Parallel Erasure Coding

Jiongtao Ye
Andrew ID: jiongtay

1 Summary

In this project, I implemented Erasure Coding, specifically Reed-Solomon Code (RS Code) in OpenMP on multi-core CPU platforms and in CUDA on GPU platforms. Detailed performance analysis was performed to quantify the speedup attained. The code resides in https://github.com/yejiongtao/parallel_ec.

2 Background

As introduced in the project proposal, Erasure Coding is a technique widely used in storage systems to provide data redundancy. Compared to traditional replications, Erasure Codes provide the same fault tolerance ability with much less space consumptions. As a tradeoff, when failure happens, the storage system enters the degrade mode, where it needs to calculate the lost data based on its related strides. Thus, the Erasure Coding calculation must be fast, to provide low latency for user requests in degrade mode. What's more, faster computation also means that given a time period, the system can do more computations of such kind, so the overhead brought by the erasure coding calculations will be smaller.

The algorithm of RS Code is described here. Firstly, the data is constructed into a matrix, called data matrix. Another matrix called coding matrix is constructed in a way that multiplying it with the data matrix gives a result matrix that contains the original data as well as the parity. As shown in Figure 1, multiplying a coding matrix of shape 6x4 and a data matrix of shape 4x4 results in a 6x4 matrix, whose first 4 rows are identical to the data matrix, and the latter 2 rows are parity.

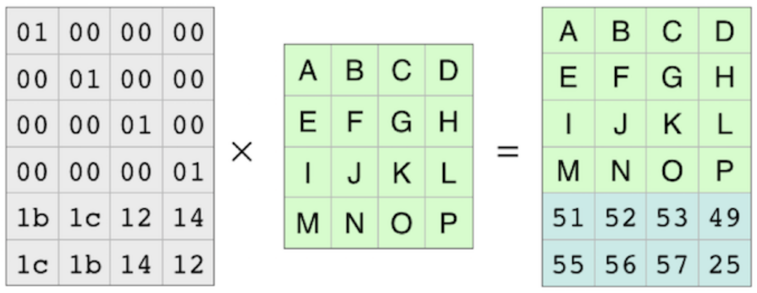

Figure 1, multiply coding matrix with data matrix. (https://www.backblaze.com/blog/reed-solomon/)

Upon data loss, some rows in the result matrix are lost. As in Figure 2, two rows are missed, but if you cross out the corresponding two rows in the coding matrix, the equation still holds.

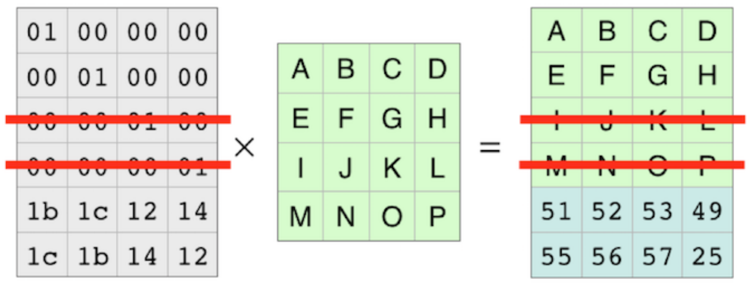

Figure 2, data loss. (https://www.backblaze.com/blog/reed-solomon/)

This brings us the approach to reconstruct the data matrix given the partial coding matrix and result matrix, which is to right-multiply the inverse of the partial coding matrix to both side of the equation, as shown in Figure 3. As a result, the left side of the equation becomes the original data matrix and we can obtain it using things at the right side of the equation. To ensure the remaining coding matrix is inversible, a Vandermonde matrix is used.

