Parallel CRC Algorithm and Implementation with CUDA

Mirco De Marchi - VR445319 Luigi Capogrosso - VR445456

***Abstract*—Cyclic redundancy checks (CRCs)** **form a powerful class of codes suited especially for the detection of burst errors in data storage and communication applications. The use of cyclic redundancy codes, referred to more briefly in this document as check words, is one of the most well known error checking mechanism. It is customarily placed at the end of a data block, occupying one or more bytes of the transmitted message. In this paper, we present a *Parallel Cyclic Redundancy Check (PCRC)* algorithm improved with CUDA. Our approach is to systematically decompose the original input message into a set of subsequences based on the *Galois field* theory.**

I. Introduction

Devices such as computers, routers, networking equipment, and components that communicate information internally and/or with other devices. For example, computers might communicate across a local area network (LAN) using Ethernet protocol, or application-specific integrated circuits (ASICs) may communicate with each other over a single or parallel bit bus. **It is important for these devices to reliably communicate and detect errors in their communication**.

One common technique for detecting transmission errors is known as **cyclic redundancy check (CRC)**. A CRC allows errors detection using only a small number of redundant bits typically sent along with the communicated information.

CRCs are based on the theory of **systematic cyclic codes**, which encode messages by adding a fixed-length check value. Cyclic codes are not only simple to implement but **have the benefit of being particularly well suited for the detection of burst errors**: contiguous sequences of erroneous data symbols in messages.

Specification of a CRC code requires the definition of so-called **generator polynomial**. This generator polynomial becomes the **divisor** in a **polynomial long division**, which takes the message as the **dividend**, the **quotient** is discarded and the **remainder** becomes the CRC code result. **The polynomial coefficients are calculated according to the arithmetic of a finite field**, so the addition operation can always be performed bitwise-parallel (there is no carry between digits).

A CRC is called an N-bit CRC when its check value is N bits long. The simplest error-detection system, the parity bit, is in fact a 1-bit CRC: it uses the generator polynomial $x + 1$ (two terms), and has the name CRC-1.

The common hardware solution is the "Linear Feedback Shift Register" (LFSR), which is a simple bitwise architecture for both encoding and deconding the message. This approach typically calculates the CRC for a N-bit message in N clock cycles. This approach is not efficient at high bit rates.

II. Backgorund

The CRC see the message A of length n as a polynomial $A(x)$ of degree $n-1$ in which every bit of message is the coefficient of the respective monomial:

$$A = [a_{n-1}, a_{n-2}, \ldots, a_2, a_1, a_0]$$

$$A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_2x^2 + a_1x + a_0$$

For example, the input data `0x25=0010 0101` is taken as:

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

Then for the CRC computation is used a polynomial generator $G(x)$ that is a polynomial of degree m, where m will be the length of the resulting CRC code. The major index coefficient is always 1, in this way the polynomial generator guarantees to be of m degree.

$$G = [g_m, g_{m-1}, \ldots, g_1, g_0]$$

In a reverse way than the message A, the polynomial generator can be represented as a sequence of bit G of size $m+1$ with the most significant bit always to 1. This is the reason why you will find the value representation of the polynomial generator G of size m, because the bit 1 to the MSB is implicit.

$$G = g_m x^m + g_{m-1}x^{m-1} + \ldots + g_1 x + g_0$$

For example, Ethernet uses the following 32-bit polynomial value:

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

The **polynomial generator value** has a **great impact on its error detection** capabilities, its design is non-trivial and requires serious mathematical knowledge.

The CRC detection code is the reminder result after dividing the original message A concatenated with m zero bits by a polynomial generator in binary modulo 2 arithmetic.

In the polynomial version this definition of CRC is equivalent to:

$$CRC[A(x)] = A(x)x^m mod 2(G(x))$$

The binary modulo 2 arithmetic makes every operation between bits independent between each other. This means that **the carry bit can be completely forgotten**. In particular the binary modulo 2 reminder of the division always generates a result that is a degree less of the divisor degree. Therefore the

CRC result will always be a polynomial of $m-1$ degree that can be viewed as a binary code of size m. In fact:

$$degree(CRC[A(x)]) = m - 1$$

The main difference between CRC code is the length m of the code generated. The most important CRC Implementations are **CRC8**, **CRC16**, **CRC32** and **CRC64**. The longer the length of the CRC code, the more likely bit errors shall be detected.

Then there are different parametrization of the CRC:

- **Polynomial generator**: normal, reversed (from the least significant bit to the most), reciprocal (reversed but respect the polynomial coefficients), reversed and reciprocal in combination and the Koopman representation (discarding the least significant bit);
- **Initial value of the CRC**: in most algorithms is 0, in some cases is set with all bit to 1, but in general could be any value;
- **Input reflected**: each input byte can be used in reverse order before being used in the calculation;
- **Output reflected**: the result can be returned in reverse order over the whole CRC value;
- **Final XOR value**: it is a value XORed to the final CRC value before being returned and, in case, after the output reflected step.

III. Lemmas

Before we present our parallel CRC algorithm, we need the following lemma, which gives the well-known congruence properties for polynomial division.

$$(A(x) + B(x))modG(x) = A(x)modG(x) + B(x)modG(x) \tag{1}$$

$$A(x)B(x)modG(x) = ((A(x)modG(x))(B(x)modG(x)))\ modG(x) \tag{2}$$

There is a system of arithmetic known as **Galois field**. A Galois field is an example of algebraic field, which has a set of elements called **numbers or field elements**, and a definition of two operations called **addition and multiplication**.

For each positive integer M, there is a Galois field called $GF(2^M)$ that has 2^M elements in it. The elements set of $GF(2^M)$ can be represented with M-bit binary numbers in the range from 0 to $2^M - 1$. The addition over Galois Field $GF(2^M)$ is defined as bit-by-bit modulo-2 addition which is equivalent to exclusive-or (i.e., XOR) operation.

