GPU HashSums: CRC32C
CS: 6230 High Performance Parallel Computing
Final Report

Akash Gangil
GTID: 902950416
agangil3@gatech.edu

Lup Peng, Loke
GTID: 903014478
luppeng.loke@gatech.edu

I. Introduction

A. Overview

Data being sent over physical mediums are susceptible to interference from external sources. These interferences may change the value of the bits being sent and may render the entire data set to be useless at the receiver end. For example, a change in a single bit of a cryptographic ciphertext may cause a cascading effect over the rest of the message after that point.

The Cyclic Redundancy Check (CRC) is used to detect accidental changes to raw data being transferred over a network medium.

B. CRC In A Nutshell

CRC works by using the remainder of a polynomial division as a checksum to verify that the content of the file is not erroneous after being transferred over the network.

The sender will first perform a polynomial division on the data using a predefined polynomial to obtain the remainder, known as the checksum. This checksum is sent together with the data to the receiver. After receiving the data, the receiver performs a polynomial division on the data with the same predefined polynomial to obtain its own checksum. The receivers checksum is then compared to the senders checksum to verify that the data has been received with no errors. If the checksums do not match, it can be assumed that the data is corrupted and actions can be taken to rectify it.

C. Issues with CRC and Network Speeds

With larger data sizes and faster network speeds, the calculation of the checksum becomes a bottleneck as the polynomial division is a non-trivial function. However, the CRC algorithm to derive the checksum can be computed in parallel on different parts of the data, and then combined together at the end. Having a parallel CRC implementation will help overcome this potential bottleneck by reducing the time needed to calculate the checksum on the sender and receiver side of the transmission.

II. Project Objectives

The objective of this project is to exploit the parallelism potential in the CRC algorithm. And compare the CRC calculation using the CRC32-MPEG and CRC32C polynomials by making use of various techniques like SIMD, OpenMP and CUDA on the Jinx Cluster in the the College of Computing at Georgia Tech.

III. Project Breakdown

1) *Preliminary Analysis and Understanding*
   During the initial phase, we will start by understanding the calculation of the CRC32C and CRC32-MPEG checksums, the different standards and how they are implemented. The various approaches will be analyzed and areas of potential parallelization will be identified.

2) *Phase 1: Determine Baseline CPU Performance*
   After understanding the hash sums, we will develop a naive CPU implementation of the CRC32C and CRC32-MPEG hash sums.

3) *Phase 2: Parallelization of CPU Implementation*
   We would then be identifying the areas in the algorithm that we can parallelize.

4) *Phase 3: Evaluating Intel CRC32 Instruction*
   In this phase we would be trying to use Intel's CRC32 instruction for computing the checksum.

5) *Phase 4: Using OpenMP for speedup*
   We would be using OpenMP to further improve the performance of the naive CPU implementation.

6) *Phase 5: Comparing other CRC32 Alternatives*
   Over here we would be evaluating a bunch of other libraries and programs which provided CRC calculation mechanisms like the Java library.

7) *Phase 6: GPU Implementation*
   We would be trying to port the CRC calculation onto

the GPU to see if we can gain further performance improvements.

Our approaches, methods and results for each phase of the project will be documented in the following sections. At the end of the report, we will analyze the data and results and come to a conclusion on our project. Source code has been added to the repository. Please check the appendix for the repository url.

IV. Project Stages

A. Preliminary Stage

1) CRC Standards: CRC is the general name of the approach to use the remainder of a polynomial division to check if a file has been changed after being transmitted. There are many standards of CRC which use different polynomials as the divisor.

2) Polynomials: Polynomials are usually written in hex for simplicity and practical uses within the program. For our project scope, the polynomials for the CRC-32C and CRC-32-MPEG standard are 0x1EDC6F41 and 0x04C11DB7 respectively.

