GF(3) Bitslicing

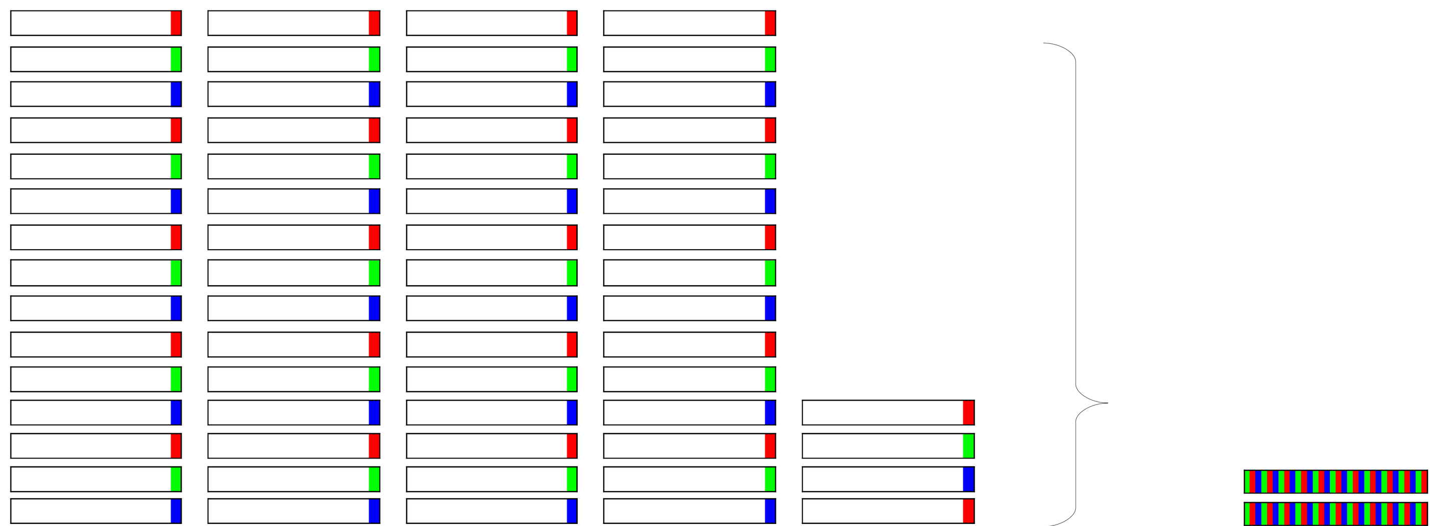
With Matrix Algorithms Oblivious to the Data Compression