Multiplication of two elements a and b in $GF(2^M)$ will produce an element $c = ab$ by polynomial multiplication modulo the irreducible polynomial G(x):

$$C(x) = A(x)B(x)modG(x)$$

where $A(x)$, $B(x)$ and $C(x)$ is the polynomial form of the elements a, b, and c, respectively.

IV. CRC algorithms

Common CRC algorithms are bitwise and bytewise.

In the bitwise 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) 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 bitwise CRC algorithm, we need to perform N loops where in each loop we need to perform the MSB bit check, modulo-2 subtraction (conditionally), and left shift.

This is a **pseudocode of the bitwise CRC algorithm implementation**:

```
// Result size: N + M.
append(message, M)

// Usually 0.
message[M-1 : 0] = crc_initial_value

// Check each bit of the message.
for i in range(0 : N)
{
   LSB = (N + M - 1) - i
   if (message[LSB] == 1)
   {
      message[LSB : LSB-(M-1)] ^=
               polynomial_generator;
   }
}

return message[M-1 : 0];
```

In bytewise CRC algorithm, one byte is checked at a time.

The bitwise algorithm is quite inefficient as it works bit by bit. For larger input data, this could be quite slow. The idea behind the CRC bytewise implementation is to precompute the CRC value for each possible byte by the fixed polynomial and store these result in a lookup table. Since the polynomial generator is fixed, the lookup table can be precomputed independently from the message and it can be directly hard-coded.

A **simple pseudocode to generate the lookup table could be**:

```
for i in range(0 : 2^{8} - 1)
{
lookup_table[i] = crc_bitwise(i)
}
```

In the first step, similar to bitwise CRC algorithm, 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.

This is a **pseudocode of the bytewise CRC algorithm implementation**:

```
// Result size: N + M.
append(message, M)

// Usually 0
message[M-1 : 0] = crc_initial_value

// Check each byte of the message.
for i in range(0:N / 8-1)
{
   LSB = (N + M - 1) - (i * 8);
   message[LSB-8 : LSB-8-(M-1)] ^=
           lookup_table[message[LSB : LSB-7]
}

return message[M-1 : 0];
```

V. The CRC lookup table optimization

VI. Parallel CRC Algorithm

In this section we present our **"Parallel Cyclic Redundancy Check" (PCRC) algorithm** that performs CRC computation for any length of message in parallel.

For a given message with any length, we first chunk the message into blocks, each of which has a fixed size equal to the degree of the generator polynomial. Then we perform CRC computation among the chunked blocks in parallel using Galois Field Multiplication and Accumulation (GFMAC).

The message is seen as divided in M bit size chunk and the following is its polynomial representation:

$$A(x) = W_{n-1}(x)x^{(n-1)M} + \ldots + W_1(x)x^M + W_0(x)$$

Where each $W_i(x)$ polynomial is a chunk of the message. From this equation and the CRC definition, we can compute the CRC for the chunked message by:

$$CRC[A(x)] = W_{n-1}x^{nM} modG(x) + \ldots + W_0 x^M modG(x)$$

Furthermore, from the Galois Field operations lemma, we obtain that:

$$W_i(x)x^{(i+1)M} modG(x) = (W_i(x) modG(x) \; x^{(i+1)M} modG(x)) modG(x)$$

```
const uint8_t crc8_lu[] = {
   0x0,  0x1d, 0x3a, 0x27, 0x74, 0x69, 0x4e, 0x53,
   0xe8, 0xf5, 0xd2, 0xcf, 0x9c, 0x81, 0xa6, 0xbb,
   0xcd, 0xd0, 0xf7, 0xea, 0xb9, 0xa4, 0x83, 0x9e,
   0x25, 0x38, 0x1f, 0x2,  0x51, 0x4c, 0x6b, 0x76,
   0x87, 0x9a, 0xbd, 0xa0, 0xf3, 0xee, 0xc9, 0xd4,
   0x6f, 0x72, 0x55, 0x48, 0x1b, 0x6,  0x21, 0x3c,
   0x4a, 0x57, 0x70, 0x6d, 0x3e, 0x23, 0x4,  0x19,
   0xa2, 0xbf, 0x98, 0x85, 0xd6, 0xcb, 0xec, 0xf1,
   0x13, 0xe,  0x29, 0x34, 0x67, 0x7a, 0x5d, 0x40,
   0xfb, 0xe6, 0xc1, 0xdc, 0x8f, 0x92, 0xb5, 0xa8,
   0xde, 0xc3, 0xe4, 0xf9, 0xaa, 0xb7, 0x90, 0x8d,
   0x36, 0x2b, 0xc,  0x11, 0x42, 0x5f, 0x78, 0x65,
   0x94, 0x89, 0xae, 0xb3, 0xe0, 0xfd, 0xda, 0xc7,
   0x7c, 0x61, 0x46, 0x5b, 0x8,  0x15, 0x32, 0x2f,
   0x59, 0x44, 0x63, 0x7e, 0x2d, 0x30, 0x17, 0xa,
   0xb1, 0xac, 0x8b, 0x96, 0xc5, 0xd8, 0xff, 0xe2,
   0x26, 0x3b, 0x1c, 0x1,  0x52, 0x4f, 0x68, 0x75,
   0xce, 0xd3, 0xf4, 0xe9, 0xba, 0xa7, 0x80, 0x9d,
   0xeb, 0xf6, 0xd1, 0xcc, 0x9f, 0x82, 0xa5, 0xb8,
   0x3,  0x1e, 0x39, 0x24, 0x77, 0x6a, 0x4d, 0x50,
   0xa1, 0xbc, 0x9b, 0x86, 0xd5, 0xc8, 0xef, 0xf2,
   0x49, 0x54, 0x73, 0x6e, 0x3d, 0x20, 0x7,  0x1a,
   0x6c, 0x71, 0x56, 0x4b, 0x18, 0x5,  0x22, 0x3f,
   0x84, 0x99, 0xbe, 0xa3, 0xf0, 0xed, 0xca, 0xd7,
   0x35, 0x28, 0xf,  0x12, 0x41, 0x5c, 0x7b, 0x66,
   0xdd, 0xc0, 0xe7, 0xfa, 0xa9, 0xb4, 0x93, 0x8e,
   0xf8, 0xe5, 0xc2, 0xdf, 0x8c, 0x91, 0xb6, 0xab,
   0x10, 0xd,  0x2a, 0x37, 0x64, 0x79, 0x5e, 0x43,
   0xb2, 0xaf, 0x88, 0x95, 0xc6, 0xdb, 0xfc, 0xe1,
   0x5a, 0x47, 0x60, 0x7d, 0x2e, 0x33, 0x14, 0x9,
   0x7f, 0x62, 0x45, 0x58, 0xb,  0x16, 0x31, 0x2c,
   0x97, 0x8a, 0xad, 0xb0, 0xe3, 0xfe, 0xd9, 0xc4
};
```

