A high performance parallel calculation of 32-bit Cyclic Redundancy Check for GP-GPU

Francesco Tosoni, Nereo Sorio

***Abstract*—A fast parallel table-based implementation for CRC (Cyclic Redundancy Check) algorithm is proposed. The proposed algorithm aims to express the highest possible level of parallelism. First, you divide the input into bytes, then you access a pre-computed lookup table for each byte in parallel. In the end, a final xor between all the elements calculated in predence is made. Also here we have looked for the most parallel approach possible. The result is the CRC of the original message.**

In particular, we made the 32 bit version of the CRC, using CUDA. Also at the end of the document is reported a performance analysis comparing our product with the equivalent serial algorithm.

I. Introduction

CRC (*Cyclic Redundancy Check*) is a checksum algorithm to detect inconsistency of data, e.g. bit errors during data transmission. Blocks of data can therefore be checked quickly, based on the remainder of a polynomial division. The calculation is then performed both by the transmitter, which appends its result to the data, and by the receiver, which compares its result with the transmitted one. This technique is very effective in detecting physical errors on the transmission channel, but not in against intentional corruption of data.

The encoding of a CRC value requires a **generator polynomial** to be defined. It will serve as a **divisor** in a **long division**, in which the data to be transmitted becomes the dividend. In this case, the quotient is of no use and therefore is discarded, while the **remainder is the CRC value**. The polynomial coefficients are calculated according to the arithmetic of a finite field, so there is no carry between digits. Obviously, as shown in the next section, the message is also encoded as a polynomial.

When talking about n-bit CRC, n is the size of the checksum value. There are several protocols for a CRC of size n but with different values: simply, the generator polynomial changes. Traditional algorithms are implemented through cyclic operations, since many times, the CRC is calculated at hardware level. The checksum value in fact, is obtained with a few simple operations repeated on each block (of fixed size) of data. The simplest error-detection system, the parity bit, is in fact a 1-bit CRC: it uses the generator polynomial x + 1 (two terms).

II. Background

A. Basics

The CRC value is usually calculated on a fixed-length bit stream. CRC algorithms treat each bit stream as a **binary polynomial** $A(x)$ and calculate the remainder $R(x)$ from the division of $A(x)$ with a standard "generator" polynomial $G(x)$.

For a n bit message $a_{n-1}a_{n-2}...a_0$ it can be treated as a polynomial as follows:

$A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_1x + a_0$

where a_{n-1} is the Most Significant Bit (**MSB**) and a_0 is the Least Significant Bit (**LSB**) of the message.

For example, the input data `0x14=00010100` is taken as:

$A(x)=0x^7 + 0x^6 + 0x^5 + 1x^4 + 0x^3 + 1x^2 + 0x^1 + 0x^0$.

Given the degree $m-1$ generator polynomial,

$G(x) = g_{m-1}x^{m-1} + g_{m-2}x^{m-2} + ... + g_0$ where $g_{m-1} = 1$ and $g_i \in \{0, 1\}$ for all $0 \leq i \leq m-2$.

For example, the generator polynomial of CRC32C (in hex `0x1EDC6F41`) is:

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

$A(x)$ is multiplied by $x^{m-1}$and divided by $G(x)$ to find the remainder.

$CRC(A(x)) = R(x) = A(x)x^{m-1} mod G(x)$

CRC value of the message is defined as the **coefficients of the remainder polynomial**. After CRC processing is completed, the binary words corresponding to $R(x)$ are transmitted together with the bit stream associated with $A(x)$. At the receiver side, CRC algorithms check whether $R(x)$ is the correct remainder. The division is performed using modulo-2 arithmetic. Additions and subtractions are "carry-less" in modulo-2 arithmetic. In this case, additions and subtractions are equal to the ***xor*** logical operation.