Bryan Youse, B. David Saunders
University of Delaware, Newark, Delaware

Thanks to NSF Grant CCR-0515197

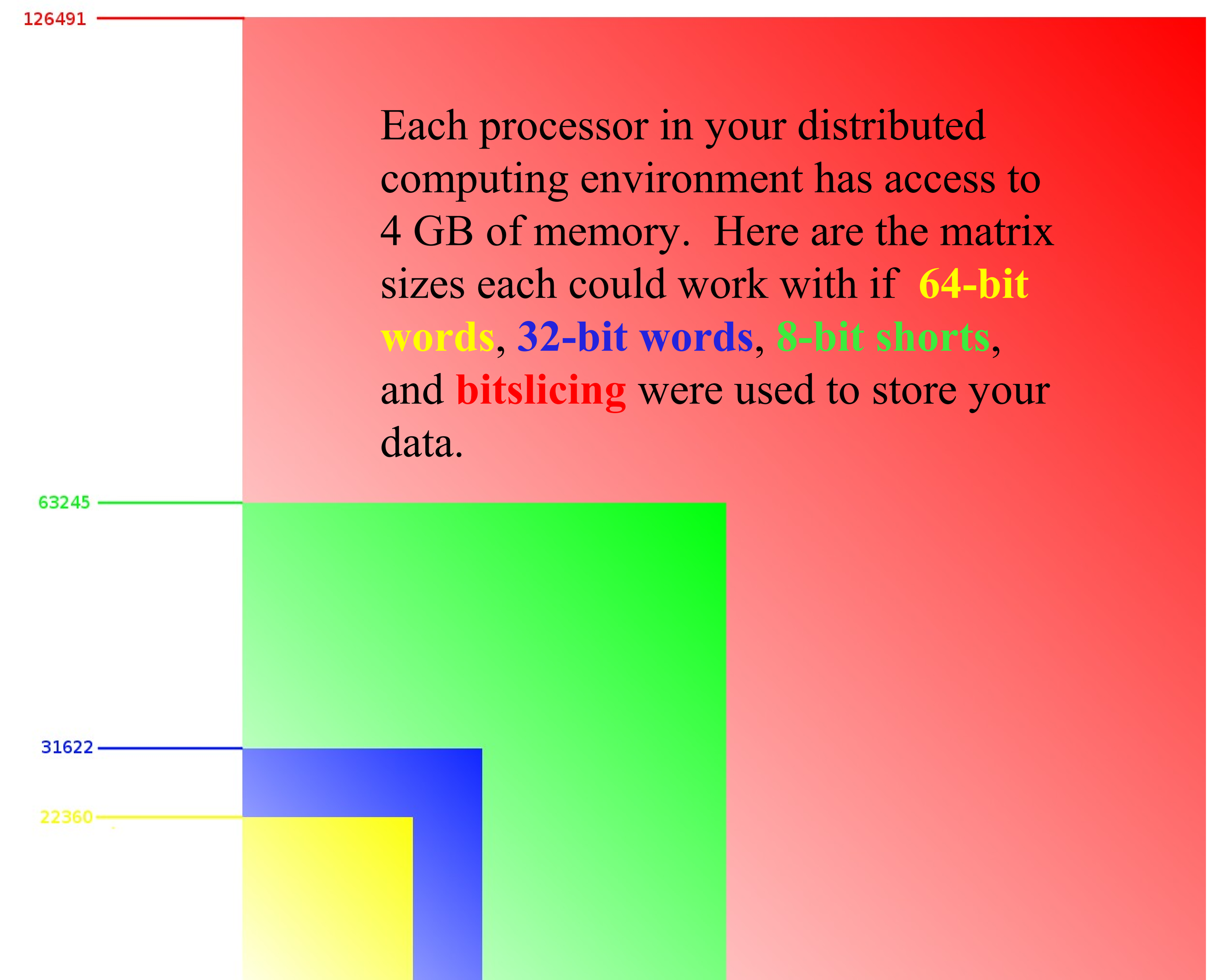
Technique

Field elements, represented by few bits, are striped across multiple machine words:



Only two bits are needed to represent elements in the field of integers mod 3. No carry bit(s) necessary due to specialized bit operations on the bit-vectors to accomplish field arithmetic. On 64-bit machines, compression ratio is thirty-two to one.

Perspective



Example

A "SlicedUnit", containing two machine words

index	0	1	2	...	18	19	20	21	22	23	...	61	62	63	
value	1	2	0	...	1	1	2	2	1	0	...	0	1	2	
bits (2^0)	1	1	0	...	1	1	1	1	1	0	...	0	1	1	machine word 1
bits (2^1)	0	1	0	...	0	0	1	1	0	0	...	0	0	1	machine word 2

This method can be generically applied to other small fields by adding more "slices" of bit-vectors. However, standard bit packing is better for any field other than F3. Bitslicing wins here due to the highly specialized arithmetic enabling the use of the best possible compression ratio: only two bits used to store each element.