When converted to binary, each bit of the binary representation determines the coefficient of the polynomial (either a 1 or 0). For example, for the CRC-32C polynomial, the hex representation would be:

0x1EDC6F41

and the binary representation would be:

1110 1101 1100 0110 1111 0100 0001

thus the corresponding polynomial would be:

$$x^{32}+x^{28}+x^{27}+x^{26}+x^{25}+x^{23}+x^{22}+x^{20}+x^{19}+x^{18}+x^{14}+x^{13}+x^{11}+x^{10}+x^{9}+x^{8}+x^{6}+1$$

On the other hand, for CRC-32-MPEG, the hex presentation is:

0x04C11DB7

and the binary representation would be:

0000 0100 1100 0001 0001 1101 1011 0111

thus the corresponding polynomial would be:

$$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x+1$$

Do note that the largest value of x (i.e. x32) is not included in the polynomial and during calculations. However, it is added implicitly in the polynomial representation in order to make it have the degree of 32.

3) Reversed Representation With Regard to Endian: During this phase of the project, we noticed that there were many CRC implementations that use a reversed version of the polynomial. In the above sub-section, our polynomials were not reversed. I.e. the left most bit is the Most Significant Bit (MSB) and the right-most bit is the Least Significant Bit (LSB).

The decision to use the reversed polynomial or not depends on the system and the way it orders/stores data in the memory, or its Endianness. There are two types of Endianness: in the Big Endian system, the Most Significant Byte is stored in the first/smallest memory address, and in the Little Endian system, the Most Significant Byte is stored in the last/biggest memory address.

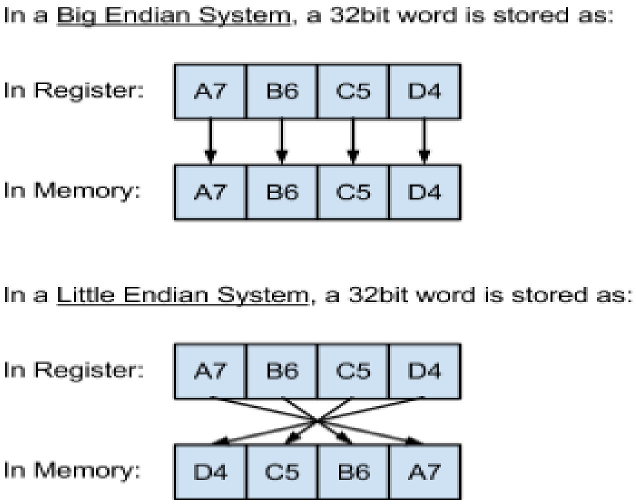

In a Big Endian System, a 32bit word is stored as:
In Register:
A7
B6
C5
D4
In Memory:
A7
B6
C5
D4
In a Little Endian System, a 32bit word is stored as:
In Register:
A7
B6
C5
D4
In Memory:
D4
C5
B6
A7

Fig. 1. Illustration of how a 32bit word is stored in different Endian systems

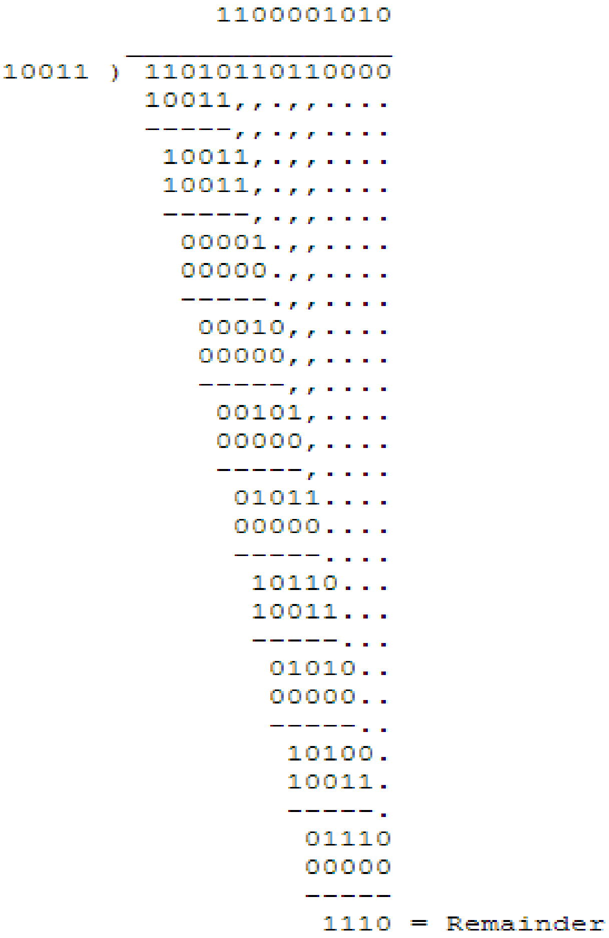
For a Big Endian system, the normal polynomial is used, and for a Little Endian system, a reversed polynomial is used. This is so that the MSB of the polynomial is always divided by the first byte of the data.

