Hardware-accelerated Galois Field Arithmetic on the ARMv8 Architecture

Markus Ongyerth
Advisor: Stephan Günther
Seminar Future Internet WS1415
Chair for Network Architectures and Services
Department of Computer Sience, Technische Universität München
Email: ongyerth@in.tum.de

ABSTRACT

A limiting factor for throughput of a network is the limit of throughput achievable with traditional routing. Network coding is a way to avoid this problem. A limiting factor for network coding is its inherent arithmetic complexity. This is particular true for high-throughput networks, but lower throughput and embedded systems suffer from the same limitations. This paper evaluates the performance, of different implementation and algorithms doing the descrete math required for network coding on an ARMv8 architecture and compares it to on an ARMv7 architecture. Since the ARMv7 is a 32bit architecture while ARMv8 is a 64bit, this benchmark shows the advantage of having larger general purpose registers. The different implementations compared in this paper also show the performance gain by taking advantage of the NEON SIMD extensions, which increase register size (even more) to 128bit.

1. INTRODUCTION

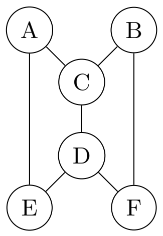
In theory, network coding allows to increase the throughput of a network to its upper bounds [6]. It uses intelligent broad- and multicasting to distribute information in a way that is more efficient than routing. To achieve this, packets are aggregated and encoded in a way that allows a receiving node to decode the original packets. One of the easiest examples to show when and how network coding increases the throughput of a network is the butterfly network which is shown in Figure 1a.

Each connection can transmit one packet of data per unit of time. The advantage of network coding in this kind of network becomes apparent when nodes A and B want to send data to nodes E and F. These nodes do not have to be the actual source or actual destination for this data but can be intermediate nodes as well. In a network that does not use any network coding this means that the connection between nodes C and D has to be used twice (once for A to F and once for B to E). If the network utilizes network coding, this can be avoided. In this case, node C receives both packets from nodes A and B separately and encode those into one common packet. Since there are only two packets to combine, this encoding can be done with a simple XOR operation. This common packet can than be transfered to node D as one packet. Node D then transmits the packet to nodes E and F and thus the connection between C and D is used only once. Nodes E and F receives the packet from A (for E) or the packet from B (for F) over their direct connection to the source node. With one of the source packets and the combined packet they are now able to decode the combined packet. Therefore obtaining both packets in a way that puts less strain on the transport medium.

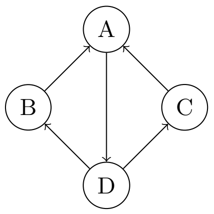
In this example there was only one situation with only two packets that could benefit from network coding. For wireless networks with their inherent broadcast nature every packet two packets sent at the same time collide with each other, voiding both packets. This slows down the network since at a given time, only one, of possibly many, nodes in range of each other can send a packet without creating a collision. As the network gets more complicated–especially in mesh-networks–this becomes a problem. With network coding information can be exchanged over a network using less transmissions. By sending fewer packets the medium can be used more efficiently which increases the overall throughput of the network.

As mentioned, in this example the encoding and decoding of packets can be done with XOR operations. For more complicated networks, it is necessary to combine more than two packets and therefore more complicated encoding and decoding algorithms have to be applied. These algorithms have an inherent computational complexity, which prevents their practical deployment. In [4] Günther et al. have created and published a library, that does efficient finite field arithmetic needed for network coding. This library contains algorithms optimized to use SIMD extensions, in the case of ARM the NEON extension. They published benchmark results created with this library on an x86 CPU and ARMv7 CPU. The ARMv7 is a Cortex A15 and has a 32bit architecture. In this paper we compare the benchmark results of the library running on an ARMv8 processor, which has a 64bit architecture. This paper does not compare results to a x86 CPU, since traditionally x86 is aimed at high power and high performance while ARM is tuned for low power consumption. Therefore the ARM CPUs have a considerable lack of performance compared to the x86 CPU. At the time this paper is written the only processor comercially available with an ARMv8 architecture is the Apple A7, which is used for the benchmarks in this paper.