B. Theorems

Two theorems to achieve parallelism in CRC computation are used:

- **Theorem 1**: Let $A(x) = A_1(x) + A_2(x) + ... + A_n(x)$ over GF. Given a generator polynomial, $CRC(A(x)) = \sum_{i=1}^{n} CRC(A_i(x))$
- **Theorem 2**: Given $B(x)$, a polynomial over GF, $CRC(x^k B(x)) = CRC(x^k CRC(B(x)))$ for any k.

The figure 1 shows the theorems more intuitively.

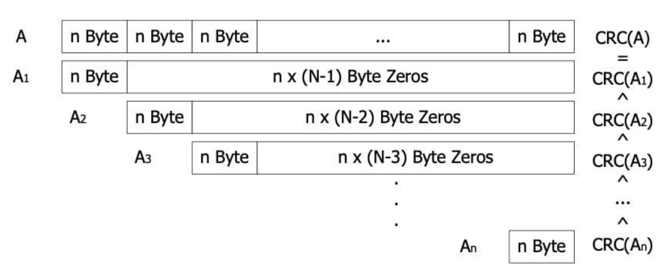

Figure 1. The two theorems.

III. Serial Algorithms

There are two main approach for implement CRC: Bit-wise and Byte-wise alghoritms.

In the **bit-wise** CRC algorithm, 1 input bit is processed at a time as the name indicates, using long division. First we append M zeros (M is the number of bits in CRC) to the original N-bit message to the least significant bit (LSB) end and define the appended message as a running message. Second, check the most significant bit (MSB) of the running message. If the MSB of the running message is 1, subtract the generator polynomial from the M most significant bits of this running message and shift the result to left by 1 bit and store the result to the running message. Otherwise, (i.e., the MSB of the running message is 0), just shift the running message to left by 1 bit and store the result to the running message. The second step is repeated N times and at the end the remainder will be the M most significant bits, which is the CRC. Note that in bit-serial CRC algorithm, we need to perform N times of operation where in each operation we need to perform checking the MSB bit, modulo-2 subtraction (conditionally), and left shift.

So far the algorithm is quite inefficient as it works bit by bit, specially in case of large inputs. In that case, this could be quite slow. The divident is the current crc byte value, and a byte can only take 256 different values. The polynomial (= divisor) is fixed. It is possible to **precompute the division** for each possible byte by the fixed polynomial and store these result in a **lookup table** as the remainder is always the same for the same divident and divisor! Then the input stream can be processed **byte by byte** instead of bit by bit.

In **byte-wise** CRC algorithm, one byte is checked at a time. In the first step, similar to bit-serial, we append M zeros to the original message after the LSB. If the original message size N is not a multiple of 8, we need to pre-append a number of 0's (in the range of 1 to 7) before the MSB to make the appended message having size of multiple of 8. We define this appended message as a running message. Second, perform table lookup based on the MSB 8 bits to find the M- bit remainder. This M-bit remainder will be $xored$ with the following MSB byte in the running message. Then we left shift the running message by 8 bits. Repeat the second step by $\lceil N/8 \rceil$ times. In total there are $\lceil N/8 \rceil$ operations with $2^8 * M$ bit of memory for table lookup. Note that the table has 256 entries each of which has M bits. The table entries **can be pre-computed since they only depend on the generator polynomial**.

Note that both bit-wise and byte-wise algorithm are explained using big endian bit order. In little endian, you have simply to switch the order of the bits in the bytes and shift to the right, without appending any 0.

Traditional table-based CRC calculation always take 4-bit or 8-bit data as input. The **Sarwate** algorithm is one of the most popular ones. By performing an xor operation between the least significant byte of the current CRC value and a new byte from the input data and by performing a table lookup, the Sarwate algorithm determines how the current CRC value is modified when a new byte is taken into consideration. The lookup table used by the Sarwate algorithm stores the remainders from the division of all possible 8-bit numbers shifted to the left by 32 bits with the generator polynomial. The C-code of the Sarwate algorithm is shown in figure 2.

```
crc = INIT_VALUE;
while(p_buf < p_end) {
  crc  = (crc >> 8) ^ table[(crc ^ * p_buf++) & 0x000000FF];
}
return crc ^ FINAL_VALUE;
```

Figure 2. Sarwate algorithm (little endian).

