INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

SIMD processing of AES on the Raspberry Pi's GPU

Yannek Rixen

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

SIMD processing of AES on the Raspberry Pi's GPU

Yannek Rixen

Aufgabensteller:	Prof. Dr. D. Kranzlmüller
Betreuer:	Tobias Guggemos Jan Schmidt
Abgabetermin:	9. Dezember 2019

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den December 18, 2019

. . . . . .
(Unterschrift des Kandidaten)

Abstract

The Raspberry Pi's GPU, called VideoCore IV 3D, is integrated on the SoC. Because Raspberry Pi's are often used without a monitor connected to it, it is massively underutilized while holding most of the Raspberry Pi's computing capability. The QPULib, a C++ library for easy programming of VideoCore IV 3Ds computing cores, called QPUs, provides an abstraction layer between the software and hardware. The Advanced Encryption Standard is of the most used encryption algorithms and is used in all kinds of applications. To take the workload from the CPU, we implemented the Advanced Encryption Standard's ECB Mode on the Raspberry Pi's GPU and built an OpenSSL-engine with it. The implementation was held back by hardware features that could not be accessed using the library without making alterations to it. In the end, we found a slight performance improvement over the CPU implementation of OpenSSL on a Raspberry Pi 2 depending on the Raspberry Pi's configuration and data length to be encrypted. With more optimizations, the results could be more promising.

Contents

1 Introduction

In times where clock speed increases are stagnating, where memory accesses are slow and manufacturing nodes seem to come to a halt, there are only a few ways left to improve computing performance: increasingly complex architectures for more instructions per clock, purpose-built ICs, and parallelization. All of them are complex, but most computing platforms already have all the preconditions implemented to parallelize on a scale far greater than any modern CPU could: A GPU. This is not just true for desktop PCs, but also laptops and even handheld devices, like mobile phones. On most of these tiny devices, the SoC (System on Chip) integrates the CPU, the RAM, and also the GPU. The Raspberry Pi is no exception. This tiny device, even though it weighs just 7 to 45grams, depending on the revision and model, and exceeds a credit card just in a single dimension, can host a full Linux system or even a Windows 10 IoT Core. While the ARM Cortex based CPU lacks speed, the VideoCore IV 3D GPU of the Raspberry Pi is a highly parallelized subsystem with loads of power, compared to its CPU.

The Advanced Encryption Standard (AES) is one of the few widespread encryption algorithms used in all kinds of applications. It is so common that AMD and Intel even implement the AES-NI instruction set, which can speed up AES encryption and decryption significantly, into their CPUs since AMD's Bulldozer and Intel's Clarkdale (first Core i-series) architectures respectively.

The Raspberry Pi's GPU is highly underutilized, especially if it is in a headless setup, without a monitor connected, and no GUI is loaded. Here lies most of its computing power. The VideoCore IV 3D GPU of the Raspberry Pi is capable of up to 24 GFLOPS (floating-point operations per second) while the CPU on the same System-On-A-Chip (SoC) is not even capable of 1 GFLOPS in the case of the Raspberry Pi 1 Model B. Currently, there is no viable implementation of AES for the VideoCore IV 3D available. To offload the work of encryption from the busy CPU to the possibly idle GPU of the Raspberry Pi, a new implementation of AES for its GPU is needed. Ideally, the implementation on the GPU can also provide a speedup over AES on the Raspberry Pi's CPU.

Approach of this Thesis

First, we have a look into Rijndael and the AES algorithm in the first chapter. At this stage, we focus on the details of the algorithm and try to get a profound understanding of it. Then, a general understanding of the architecture of the Raspberry Pi's GPU hardware is required to be able to get to an in-depth view of the VideoCore IV 3D architecture and its instruction set. We explain the functionality and structure of the QPULib, a library that we use for our implementation, and get to know its features and limitations. At the end of chapter two, we introduce some related work.

In the third chapter, two concepts for extending the QPULib are shown, and one of them implemented. This implementation comes in handy later in the actual implementation of AES in the fourth chapter. There, two AES different approaches to implementing the AES on the Raspberry Pi's GPU using the QPULib are implemented.

In the fifth chapter, the latter implementation is tested and compared to the OpenSSL's CPU implementation on the Raspberry Pi. The thesis is closed with summing up the results and a look into future work.

