GF GPU

Galois Field Arithmetic on GPU

Problem Description

Fault Tolerant Disk Systems

Large disk systems need to be able to tolerate disk failures. The naïve solution would be to have duplicate copies of the data to be stored. However, in order to tolerate any two disk failures, each piece of data would require an extra two copies to be stored, effectively tripling the cost of data storage.

Using intelligent erasure codes, we can ensure full data recovery with only a nominal increase in data storage cost. To tolerate any single disk failure, we can add a single disk to the system in order to increase our tolerance. The simplest code would store the XOR of the data disks on the additional disk allowing for any failed disk's data to be recalculated from the remaining information. The problem lies in scaling this solution's tolerance beyond a single disk. In order to properly generalize the method, we require a finite, closed algebra so that we can guarantee unique inverses to multiplications, thus ensuring the division is possible. Galois Fields provide exactly the solution we require.

Galois Fields

In *Galois Fields*, addition is simply binary XOR; multiplication is a bit more complicated, however. Given two numbers *a* and *b*, we can multiply them as normal to begin with. However, there is no guarantee that the product is contained within the finite field. The product is then reduced by XORing a predetermined irreducible polynomial shifted to align with the highest non-zero bit in the value. The process is repeated until the end result is a number wholly contained within the field size.

In order to utilize these mathematics methods, we construct a coding matrix using an upper identity (to preserve the data being stored) and a sequence of coding coefficients below. This matrix is then multiplied onto sectors of data in vector format to produce the coding sectors in the bottom of the output vector. By storing the coding sectors on the added disks, the desired redundancy is achieved. Recovery of missing data involves the inversion of a submatrix of the coding matrix determined by the remaining disks and a similar matrix vector product.

The process of encrypting or decrypting these coding sectors is essentially a matrix-vector multiplication problem. However, due to the complex nature of the arithmetic in use, the usual optimizations for matrix operations cannot be applied easily. Most systems convert the matrix-vector operations into a sequence of single element operations, often times multiplying entire disks by a single value at a time. This parallel problem is one that takes a serial processor a substantial length of time to calculate. We believe that the highly parallel nature of a GPU will lend itself well to Galois Field arithmetic for use in erasure coding systems.

Serial Solution

In our previous experience, we have found the GF-Complete library to be one of the highest performing CPU libraries to implement Galois Field arithmetic. The library has several methods in which it performs the desired calculation. The following provide the basics for the methods implemented in GF-Complete[1].

SHIFT

The most naïve method of performing Galois Field multiplication, SHIFT is also the simplest to implement. The product of two values, *a* and *b*, are calculated as normal with the exception that addition is replaced with XOR to combine the products of each single multiplication. This is called a *carry-less multiplication*. The result, if too large initially, is then reduced into the field size by XORing the given irreducible polynomial shifted to align with the highest non-zero bit in the product. This process of shifting the polynomial is repeated until the resulting value is wholly contained within the field size. The following is a pseudocode example:

```
for i = 0 to w-1:                       // Iterate over all bits in a
    if the i-th bit of a is set:
        answer ^= b << i
for i = 2w-2 down to w:                 // Iterate over overflow bits in answer
    if the i-th bit of answer is set:
        answer ^= IP << i
```

GROUP

GROUP groups together bits and uses a small lookup table with bit shifting to perform multiple steps of SHIFT. It does the same for reduction steps. The number of bits we simultaneously check during the multiplication and reduction steps can be configured separately.

BYTWO_p and BYTWO_b

In the SHIFT implementation, reduction occurs as the final step of computation. In both BYTWO implementations, the multiplication and reduction phases are interleaved. That is, one factor is added and then the answer is reduced. In BYTWO_p, the intermediate products are multiplied by 2 before reducing. In BYTWO_b, however, *b* is incrementally multiplied by 2 rather than the product. The key difference between the two is that BYTWO_b will end after the highest set bit is checked. This means that it will end without needing to check every bit as in BYTWO_p. This is extremely fast if multiplying by 2.