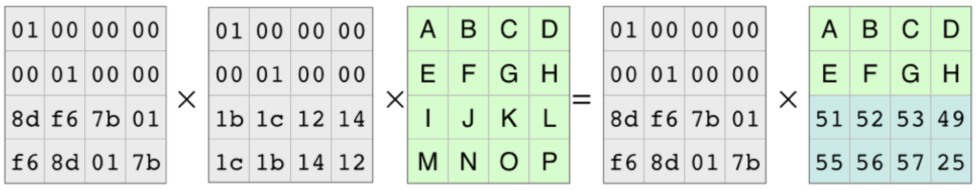

Figure 3, reconstruct data matrix from loss. (https://www.backblaze.com/blog/reed-solomon/)

Another concept that needs mentioning is Galois Field. The operations mentioned above all happen in a Galois Field, which adds some complexities to the implementation, as well as room for optimizations. For example, multiplications in Galois Field may require exponential and logarithmic operations, which are computationally expensive. In BlackBlaze's Java implementation of RS Code (https://github.com/Backblaze/JavaReedSolomon), which I'll use as reference, lookup tables are used for multiplication, logarithm and exponent operations in the Galois Field, instead of computing them every time. Such approach is also explored and analyzed in this project.

To summarize, the main task to optimize and parallelize is matrix multiplications in Galois Field. What's more, the characteristics of the workload, e.g. the matrix sizes, also influence the optimization decisions and the parallel performance, which will be elaborated soon.

3 Approach

In BlackBlaze's sequential Java implementation (https://github.com/Backblaze/JavaReedSolomon), they implement various loop orderings for matrix multiplication, claiming that different orderings would have different performance on different machines. As in Figure 4, there're three loops in a matrix multiplication operation: looping over all rows of the left matrix, all columns of the right matrix, and all elements in the related row and column. For simplicity, we call them num_output, num_byte and num_input respectively. A typical matrix multiplication implementation would iterate num_output in the outermost loop, num_byte in the middle, and num_input in the innermost loop. However, the three loops can actually be in any order, yielding 6 different orderings in total.

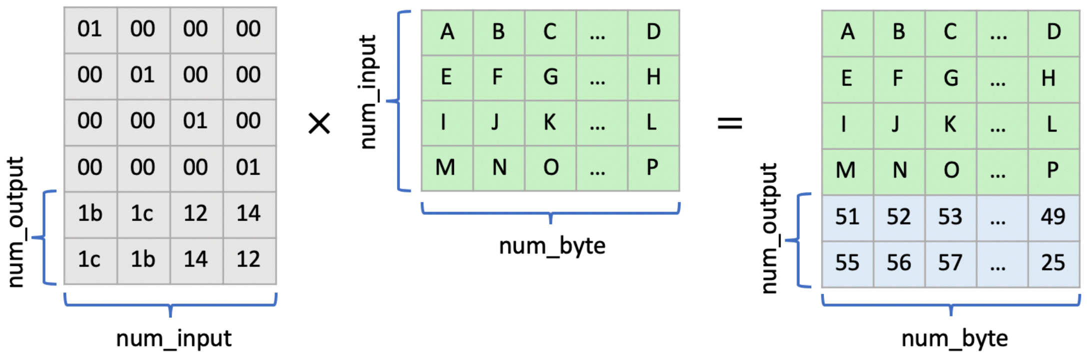

Figure 4, demonstration of loop orderings of matrix multiplication.

As the first part of the project, I re-implement the program in C language, trying out all the loop orderings. Not surprisingly, on GHC clusters, those with num_byte in the innermost loop result in the best performance. The reason is that in a typical RS Coding process, num_input and num_output are relatively small, e.g. num_input = 17 and num_output = 3 for a RS(17, 3) code, while num_byte is much larger, usually of the scale of hundreds of thousands. Adding the fact that the matrix is represented in a row-major manner, as a result, having num_byte in the innermost loop gives the best cache locality and thus the best performance. Detailed experiment results are listed in the next section.

The second step of the project is to parallelize the program with OpenMP. I choose the best loop ordering as baseline, where num_byte is in the innermost loop. Since num_byte is large, I parallelize the program by splitting num_byte into multiple chunks and assign them to different compute units, as demonstrated in Figure 5 below.