2 Background

In this chapter, we are trying to gather all the necessary information for our implementation. We need to know what and how we can parallelize using the Raspberry Pis GPU. First, we need to figure out the Advanced Encryption Standard, what it does, and how it works. An early idea of the restrictions by the Raspberry Pis hardware is an obvious benefit to the first success of the implementation that should be standing at the end of this thesis. We also need to figure out if there is any software available that provides an abstraction layer for the hardware and could, therefore, help with more straightforward implementation. At the end of this chapter, we are having a look at other works that are closely related to ours.

2.1 The Advanced Encryption Standard

The Advanced Encryption Standard is the successor of the Data Encryption Standard. The Rijndael algorithm has been selected from a total of 14 candidates accepted to The First Advanced Encryption Standard Candidate Conference held by the National Institute of Standards and Technology (NIST) in August 1998. After the conference, these 14 algorithms were evaluated by NIST for security, cost of implementation, and algorithm- and implementation characteristics. Only non-licensed and non-patented algorithms would be accepted as candidates.

At the second AES conference, held in Rome in March 1999 together with the Fast Software Encryption Workshop, it became clear that Rijndael was not only one of the fastest on a then-popular Intel Pentium processor, but it also fitted 8-bit processors exceptionally well. Later, in August 1999, Rijndael was announced to be one of the five finalists and on the 2nd October 2002, to be the winner of the AES selection [DR02].

> Rijndael appears to be consistently a very good performer in both hardware and software across a wide range of computing environments regardless of its use in feedback or non-feedback modes. Its key setup time is excellent, and its key agility is good. Rijndael's very low memory requirements make it very well suited for restrictedspace environments, in which it also demonstrates excellent performance. Rijndael's operations are among the easiest to defend against power and timing attacks. Additionally, it appears that some defense can be provided against such attacks without significantly impacting Rijndael's performance. Finally, Rijndael's internal round structure appears to have good potential to benefit from instruction-level parallelism. ([DR02, 23])

2.1.1 Rijndael

The Rijndael algorithm is a round-based block cipher developed by Vincent Rijmen and Joan Daemen from Belgium. Its block length can be any multiple of 32 bits in the range of

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Figure 2.1: Byte Order for 128-bit block size and key-lengths in Rijndael

0	4	8	12	16	20
1	5	9	13	17	21
2	6	10	14	18	22
3	7	11	15	19	23

Figure 2.2: Byte Order for 192-bit block size and key-lengths in Rijndael

128 to 256 bits. The block that the Rijndael algorithm operates on is called state. The one-dimensional byte vector from main memory is represented as a matrix with the byte order shown in 2.1. This applies to all steps of the algorithm and also the key. While Rijndaels block length and key size can be any multiple of 32 bits in the range of 128 to 256, AES limits the block length to 128 bits and key lengths of either 128 bit, 192 bit or 256 bit. These variations are known as AES128, AES192 and AES256 respectively [DR02, 31].

2.1.2 Algorithm Overview

For all operations the byte order of the state matrix is as shown in figure 2.1. This also applies for the key. For keys longer than 128-bit, the block is extended by columns to the right. The algorithm starts with a key-addition described in 2.1.2, followed by several rounds of byte-substitution, row-shifting, column-mixing, and further key-additions. The exact number of rounds for each algorithm variant is shown in table 2.1. The final "round" lacks the Mix Columns step.

```
1 function aes_encrypt(char * state, char * expanded_key, int max_rounds) {
2     int rounds = 0; // 10 for 128bit key size; 12 for 192bit key size; 14 for
      256bit key size
3
4     add_round_key(&state, &expanded_key, round);
5
6     while(round++ < max_rounds) {
7         sub_bytes(&state);
```

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

Figure 2.3: Byte Order for 256-bit block size and key-lengths in Rijndael

```
        shift_rows(&state);
        mix_columns(&state);
        add_round_key(&state, &expanded_key, round);
    }

    sub_bytes(&state);
    shift_rows(&state);
    add_round_key(&state, &expanded_key, round);
    return;
}
```