Figure 1. LOOKUP TABLE USED IN CRC8 BYTEWISE

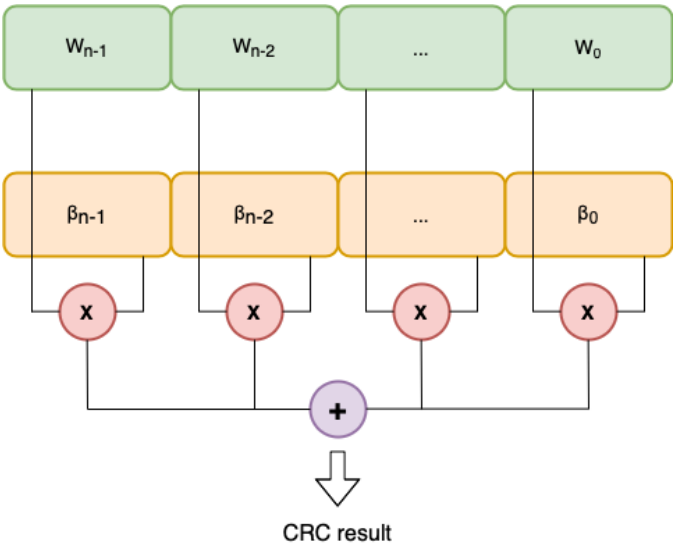


Figure 2. ILLUSTRATION OF PARALLEL CRC ALGORITHM OVER CHUNK OF M BITS

The degree of the polynomial $W_i(x)$ for each chunk is $M-1$, since it is less than M, it means that $W_i(x) modG(x) = W_i(x)$. On the other hand the beta coefficients can be defined as $\beta_i = x^{(i+1)*M} modG(x)$ for $i = 0, 1, \ldots, n-1$. Therefore we have:

$$CRC[A(x)] = W_{n-1} \otimes \beta_{n-1} \oplus \ldots \oplus W_0 \otimes \beta_0$$

Note that the operations $\otimes$, $\oplus$ in the above equation is Galois Field multiplication, addition over $GF(2^M)$, respectively.

The following algorithm is illustrated in Figure 2.

1) Load N-bit message in the chunked form of (W_{n-1}, W_{n-2}, W_0) where each chunk is M bits. (Note: $N = nM$).
2) Initially setup the generator polynomial $G(x)$ and its degree M. In the same time, pre-compute the beta factors $(\beta_{n-1}, \beta_{n-2}, \beta_0)$ which depend only on the polynomial and its degree.
3) Perform n-pair Galois field multiplications in parallel and then XOR the products. **This generates the CRC results**.

Note that beta factors $(\beta_{n-1}, \beta_{n-2}, \beta_0)$ do not depend on the incoming message but only on the generator polynomial. Since the polynomial is static, this should be pre-computed once and used repeatedly in the CRC calculation.

A **simple pseudocode to generate the beta array could be**:

```
for i in range(0 : N)
{
   shift_buffer = 1 << (M * (i + 1));
   beta[i]] = mod2(shift_buffer,
                   generator_poly);
}
```

Each thread is executed over a M bit size chunk of the original message and performs the following operations:

```
// Get the data of this thread.
W_i = orginal_message[global_index];
beta_i = beta[global_index];

// Perform binary modulo 2 multiplication.
mul = mod2_mul(W_i, beta_i);

// Perform binary modulo 2 reminder.
mod = mod2_mod(mul, generator_poly);

// Copy in shared memory.
shared_memory[threadX] = mod;
sync();

// XOR all data in shared memory.
return xored(shared_memory);
```

In our implementation, the last operation, that performs the Galois Field addition, XORing all the results of the Galois Field multiplication calculated from each thread, has been done in 2 ways:

1) The standard implementation takes the first thread, stop all the other and performs the XOR operation on all shared memory. This solution is more divergent because one thread is in execution while the other has nothing to do.
2) The reduction implementation takes the first half of all thread and performs iteratively the XOR operation between a pair of values in shared memory. In this way the workload is more spread across all threads and there is less divergence.

Finally, since the thread execution is done in blocks, the result of the device execution will be an array of partial results.

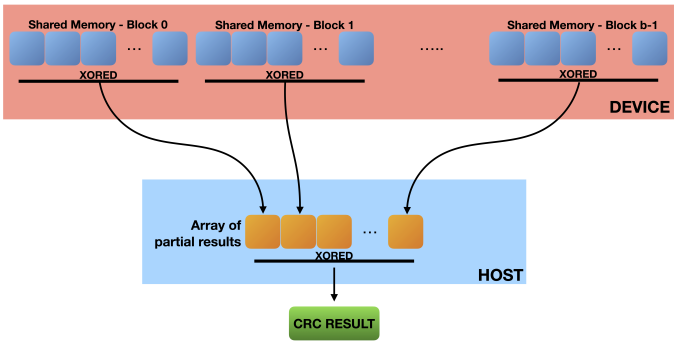


Figure 3. XOR OPERATIONS PERFORM BY DEVICE IN RELATION WITH THE ONE PERFORMS BY HOST

Each value of this array will be the XORed value of all the Galois Field multiplication of the message chunks of that specific thread block. Therefore the host has to perform the XOR operation over the array of partial results to find the resulting CRC code. The Figure 3 shows how the CRC result is obtained by the xoring computation over the thread blocks.

