunk value has been created, the current system never deallocates it. A garbage collection scheme would be needed to prevent continuous growth of memory use over long sequences of computations.

4 Implementation And Performance

The current PBP library for bit-serial SIMD computation is implemented as 3,644 lines of portable C++ source code. It supports `pfloat` and `pint` classes with a wide variety of primitive operations and currently runs on a single processor core using bitwise parallelism within either 32-bit or 64-bit words. This is much narrower than the desired hardware parallelism width, and also makes power savings unmeasurable, but hardware directly implementing PBP execution[9] is not yet available. The chunk size for the library may be any power of 2 no smaller than the host word size, and the maximum supported *nproc* is 4294967296.

For the `pint class`, operations include: conversion to/from C++ `int`, reading from and writing to a variable in a selected PE; scatter and gather; initialization to a range of values; logical NOT, AND, OR, and XOR; bitwise NOT, AND, OR, and XOR; comparisons for EQ, NE, GT, LT, GE, and LE; shift right and left; negation, absolute value, addition, subtraction, multiplication, division, and remainder; SIMD ANY and ALL reductions; reductions and scans (parallel prefix) for AND, OR, XOR, addition, multiplication, minimum, and maximum; and sort to increasing or decreasing order. Where appropriate, the C++ operators have been overloaded so that `pint` behaves like a built-in type.

The `pfloat class` implements most of the same operations implemented for `pint`, but not bitwise logic nor remainder. In addition, it implements reciprocal, exponentiation, logarithm, square root, sine, cosine, tangent, and arctangent. The maximum settable precision for a mantissa is 32 bits, although some operations work correctly only for 16 or fewer mantissa bits. The exponent dynamically sizes, and can be as large as 32 bits. C++ operators also have been overloaded for `pfloat`.

At this writing, we have not yet run any significant applications, but have benchmarked several simple programs and the `pint` library validation suite. Given that the PBP code is only using a hardware parallelism width of 32 or 64, one would expect that the bookkeeping overhead would make SIMD code run slower than optimized sequential code on the same processor. However, even for simple programs executed with modest *nproc* values, the PBP run times were within a factor of 2-3$\times$ faster or slower than the optimized sequential code. For the `pint` library validation suite, one would expect poorer performance than from most application codes due to the higher entropy associated with testing all the different library routines. In order to better understand the performance, the PBP library was augmented with various performance counters and the validation suite was run ten times, each with freshly created random data, for each of 8 sets of parameters.

Table 4. Active gate counts for 32-bit word operations vs. proposed PBP model.

nproc	Chunk bits	Gates (Words)	Gates (PBP)	Ratio
65536	256	12279113318	3209523	3826:1
262144	256	55522282700	3141452	17674:1
262144	512	55520002048	6563379	8459:1
1048576	256	252845228032	3135360	80643:1
1048576	1024	252876370739	13902438	18189:1
4194304	2048	1154496017203	29179904	39565:1
16777216	4096	5277432676352	61104947	86366:1
67108864	8192	24153849174425	128459571	188027:1

The measured performance of the validation suite is summarized in Table 4. The first column gives the number of virtual PEs (*nproc*) used and the second column specifies how many bits were in each chunk, which is essentially giving the equivalent of the warp size in GPU terminology. The validation suite creates random data scaled in proportion to the number of bits in a chunk, thus entropy of the test data increases with larger chunk sizes, whereas increasing *nproc* multiplies the total amount of work to be done, but has no direct effect on entropy of the test data. The two “Gates” columns respectively show the average total number of active gate operations that were needed to perform the validation suite’s computations. The “Gates (Words)” column measures the number of gate actions assuming that each pint was treated as typical older bit-serial SIMD systems commonly did, using a fixed gate sequence to handle each value as if it were holding up to 32 bits; it should be noted that this is still a far lower number of gate actions than would be counted using non-bit-serial hardware because, for example, it assumes addition is done by ripple carry rather than by a much more complex circuit (e.g., implementing carry lookahead) as is commonly used in word-oriented arithmetic. As can be seen from the “Gates (PBP)” column numbers, the method proposed in this paper dramatically reduces the number of gate actions used to perform the exact same computation. The rather surprising ratios are given in the final column, making it obvious that the savings from operating only on active bits and performing various bit-level optimizations at runtime *can be* far more than the 32:1 best case that one might have expected.