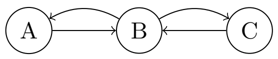
The remainder of this paper is organized as follows: first Section 2 describes Galois fields. Section 3 introduces and describes the hardware in this paper, while Section 4 introduces the algorithms used. In Section 5 the benchmark



(a) Butterfly network



(b) Diamond network



(c) Wireless relay

results are presented and evaluated especially in comparison to the results in [4] made on the ARMv7 CPU. In Section 6 a few general remarks are made. Section 7 concludes the paper.

2. GALOIS FIELD

A Finite field, also called Galois field. We denote them as GF(p^n), where p is a prime number and n is a positive integer. This is possible, because the number of elements in a finite field is always a power of a prime number and all finite fields with the same size are isomorphic [4]. In this paper only binary extensions fields are considered, i.e., where the number of elements in the field is of order $q = 2^n$. Elements of this field can be expressed as polynomials over $\mathbb{F}_2$ of degree $n-1$, i.e.,

$$F_q[x] = \left\{ \sum_{i=0}^{n-1} a_i x^i \middle| a_i \in \mathbb{F}_2 \right\}. \tag{1}$$

The coefficients $a_i \in \mathbb{F}_2$ are represented by individual bits which allows for efficient processing on today's processor architectures. For the scope of this paper we focus on the finite fields of order $n = \{2, 4, 16, 256\}$, namely GF(2), GF(2^2), GF(2^4) and GF(2^8). These are the most important fields for network coding since the overhead for their coefficients is still kept relatively small compared to larger fields such as GF(2^{16}) and GF(2^{32}) and their elements naturally fit into processor registers.

Addition of $a, b \in F_q[x]$ is defined as:

$$a(x) + b(x) = \sum_{i=0}^{n-1} a_i x^i + \sum_{i=0}^{n-1} b_i x^i = \sum_{i=0}^{n-1} (a_i + b_i) x^i. \tag{2}$$

Note that coefficients are added according to the rules of the additive group associated with $\mathbb{F}_2$, meaning that addition is done modulo 2. Addition modulo 2 reduces to a simple XOR operation.

For Multiplication a polynomial r of degree n, that is irreducible over $F_q[x]$ is required. Irreducible means, that the polynomial r cannot be expressed as the product of two polynomials in $F_q[x]$. Such a polynomial is guaranteed to exist as shown in [5], but generaly not unique. Multiplication in the obtained finite field depends on the polynomial g. The product of $a, c \in F_q[x]$ is the unique remainder

$$b(x) = (a(x) \cdot c(x)) \bmod r(x). \tag{3}$$

The polynomial r is sometimes called *reduction polynomial* since it constrains the maximum degree of the result $b \in F_q[x]$. Because r is irreducible, it is guaranteed that $a(x) \cdot b(x)$ does not equal $r(x)$. This ensures that the reduction does not reduce a polinomial to zero and therefore that the multiplication result is–except for commutativity–unique.

Data words of n bit length are expressed as polynomials $a \in F_q[x]$. A vector $\underline{a} \in F_q^k[x]$ is used as representation of a data packet of length kn bit. A *generation* of N source packets can then be written as matrix $A = [\underline{a}_i \dots \underline{a}_N]$. A coded packet is obtained by

$$\underline{b} = Ac = \sum_{i=1}^{N} \underline{c_i a}_i, \tag{4}$$

where $c^T = [c_1 \dots c_N] \in F_q^N[x]$ denotes a vector of random coefficients which are drawn independently uniformly distributed from $F_q[x]$.

3. THE HARDWARE USED

Feature	Apple A7	Exynos5
Frequency	1.4 GHz	1.4 GHz
Cores	2	4(+4)
L1 Cache	64 kB/64 kB	32 kB/32 kB
L2 Cache	1 MB/	2 MB

Table 1: The hardware specifications of the devices used. Apple specification extracted from [1] and [3], while the Exynos5 specifications are from [4]

The Apple A7. The device used for the benchmarks that are newly made for this paper is an Apple iPad Mini, second generation. This device is used because at the time this paper is written (September 2014) the Apple A7 is the only commercially available processor with an ARMv8 and therefore an 64bit ARM architecture. There are different devices that use an Apple A7 processor, but the frequency the processor runs on does not differ much between the devices (1.3 GHz to 1.4 GHz). The frequency on the iPad matches the frequency of the board used for comparison.