VII. Performance analysis

Our tests have been performed on the NVIDIA Tesla K40c GPU with the following key features:

- Architecture: Kepler;
- N of multiprocessor: 15;
- Number of GPUs: 1 x GK110B;
- Number of CUDA cores: 2880;
- Threads max per multiprocessor: 2048;
- Threads max per block: 1024;
- Warp size: 32;
- L1 Cache: 16 KB (per SMX);
- L2 Cache: 1536 KB;
- Base Clock: 745 MHz;
- Boost Clock: 876 MHz;
- Memory type: GDDR5;
- Maximum global memory amount: 12 GB;
- Memory bus width: 384 Bit;
- Memory clock speed: 6008 MHz;
- Memory bandwidth: 288.4 GB/s;
- Shared memory per block: 48 KB;
- Registers per block: 65536;
- Constant memory: 64 KB;
- Max power consumption: 235 W;
- Processor core clock: 745 MHz.

Our CUDA kernels of the parallel CRC, that we have designed based on the considerations explained above, counts a **number of registers equal to 11** for both the standard implementation and the reduction implementation. With a so low number of registers, our PCRC implementation won't never have problems in registers allocation, unless very old GPUs were used.

The shared memory size is equal to the block size, but each entry of the shared memory is M-bit size, that is the size of the CRC code generated. As consequence, the shared memory allocated for each block of thread is $M * BlockSize$ bits. Considering a $BlockSize = 1024$ and a possible 64 bit

implementation of PCRC, with $M = 64$, the shared memory per block resulting will be of size $64 * 1024 = 65536bit = 8192byte = 8KB$. A **shared memory maximum size of 8 KB**, also in this case, shouldn't give problem of memory allocation in most of the still existing GPU's architectures. Moreover, with the test analysis listed below, we can see that we obtain better performance results with a low block size value, therefore the size of the shared memory per block also decreases.

With our software implementation we have achieved the following results. Consider that all **the speedup obtained includes, in the time calculation, the overhead of memory data exchange between host and device**.

A. $N = 2^{16}$. $BlockSize = 64$. $StreamDim = 4$. $SegSize = N/StreamDim$

Test PCRC 64 blocksize	Speedup
PCRC8	34.71340
PCRC8 with reduction	37.40804
PCRC8 with task parallelism	27.86828
PCRC8 bytewise comparison	2.58712
PCRC8 bytewise comparison with reduction:	2.61057
PCRC8 bytewise comparison with task parallelism	1.93434
PCRC16	34.98040
PCRC16 with reduction	40.59672
PCRC16 with task parallelism	26.11727
PCRC16 bytewise comparison	3.44060
PCRC16 bytewise comparison with reduction:	3.85280
PCRC16 bytewise comparison with task parallelism	1.51222
PCRC32	20.23205
PCRC32 with reduction	20.01593
PCRC32 with task parallelism	8.46386
PCRC32 bytewise comparison	1.78715
PCRC32 bytewise comparison with reduction:	1.86166
PCRC32 bytewise comparison with task parallelism	0.71914

B. $N = 2^{16}$. $BlockSize = 128$. $StreamDim = 4$. $SegSize = N/StreamDim$

Test PCRC 128 blocksize	Speedup
PCRC8	34.63720
PCRC8 with reduction	37.41059
PCRC8 with task parallelism	28.19650
PCRC8 bytewise comparison	2.24552
PCRC8 bytewise comparison with reduction:	2.70207
PCRC8 bytewise comparison with task parallelism	2.02773
PCRC16	34.14637
PCRC16 with reduction	40.48397
PCRC16 with task parallelism	25.37525
PCRC16 bytewise comparison	3.57066
PCRC16 bytewise comparison with reduction:	3.60613
PCRC16 bytewise comparison with task parallelism	1.49754
PCRC32	20.41607
PCRC32 with reduction	21.54121
PCRC32 with task parallelism	8.51250
PCRC32 bytewise comparison	1.89645
PCRC32 bytewise comparison with reduction:	1.85455
PCRC32 bytewise comparison with task parallelism	0.79350

C. $N = 2^{16}$. $BlockSize = 256$. $StreamDim = 4$. $SegSize = N/StreamDim$

Test PCRC 256 blocksize	Speedup
PCRC8	35.11670
PCRC8 with reduction	37.36065
PCRC8 with task parallelism	29.85230
PCRC8 bytewise comparison	2.66612
PCRC8 bytewise comparison with reduction:	2.71714
PCRC8 bytewise comparison with task parallelism	2.00425
PCRC16	33.09628
PCRC16 with reduction	39.25975
PCRC16 with task parallelism	27.58368
PCRC16 bytewise comparison	3.53824
PCRC16 bytewise comparison with reduction:	3.66416
PCRC16 bytewise comparison with task parallelism	1.54122
PCRC32	20.29566
PCRC32 with reduction	21.39834
PCRC32 with task parallelism	7.98474
PCRC32 bytewise comparison	1.75776
PCRC32 bytewise comparison with reduction:	1.80291
PCRC32 bytewise comparison with task parallelism	0.68456

D. $N = 2^{16}$. $BlockSize = 128$. $StreamDim = 2$. $SegSize = N/StreamDim$

Test PCRC 128 blocksize	Speedup
PCRC8 with task parallelism	32.97171
PCRC8 bytewise comparison with task parallelism	2.48787
PCRC16 with task parallelism	30.44083
PCRC16 bytewise comparison with task parallelism	2.80089
PCRC32 with task parallelism	7.35099
PCRC32 bytewise comparison with task parallelism	0.72902

