A Quantum-Inspired Model For Bit-Serial SIMD-Parallel Computation

Henry Dietz, Aury Shafran, and Gregory Austin Murphy

University of Kentucky, Lexington KY 40506, USA,
hankd@engr.uky.edu,
WWW home page: http://aggregate.org/hankd

Abstract. Bit-serial SIMD-parallel execution was once commonly used in supercomputers, but fell out of favor as it became practical to implement word-level operations directly in MIMD hardware. Word-level primitive operations simplify programming and significantly speed-up sequential code. However, aggressive gate-level compiler optimization can dramatically reduce power consumed in massively-parallel bit-serial execution without a performance penalty. The model described here, Parallel Bit Pattern Computing, not only leverages gate-level just-in-time optimization of bit-serial code, but also uses a quantum-inspired type of symbolic execution based on regular expressions to obtain a potentially exponential reduction in computational complexity while using entirely conventional computer hardware.

Keywords: bit-serial SIMD, quantum computing, qubit, logic optimization, regular expressions, just in time compilation, C++

1 Introduction

Bit-serial SIMD supercomputing is not a new topic for the Languages and Compilers for Parallel Computing (LCPC) community; much of the work presented in the first two decades of this workshop series targeted such machines. However, the current work is largely focused on applying compiler technology at the level of individual gate operations, and such techniques are far less well studied. The 2017 "How Low Can You Go?" paper [1] was an attempt to inspire more work in that direction, and much of what it suggested is implemented in the system described in this paper.

The *parallel bit pattern* model of computation[2] shares two important properties with quantum computing:

– Both quantum computing and parallel bit pattern computing provide execution mechanisms that have the potential for a single unit of computational work to produce results for exponentially many data values using the concepts of *superposition* and *entanglement*. Quantum computers seek

these benefits by implementing *qubit*s using quantum phenomena. In contrast, *pattern bit*s, or *pbit*s, use **symbolic computation on a compressed bit vector representation – a bit pattern**, which can be manipulated efficiently using conventional computer hardware.
– Both focus on optimizing computations at the gate level. Quantum computers are directly programmed at that low level, expecting programmers to manually optimize the gate-level code. In contrast, parallel bit pattern computing leverages **gate-level compiler optimization technology at runtime** to allow not only programming at the gate level, but also at a higher level, using relatively conventional operators and data types including variable-precision integers: *pint*s.

Parallel bit pattern computing is neither a simulation of quantum computing nor a compatible replacement for quantum hardware. It offers a new high-level programming model, and bit-serial parallel execution model, that together enable conventionally-constructed computers to efficiently use superposition and entanglement to implement a large class of quantum-inspired algorithms. The model is fundamentally stronger than quantum models because it allows non-destructive measurement and values may be maintained for arbitrarily long without decoherence. Of course, all this is accomplished while aggressively using compiler optimization to dramatically reduce the total number of gate-level operations that must be executed to perform each computation, thus potentially reducing the power consumed.

1.1 Representation of Entangled Superposition

Figure 1 shows three different ways in which superposed values can be represented. The value of a *qubit* is commonly modeled as a real-valued, two-dimensional, probability density function: the Bloch Sphere[3]. Instead of using that model, an e-way entangled *pbit* value can be represented as an array of 2^e values (AoV), in which each possible multi-bit value is an element: an entangled pair of *pbit* initialized to equiprobable {0, 1, 2, 3} is shown. However, the AoV

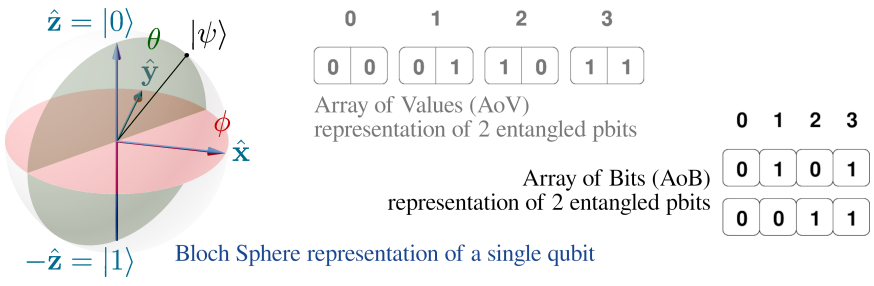

Fig. 1. Representations: Bloch Sphere *qubit*; AoV and AoB 2-way entangled *pbit*

layout does not provide the benefits sought by bit-serial execution. Thus, consider *turning* that representation – a trick used to integrate word-based floating point units with massively-parallel bit-serial execution in the Thinking Machines CM2[4]. Thus, the value of each *pbit* is an array of 2^e bits (*AoB*), entangled values are bits with the same array index (i.e., using the same *entanglement channels*), and value probabilities are not real numbers, but always integer parts per 2^e.