IV. Parallel Algorithms

In this section, we present the parallel version of the Sarwate algorithm taken as serial reference. We decided to parallelize this algorithm because we thought it was interesting to try to improve the performance of an algorithm that already contains an optimization. So the goal is to **parallelize the CRC calculation** of a character string trying to keep us close to the **tabular approach**, so without too many calculations.

The message is taken as input and divided into pieces of 1 byte each. This is because, adopting also here the optimization of the table, **the bigger the size of the piece, the bigger the table is**. In particular, as it happens in the serial, having segments of 1 byte and wanting a CRC of 32 bits, a table of 256 elements of 32 bits each is formed. As said in the previous section, the serial algorithm uses a table of 256 elements for an input bit stream of any length. This is because, at each iteration of the algorithm, i.e. at each calculation for one byte, the value of the current shifted CRC is updated (via xor operation) with the value just taken from the table.

Here instead, the purpose is to **first make all the accesses to the table** in the first kernel method **and then make** xor **of all these intermediate values**. However, to do this we need a table that allows to have intermediate values in the right position (here no shift of the values to the right is done, since we work in little endian, in the final cumulative xor of the values). The first 4 bytes of message are $xored$ with the initial CRC value (it depends on the protocol used, for example, in CRC32 is `0xffffffff`, and also the final word is the same) while all the others are taken without further transformations. Since we are in little endian, groups of 4 bytes are taken and we act in the opposite way to how we would normally do, i.e. we take first the LSB and then the MSB.

Given the strong similarity of operation, we adopted the same solution used by the "`slice-16`" algorithm proposed by Intel[1] to generate our table. This is necessary because **every time we proceed with the bytes of the message, we must take into consideration the shift and always have a table of 256 elements**. So, we generate the first table in a classic way (CRC of each 8 bit combination) and for the next ones we perform the following operation for each element of each table: we take the index of the current value to be calculated on the previous table, we shift it by 8 bits and we execute a xor with the value in the first table indexed by the current value of the previous table. In this way, we can handle the mid-values of CRC separately. This is because of the second theorem we had mentioned in previous section.

Note that each table access is made through **row-major layout**: Cuda does not allow the reference to multidimensional arrays but only with a unique index. In the table, therefore, the access will not take place in a classical way (as if it were a matrix) but each processed byte has its range of 256 elements.

The only thing that may be difficult to manage for some devices is the amount of memory that occupies the generated table. In fact, since this is an experimental and therefore parametric version, **the size of the table depends on the size of the input bit stream**. For example, with an input stream of `2 MB`, a table of about `2 GB` is generated. This is an expensive set of data to manage in memory. However, we know that CRC is usually calculated on **fixed size input stream**, so the table, no matter how big it is, you can **pre-compute** it only once and always have it in memory as a **constant**. In this way you avoid expensive data transfers from host to device. Actually, this is the only limit imposed by this approach.

Moving on, what remains to do is to combine the intermediate values in the final CRC via xor operation. The second kernel method accomplishes this last step and three different versions have been developed:

- The first one differs a bit from what was proposed in the reference paper[2]: the only operations performed by the device are performed by the first kernel method. The final xor is done by the host. This solution is the simplest of the three, but also the most trivial, because you simply make one xor after another in a sequential way. As a result, there are **two main disadvantages**: it is necessary to retransfer the array containing the intermediate CRC values from device to host (a time-consuming operation) and moreover, you lose the parallelism that is implemented in the first kernel method. This, in fact, turns out to be the least performing solution, although the easiest one.
- The second and the third ones implement the optimization technique called **reduction**. It is used in parallel field to significantly reduce the execution time of algorithms that must **process very large arrays, keeping as many GPU multiprocessors as possible employed**. Each block of threads is assigned a portion of arrays to work on separately and the invocation of the kernel method (multiple) is the synchronization point between threads (in Cuda this is implicit). In particular, a reduction with sequential addressing is implemented (fig.3). It works like this: the entire array of intermediate CRCs (same size as the input) is split within each block of threads. Using an array in **shared memory** as support, in every block, each thread makes an xor between two elements and, at the end, blocks store the result of their under computation in the global memory of the device, updating the array in DRAM. Each block of threads is assigned a portion of array to work on separately and the invocation of the kernel method (multiple) is the **synchronization point** between threads (in Cuda this is implicit). In this way, each time the kernel method is invoked with fewer and fewer blocks needed, since the data involved are less and less and stored in the first part of the array. The number of executions of the kernel method is calculated before, based on the size of the input (for a fixed size there would be no need to calculate it). Also, there is no need to transfer data from device to host except for the final result. Obviously, this will be $xored$ with the final word in host.
- The last variant of the second kernel method, is also a reduction but slightly different. You take blocks of smaller size while the size of the array in shared memory is the same size as before. The differences are in how the temporary shared memory array is filled and therefore how the calculations are done. This is more efficient variant than the others, although it's similar to the previous one.

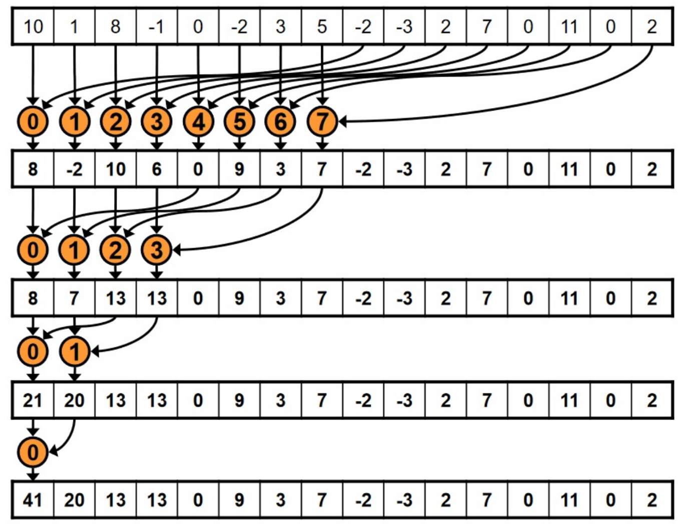

Figure 3. Example of first reduction.

V. Results

Testing device specifications:

- Device Name: GeForce RTX 2070 Super
- Producer: Nvidia
- Amount of cuda cores: 2560
- Clock speed: 1605 MHz (1770 boost)
- Memory: 8 GB GDDR6
- Streaming Multi Processor: 40

The results produced by tests are shown below. In the table, the numbers identifying the parallel versions refer to the order in which they were introduced in the previous section.

Note that, except for the serial program and the first parallel version, the size of the invoked thread blocks strongly depends on the input bit stream size. In fact, the first and the third input, are respectively 128^2 and 128^3, while the second is 256^2. So, in the last two versions 128 thread blocks have been used in the first and third test and 256 blocks for the second test. However, this block solution was also adopted in the first version to **minimize the divergence of threads**.

Algorithm	Input size	Table size	Overall time	Speedup	Overall time (with data transfer)	Speedup
Sarwate	16 kB	1 kB	0.1 ms	1x	-	-
Parallel 1	16 kB	16 MB	0.1 ms	1x	0.1 ms	1x
Parallel 2	16 kB	16 MB	0.028 ms	3.5x	0.04 ms	2.2x
Parallel 3	16 kB	16 MB	0.022 ms	4.4x	0.04 ms	2.2x
Sarwate	64 kB	1 kB	0.3 ms	1x	-	-
Parallel 1	64 kB	64 MB	0.3 ms	1x	0.3 ms	1x
Parallel 2	64 kB	64 MB	0.04 ms	7.5x	0.1 ms	4.9x
Parallel 3	64 kB	64 MB	0.032 ms	9.1x	0.057 ms	5.2x
Sarwate	2 MB	1 kB	8.3 ms	1x	-	-
Parallel 1	2 MB	2 GB	9.8 ms	0.8x	10.1 ms	0.8x
Parallel 2	2 MB	2 GB	0.81 ms	10.2x	1.1 ms	7.7x
Parallel 3	2 MB	2 GB	0.8 ms	10.5x	1.1 ms	7.8x