E. $N = 2^{16}$. $BlockSize = 128$. $StreamDim = 8$. $SegSize = N/StreamDim$

Test PCRC 128 blocksize	Speedup
PCRC8 with task parallelism	11.97966
PCRC8 bytewise comparison with task parallelism	1.20074
PCRC16 with task parallelism	11.68695
PCRC16 bytewise comparison with task parallelism	1.58960
PCRC32 with task parallelism	6.38649
PCRC32 bytewise comparison with task parallelism	0.65893

F. $N = 2^{16}$. $BlockSize = 128$. $StreamDim = 4$. $SegSize = N/8$

Test PCRC 128 blocksize	Speedup
PCRC8 with task parallelism	17.13281
PCRC8 bytewise comparison with task parallelism	0.47928
PCRC16 with task parallelism	16.70620
PCRC16 bytewise comparison with task parallelism	1.51226
PCRC32 with task parallelism	7.47568
PCRC32 bytewise comparison with task parallelism	0.52763

G. $N = 2^{10}$. $BlockSize = 128$. $StreamDim = 4$. $SegSize = N/StreamDim$

Test PCRC 128 blocksize bitwise limit	Speedup
PCRC8	0.60038
PCRC8 with reduction	1.29636
PCRC8 with task parallelism	0.78312
PCRC16	0.93035
PCRC16 with reduction	1.28736
PCRC16 with task parallelism	0.95668
PCRC32	1.03503
PCRC32 with reduction	0.75085
PCRC32 with task parallelism	0.38226

H. $N = 2^{13}$. $BlockSize = 128$. $StreamDim = 4$. $SegSize = N/StreamDim$

Test PCRC 128 blocksize bytewise limit	Speedup
PCRC8 bytewise comparison	0.36369
PCRC8 bytewise comparison with reduction:	0.95555
PCRC8 bytewise comparison with task parallelism	0.90212
PCRC16 bytewise comparison	0.48797
PCRC16 bytewise comparison with reduction:	1.44774
PCRC16 bytewise comparison with task parallelism	1.02833
PCRC32 bytewise comparison	0.46930
PCRC32 bytewise comparison with reduction:	0.68780
PCRC32 bytewise comparison with task parallelism	0.28820

I. NVIDIA Nsight Graphics

NVIDIA Nsight Graphics is a low overhead performance analysis tool designed to provide insights developers need to optimize their software.

The Figure 4 shows the standard algorithm profiling where you can see on the left the kernel execution rates compared to memory transfer.

In the Figure 5 you can see how this time kernel execution prevails because there is less divergence.

With the Figure 6 and 7 you can see that execution is split over multiple streams and that in some cases, in the timeline, kernel execution and memory transfer occurs in parallel.

VIII. Conclusions

This paper shows a systematic method to calculate CRC in parallel using the Galois Field property. The method can be easily expanded to all feasible parallel-input bits for fast CRC calculation without added complexity in the developing process or practical limitation like the size of lookup tables needed in other approaches.

Theoretically this level of performance is hundreds of times faster than bitwise CRC algorithm or tens of times faster than bytewise parallel CRC algorithm.

The implementation with tasks parallelism does not give better performance because the overhead of the technique is not positively compensated on the improvement that leads to performance. Anyway the best result with tasks parallelism is obtained for $StreamDim = 2$ and $SegSize = N/StreamDim$, with a *25x* average speedup compared to bitwise and a *2x* average speedup compared to bytewise.

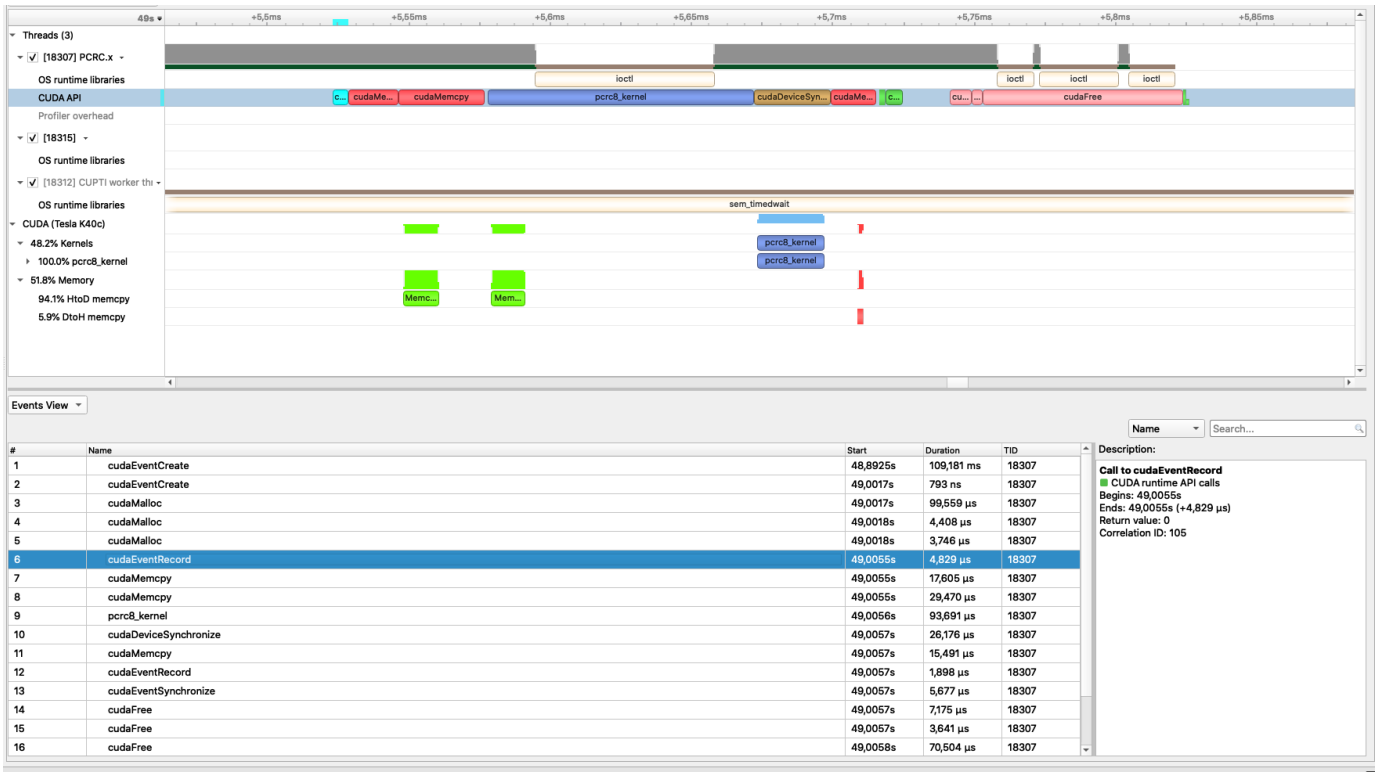
In general the best performances that we obtained are with $BlockSize = 128$. The standard kernel implementation give us a *34x* average speedup compared to bitwise CRC algorithm and a *2.5x* average speedup compared to bytewise. Using the reduction algorithm that reduces the divergence of the threads, we obtain an average increase of the performance compared to the standard solution of some speedup units, that is *40x* speedup compared to bitwise CRC and *3.5x* speedup compared to bytewise.