Specializations

Our implementation of bit slicing for F_3 actually uses bit pattern "11" to represent field element 2. Addition now consists of just six bit-operations, which is an enormous benefit.

```
add(SlicedUnit lhs, SlicedUnit rhs)
  a = lhs.bit0 ^ rhs.bit1
  b = lhs.bit1 ^ rhs.bit0
  s = a ^ lhs.bit1
  t = b ^ rhs.bit1
  answer.bit0 = a & b
  answer.bit1 = s | t
```

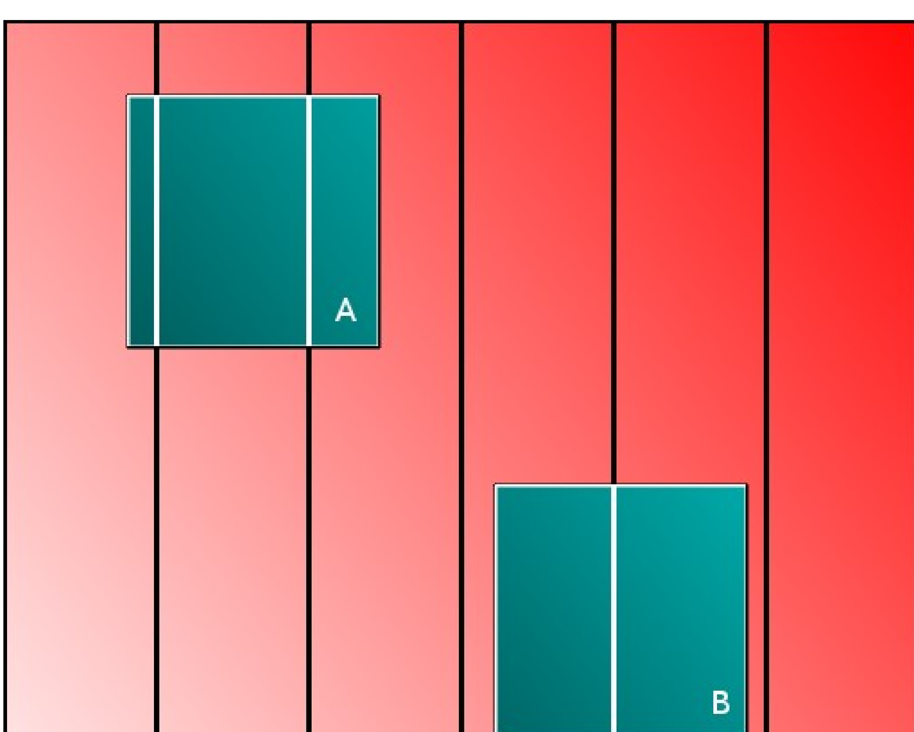
Multiplication is even more straightforward. Two is the only non-obvious multiplicand in this field, but it simply acts to flip-flop all non-zero values from one-to-two and vice-versa. Accomplishing this for a SlicedUnit is easy:

```
smul(SlicedUnit s)  //  multiply by two
  s.bit1 ^= s.bit0
```

[1]

Application

The immediate application is computing the rank of a large order matrix [2]. This calls for vector add, mul, and their tandem (axpy). Due to the size of the matrices, blocking and sub-matrixing is needed for computation . This poses additional challenges to our compressed format:



The black lines represent SlicedUnit boundaries. Submatrices A and B illustrate the problem of unaligned data. If an algorithm called for the two matrices to added, for example, the naïve approach of iterating through each SlicedUnit in each matrix and adding them together would fail. The extra steps of aligning one of the two to be in agreement with the other would need to occur first. This boils down to bit shifting and masking. We are working on experiments to show the penalty for these necessities.

Outlook

Our aim is to implement a complete arithmetic suite for bitslicing over GF(3). In addition to the space efficiency outlined above, the computational benefits are enormous. Below is the speedup for the three implemented operations relative to existing implementations using uncompressed storage.

ADD: 74.99	MUL: 133.39	AXPY: 63.54

We were able to compute the rank of a 3^{14} order matrix, computing with blocks of size 2^{15}. We are working on a parallel implementation to compute the rank the next matrix in the sequence, of order 3^{16}. Goal, in progress: make the storage scheme invisible to high-level matrix algorithms, so they can enjoy the space- and time-saving benefits of bitslicing.

[1] Tomas J. Boothby and Robert W. Bradshaw. Bitslicing and the method of four russians over larger finite fields. CoRR, 2009.
[2] B. D. Saunders and B. Youse. Large matrix, small rank. In Proc. of ISSAC'09, pages 317–324. ACM Press, 2009.