TABLE

Since we have a finite amount of elements, we can compute every possible multiplication ahead of time and simply lookup the answers when needed. The problem with this method is the large amount of space required to store the table so it is only feasible with small word sizes.

LOG TABLE

By exploiting the laws of logarithms, we can exchange the weakness of the TABLE method's size with a few simple arithmetic operations. The product of values *a* and *b* can be expressed as $a*b=2^{lg(a)+lg(b)}$. Given the well behaved properties of these finite fields, calculating the tables of logarithms and antilogarithms can be done quite easily by exploiting generators of the field. With these tables precomputed, a single multiplication is reduced into three table lookups and a standard addition. Unlike TABLE's storage requirements of $w*2^{2w}$ bits, the storage requirements of LOG TABLE are $3w*2^{w}$ bits of storage space, a very significant improvement.

SPLIT TABLE

In much the same way that polynomials can be multiplied together, single numbers can be multiplied. By breaking a value into constituent parts, these smaller parts can be multiplied together and recombined to allow for several smaller multiplications in place of a single larger multiplication. For instance,

$$42*73=(00101010)*(01001001)=(0010*2^4+1010)*(0100*2^4+1001)$$
$$=(0010*0100)*2^8+(0010*1001)*2^4+(1010*0100)*2^4+(1010*1001)$$

By using a table for a smaller field size, we can use smaller multiplications to solve a much larger problem. In this example, we have reduced the need for a table of order 2^{16} into 2 tables (one for the high 4-bits and one for the low 4-bits) of order 2^{12}. This method is one that sees a very favorable speed increase in higher order field sizes but is lacking when the field size is already fairly small.

Related Work

As previously stated, this project seeks to recreate the results of GF-Complete by utilizing the GPU as an accelerator, rather than relying solely on the CPU's capabilities. GF-Complete used SSE instructions for parallel table lookups for several methods and achieved incredible results because of it. While the repository for the library is no longer available, the code and documentation has been archived[2]. As of this moment, we are unaware of any currently available Galois Field solutions utilizing GPU accelerators.

Parallelization Strategy

The process of using Galois Field arithmetic is already very straight forward. The entire process amounts to multiplying vast data arrays by a single value, albeit with a non-standard definition of multiplication. Since the desired result is so direct, there is no need for complicated codes to avoid the typical parallel pitfalls; the problems don't exist. As a result, a lot of the GPU kernels are similar to their serial counterparts. As a general rule, each thread handles a single data point in the given data arrays and performed the requested multiplication method on the single point.

Parallel Optimization

SHIFT

In each original shift implementation, we loop over every bit in the given constant *c* and check whether that bit is set or not. If it is, we do the XOR operation. Instead, we tried to optimize by having each block's thread 0 check at the beginning and store the location of set bits in *c* to an array. Now the for loop can just loop over this array and XOR each time. This is in hopes that the number of set bits in *c* is less than the total number of bits in *c*.

In practice, the performance increase would depend on the constant that is used in the carry-less multiply. If the constant is some power of 2, it would have only one bit set high in its binary representation. This is the best case scenario as it reduces the number of bits to calculate for from 8, 16, or 32 to 1. In contrast, the worst case scenario is when every bit in *c* is set and the "average" case scenario would be half of the bits in *c* set.

We found that the resulting performance is somewhat better for the best case scenario, but actually worse for the worst case scenario. In the average case scenario, performance was negligibly better. Overall this optimization might not be worth doing since it results in poorer performance half of the time.

SPLIT/TABLE

Both split and table implementations simply do table lookups, which gives them their huge speed when unoptimized. In both cases, it's obvious to try an use shared memory to optimize the table lookup instead of accessing global memory for every element. This is

especially apparent in the split implementations, because as word size increases, the number of table lookups increases. Additionally in split, we changed the multiple separate XOR operations into a single long chain of XOR operations. The idea behind this was that the compiler would optimize the chains of operations into single register values before running the kernel.