In detail, what we have seen during the execution of the tests is that $N = 2^9$ and $N = 2^{10}$ is the limit between positive and negative speedup of the PCRC implementations compared to bitwise CRC, while $N = 2^{12}$ and $N = 2^{13}$ is the limit between positive and negative speedup of PCRC implementations compared to bytewise CRC.

All our tests were performed with the input $N = 2^{16}$. With input from $N = 2^{10}$ up to $N = 2^{16}$ we have an exponential increase in speedup. With $N > 2^{16}$ we have that the speedup still increases, but with a linear trend with respect to the size of N.

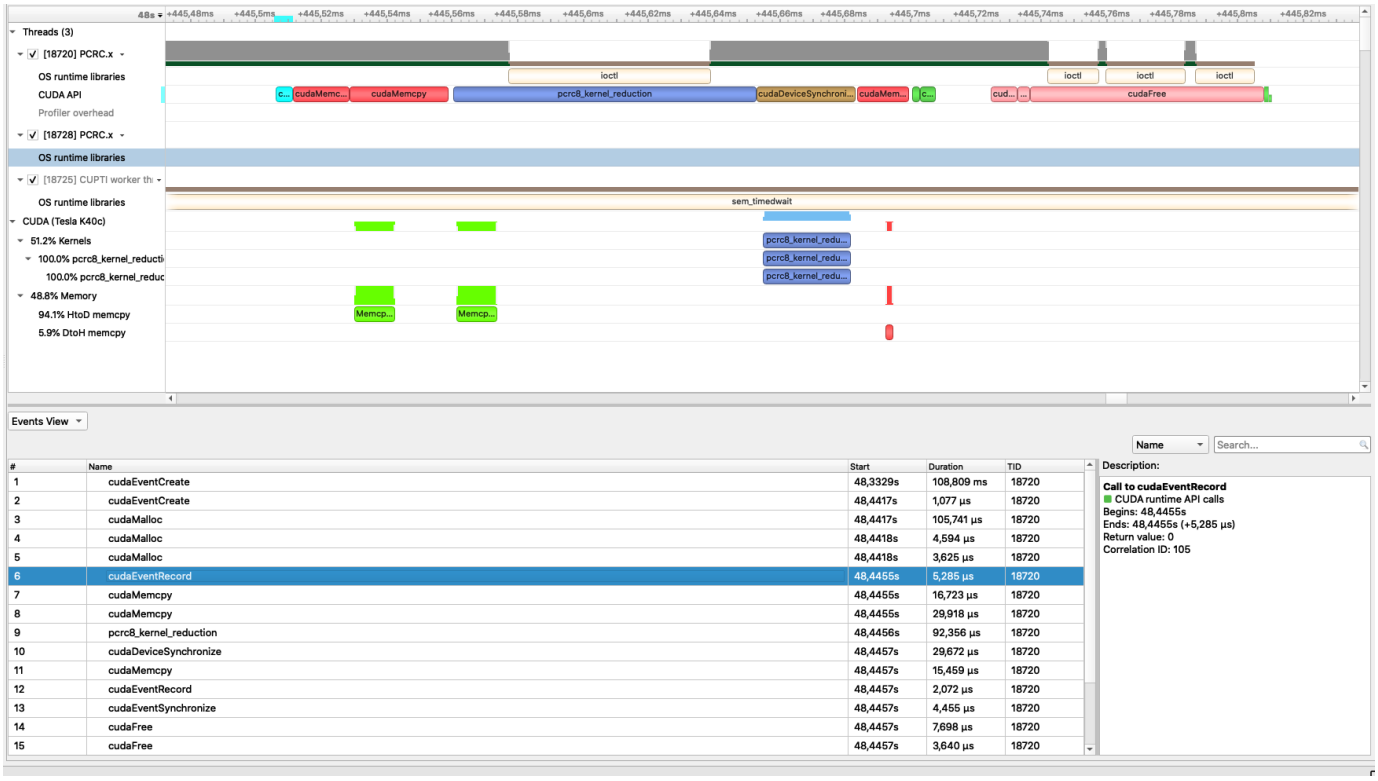
Based on our PCRC data input that we used for test, we can say that our CRC solution could be a really good choice, with a significant increase in performance, for devices using Ethernet protocol (maximum packet size 2^{16}), but not for embedded devices using protocols like ZigBee (maximum packet size 2^7).