The *AoB* representation offers one more huge benefit: low entropy. The bit values often have relatively simple repeating patterns, which we can compress by representation as a *regular expression* (*RE*). In the *AoB* example above, {0,1,0,1} can reduce to $(01)^2$ and {0,0,1,1} is $0^2 1^2$ by simple *run-length encoding*. By storing and operating directly on REs, parallel bit pattern computing reduces both storage requirements and computational complexity by as much as an exponential factor... essentially the same goal sought by quantum computing, but achieved using partially symbolic parallel execution on conventional hardware.

1.2 A `pbit`-level Example

Even at the *pbit* level, storage space for values is automatically managed. Also unlike quantum computers, programs are not restricted to using reversible gates like `NOT` (Pauli X), `CNOT`, `SWAP`, `CCNOT` (Toffoli), and `CSWAP` (Fredkin). They can be used, but so can conventional gates and fanout. For example, a 1-bit full adder computing `a+b+cin` to produce `sum` and `cout` could be:

```
pbit sum = pbit_xor(pbit_xor(a,b),cin);
pbit cout = pbit_or(pbit_and(a,b),pbit_and(pbit_xor(a,b),cin));
```

Of course, `pbit_xor(a,b)` will be evaluated only once, but the really interesting thing is that values for `a`, `b`, and `cin` can be 3-way entangled superpositions of all 8 possible input values. To do this, each input must be given a Hadamard value on it's own *entanglement channel* (a concept unique to parallel bit patterns):

```
pbit a = PBIT_H(0);    // a is (01)+
pbit b = PBIT_H(1);    // b is (0011)+
pbit cin = PBIT_H(2);  // cin is (00001111)+
```

The result of performing the add is thus that `sum` gets the value `(01101001)+` and `cout` gets `(00010111)+`. Unlike quantum computers, this entangled superposition does not collapse into a single value when measured; any or all of the values can be read. In fact, the entire probability distribution can be read without need to repeat the computation: in this case, the 2-bit {`cout`,`sum`} results would be 1/8 {`0,0`}, 3/8 {`0,1`}, 3/8 {`1,0`}, and 1/8 {`1,1`}.

These REs can be simplified using run-length encoding: `cout` is $(0^3 1^1 0^1 1^3)+$. Execution walks these RE patterns without expanding them to AoB form. Overhead of this symbolic manipulation is reduced by using larger symbols in the RE; rather than patterns of individual bits, the current prototype treats each

4096-bit *chunk* as a symbol. This also allows massively-parallel execution of gate operations over the bits within each chunk, and *applicative caching* can avoid recomputation when a chunk result is available from a prior computation.

1.3 Two `pint`-level Examples

Moving up to the `pint` level, consider the problem of computing the square root of the 16-bit value 29929, which is 173. Rather than using a conventional algorithm, this can be computed by squaring all 8-bit values and selecting only the values that produced 29929. The complete code is simply:

```
int main(int argc, char **argv) {
  pint_init();
  pint a = pint_mk(16, 29929);  // 16-pbit value 29929
  pint b = pint_h(8, 0xff);     // H(0) .. H(7)
  pint c = pint_mul(b, b);      // square them
  pint d = pint_eq(c, a);       // where square equals 29929
  pint e = pint_mul(d, b);      // make non-sqrts all 0
  pint_measure(e);              // prints 0, 173
}
```

Notice that multiplying two 8-bit values naturally produces a 16-bit result (which here is 8-way entangled). This *pint* computation is implemented by just 310 gate-level *pbit* operations. The obvious algorithm to find all factors of 221 is similar, but creates a 16-way entangled space from two 8-way entangled values:

```
int main(int argc, char **argv) {
  pint_init();
  pint a = pint_mk(8, 221);    // 8-pbit value 221=13*17
  pint b = pint_h(8, 0x00ff);  // H(0) .. H(7)
  pint c = pint_h(8, 0xff00);  // H(8) .. H(15)
  pint d = pint_mul(b, c);     // multiply them
  pint e = pint_eq(d, a);      // where product equals 221
  pint f = pint_mul(e, b);     // make non-factors all 0
  pint_measure(f);             // prints 0, 1, 13, 17, 221
}
```