The Endianness of the system is determined by the processor. For Jinx, it uses Intel x86 processors, which are Little Endian systems. In order for the project to be as robust as possible, it uses the normal polynomials but with a reflection step on the data before being used to calculate the CRC. We use this approach such that we are able to cater to both Little Endian and Big Endian systems by simply enabling or disabling the reflection step before and after the CRC computation.

4) Approaches to Calculate CRC Sequentially: There is no easy way to calculate the remainder of a polynomial division, as every division is dependent on the remainder of the previous division. We have derived three sequential ways to calculate the CRC checksum given data and a divisor.

1) *Bit-by-Bit* In this approach, we first take x number of bits of the data and divide it with the polynomial, where x is the number of bits for the divisor polynomial.

After which, we add y number of bits to the remainder such that: y + remainder bits = x bits. We keep doing this until the remaining number of bits in the data is less than the divisor polynomial. This value is the CRC checksum.



```
            1100001010
        ______________
10011 ) 11010110110000
        10011,,.,,....
        -----,,.,,....
         10011,.,,....
         10011,.,,....
         -----,.,,....
          00001.,,....
          00000.,,....
          -----.,,....
           00010,,....
           00000,,....
           -----,,....
            00101,....
            00000,....
            -----,....
             01011....
             00000....
             -----....
              10110...
              10011...
              -----...
               01010..
               00000..
               -----..
                10100.
                10011.
                -----.
                 01110
                 00000
                 -----
                  1110 = Remainder
```

Fig. 2. Example of Polynomial Division. In this case, the data is the dividend and the CRC polynomial is the divisor. The Remainder is the CRC Checksum

2) *Byte-by-Byte/Table Lookup* This method is an improvement from the previous - instead of 1 bit, we take 1 byte (or 8 bits) at a time and do the polynomial division. When the division has run out of data bits, we take another 1 byte and continue with the division. While this approach still seems naive, it is a stepping stone to the third approach.

   Instead of computing the values during runtime, we precompute all possible remainders that can be derived from the polynomial divisor and store it in a lookup table. Using x bits of the data as the lookup table index, the algorithm checks the lookup table and obtains the polynomial division remainder.

   While this approach is fastest of the three, it requires more memory to store the lookup table during runtime.

3) *Intel PCLMULQDQ Instruction*

   In all 2010 Intel Core processors and onwards, there is a new Intel PCLMULQDQ instruction that does a carry-less multiplication of two 64-bit polynomials over the finite field GF(2). This new instruction allows polynomial multiplications to be done much more efficiently than previous instructions.

   While this new instruction was initially meant for the purposes of encryption, another usage could be for calculating the CRC Checksum of a data block. As this instruction is a hardware instruction that is designed specially for doing polynomial multiplication operations, it translates to much faster run time for calculating the CRC Checksum for our case. A SSE4 intrinsic named `_mm_crc32_u32` uses this instruction to calculate the crc.

5) Potential Areas for Parallelization : In Methods 1 and 2 illustrated above, it is almost impossible to parallelize the algorithm as every division depends on the remainder of the previous division. However, Method 3 does give a ray of hope - it may be possible to: 1) break up the data into smaller chunks, 2) calculate the CRC checksum on these chunks independently, and 3) finally combine the individual checksums back into a larger checksum that corresponds to the original data

B. Phase 1: Determine Baseline CPU Performance

In this section, we implement the above approaches and run them with different data sizes to get a baseline CPU performance that we can use as a benchmark for future experiments. We implemented the bit-by-bit and the table lookup (byte-by-byte) methods to calculate the CRC checksum as explained in the previous section. The functions can be found in `crc.c` in the source code repo.