The problem imposed by using a device with an Apple A7 is that Apple has not published much information about the processor's specification or even its frequency. Fortunately, others are interested in the technical specification of these devices as well. *Anandtech* has published an article, about the iPhone 5s and later the iPad Mini, which both use the same Apple A7 SOC, analyzing the processor specification, of this platform. The information in 1 is extracted from those articles.

The Exynos 5 Octa. The device used for comparisons in this paper, is a ODROID-XU Lite development board. Table 1 displays the specification of this device.

The Exynos5 Octa follows the ARM "big.LITTLE" system and actually has 4 "big" cores and 4 "small" cores, but only one of these groups is active at a time. This benchmark was always executed on the faster cores. The core-count itself, however, does not really have an impact on the results since the library is single-threaded.

Looking at raw numbers, the two processors are similar for this benchmark. The A7 has twice as much L1 cache as the Exynos, but the Exynos has twice the L2 cache. The two big differences between the two platforms are the core count - 2 to 8 or rather 4 - and the register width of the processors. But as mentioned before, the advantage in core-count does not matter for this benchmark since it is aimed at single-core throughput, and the difference in word width is one of the main points why those platforms are compared to each other.

4. THE ALGORITHMS

For a base performance to compare against, a simple table lookup algorithm is used. For this table lookup all possible products of two elements of the Galois field are precomputed and saved in an array. The multiplication is then done by retrieving those values from the array.

The *imul* [4] algorithm does not require SIMD extensions and can therefore be implemented using only general purpose registers. It can also benefit from wider SIMD registers if available. It is suitable to run on microarchitectures that does not have SIMD extensions. The downside to this algorithm is that it scales badly with the degree of the finite field used. The library contains different implementations of this algorithm, which differ in the register size used. There is a 32 bit version and a 64 bit version. Both those versions only use general purpose registers. There is also a version using SIMD registers and instructions. Those registers are at least 64 bit–128 bit on both platform used in this paper.

The *shuffle* [4] algorithm benchmark results are not considered in this paper since the benchmark for the iPad has to be built, with the **Apple-llvm** compiler, which currently does not support the intrinsics used for this algorithm. (September 2014)

5. MEASUREMENTS

The linear encoding throughput is compared using a generation size of *N= 16*. The *throughput* is defined as the total size of encoded packets over a time interval measured in Gbit/s. The benchmark is done for packet sizes ranging from 128 B to 8 MB. As baseline performance the table lookup is used. The packet size has a significant impact on the performance since there is overhead that has to be done for every packet. This overhead amortizes for larger packets. Therefore, a larger packet size yields a higher throughput. This general assumption holds true until the memory requirements of the working set exceed the cache sizes.

When the cache sizes are reached, the throughput becomes limited by memory performance. Figures 1a to 1d show the performance of the Apple A7 compared to the Exynos5 in GF(2) and GF(2^2). Generally speaking, the A7 is about 2 to 3 times faster than the Exynos5. This rule of thumb is not true for every packet size and at its peak the Exynos5 is even faster than the A7. The change in throughput for varying packet sizes is rather similar for both processors used.

As expected, the throughput increases until the memory requirements of the working set hit the L1 cache limit. After this point the throughput remains about the same until the memory required for the working set hits the size of the L2 cache. At this point there is a second degradation in throughput, since main memory performance now has an effect on the algorithms. This degradation of throughput is most visible in the NEON implementation. The Exynos5s performance drops drastically to about half its value. The A7s memory does not throttle the performance as significant, but the impact is still visible. A bigger difference between the two platforms is visible when the difference between the 64 bit and the 32 bit on a single platform implementations is analyzed. On the Exynos5 the *imul 64bit* is at best as fast as the *imul 32bit* or even slower. This is caused by the lack of native 64bit operations, which cut the performance on 64bit numbers at least in half. The A7, which has support for native 64bit operations, shows that this assumption is correct. On the A7 , the throughput behaves comparable to the results [4] got on `x86_64`. Namely the *imul 64bit* algorithm is about 1.5 times to twice as fast as the *imul 32bit*. This performance increase is possible because the CPU is able to process twice the amount of data per instruction than it can process in a single instruction on 32bit registers. This advantage gained by using larger registers is visible by looking at the *imul NEON* implementation as well. This implementation uses NEON SIMD extensions and therefore has 128bit wide registers, which makes it another 1.5 to 2 times faster than *imul 64bit*.