The number of values found in this measurement trivially determines primality. Any value thus factored will list at least 0, 1, and itself; if that is all, the number is prime. If there are four listed, then the number is the square of a prime. If there are five, the number is the product of two primes – the prime factors. Of course, much more efficient algorithms are possible, but the elegance of this example is a compelling argument for investigating this model further.

The remainder of this paper discusses some of the more interesting aspects of the current implementation of the parallel bit pattern computing model.

2 The Prototype Implementation

The latest prototype implementation consists of 2,713 lines (59KB) of C source code, originally written by Dietz and significantly improved by Shafran in Spring 2020. There are expected to be five major layers in the implementation of this model, four of which are operational at this writing. The lowest level is the chunk management. Above that layer is the factored bit parallel (FBP) or pattern layer, which manages regular expression values of *pbit*s. The `pbit` layer is next, constructing optimized DAGs (directed acyclic graphs) for *pbit*-level computations. The `pint` layer handles arithmetic and other operations on variable-precision multi-*pbit* signed and unsigned integer values. The top layer, which is not yet complete, essentially wraps the `pint` layer in C++ constructs that allow `pint`s to be directly manipulated in a C++ program as though they were a built-in data type. These layers are described in the following subsections.

2.1 The Chunk Management Layer

As mentioned in Section 1.2, the REs are currently expressed as patterns of 4096-bit chunk values within a potential 4294967296-bit AoB representation for 32-way entanglement. The chunk management layer implements a pool for allocation of chunk data blocks in an aligned, contiguous, region of memory. The data is kept separate from the management structures to ensure optimal alignment with cache lines, page table entries, and parallel execution structures. In Spring 2020, Murphy began work on parallel evaluation within a CUDA GPU, but parallel execution is currently within the host processor.

Chunks are indexed by a hash table containing many buckets (to keep loading light), each of which heads a dual-linked list of hash entries. Each hash entry not only points at the corresponding chunk data, but also contains a reference count tracking how many higher-level structures still have live pointers to this chunk entry. Duplicate chunks are recognized and only unique live chunks are stored. Reusing chunk memory as soon as possible is intended to improve cache and translation lookaside buffer performance.

2.2 The Factored Bit Parallel (FBP) Pattern Layer

The representation of a *pbit* value as a regular expression in which chunks are the basic symbols is managed by the factored bit parallel (FBP) layer.

As Table 1 shows, many FBP operations still have worst-case complexity that is exponential. Note that, using a more conventional (e.g., AoV) model of computation, all of the *pbit* operations would have at least 2^{32} work complexity, and there would be more total work to perform because bit-serial optimizations[1] would not have been applied. In contrast, RE-based FBP makes 2^{32} an unlikely worst case. The 2^{20} limits come from operations acting only on the symbols within a regular expression, rather than operating (in parallel) on the 4096 bits in each chunk. Lower-entropy regular expressions and applicative caching of chunk operations make the expected complexities far lower; any operation with

Time	Work	Operations
1	1	`SWAP` gate; `ALL`, `ANY` reduction; non-destructive measurement
$1..2^{20}$	$1..2^{20}$	`DUP`; `POP`ulation count; simplify regular expression
$1..2^{20}$	$1..2^{32}$	`CSWAP`, `CCNOT`, `NOT`, `AND`, `OR`, `XOR` gates

Table 1. Complexities of 32-way entangled FBP operations with 4096-bit chunks

no more than 12-way entanglement takes unit time, and a symbol repeated N times in an FBP regular expression typically would be evaluated only once.

Contrast these complexities with a true quantum computer supporting 32-way entanglement (which none yet support). Complexity would be constants for `SWAP`, `CSWAP`, `CCNOT`, and `NOT`. However, the other operations are not directly implementable. In fact, the *no cloning theorem* implies implementing operations like `POP` or even non-destructive measurement is impossible. Many complex quantum algorithms, such as Shor's algorithm[5], owe their complexity to statistically approximating such operations (typically using phase interference).

