GPU Hashsums

Georgia Institute of Technology

Akash Gangil　　Lup Peng, Loke

CSE 6230: HPC Tools and Applications

Final Project, Fall 2013

Goal

1. Exploit the parallelism potential of the CRC checksum calculation algorithm using GPU .
2. Benchmark against various techniques like SIMD and OpenMP, as well as third party implementations like zlib and Java.

Introduction

Data being sent over physical mediums is susceptible to interference which can corrupt the bits in a message.

Cyclic Redundancy Check (CRC) is used to detect these accidental changes to raw data being transferred over a network medium. CRC uses the remainder of a polynomial division as a checksum to verify the message integrity after being transferred over the network.

The sender's checksum is sent together with the data and the computed receiver's checksum is compared against the former. A matching pair of values would mean that the data had been sent successfully.

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 combined at the end. Having a parallel CRC implementation will help overcome this potential bottleneck by reducing the calculation time required.

This would result in faster network speeds without sacrificing on the correctness of the data.

Project Breakdown

Phase 0: Preliminary Analysis and Understanding
Started by understanding the CRC32C and CRC32-MPEG checksum calculations, the different standards and how they are implemented. The various approaches and areas of potential parallelization will be identified.

Phase 1: Determine Baseline CPU Performance
Developed a naive CPU implementation of the CRC32C and CRC32-MPEG hash sums, using the Bit-by-Bit, Byte-by-byte and Table Lookup approaches.

Phase 2: Parallelization of CPU Implementation
Identified the areas of algorithm which we can parallelize, we reused zlib's crc combine function in this process.

Phase 3: Evaluating Intel® CRC32 Instruction (PCLMULQDQ)
Intel's CRC32 instruction for calculating CRC32 checksums is used to give an idea of performance in hardware implementations.

Phase 4: Using OpenMP for Speedup
In addition to the above Intel instruction, OpenMP directives are used for further speed up the algorithm.

Phase 5: Comparing other CRC32 Alternatives
Existing third party implementations and libraries, Java and zlib, are evaluated for benchmarking performance.

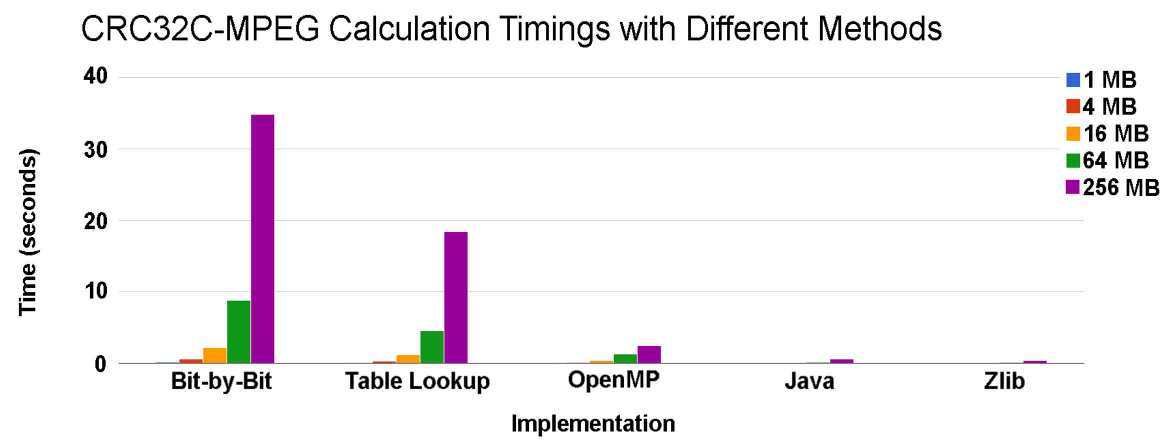
Phase 6: GPU Implementation using CUDA
Implemented the CRC Checksum calculation algorithm on the GPU.