Listing 2.1: AES encryption function

Decryption The decryption block cipher is exactly the encryption block cipher in reverse, with all function calls reversed and all functions implement their counterparts inverse functionality. The block lengths, key lengths and numbers of rounds stay the same.

```
function aes_decrypt(char * state, char * expanded_key, int rounds) {
    int round = rounds; // 10 for 128bit key size; 12 for 192bit key size; 14
    for 256bit key size

    add_round_key(&state, &expanded_key, round);
    shift_rows_inverse(&state);
    sub_bytes_inverse(&state);

    while(round--> 0) {
        add_round_key(&state, &expanded_key, round);
        mix_columns_inverse(&state);
        shift_rows_inverse(&state);
        sub_bytes_inverse(&state);
    }

    add_round_key(&state, &expanded_key, round);
    return;
}
```

Listing 2.2: AES decryption function

length of input key		rounds	length of expanded key	multiple of input key length
bits	bytes		bytes	
128	16	10	176	11
192	24	12	208	8
256	32	14	240	7

Table 2.1: Key lengths in AES

Expand Key

In AES, the key length can be either 128, 192, or 256 bits. Together with the changing number of rounds depending on the key length, they form the differentiation between AES128, AES192, and AES256, respectively. The length of the expanded key in bytes is a fixed multiple of the block size for each of the input key lengths so that in each round a different part of the expanded key can be applied to the state matrix.

Add Round Key

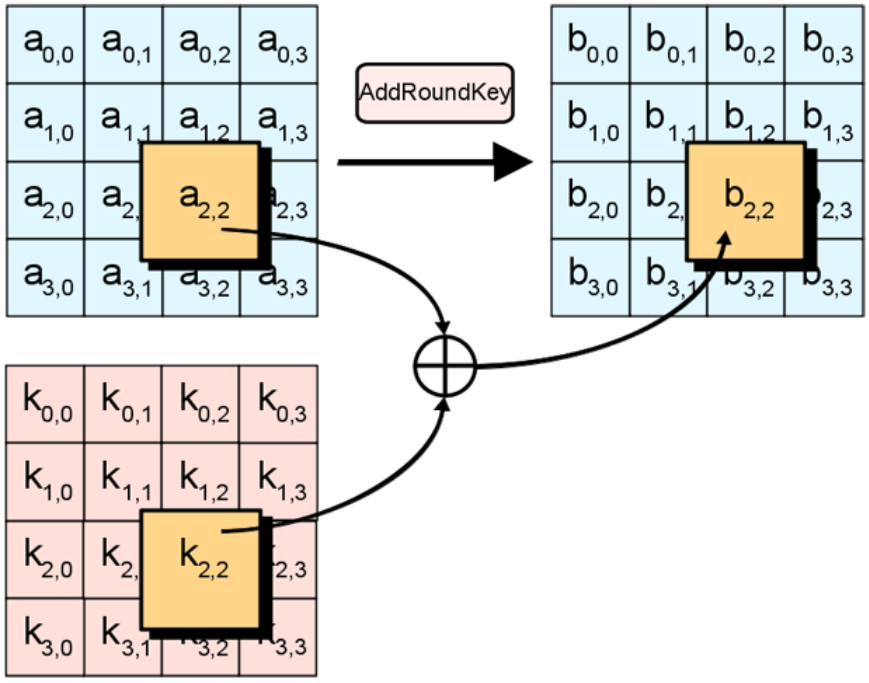

Figure 2.4: Add Round Key visualisation [Haa08]

The application of the expanded key to the state matrix is a simple XOR of consecutive 16 bytes with the state matrix beginning with the first 16 bytes of the input key.

Algorithm 1 Add Round Key

Input: $expanded\_key[rounds][16]$
$state\_matrix[16]$
Output: $state\_matrix[16]$
for $0 \leq i < 16$ **do**
$state\_matrix[i] = state\_matrix[i]^e xpanded\_key[round][i]$
end for
return $state\_matrix$

Sub Bytes

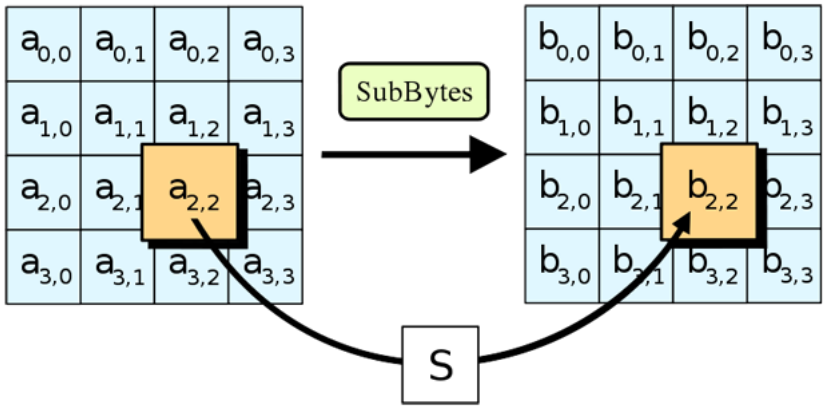

Figure 2.5: Sub Bytes visualisation [Haa08]