The table illustrates how, even with contained input, our implementation still obtains the result in less time than the serial algorithm. More detailed results on the two parallel versions are shown in the images 4 and 5. In these results it is evident that the cost of data transfer from host to device is very long. This is due to the transfer of both the entire input bit stream and the table (much larger than the input) to the device. In the results table, in the column that shows the time with data transfer, the transfer of the table is not considered. However, we think it is important to emphasize that this cost is simply due to the fact that the table is **dynamically generated at each execution** by the host. This is because it was convenient in the development phase to always have the table that fits the input bit stream size always different. In a not experimental use, that mean with the fixed input bit stream size, the table would always be the same, therefore constant and already present in device without the need to pass it to every single execution.

VI. Conclusions

As a project it was very interesting, because we had the opportunity to approach a problem that lends itself to any application and tried to improve it using Cuda. We experimented with different approaches, choosing the path that seemed to us to bring better results, especially in terms of time. Regarding this aspect we can be satisfied because we have obtained good results even if the Sarwate serial algorithm is already very performing. As for the energy consumption and resources used, we realize that our approach is a bit more invasive especially with regard to the memory used. Unfortunately, the convenience of simply accessing to a lookup table instead of calculations is paid with more memory used. We knew this even before we started making the parallel version but we are sure it is a cost affordable for a device with a good GPU capacity. An additional optimization on a strictly functional and not algorithmic level, would be to realize also the generation of the table in device. This would avoid switching the table from host to device: the 256 elements can be calculated in parallel, one table at a time.

Another approach, totally different from this one, would be to leave the table lookup optimization and perform each CRC of each byte, always taking into account the position and order of the bytes to be respected. This would use less memory, besides not passing the table in devices. However, we don't think you can be better than our version in terms of performance by implementing a similar approach.

References

[1] Intel, "Choosing a crc polynomial and associated method for fast crc computation on intel processors," 2012.

[2] W. L. Huo, Li, "High performance table-based architecture for parallel crc calculation," 2015.

```
==25242== NVPROF is profiling process 25242, command: ./crc32-prl
2097152
CRC32 host:     8.4 ms
CRC32 device:   0.8 ms
Speedup: 10.3x

0xbb30ee63 - 0xbb30ee63
<> Correct

==25242== Profiling application: ./crc32-prl
==25242== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.64%  221.45ms         2  110.72ms  190.24us  221.26ms  [CUDA memcpy HtoD]
                    0.30%  674.50us         1  674.50us  674.50us  674.50us  crc32kernel(unsigned char*, int, unsigned int*, unsigned int*)
                    0.06%  132.70us         3  44.234us  1.6320us  128.90us  xorkernel(unsigned int*)
                    0.00%  1.8880us         1  1.8880us  1.8880us  1.8880us  [CUDA memcpy DtoH]
      API calls:   48.92%  221.61ms         2  110.81ms     839ns  221.61ms  cudaEventCreate
                   48.92%  221.59ms         3  73.864ms  19.715us  221.35ms  cudaMemcpy
                    0.69%  3.1458ms         2  1.5729ms  1.2126ms  1.9332ms  cudaFree
                    0.58%  2.6099ms         3  869.96us  101.81us  2.1231ms  cudaMalloc
                    0.34%  1.5263ms       194  7.8670us     570ns  327.88us  cuDeviceGetAttribute
                    0.31%  1.4016ms         2  700.78us  693.22us  708.34us  cuDeviceTotalMem
                    0.18%  830.32us         1  830.32us  830.32us  830.32us  cudaEventSynchronize
                    0.04%  174.48us         2  87.239us  66.922us  107.56us  cuDeviceGetName
                    0.01%  43.753us         4  10.938us  4.1450us  27.593us  cudaLaunchKernel
                    0.00%  17.038us         2  8.5190us  2.0220us  15.016us  cudaEventRecord
                    0.00%  7.8230us         2  3.9110us  2.3560us  5.4670us  cuDeviceGetPCIBusId
                    0.00%  6.0670us         3  2.0220us     920ns  2.6750us  cuDeviceGetCount
                    0.00%  4.4580us         4  1.1140us     603ns  1.8140us  cuDeviceGet
                    0.00%  4.4230us         2  2.2110us  1.4270us  2.9960us  cudaDeviceSynchronize
                    0.00%  2.5400us         2  1.2700us     493ns  2.0470us  cudaEventDestroy
                    0.00%  1.9120us         2     956ns     713ns  1.1990us  cuDeviceGetUuid
                    0.00%  1.7140us         1  1.7140us  1.7140us  1.7140us  cudaEventElapsedTime
                    0.00%     287ns         1     287ns     287ns     287ns  cudaGetLastError
```