2.3 The `pbit` Layer

Quantum computing compilation and/or simulation environments generally define, and expose to users, some simple syntax for expressing operations on *qubit*s: a "quantum assembly language." For example, Quil[6], OpenQASM[7], and cQASM[8] all implement similar syntax for specifying operations on *qubit*s. However, that approach is not well suited to specification of FBP operations. One problem is the mismatch between basic operations provided: the various quantum assembly languages all provide direct operations on quantum wave functions and only adiabatic logic gates, whereas FBP does not model wave functions at all and provides a variety of both adiabatic and conventional types of logic gates. However, there is a larger incompatibility: `pbit` layer operations are normally not textually represented in a program, nor are they static; aggressive optimization and `pbit` (register) allocation are done at runtime.

As is discussed in Section 2.4, the `pbit` layer is really intended to serve as an internal framework for just-in-time compilation and optimization of work specified at the `pint` level. When specifying a computation using sequences of operations on multi-bit integers, as was observed by Dietz[1], it is common that a very large fraction of the intermediate bit-level operations will end-up being unnecessary. Logic optimization can symbolically recognize and remove many of these operations at compile time without ever incurring the overhead of constructing and evaluating FBP data structures. Thus, the `pbit` layer is literally an optimizing compiler used to cheaply remove as many unnecessary operations as possible before causing any FBP-layer evaluation.

Although `pbit` operations should look a lot like the FBP operations that are used to implement whatever computation remains, there is no need to use every type of instruction that the underlying machine supports. The current `pbit` layer simplifies optimizations by decomposing all operations into `ANY`, `NOT`, `OR`, `AND`,

and `XOR`. The only constants available are 0, 1, and Hadamard superpositions for up to 32-way entangled *pbit*s (i.e., `H(0)` .. `H(31)`).

Various algebraic simplifications are performed on-the-fly as `pbit` expression DAGs are created. For example, `AND` of anything with the constant 1 does not create an AND gate, but returns the other operand. A few multi-level simplifications also are performed, such as removal of `NOT NOT` and recursive searches to see if an item being ORed or ANDed into a sequence of that operation has already been included – e.g., (a `AND` (b `AND` (a `AND` c))) becomes just (b `AND` (a `AND` c)). Every potential operation also has its operand order normalized and a new operation will only be generated if that normalized computation is not already an available expression.

Originally, to maximize the probability of finding available expressions, no `pbit` DAG operation created during the expression compilation process was ever deleted. However, the latest version greedily reclaims no-longer-referenced `pbit` data structures to reduce memory usage. When the `pbit` layer is initialized, only 0, 1, and the Hadamard superpositions are available, but the set of available expressions grows as calls are made to compile additional operations. When the value of a `pbit` is demanded, the DAG producing that value is evaluated by executing a simple bottom-up tree walk that decorates the DAG with the results from executing each operation using the FBP layer. Values shared between DAGs are evaluated only once because the first walk to visit a node decorates it with a pointer to the FBP result, thus making it the bottom node in that walk. The nodes that correspond to dead code are not reachable via any walk, hence they are never evaluated using FBPs.

2.4 The `pint` Layer

In most quantum computer programming systems, the next level up from the quantum assembly languages described in the previous section is one in which quantum computations are still specified at the level of individual operations on *qubit*s, but the quantum manipulations are embedded in a full-featured conventional programming language. For example, both IBM's Qiskit[9] and Microsoft's Q#[10] essentially add a variety of functions to existing languages to allow *qubit*-level specification of computations. Higher-level (e.g., integer) operations must be built using the primitive operations. The system described in the current work also augments a conventional language (C/C++), but the `pint` layer directly understands integer operations.

A `pint` is represented as a data structure which contains an array of `pbit` references, a current precision, and a flag specifying if the value is signed (as opposed to unsigned). All the usual integer operations are supported for `pint` containing from 1 to 32 `pbit`s.

Lowering operations on `pint` to operate on `pbit`s is a lot like lowering operations on integers to gate-level code operating on individual bits. Some multi-bit integer operations are trivially lowered to operations on individual bits. For example, bitwise `AND` of two `pint` values trivially produces a result using `AND`s of corresponding component `pbit` values from the two operands. Other operations

are significantly less straightforward. For example, addition of two `pint` values performs a sequence of `pbit` operations that is equivalent to implementing a ripple carry adder circuit. Multiply builds upon that to implement a purely combinatorial shift-and-add circuit.