Using shared memory resulted in great speed and throughput increases for both table and split. Split with 32-bit words increased from about 15.6 GB/s to just under 30 GB/s, nearly double the throughput.

BYTWO

The original BYTWO_p implementation contains an if/else block in its main loop. We reasoned that this block would cause divergence. Even though there is only a single instruction in either side of the branch, transforming the block into a single logic calculation would totally remove any divergence caused by that block. Consequently this would decrease execution time. We first tried this calculation as a replacement:

```
prod = prod << 1 ^ IP08 * (prod >> 7);
```

The 16- and 32-bit word kernels used appropriate substitutions for *IP08* and the 7-bit right shift. However, this resulted in slower speeds for the 8- and 16-bit kernels. The 32-bit kernel had approximately the same performance.

We didn't understand why this was the case. Researching bitwise operations on GPUs resulted in many examples of it being faster than branching. We then hypothesized that the multiplication of the irreducible polynomial and the shifted *prod* caused a slowdown. That is, we guessed the integer multiplication was slower than the original bit-shifting in branches. Since we only used the multiplication as a “mask”, that is we only wanted a result of 0 or 1 from it, we replaced it with signed bit-shifting and bitwise AND:

```
int8_t sprod;
//...
sprod = prod;
prod = prod << 1 ^ IP08 & (sprod >> 7);
```

This did not change the speed for 8-bit words. However, 16-bit word speeds now matched or slightly exceeded the speed when using the if/else block. And again, 32-bit speeds were approximately the same.

LOG TABLE

For LOG TABLE, we attempted using shared memory to hold the log and antilog tables. However, only the tables for 8-bit word sizes would fit. For 8-bit words, we need $2^8 = 256$ elements for both tables, which easily fits in shared memory since each element is a single byte in size. However, for 16- and 32-bit words, this becomes unwieldy. These would require 2^{16} and 2^{32} elements, but the elements are 2 and 4 bytes in size, respectively. This would equate to 128 KB and 16 GB, which are far too large to fit in shared memory. However, using shared memory for 8-bit words did improve throughput

substantially.

Additionally, we tried to remove the modulus operator in the main kernel. This operator was included to make sure that the index into the *antilog* tables was not past the end of the table. We instead made the *antilog* table twice as large. Each “half” of the new doubled tables would contain the same data. Since the indexes would only be calculated up to 510, we could safely index any number without modulus. However the overhead of setting up the new tables caused quite a bit of divergence and overall slowed down the kernel by quite a bit.

GROUP

GROUP was relatively simple to optimize, though it was difficult to extensively optimize. We were easily able to move the two tables into shared memory since they were each only 16 elements long. We were concerned that having only 16 threads loading elements into shared memory would cause divergence with our blocks of 512 threads. However it did result in a moderate increase in throughput.

Since the two loops inside the GROUP kernels are constant time, we decided to try using one of CUDA's compiler optimizations: `#pragma unroll`. By placing this above a loop that runs over a constant, compile-time-known range, we can force the compiler to unroll the loop. That is, it will simply compile as if we had copy-pasted the instructions inside the loop multiple times. However we saw no speed increase from this, and found that the compiler was already making this optimization for us.

Evaluation and Results

The figures below show the performance comparison between CPU-bound GF-Complete, naïve GF GPU implementations, and optimized GF GPU implementations. GF-Complete was run on an Intel Core i7 3770 @ 3.40 GHz and GF GPU kernels were run on an Nvidia Tesla C2050. The comparison is the average throughput for each implementation using the same test case. Each test on the GPU used the following parameters for consistency:

- Constant c = 128
- Random data generation *seed* = 10
- Data size in *bytes* = 50,000,000 (50 MB)

Note that not all techniques are implemented at all word sizes. This is either due to it being missing in the case of GF-Complete, or the technique being too memory intensive for the GPUs at our disposal.