From these results we can say that the PCRC speedup is always more than *2.5x* with a maximum of *40x*.



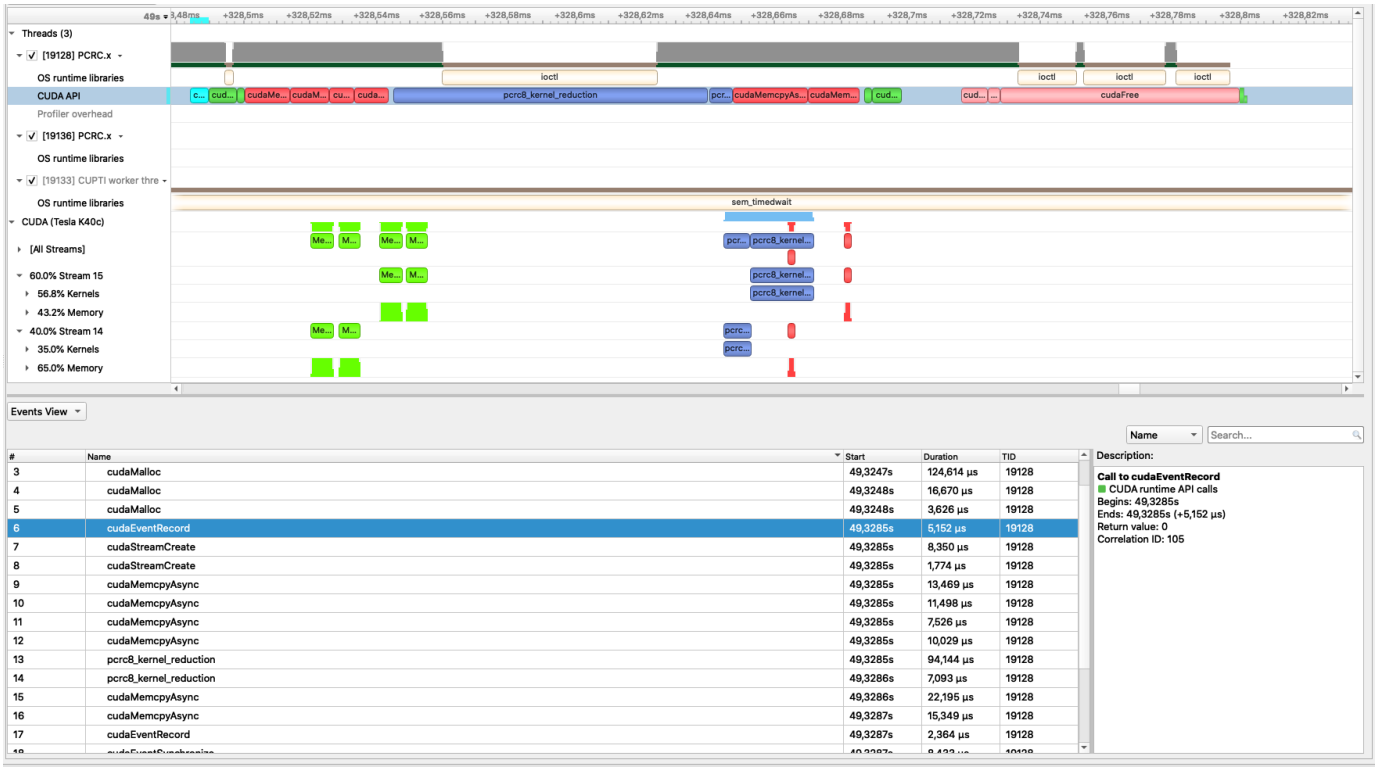
#	Name	Start	Duration	TID
1	cudaEventCreate	48,8925s	109,181 ms	18307
2	cudaEventCreate	49,0017s	793 ns	18307
3	cudaMalloc	49,0017s	99,559 µs	18307
4	cudaMalloc	49,0018s	4,408 µs	18307
5	cudaMalloc	49,0018s	3,746 µs	18307
6	cudaEventRecord	49,0055s	4,829 µs	18307
7	cudaMemcpy	49,0055s	17,605 µs	18307
8	cudaMemcpy	49,0055s	29,470 µs	18307
9	pcrc8_kernel	49,0056s	93,691 µs	18307
10	cudaDeviceSynchronize	49,0057s	26,176 µs	18307
11	cudaMemcpy	49,0057s	15,491 µs	18307
12	cudaEventRecord	49,0057s	1,898 µs	18307
13	cudaEventSynchronize	49,0057s	5,677 µs	18307
14	cudaFree	49,0057s	7,175 µs	18307
15	cudaFree	49,0057s	3,641 µs	18307
16	cudaFree	49,0058s	70,504 µs	18307