The primary complication in implementing these `pint` operations at the `pbit` level lies in the fact that precision and signedness can dynamically vary. It does not make sense to bitwise `OR` values of different precision; the less precise one should be promoted to have the same number of `pbit`s as the more precise one. If two k-`pbit` `pint` values are added, the result generally has $k + 1$ `pbit`s. On the other hand, if the two unsigned integers being added are 0 and 1, only a single bit is needed to express that the result is 1. Implementation of `pint` operations involves a variety of automatic promotion and precision-minimization operations.

When the `pint` layer is initialized, all the layers below also are initialized. Operations on `pint` simply compile DAGs for the component `pbit` operations. At the end of a sequence of `pint` operations, a call to evaluate each `pint` will cause the component `pbit` DAGs to be evaluated and decorated with references to their FBP results. Arbitrarily complex intermediate steps combining `pint` values do not cause any computation until it is demanded by calling for evaluation of a particular `pint`, e.g., by measuring the value. Measurement results can be printed, but normally would be storing a single value into an ordinary `int` or all superposed values into an `int` array.

3 Conclusion

The current work begins by describing, and giving a few motivating examples for, the quantum-inspired parallel bit pattern model for energy-efficient execution using conventional computer hardware. The efficiency comes partly from extensive gate-level optimization implemented using just-in-time compilation, but also from use of symbolic computation on regular grammars to obtain the quantum-like property of a single operation on an entangled, superposed, value producing up to exponentially many results. The structure of a prototype implementation is also detailed.

Although the prototype system is operational, it is not yet complete: we are improving/debugging the system and implementing a C++ wrapper, and plan an open source release. We are working on offloading the massively-parallel evaluation of chunks to a GPU. Dietz also has created a greatly simplified parallel bit pattern computer architecture called *Tangled*, which provides coprocessor support for parallel AoB chunk operations and is being implemented in Verilog by the students taking his undergraduate CPE480 Computer Architecture course at the University of Kentucky in Fall 2020. In the more distant future, we envision compiler technology for automatic parallelization targeting this new model.

References

1. Dietz, Henry G.: How Low Can You Go?. 30th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2017), College Station, Texas, 8 pages (October 11, 2017)
2. Dietz, Henry: Parallel Bit Pattern Computing, IEEE 2019 Tenth International Green and Sustainable Computing Conference (IGSC), DOI: 10.1109/IGSC48788.2019.8957188 (2019)
3. Rieffel, E. and Polak, W.: An introduction to quantum computing for non-physicists, ACM Computing Surveys (CSUR), 32(3), 300-335 DOI: 10.1145/367701.367709 (2000)
4. Tucker, L. W. and Robertson, G. G.: Architecture and applications of the Connection Machine. IEEE Computer, Volume 21, Number 8, 26–38 (August 1988)
5. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring, Proceedings 35th Annual Symposium on Foundations of Computer Science. IEEE Comput. Soc. Press: 124–134 DOI: 10.1109/sfcs.1994.365700 (1994)
6. Smith, R.S.; Curtis, M.J.; and Zeng, W.J.: A practical quantum instruction set architecture. arXiv preprint arXiv:1608.03355 (2016)
7. Cross, A.W.; Bishop, L.S.; Smolin, J.A.; and Gambetta, J.M.: Open quantum assembly language. arXiv preprint arXiv:1707.03429 (July 13, 2017)
8. Khammassi, N.; Guerreschi, G. G.; Ashraf, I.; Hogaboam, J. W.; Almudever, C. G.; and Bertels, K.: cqasm v1. 0: Towards a common quantum assembly language. arXiv preprint arXiv:1805.09607 (2018).
9. Wille, R.; Van Meter, R.; and Y. Naveh: IBM's Qiskit Tool Chain: Working with and Developing for Real Quantum Computers. Design, Automation & Test in Europe Conference & Exhibition (DATE), Florence, Italy, 1234–1240 (2019)
10. Svore, K. M.; Geller, A.; Troyer, M.; Azariah, J.; Granade, C.; Heim, B.; Kliuchnikov, V.; Mykhailova, M.; Paz, A.; and Roetteler, M.: Q#: Enabling scalable quantum computing and development with a high-level domain-specific language. arXiv preprint arXiv:1803.00652 (2018)