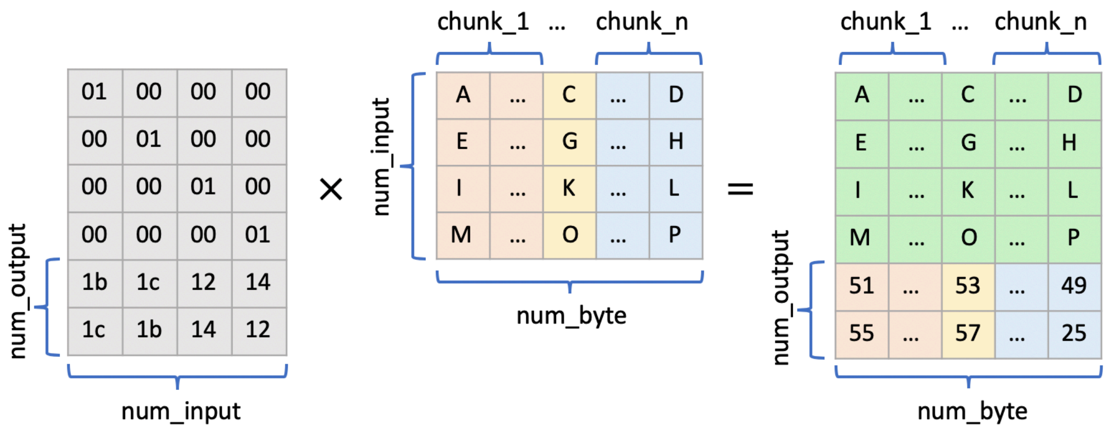

Figure 5, demonstration of chunk splits.

I first try exploiting SIMD, which is supported since OpenMP 4.0. The result shows no performance improvements compared to the sequential baseline which is compiled with -O3. The reason is modern compilers like GCC already attempt to utilize SIMD. Apparently for this particular program, its attempt to use SIMD is successful because there's not much inter-loop dependencies. Then I try using OpenMP for multi-threading, by assigning each chunk to different threads. The speedup is promising for the multi-threading version. See next section for detailed analysis.

As a third step, I implement a CUDA version. Putting it simply, each CUDA thread is responsible for a chunk of consecutive bytes. For analysis, I explore the influences of different chunk size and data size. I also analyze the cost of different parts of the CUDA program, whose interesting results will be shown later.

4 Results

4.1 Experiment Setup

Experiments are conducted on the GHC cluster. The machine is equipped with an eight-core, 3.2 GHz Intel Core i7 processor, with support of AVX2 vector instructions. GCC 7.3.0, with support for OpenMP 4.5.0, is used for compilation. Each of the machines also contains a NVIDIA GeForce GTX 1080 GPU, supporting CUDA compute capability 6.1. Unlike Latedays, GHC machines are of shared use, so multiple runs are conducted when evaluating performance.

For most of the experiments, RS(17, 3) is used, and the num_byte of the input data is 200,000 bytes. For some cases, I increase num_byte to 10 x 200,000 and 100 x 200,000 to analyze the influences on performance of the input size.

4.2 Loop Ordering and Lookup Table

Figure 6 shows the results for different loop orderings. For example, Byte_Input_Output means iterating num_byte in the outermost loop, num_input in the middle, and num_output in the innermost. Besides loop orderings, the influences of lookup tables, inline functions, and different programming languages are also explored here.