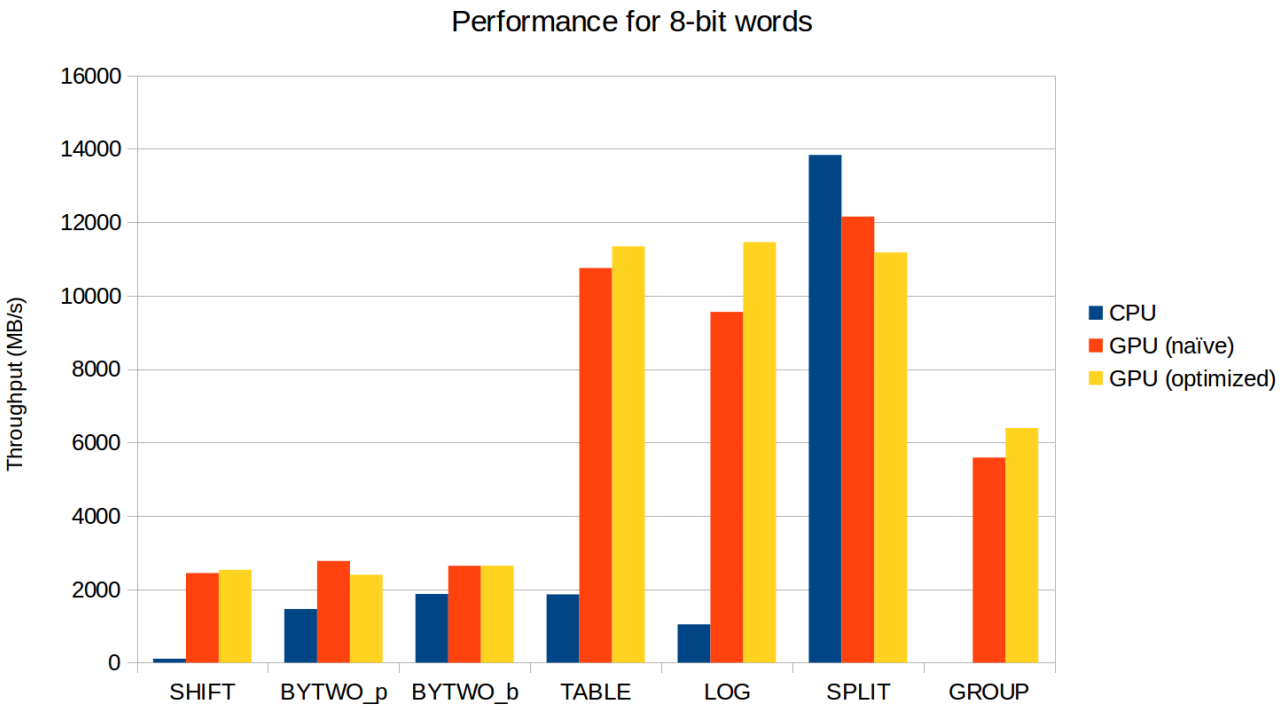

Figure 1: Throughput comparison for the 7 techniques with 8-bit words. Note that GROUP at word size 8 is not implemented in GF-Complete