In the Sub Bytes step, every byte of the state matrix is substituted by a byte from a non-linear substitution box. This substitution box is filled with each multiplicative inverse in the Galois Field $GF(2^8)$ at the position of its original value.

$$SBox[x] = x^{-1}$$

For the inverse of this step an inverse substitution box is used, where the index and its corresponding value are swapped to form a new inverse substitution box.

$$inverseSBox[x] = SBox[x^{-1}]$$

These substitution boxes are used as lookup tables. This way we don't have to compute each value on the fly every time.

Algorithm 2 Add Round Key

Input: $sub\_box[256]$
$state\_matrix[16]$
Output: $state\_matrix[16]$
for $0 \leq i < 16$ **do**
$state\_matrix[i] = sub\_box[state\_matrix[i]]$
end for
return $state\_matrix$

Shift Rows

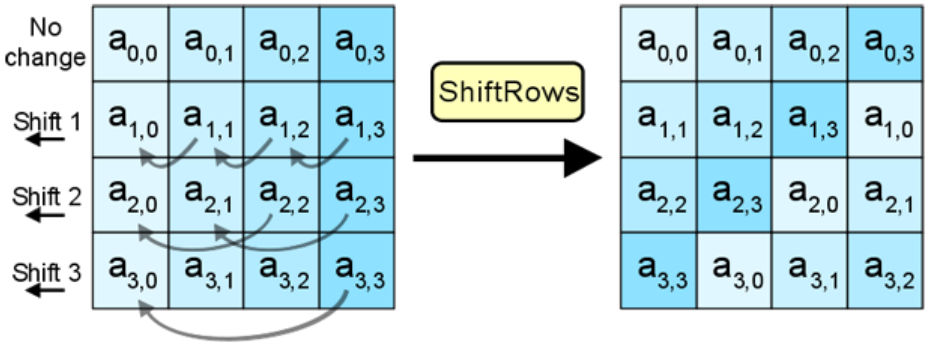

Figure 2.6: Shift Rows visualization [Haa08]

In shift rows, each row of the state matrix is rotated by its row's index, meaning the first row is not shifted left, the second row by one Byte, the third by two Bytes and the fourth by three Bytes.

Algorithm 3 Shift Rows

Input: $state\_matrix[4][4]$
Output: $state\_matrix[4][4]$
for $0 \leq i < 4$ **do**
$rotate\ state\_matrix[i]\ left\ by\ i$
end for
return $state\_matrix$

Mix Columns

Most of the complexity of the AES algorithm comes from the mix columns operation. Each column of the state matrix is multiplied by a circulant maximum-distance-separable (MDS) matrix in the Rijndael finite Galois Field $GF(2^8)$. The Galois multiplication, with the second operand being 2, is called "xtime" [DR02, 53]. Multiplications with constants in the Galois Finite Field can be implemented using chaining of the xtime function and exclusive-ors with the state in between. Addition in the Galois Field is a simple exclusive-or operation [DR02, 54]

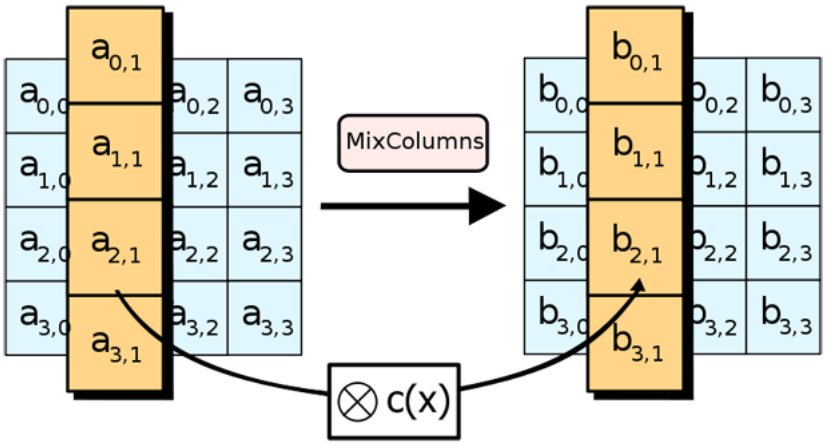

Figure 2.7: Mix Columns visualisation [Haa08]