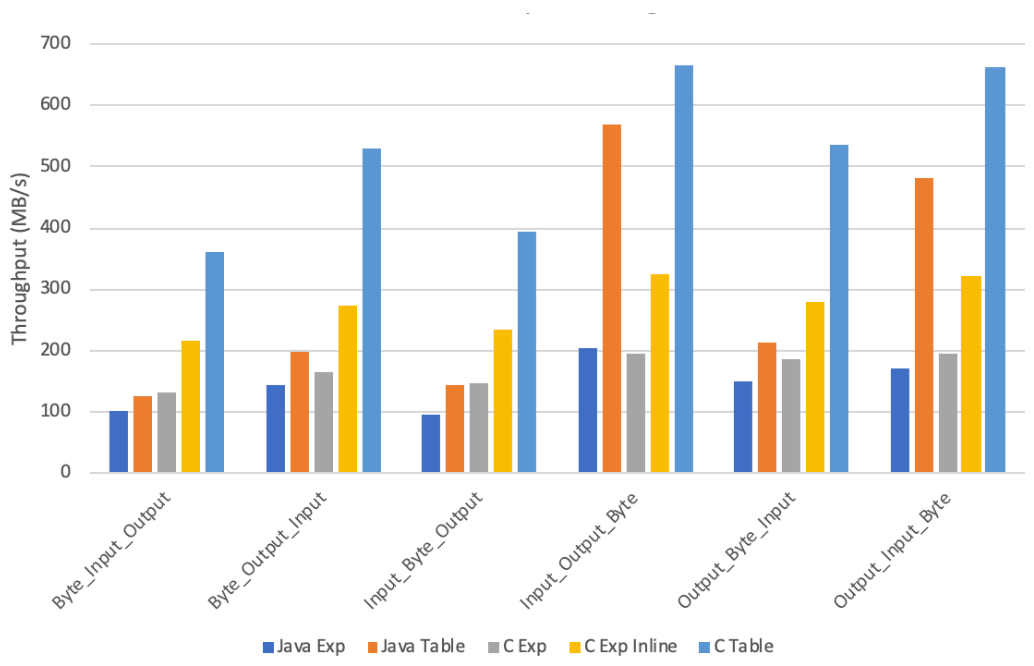

Figure 6, effects of loop orderings, lookup tables, inline functions and programming languages.

A number of interesting observations can be made from the figure. For each observation, I attempt to rationalize it with plausible explanations:

1) The performance of a specific loop ordering is mainly dependent on the data size of the innermost loop. Because the larger the innermost loop, the more locality existed to utilize. In this case, num_byte is the largest, so Input_Output_Byte and Output_Input_Byte perform the best. In contrary, Input_Byte_Output and Byte_Input_Ouptut have the worst performance, due to the fact that num_output is the smallest.
2) Using a lookup table (Java Table, C Table in the figure) for multiplications in Galois Field brings much performance improvement, compared to those directly compute every time (Java Exp, C Exp, C Exp Inline).
3) Using inline functions in C improves performance by around 1/3.
4) With specific loop orderings with lookup tables, the C version doesn't outperform the Java version by a large degree (Input_Output_Byte and Output_Input_Byte). Since the C version is using SIMD, we can deduce that modern JVM tries to utilize SIMD as well. However, for other orderings, C program performs much better than Java.

4.3 Multi-threading with OpenMP

Parallelized using the approach described in the section 3, the program gains major performance boosts. Figure 7 demonstrates the performance of different number of threads. There're multiple aspects worth mentioning in the figure:

1) Because the inner loop is computationally intensive and doesn't have much inter-loop dependency, from 1 thread to 7 threads it shows a nearly linear speedup.
2) The GHC machines are equipped with an 8-core processor, supporting 2-way hyperthreading. In theory, running with 8 threads should give a very good performance. However, as in the figure, the performance goes down. There are a few reasons I can think of about how this could happen, including caching issue and the influence of other background threads. But most of them should happen in the same way for 16 threads as well, since in this system, using 8 threads is pretty similar to using 16 threads. Interestingly, as shown in the figure, the 16-thread version is not facing the same issue. I suspect it has something to do with the implementation details of OpenMP, which I haven't figure out yet.

3) When number of threads exceeds number of cores in the system, specifically, with 9 threads, the performance goes down. The reason is that two of the threads will have to run on the same processor. Although with hyperthreading, a computationally intensive program wouldn't benefit much from it. As a result, those two threads take a longer time to finish, dragging down the whole system's performance.
4) From 9 threads to 16 threads, the problem size assigned to each thread becomes smaller. In the meantime, each core has at most threads running simultaneously. Based on these two facts, the time each core spends gets shorter, giving an increase in performance.
5) Starting from 17 threads, the number of threads exceeds the number of computing units, including hyperthreading. OS has to switch in and out different threads, yielding a major performance decrease.