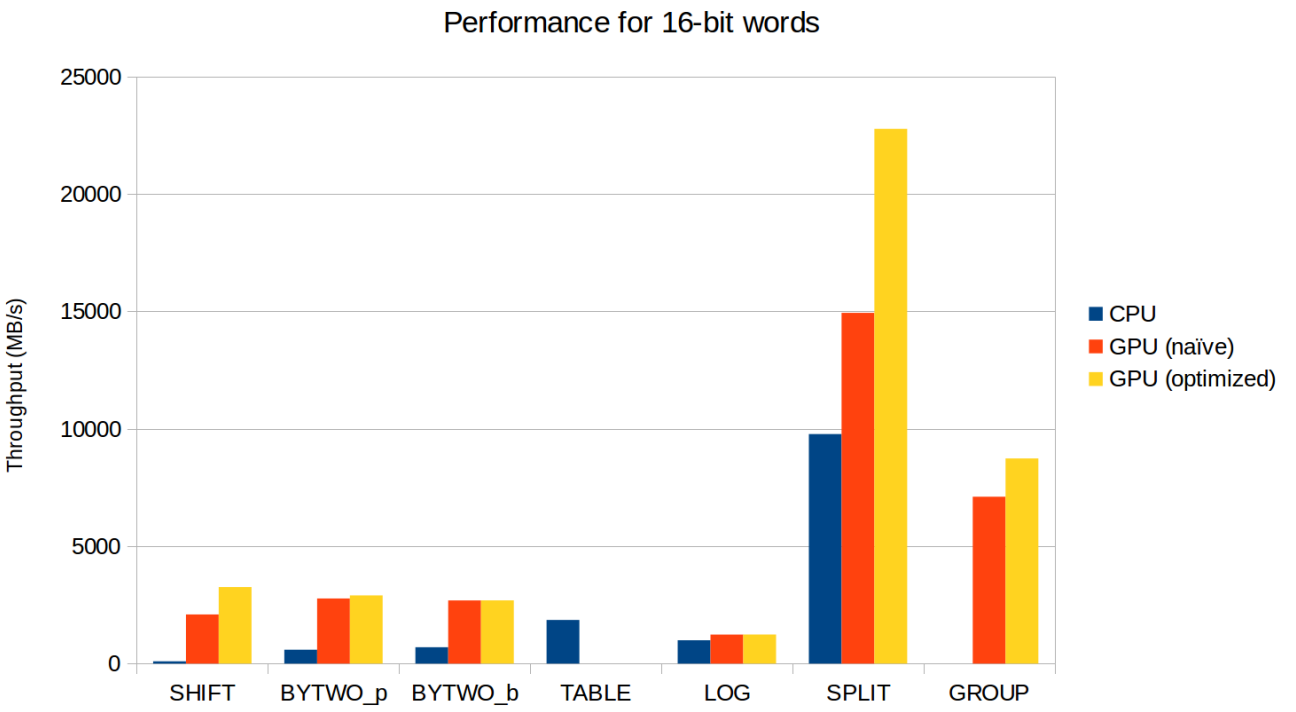

Figure 2: Throughput comparison for 16-bit words. Here the optimization for SPLIT takes better effect