Figures 1e to 1h show the benchmark results for GF(2^4) and GF(2^8). In these larger fields the throughput does not change as much with packet size. It seems like the throughput gets more consistent with the size of the finite field the arithmetic is done in. The A7 shows nearly no change when it hits the cache sizes. Because the throughput gets more limited by the CPU performance and less by memory performance. The impact on throughput by exceeding cache size gets less significant for larger field sizes on the Exynos5 as well even though, it isn't as good as on the A7. The difference between the two processors does not divert much from the observations made for smaller field sizes. The A7 is still about 2 to 3 times faster and benefits from 64bit general purpose registers when possible.

Next to the cache limits, another interesting packet size is 1024Kib, especially in comparison of the two processors. The A7 has an unusual big drop in throughput here while the Exynos5 has an irregular increase of throughput at this point. So far there is no conclusive explanation for this anomaly on either of the platforms.

6. REMARKS

[4] concluded, that the trend to heterogenous microarchitectures where both CPU and (integrated) GPU access the same memory might bring a similar gain when low level arithmetic is outsourced to the graphics processor.

It seems that there is an SRAM cache on the A7 SOC [2], that may be also accessible by the integrated GPU. For now this is guesswork and there is no easy way to prove and or test since Apple does not release any information about the SOC. But potentially this may be used to transfer data to and from the GPU and bypass the usual high overhead induced by memory transfer from CPU to GPU, which makes these operations on GPU, far more viable.

Something interesting to note about the two platforms compared in this paper is, although with different frequencies both of the platforms have an actual real live use case and those use cases are rather similar: they have been used as SOC for a smartphone. The Apple A7 is also used in the iPhone 5s and the Exynos5 is used in one version of the Samsung Galaxy S4. Both have been released in 2013 and competed on the market. Comparing the two platforms with this benchmark is not fair, since the benchmark only uses a single core and the Exynos5 which is significantly slower in this paper has more cores.

7. CONCLUSION

With scalar implementation on general purpose registers all field sizes larger than GF(2^2) are limited to less than 1 Gbit/s on the A7. Using SIMD extensions GF(2^4) reaches 1 Gbit/s. The performance of the *imul NEON* is about twice the performance of the *imul 64 bit*. The *shuffle* algorithm provided by *libmoepgf* cannot be used for a benchmark on the A7 yet. This algorithm outperforms the *imul* implementations on all tests performed [4]. It will be interesting to see whether or not the *shuffle* algorithm outperforms the algorithms on the A7 as well, and, if it does, what kind of performance it achieves on the A7.

References

[1] *Anandtech: The iPhone 5s Review*, http://www.anandtech.com/show/7335/the-iphone-5s-review/3