5 Conclusion

The current work has introduced and explored the concept of wordless integer and floating-point computation, in which precision varies dynamically and aggressive symbolic bit-level optimizations are performed at runtime – all with the goal of minimizing the total number of gate actions needed to perform each computation. The high overhead of precision bookkeeping is managed by applying these types only to massively-parallel data structures being operated upon in a SIMD fashion. The quantum-inspired PBP model is shown to have the potential to be a dramatically more efficient virtualized SIMD execution model by combining this bit-level optimization with the ability to skip chunks of SIMD computation not only if all PEs were disabled for the chunk, but also if equivalent computations had been performed by *any* PEs before. The preliminary results shown here suggest *4-6 orders of magnitude reduction in gate actions per computation* is feasible.

This work is still at an early stage, largely because neither PBP hardware nor bit-serial SIMD computers is readily available, but the portable C++ library implementation will soon be released as open source. Beyond that, the highest priority is resolving the issue of how to garbage collect chunks that are no longer needed.

References

1 Cocke, J., Schwartz, J. T.: Programming Languages and Their Compilers, Preliminary Notes, Second Revised Version. Courant Institute of Mathematical Sciences, New York University (1970)

2 Dietz, H. G.: How Low Can You Go? In: Rauchwerger, L. (ed) Languages and Compilers for Parallel Computing (LCPC) 2017, LNCS, vol. 11403, pp. 101-108. Springer (2017) DOI 10.1007/978-3-030-35225-7_8

3 Klerer, M., May, J.: A user oriented programming language. The Computer Journal, vol. 8(2), pp. 103-109 (1965), DOI 10.1093/comjnl/8.2.103

4 Reddaway, S. F.: DAP - a distributed array processor. Proceedings of the 1st annual symposium on Computer Architecture, ACM Press, 61–65 (1973)

5 Batcher, K. E.: STARAN parallel processor system hardware. National Computer Conference, pp. 405-410 (1974)

6 Batcher, K.: Design of a Massively Parallel Processor. IEEE Transactions on Computers, Volume C-29, Issue 9, 836–840, (September 1980)

7 Tucker, L. W. and Robertson, G. G.: Architecture and applications of the Connection Machine. IEEE Computer, Volume 21, Number 8, 26–38 (August 1988)

8 Morely, R. E. and Sullivan, T. J.: A massively parallel systolic array processor system. Proceedings of the International Conference on Systolic Arrays, 217–225 (1988)

9 Dietz, H., Eberhart, P., Rule, A.: Basic Operations And Structure Of An FPGA Accelerator For Parallel Bit Pattern Computation. 2021 International Conference on Rebooting Computing (ICRC), 2021, pp. 129-133, DOI 10.1109/ICRC53822.2021.00029.

10 IEEE 1364-2001, IEEE Standard Verilog Hardware Description Language, https://standards.ieee.org/ieee/1364/2052/ (2001)

11 IEEE 1076-2019, IEEE Standard for VHDL Language Reference Manual, https://standards.ieee.org/ieee/1076/5179/ (2019)

12 Granlund, T.: GNU MP 6.0 Multiple Precision Arithmetic Library. Samurai Media Limited, Hong Kong (2015)

13 BigDigits multiple-precision arithmetic source code, https://www.di-mgt.com.au/bigdigits.html , last accessed 2022/8/15

14 Macheta, J., Dąbrowska-Boruch, A., Russek, P., Wiatr, K. (2017). ArPALib: A Big Number Arithmetic Library for Hardware and Software Implementations. A Case Study for the Miller-Rabin Primality Test. In: Wong, S., Beck, A., Bertels, K., Carro, L. (eds) Applied Reconfigurable Computing (ARC) 2017, LNCS, vol 10216, pp. 323-330. Springer (2017) DOI 10.1007/978-3-319-56258-2_28

15 IEEE 754-2019, IEEE Standard for Floating-Point Arithmetic, https://standards.ieee.org/ieee/754/6210/ (2019)

16 Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro vol. 28, issue 2, (2008) DOI 10.1109/MM.2008.31