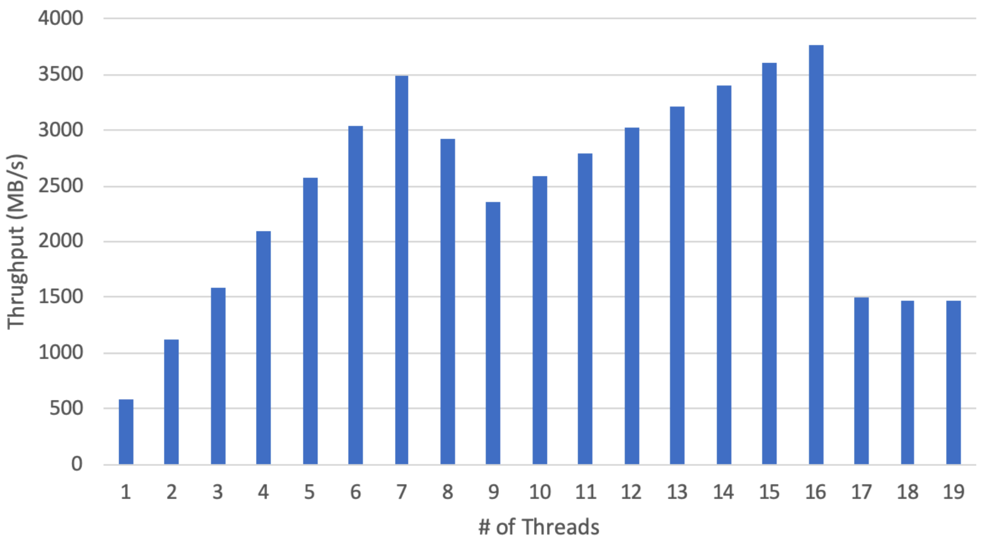

Figure 7, performance analysis of multi-threading.

4.4 CUDA on GPU

Figure 8 gives the performance comparison among the sequential, multi-threaded and CUDA implementations.

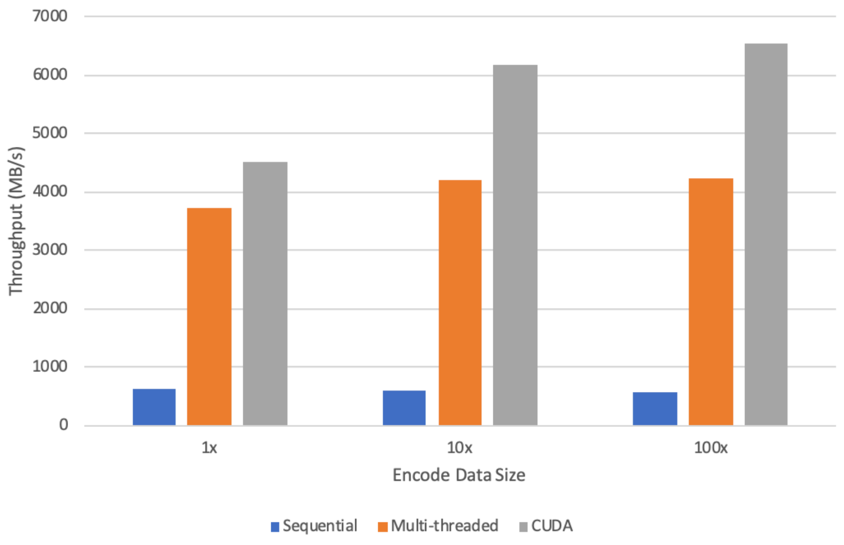

Figure 8, performance comparison of sequential, multi-threaded and CUDA versions.

We can see that for the default data size (1x), CUDA doesn't have much speedup from the multi-threaded version. The reason is shown in Figure 9, where the actual computation kernel only takes around 40% of the total time of the CUDA program. Memory operations like allocating space and copying data cost more than a half. According to Amdahl's Law, no matter how fast the kernel could be, the overall performance wouldn't improve much due to the memory operations.