Also to generate the messages of varying lengths we wrote a small C program `randomMessage.c` which would take in the size of the message in bytes and genrate the message in the file `input.in`. Its on this value the checksum is computed.

C. Phase 2: Parallelization of CPU Implementation

Identifying the parallel region in the CRC algorithm was probably the hardest part of our project. Initially we thought it to be naive but realized it wasn't that simple. Eventually, we follow the approach where we split the message into equal sized blocks and compute the CRC checksum on each of these blocks. We then combine these checksums to get the final CRC value for the entire message.

To combine the checksums of the subparts we used zlib's crc32_combine function which is used by many other libraries like Ruby. Though we found out that even this has certain limitations which we have noted down in our *Findings* section next.

D. Phase 3: Evaluating Intel CRC32 Instruction

We used Intel's SSE 4 intrinsic `_mm_crc32_u32` to calculate the CRC checksum of the message. Since it was a hardware supported instruction it showed good performance. Only downside was that it only supported CRC-32C-SCSI polynomial and there wasn't a way to configure that.

The source code for this is in `crc_intel.c`

E. Phase 4: Using OpenMP for speedup

In another of our evaluations we used OpenMP. We divided the message into blocks and made each thread compute the crc for those blocks. We then combined the resulting block crc's to get the crc for the final message.

From performance perspective, we were able to compute the CRC-32C-SCSI checksum in a lesser time by combining the intel instruction and OpenMP which helped us achieve better performance.

F. Phase 5: Comparing other CRC32 Alternatives

We checked the CRC32 alternative for two reasons:

1) To verify our computed CRC checksum gives the correct answer.
2) To benchmark against other implementations which allow users to compute CRC32C

We mainly compared Java's CRC32 implementation `CRC32.java` and zlib's CRC32 implementation `zlibCRC32.c`.

G. Phase 6: Simple GPU Implementation

To parallelize the calculation of the CRC checksum, we have used CUDA to take advantage of the computation power of the GPUs within the Jinx cluster. In our approach, we have a driver/main file on the CPU that reads in the data from the input file and copies it over to the global memory of the GPUs. After which, blocks containing CUDA threads are spawned and each thread within the GPU kernel takes in an equal portions of data to calculate the CRC for.

After calculating the partial CRC for the small chunk of data they are assigned to, the threads then perform reduction via interleaved addressing to combine the CRC values. Ultimately, the final CRC value for the data will be calculated from the repeated reductions. The final CRC value is then copied back into the CPU and the value is returned to the calling function.

For this implementation, we only spawn a single block of 16 threads so as to provide a reliable and consistent benchmark against the other implementations, in particular for the OpenMP implementation. The interleaved addressing approach is used because the order of combination of the CRC chunks is important. In addition, highly divergent branching will not be of concern here as the smallest unit of execution for the GPU (i.e. 1 warp) comprises of 32 threads.

V. Experiments and Results

A. Methodology

For all the experiments, we use the exact same data type with data sizes of 1MB, 4MB, 16MB, 64MB and 256MB. The timing for this experiment was computed using a timer inserted within the program and each program was run 20 times, with the lower 5 and the upper 5 iterations omitted to remove factors, such as varying CPU loads, that may skew the results.