Figure 4. Some details about the first reduction.

```
==29520== NVPROF is profiling process 29520, command: ./crc32-prl
2097152
CRC32 host:     8.4 ms
CRC32 device:   0.8 ms
Speedup: 10.9x

0xbb30ee63 - 0xbb30ee63
<> Correct

==29520== Profiling application: ./crc32-prl
==29520== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.69%  222.36ms         2  111.18ms  188.03us  222.18ms  [CUDA memcpy HtoD]
                    0.29%  648.42us         1  648.42us  648.42us  648.42us  crc32kernel(unsigned char*, int, unsigned int*, unsigned int*)
                    0.02%  47.200us         3  15.733us  1.5680us  43.616us  xorkernel(unsigned int*)
                    0.00%  1.9520us         1  1.9520us  1.9520us  1.9520us  [CUDA memcpy DtoH]
      API calls:   50.96%  222.50ms         3  74.165ms  19.270us  222.26ms  cudaMemcpy
                   47.29%  206.47ms         2  103.23ms  1.0130us  206.47ms  cudaEventCreate
                    0.69%  3.0237ms         2  1.5118ms  1.2190ms  1.8046ms  cudaFree
                    0.59%  2.5950ms         3  864.99us  98.987us  2.1215ms  cudaMalloc
                    0.18%  783.49us         1  783.49us  783.49us  783.49us  cudaEventSynchronize
                    0.14%  623.37us       194  3.2130us     187ns  137.81us  cuDeviceGetAttribute
                    0.11%  488.24us         2  244.12us  242.71us  245.53us  cuDeviceTotalMem
                    0.02%  72.702us         2  36.351us  27.590us  45.112us  cuDeviceGetName
                    0.01%  45.998us         4  11.499us  4.2470us  30.561us  cudaLaunchKernel
                    0.00%  12.172us         2  6.0860us  2.1160us  10.056us  cudaEventRecord
                    0.00%  5.1520us         2  2.5760us  2.0660us  3.0860us  cuDeviceGetPCIBusId
                    0.00%  4.2530us         2  2.1260us  1.3810us  2.8720us  cudaDeviceSynchronize
                    0.00%  2.6960us         2  1.3480us     497ns  2.1990us  cudaEventDestroy
                    0.00%  1.7120us         1  1.7120us  1.7120us  1.7120us  cudaEventElapsedTime
                    0.00%  1.6980us         3     566ns     285ns     900ns  cuDeviceGetCount
                    0.00%  1.2860us         4     321ns     184ns     659ns  cuDeviceGet
                    0.00%     686ns         2     343ns     318ns     368ns  cuDeviceGetUuid
                    0.00%     199ns         1     199ns     199ns     199ns  cudaGetLastError
```

Figure 5. Some details about the second reduction.