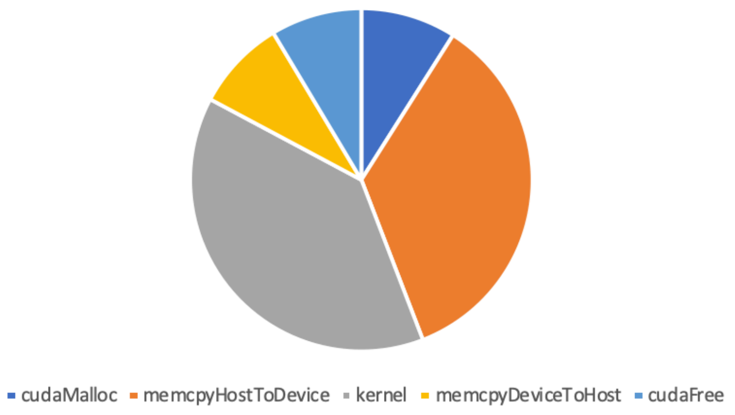

Figure 9, cost of different parts of the CUDA program.

Another insight from Figure 8 is that the performance of the CUDA program increases with larger input data size, for the reason that memory operations like allocations are done once for all the data. With larger input size, the percentage of time spent on those operations gets smaller, and thus more performance gain from the kernel.

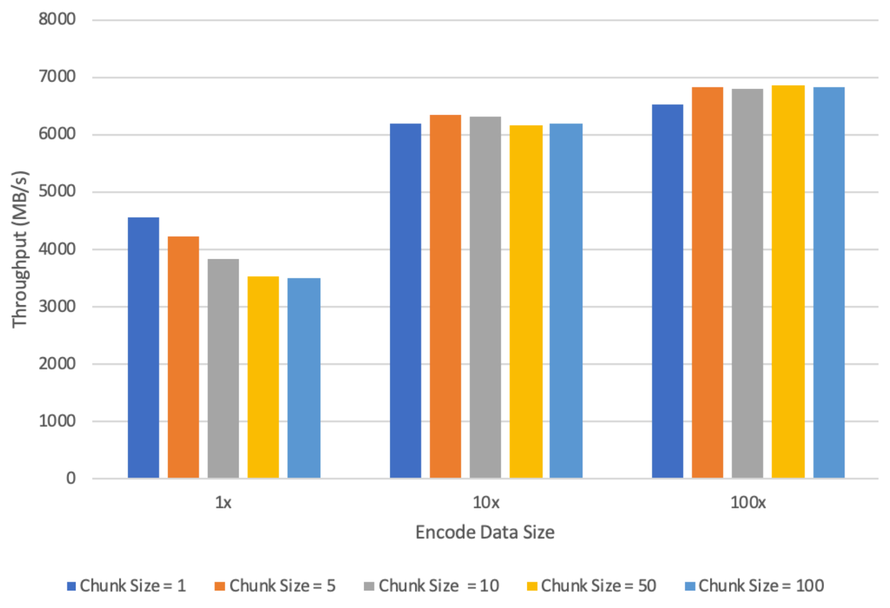

Figure 10, influences of chunk size.

Lastly, in Figure 10, I explore the influence of chunk size in the CUDA program. Chunk size refers to the width of a chunk assigned to a CUDA thread, as illustrated in Figure 5. With 1x input data size, smaller chunk size performs better, because the input data is relatively small, so with large chunk size, the total number of threads would be too small to utilize the compute units of GPU. With larger input data size, the optimal number of chunk size gets larger. The reason is now that the input size is large enough for parallelism, if the chunk size is too small, the overhead of scheduling will harm the overall performance.

5 References

BlackBlaze’s sequential Reed Solomon Code: https://github.com/Backblaze/JavaReedSolomon.
Introduction to RS code: https://www.backblaze.com/blog/reed-solomon/.