B. Testing Machine Configuration

All the testing except the Java CRC calculation was done on the Jinx Cluster which has the following configuration:

```
Number of cores : 16
vendor_id : GenuineIntel
cpu family : 6
model : 26
model name : Intel(R) Xeon(R) CPU E5520  @ 2.27GHz
stepping : 5
cpu MHz : 2260.994
cache size : 8192 KB
physical id : 0
siblings : 8
core id : 3
cpu cores : 4
apicid : 7
initial apicid : 7
fpu : yes
fpu_exception : yes
cpuid level : 11
wp : yes
flags : fpu vme de pse tsc msr pae mce
  cx8 apic sep mtrr pge mca cmov
  pat pse36 clflush dts acpi mmx
  fxsr sse sse2 ss ht tm pbe
  syscall nx rdtscp lm constant_tsc
  arch_perfmon pebs bts rep_good
  xtopology nonstop _tsc aperfmperf
  pni dtes64 monitor ds_cpl vmx est
  tm2 ssse3 cx16 xtpr pdcm dca sse4_1
  sse4_2 popcnt lahf_lm ida dts
  tpr_shadow vnmi flexpriority ept vpid
bogomips : 4521.28
clflush size : 64
cache_alignment : 64
address sizes : 40 bits physical, 48 bits virtual
```

Data Size(MB) / Method	Bit-by-Bit	Table Lookup	OpenMP	Java	Zlib
1	0.136671	0.075837	0.024169	0.002	0.001562
4	0.545223	0.293634	0.09146	0.005	0.006029
16	2.18674	1.1425	0.352656	0.035	0.025474
64	8.83211	4.57995	1.32494	0.123	0.095719
256	34.8425	18.3811	2.4542	0.556	0.385955

TABLE I
CRC32C-MPEG Timings with different methods, Data Size in MB's and Time is in seconds

Data Size(MB) / Method	Bit-by-Bit	Table Lookup	SIMD	OpenMP with SIMD
1	0.136704	0.071323	0.001386	0.002441
4	0.546632	0.285247	0.005469	0.005359
16	2.18705	1.14125	0.022826	0.029838
64	8.7534	4.57043	0.091564	0.037578
256	35.0179	18.2688	0.362553	0.127546

TABLE II
CRC32C-SCSI Timings with different methods, Data Size in MB's and Time is in seconds

Data Size(MB)	GPU Timings
1	0.0973021
4	0.3478018
16	1.241307
64	4.367201
256	14.2852039

TABLE III
CRC-32C-MPEG timings using our GPU implementation

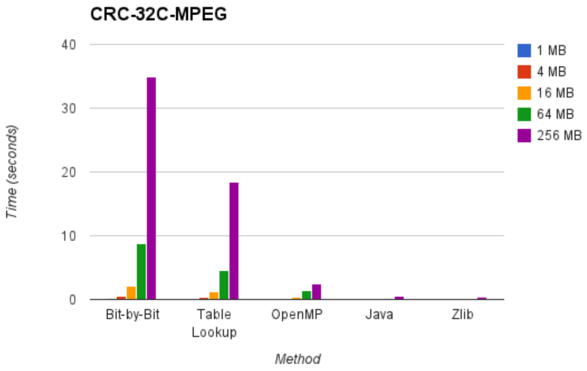


Fig. 3. CRC32C-MPEG calculation timings with different methods

VI. Analysis

- As can be seen from Table 1 our timings for CRC-32C-SCSI calculation using OpenMP and SIMD improve upon the intel's instruction. Therefore using OpenMP with hardware instructions can give an additional performance boost to the user.
- From Table 3 we see that the time taken for the CUDA implementation increases together with the size of the data. While the experiments for the smaller data sizes yield optimistic results, the time taken increases exponentially as the data size increases. But is it surprising to note that for the larger data sizes, the time taken for the parallel implementation on GPU is longer than some sequential implementations, like that of OpenMP and Java. The reason for this is due to the long duration of global memory access times. Because of the large input data sizes, we are unable to take advantage of the shorter access times of the shared memory. In addition to that, the CRC combination function is not a trivial operation and the increasing partial data sizes of the CRC that the partial CRC function degrades the overall execution time as well.
- We realized that our bottleneck was the combining of the resulting crc's of the sub blocks. As we increased the number of threads, the number of subpart crc's would increase. And since our combination was still sequential this would result in performance degradation. We therefore set the number of threads to 16.

VII. Findings

A. General

- From Fig.3 and Fig.4 we can say that the timings for the checksum calculation using CRC-32C-MPEG and CRC32-C-SCSI were same for all the methods except in the case of Intel's instruction which has the CRC32-C-SCSI polynomial built-in.
- Almost all the techniques work for calculating both CRC-32C-MPEG and CRC-32C-SCSI without any major changes.