Results

CRC32C-MPEG

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

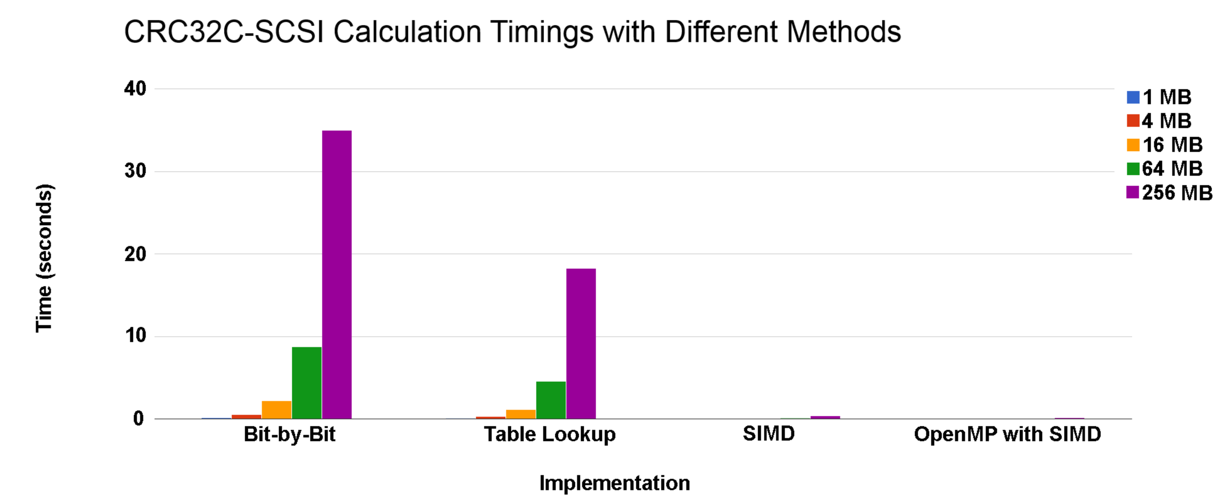
TABLE I
CRC32C-MPEG TIMINGS WITH DIFFERENT METHODS, DATA SIZE IN MB'S AND TIME IS IN SECONDS



CRC32C-SCSI

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

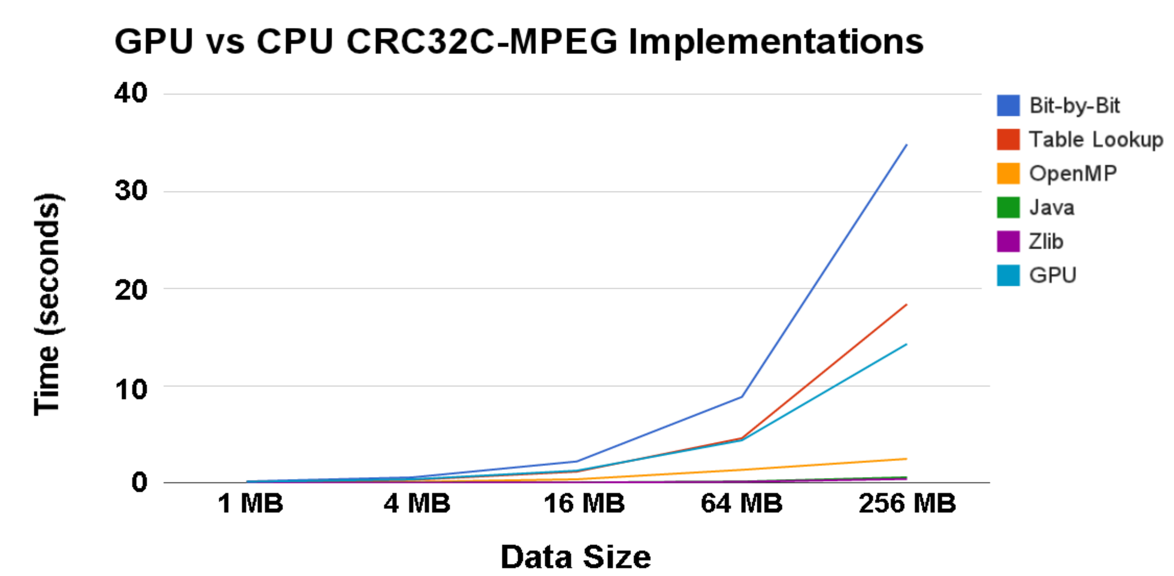
TABLE II
CRC32C-SCSI TIMINGS WITH DIFFERENT METHODS, DATA SIZE IN MB'S AND TIME IS IN SECONDS