Algorithm 4 Mix Columns

Input: $state\_matrix[4][4]$
Output: $state\_matrix[4][4]$
for $0 \leq i < 4$ **do**

$$\begin{bmatrix} state\_matrix_{0,i} \\ state\_matrix_{1,i} \\ state\_matrix_{2,i} \\ state\_matrix_{3,i} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} * \begin{bmatrix} state\_matrix_{0,i} \\ state\_matrix_{1,i} \\ state\_matrix_{2,i} \\ state\_matrix_{3,i} \end{bmatrix}$$

end for
return $state\_matrix$

2.1.3 Modes of Operation

AES is a block cipher algorithm. The algorithm describes how a single block of the size of its state matrix is to be encrypted. The mode of operation describes how a multiple of blocks are processed and linked using the given block cipher algorithm. The encryption and decryption modes CTR and ECB do not depend on any previous blocks. Therefor several blocks can be encrypted or decrypted at the same time. Encryption of CBC, PCBC, CFB, and OFB modes depend on all previous blocks of ciphertext and are not parallelizable for more than a single block at a time. Internally, every single block can be encrypted in parallel, but the parallelization of multiple blocks is possible only in some modes. In PCBC and OFB modes, decryption is also not parallelizable, while in CFB and CBC modes, data can be decrypted in parallel. In modes where decryption is parallelizable, random read access on the encrypted data is possible since these are not relying on any previous blocks. In contrast, for reading access to data encrypted with non-parallelizable decryption, all of the data up to the point interest have to be decrypted.

Electronic Codebook (ECB)

Electronic Codebook is the most straightforward mode of AES. Each block of ciphertext has one unaltered block of plaintext as the input. The main disadvantage of ECB is the same

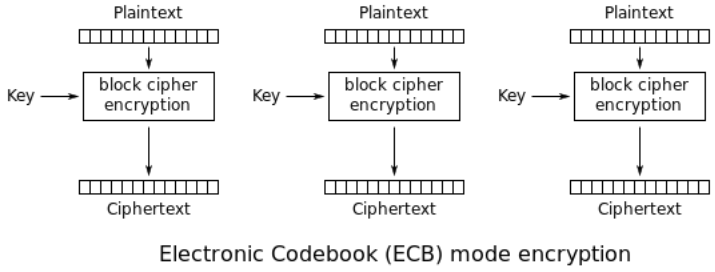

Figure 2.8: EBC Mode [Com07]

16bytes of data is always encrypted to the same ciphertext. In other words, a repeated block in the plaintext is encrypted to repeated blocks in the ciphertext. This is called 'ghosting'. ECB is fully parallelizable, as no block any dependency on any previous blocks.

Cipher Block Chaining (CBC)

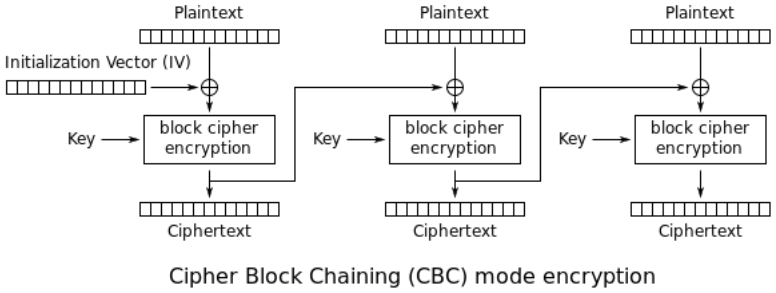

Figure 2.9: CBC Mode [Com07]

In Cipher Block Chaining mode, each plaintext is altered by an excluding-or operation with the ciphertext of the block before. Because of diffusion from factoring in previous blocks, blocks of the same plaintext are not identifiable in the ciphertext, meaning no ghosting occurs. Every block of data fed to the AES block cipher depends on the previous block of ciphertext, and therefor only one block can be processed at a time. The first round is initialized with an Initialization Vector. Decryption, on the other hand, is parallelizable in CBC mode, since the exclusive-or operation with ciphertext is done after the decryption rounds.