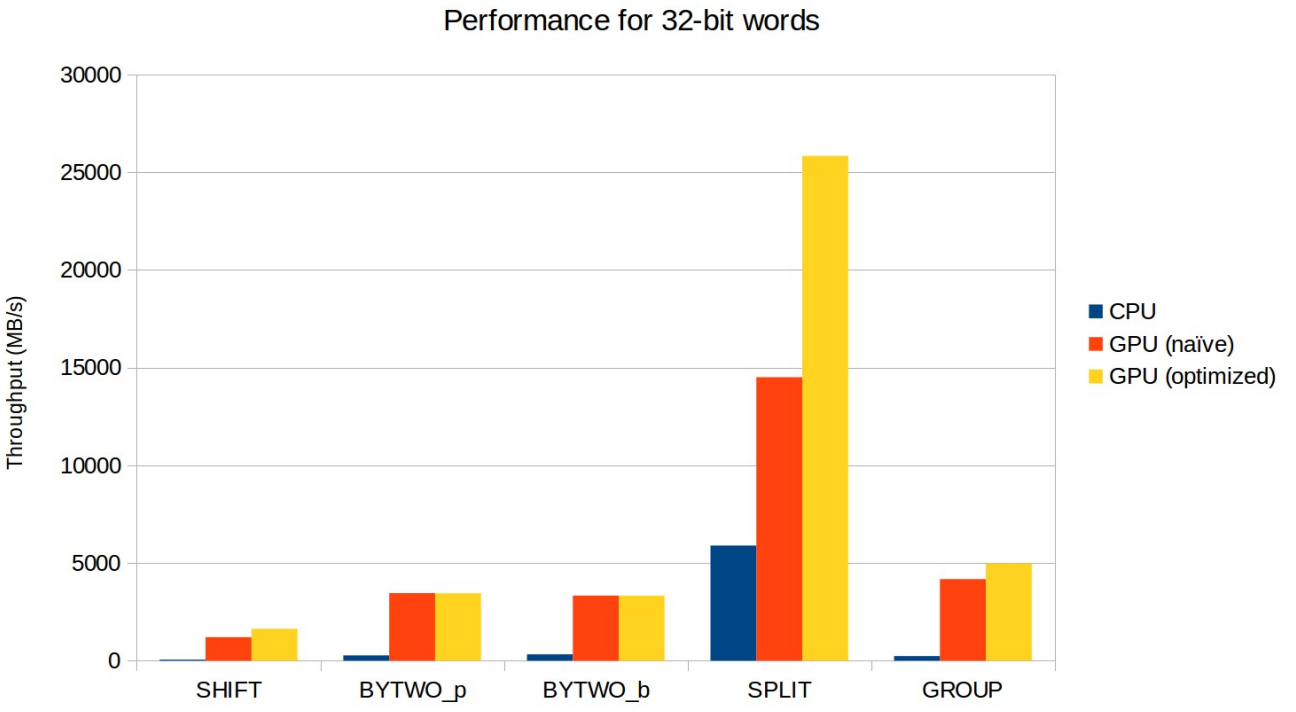

Figure 3: Throughput comparison for 32-bit words

The most immediately obvious result is that using a GPU, or a parallel implementation at all, greatly increases performance across the board. Even if the implementation is fairly naïve, we can still significantly decrease the time taken to encode large amounts of data. Optimizing the code has varying results depending on the technique being used. The clear example in the figures above are SPLIT. Optimizations for SPLIT show decreased performance for small word sizes, but outstanding increases at 16- and 32-bit word sizes.

Another note is that because we used average throughput as our comparison metric, we can produce differing results by increasing and decreasing the amount of data we encode. In these results we used 50 MB of generated data for our batch jobs. However, during testing, we ran some kernels with 1 GB of generated data and saw very high throughputs. For example, TABLE at 8-bit word size was capable of reaching approximately 14 GB/s, and SPLIT at 32-bit word size was capable of reaching just under 30 GB/s.

What these results show is that with a single GPU, we are capable of increasing data encoding performance by up to 5–10 times depending on the technique implemented. One of the reasons to use erasure codes is to decrease the amount of storage, and therefore monetary cost, needed for fault tolerance. Though adding a GPU would require a (relatively small) monetary cost, it greatly decreases the time expended on encoding data. For large disk arrays, the time saved will likely compensate for the GPU.

Conclusion

Galois Field arithmetic is a highly parallel problem that takes a serial CPU code quite a long time to execute under normal conditions. Though we can utilize SSE vectorized instructions on Intel CPUs to somewhat parallelize the data, we see a significant increase in performance by moving the computation to a GPU. Within the seven encoding techniques explored in GF GPU, we can see up to a 10 times increase in average throughput. Data encoding for fault tolerance is a very common practice and introducing GPU hardware to the workflow is a straightforward and relatively low-cost method to decrease time spent.

For future work, it would be of interest to implement CUDA streams to stream an entire disk worth of data to the GPU. Inspired by the large integer number multiplication and division project suggestion, we could possibly do the same for Galois Fields.

References

1. A Complete Treatment of Software Implementations of Finite Field Arithmetic for Erasure Coding Applications http://web.eecs.utk.edu/~plank/plank/papers/UT-CS-13-717.html
2. GF-Complete http://web.eecs.utk.edu/~plank/plank/www/software.html