B. Zlib's CRC32C Implmentation

In our project, we have adopted the partial CRC combination function from the zlib library, which is a free software library for public use. However, through our experiments, we have noticed that the CRC combination function has some limitations and flaws. The combination function takes in two partial CRC values and a third parameter

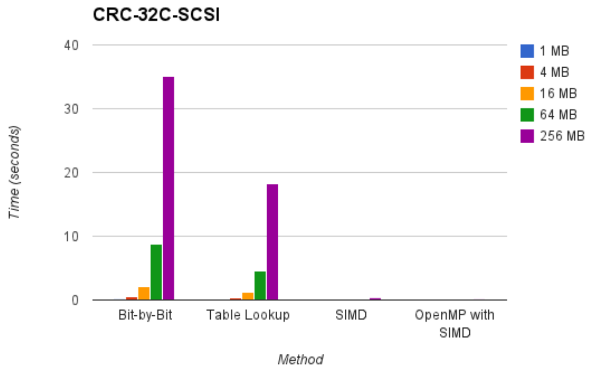


Fig. 4. CRC32C-SCSI calculation timings with different methods

as the size of the original data for the second CRC value.

- *Data Sizes for the Partial CRC* : We first noticed that while the function allowed the sizes to be unequal, it was only tolerant to a certain amount plus minus the size of the data for the first CRC. For example, if the data size belonging to the first CRC value is 512 bytes, then the second CRC size cannot be too small (e.g. 300 bytes) or too big (900 bytes).

- *Wrong Results on Specific Sizes*:The second observation was that the CRC combine function produced wrong checksums for particular values of data separation. In our experiments, we observed that given a data block of 1023 bytes, splitting the data into two parts of 643 bytes and 380 bytes produced a wrong CRC checksum after combination. We are unable to derive the reason for this phenomenon.

- *Wrong CRC Checksum after Combining Beyond a Certain Number of Times*: In one of our experiments with loop unrolling, we found out that the partial CRC combine function gave a different checksum result after being recursively executed for more than 38 times on the same base data. Despite extensive experimentation, we are unable to derive the cause for this.

VIII. Future Work

- Currently the data we calculate the CRC for resides in memory, we could extend it to calculate CRC for streaming data.
- In light of our findings for the zlib librarys partial CRC combination function, future projects for exploration of CRC calculation on GPUs may want to implement their own CRC combination function as this would provide the reliability that the zlib library function cannot.

IX. Conclusion

- We achieved a better performance for CRC32C-SCSI checksum calculation. Therefore provided a CRC calculation is supported by the hardware we can gain significant performance improvements using OpenMP.
- Since the current bottleneck is to combine the crc's of the resulting sub-messages, we need to balance out the number of subparts into which we should divide our message for computing the checksum.
- Based on our experimental data, we have come to the conclusion that the CRC Checksum calculation on GPUs would be feasible for large sizes of data. However, in this project it is important to note that we are using a specialized case comprising of only 16 threads. This is so that we can provide a fair benchmark against our OpenMP implementation. Further experiments may consider using up to 1024 threads per block, and spawning multiple blocks for a higher degree of parallelization. However, we predict that the global memory access time would still be the main bottleneck in the parallel execution on GPUs. New approaches to take advantage of another memory model or type, such as shared memory, would be required in order achieve a better speedup over current results.

References

[1] Fast CRC Computation for Generic Polynomials Using PCLMULQDQ, http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/fast-crc-computation-generic-polynomials-pclmulqdq-paper.html
[2] A Painless Guide to CRC Error Detection Algorithms, http://www.ross.net/crc/
[3] Speeding up CRC32C computations with Intel CRC32 instruction, Speeding up CRC32C computations with Intel CRC32 instruction
[4] GoLang CRC Sample Program, http://play.golang.org/p/pgNZnDVA73
[5] zlib, A Massively Spiffy Yet Delicately Unobtrusive Compression Library http://zlib.net/

Appendix

Bitbucket Repository: https://bitbucket.org/luppy_2/cse6230-final-project-gpu-hash-sums