Propagating Cipher Block Chaining (PCBC)

In PCBC, each plaintext is altered by an XOR with the plaintext- and the ciphertext of the previous block. Only a single block can be processed at a time since each block depends on the ciphertext of the previous block. The first block is initialized with an Initialization Vector. Unlike CBC, PCBC decryption is not parallelizable, since decryption depends on the plaintext of the previous block.

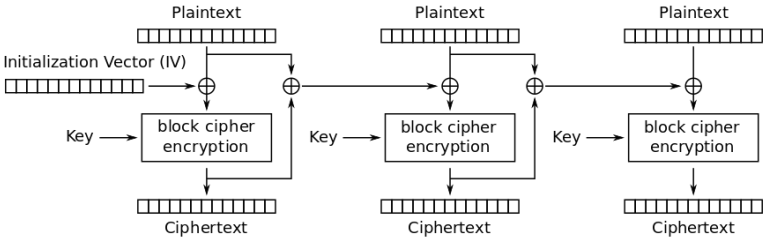

Figure 2.10: PCBC Mode [Com07]

Cipher Feedback (CFB)

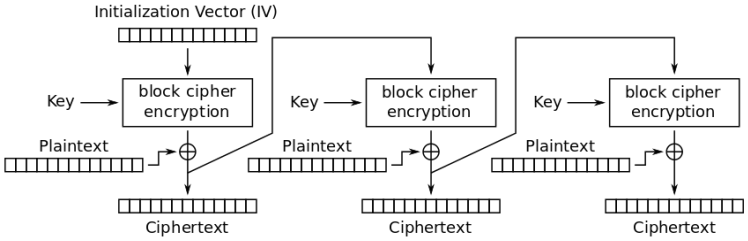

Figure 2.11: CFB Mode [Com07]

In CFB mode, each cipher output is altered by an excluding or operation with the plaintext of its block. The cipher input is ciphertext of the previous block, therefor CFB encryption is not parallelizable. The first block is initialized with an Initialization Vector. CFB decryption uses the AES encryption block cipher. In CFB, decryption is parallelizable since the XOR operation with the ciphertext is not dependent on the plaintext of the previous block.

Output Feedback (OFB)

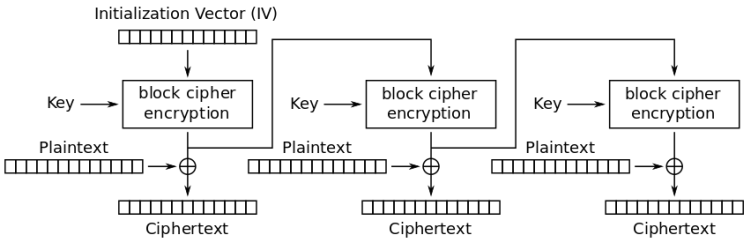

Figure 2.12: OFB Mode [Com07]

The OFB mode encryption feeds the block cipher output to the next block cipher before altering it by an excluding-or operation with the plaintext to form the ciphertext. OFB decryption uses the AES encryption block cipher and is not parallelizable due to the dependency on the block cipher output of the previous block.

Counter (CTR)

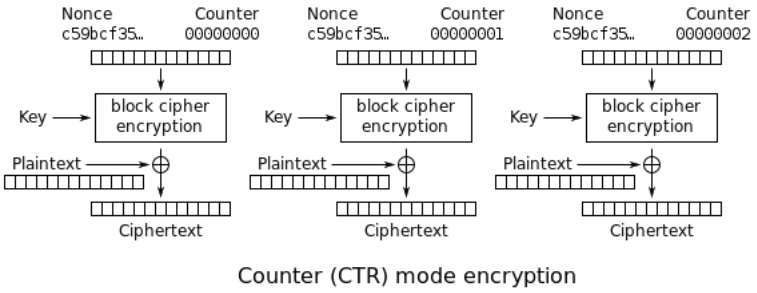

Figure 2.13: CTR Mode [Com07]

In CTR mode, an Initialization Vector is block ciphered and only then altered by an excluding-or operation with the plaintext of the corresponding block to form the ciphertext. For the next block, the initialization vector is incremented by one. Since the initialization vector for every block is predictable when known, CTR encryption is parallelizable. CTR decryption uses the AES encryption block cipher. For decryption, the initialization vector is fed to the encryption block cipher and an excluding-or operation performed on the output to get the plaintext. Counter mode is effectively a stream cipher operation mode in that it doesn't decrypt the plaintext using the block cipher, but rather encrypts the counter and exclusive-ors it with the next block of 16 bytes of plaintext.