Parallel GPU via CUDA

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



Analysis

As seen from Table 1, using OpenMP directive with Intel's PCLMULQDQ instruction gives the best performance over all the other implementations. This is because PCLMULQDQ is a hardware instruction which is much faster than software instructions.

While the time taken for the parallel implementation with CUDA increases proportionally with the input data size (Table 3), the speedup achieved is dismal (approximately 2.45).

A few areas that cause the low speedup are:

1. Parallel processing overheads arising from creating the GPU kernels and spawning the threads.
2. Threads accessing the global memory for the input data, which has a high access time as compared to shared memory.
3. Computation bottleneck during the non-trivial and sequential partial checksum combination.
4. Small data size results in communication time outweighing the computation time.

Increasing the number of thread to reduce the amount of computation time per thread is possible, but the communication time between threads and accessing the GPU's global memory would still be a key factor in low speedups.

Findings

The different CRC polynomials due to the different standards only affected the polynomial used. Hence the CRC function can be very flexible in adapting to multiple standards.

In the project, we adapted the partial CRC checksum combination function from the zlib library. However, we have noticed that it has some limitations and flaws:

1. Restrictions on the Data Size for Partial CRC : The function allows the data sizes of the partial checksums to be unequal, but only tolerant to a certain amount. The second CRC checksum data size cannot be too big or too small than the first CRC checksum data size.
2. Wrong Results on Specific Data Partition Size : The function produces wrong checksums for particular values of data separation. In our experiments, a data block of 1023 bytes split into two parts of 643 and 380 bytes produced a wrong CRC checksum after combination.
3. Wrong Checksum after Recursive Calls : The function produces wrong checksums after being recursively executed for more than 38 times with the same base data.

Conclusion

In the domain of **sequential implementations**, the parallel speedup increased proportionally with the input data size.

Using existing libraries and specialized instructions gave the most speed up:

- Zlib library gave the best performance timing for the CRC32C-MPEG polynomial. (speedup approximately 90 times)
- **OpenMP with Intel's PCLMULQDQ hardware instruction gave the best speedup for the CRC32C-SCSI polynomial (speedup approximately 270 times)**

For the **parallel GPU implementation**, using 16 threads:

- the parallel CRC Checksum calculation on GPUs achieved a speedup of approximately 2.45

The main bottleneck in computation is the recombination of the partial CRC checksums calculated by each thread - this must be done in order of the original data.

To achieve the best performance, there must be a compromise between the number of threads and the input data size. Having more threads results in more communication overheads - this means that the data size would have to be larger to achieve a speedup.

Further experiments may consider using up to 1024 threads per block, and spawning multiple blocks for a higher degree of parallelization.

But we predict that the global memory access time would still be the main bottleneck in the parallel execution.

New approaches to take advantage of another memory mode or type (e.g. shared memory) would be required in order achieve a decent speedup over sequential implementations.

Acknowledgements

We would like to thank Professor Richard Vuduc for providing us the opportunity to do this project and the TA's Marat Dukhan and Jee Choi for their help during the course of this project.

We would also like to thank the College of Computing Department at Georgia Tech for providing us with access to the Jinx cluster on which we ran our experiments.

Contact information

Akash Gangil
GT ID: 902950416
agangil3@gatech.edu

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