Figure 4. NVIDIA NSIGHT GRAPHICS PCRC8



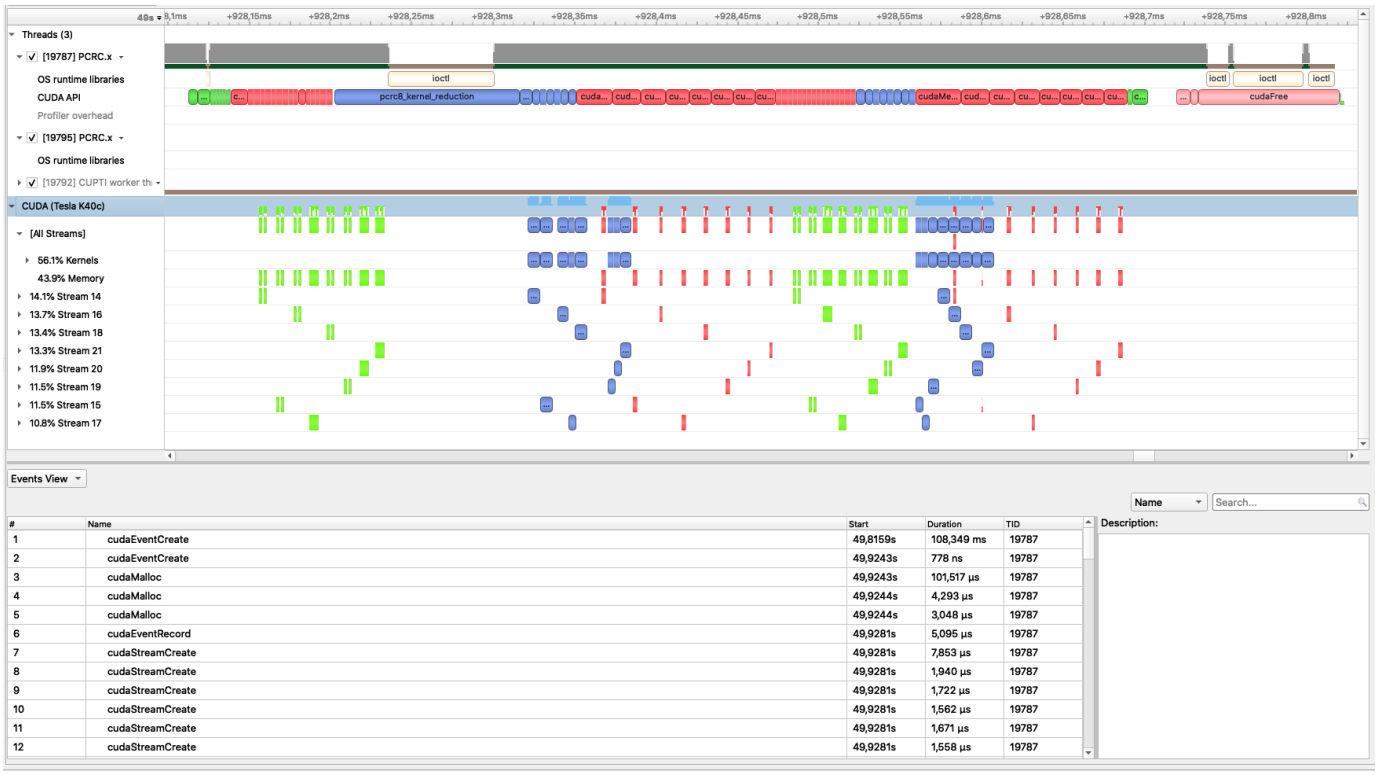
#	Name	Start	Duration	TID
1	cudaEventCreate	48,3329s	108,809 ms	18720
2	cudaEventCreate	48,4417s	1,077 µs	18720
3	cudaMalloc	48,4417s	105,741 µs	18720
4	cudaMalloc	48,4418s	4,594 µs	18720
5	cudaMalloc	48,4418s	3,625 µs	18720
6	cudaEventRecord	48,4455s	5,285 µs	18720
7	cudaMemcpy	48,4455s	16,723 µs	18720
8	cudaMemcpy	48,4455s	29,918 µs	18720
9	pcrc8_kernel_reduction	48,4456s	92,356 µs	18720
10	cudaDeviceSynchronize	48,4457s	29,672 µs	18720
11	cudaMemcpy	48,4457s	15,459 µs	18720
12	cudaEventRecord	48,4457s	2,072 µs	18720
13	cudaEventSynchronize	48,4457s	4,455 µs	18720
14	cudaFree	48,4457s	7,698 µs	18720
15	cudaFree	48,4457s	3,640 µs	18720

Figure 5. NVIDIA NSIGHT GRAPHICS PCRC8 REDUCTION



#	Name	Start	Duration	TID
3	cudaMalloc	49,3247s	124,614 µs	19128
4	cudaMalloc	49,3248s	16,670 µs	19128
5	cudaMalloc	49,3248s	3,626 µs	19128
6	cudaEventRecord	49,3285s	5,152 µs	19128
7	cudaStreamCreate	49,3285s	8,350 µs	19128
8	cudaStreamCreate	49,3285s	1,774 µs	19128
9	cudaMemcpyAsync	49,3285s	13,469 µs	19128
10	cudaMemcpyAsync	49,3285s	11,498 µs	19128
11	cudaMemcpyAsync	49,3285s	7,526 µs	19128
12	cudaMemcpyAsync	49,3285s	10,029 µs	19128
13	pcrc8_kernel_reduction	49,3285s	94,144 µs	19128
14	pcrc8_kernel_reduction	49,3286s	7,093 µs	19128
15	cudaMemcpyAsync	49,3286s	22,195 µs	19128
16	cudaMemcpyAsync	49,3287s	15,349 µs	19128
17	cudaEventRecord	49,3287s	2,364 µs	19128

Figure 6. NVIDIA NSIGHT GRAPHICS PCRC8 TASK PARALLELISM - CASE 1



#	Name	Start	Duration	TID
1	cudaEventCreate	49,8159s	108,349 ms	19787
2	cudaEventCreate	49,9243s	778 ns	19787
3	cudaMalloc	49,9243s	101,517 µs	19787
4	cudaMalloc	49,9244s	4,293 µs	19787
5	cudaMalloc	49,9244s	3,048 µs	19787
6	cudaEventRecord	49,9281s	5,095 µs	19787
7	cudaStreamCreate	49,9281s	7,853 µs	19787
8	cudaStreamCreate	49,9281s	1,940 µs	19787
9	cudaStreamCreate	49,9281s	1,722 µs	19787
10	cudaStreamCreate	49,9281s	1,562 µs	19787
11	cudaStreamCreate	49,9281s	1,671 µs	19787
12	cudaStreamCreate	49,9281s	1,558 µs	19787

Figure 7. NVIDIA NSIGHT GRAPHICS PCRC8 TASK PARALLELISM - CASE 2