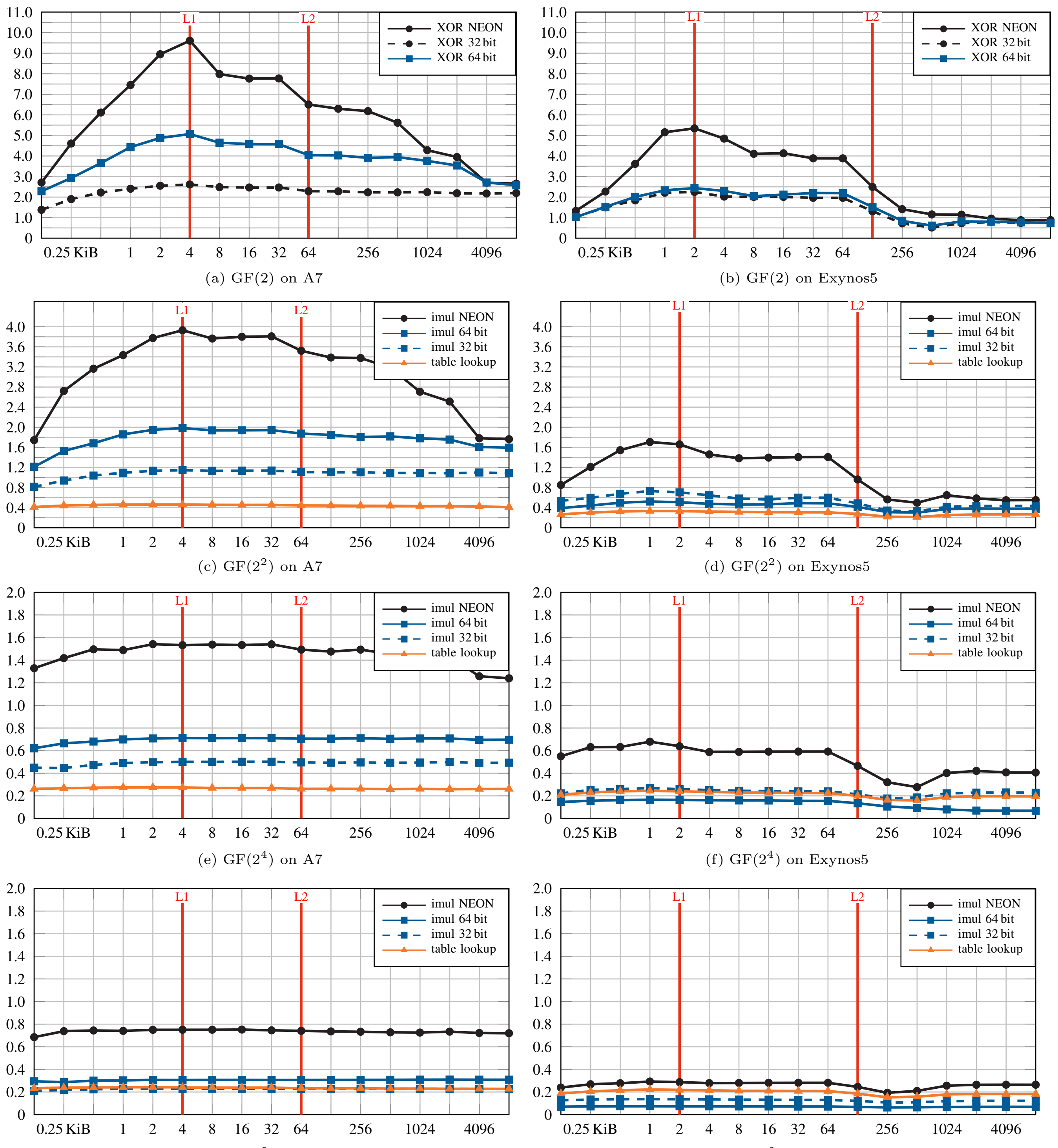
[2] *Anandtech: The iPad Air Review*, http://anandtech.com/show/7460/apple-ipad-air-review/2

[3] *Anandtech: The iPad Air Review*, http://anandtech.com/show/7460/apple-ipad-air-review/3

[4] Stephan M. Günther, Maximilian Riemensberger, Wolfgang Utschick: Efficient GF Arithmetic for Linear Network Coding using Hardware SIMD Extensions

[5] D. Hankerson, A. Menezes, and S. Vanstone: Guide to Elliptic Curve Cryptography, 1st ed., Jan. 2004.

[6] Shuo-Yen Robert Li, Senior Member, IEEE, Raymond W. Yeung, Fellow, IEEE, and Ning Cai: Linear Network Coding, IEEE transactions on infromation theory, vol. 49, no. 2, february 2003



(a) GF(2) on A7

(b) GF(2) on Exynos5

(c) GF(2^2) on A7

(d) GF(2^2) on Exynos5

(e) GF(2^4) on A7

(f) GF(2^4) on Exynos5

(g) GF(2^8) on A7

(h) GF(2^8) on Exynos5

Figure 1: Encoding throughput [Gbit/s] **for packet sizes varying from** 128 B **to** 8 MiB **in a generation of size 16 on Apple A7 (left) and Samsung Exynos5 Octa (right) both at** 1.4 GHz **with marks for both L1 and L2 cache sizes.**