Mode	Encryption parallelizable	Decryption parallelizable	random read access
ECB	yes	yes	yes
CBC	no	yes	yes
PCBC	no	no	no
CFB	no	yes	yes
OFB	no	no	no
CTR	yes	yes	yes

Table 2.2: Parallelization capability of AES modes

2.2 Raspberry Pi

The Raspberry Pi is an accessible single-board computer. The heart of the Raspberry Pi is a Broadcom BCM2835, BCM2836 or BCM2837/BCM2837B0 System-on-a-Chip (SoC) that integrates an ARM CPU with either one core in case of the Raspberry Pi 1 and Raspberry Pi Zero or a quad-core in case the Raspberry Pi 2 and 3. Integrated with the Broadcom SoC

of all Raspberry Pis of Series 1 to 3 is a Broadcom VideoCore IV 3D system. Raspberry Pis are often used without a monitor connected. In this case, the 3D system remains unused, and with it, most of the Raspberry Pis compute capability, as shown in figure 2.3.

Processing Unit	Frequency	Cores	parallelism	OPS
ARM1176JZF-S (Pi 1 / Zero)	700-1000MHz	1	1	0.7 - 1G
Cortex-A7/-A53 (Pi 2/3)	900-1400MHz	4	1	3.6 - 5.6G
VideoCore IV 3D (Pi GPU)	250-400MHz	12	4*2 ALUs	24 - 38.4G

Table 2.3: Compute capability the Raspberry Pi

VideoCore IV

The VideoCore IV 3D architecture's main purpose is, of course, processing and displaying of graphics. All of the subsystems of the VideoCore IV 3D are specially designed for processing graphics, and only some of them can be used for general-purpose computing. In figure 7.1, all subsystems of the VideoCore IV 3D are shown. The ones that are not of interest to general-purpose computing are greyed out [Bro13].

Slice

Slices are wrappers for four QPUs. Each Slice shares two Texture and Memory Lookup Units, a Uniforms Cache, an Instruction Cache, and a Special Function Unit that can be accessed by the four QPUs on the Slice. Per Slice, a single program can run at any given time.

Vertex Pipe Memory

The Vertex Pipe Memory (VPM) is a virtual 48kB cache, with each QPU having access to 4kB. Each of the QPUs have their own FIFOs, one for reads and one for writes to through the Vertex Pipe Memory. Each of the read and write FIFOs can store up to two queued vectors. For reading from the VPM, an access mode, either horizontal or vertical, has to be selected and a configuration of either packed or laned data chosen. This configuration must be written to the VPMVCD_RD_SETUP register. For writing through the VPM, a similar configuration has to be written to the VPMVCD_WR_SETUP register. The VPM can be set up to fill the read FIFO of a QPU with up to 16 consecutive reads from the main memory automatically. Setting up a read requires three instructions of latency before actually reading the data from the VPM. A read from the VPM stalls the QPU until data is available.

VPM Access Data written to the VPM can be configured to be read in different ways by the QPUs. Either a horizontal or a vertical configuration can be selected and the chunk of data to read configured by its Y and B values in a horizontal configuration, or by the Y, X, and B/H values in case of the vertical configuration. Two visualizations of the access modes are available in figure 7.3 and 7.4.

Vertex Cache Manager & DMA

Reads from the main memory by the VPM are executed via the Vertex Cache Manager (VCM) and Vertex Cache DMA (VCD). The VCD and VCM place data read from the main memory into the VPMs QPU read FIFO. For general-purpose computing, the only the VCD is utilized. When using direct memory access (DMA), we can only ever fetch a sequence of 16 32-bit words from the main memory. Each word referenced in the elements of the pointer vector is fetched from the main memory or the level 2 cache individually.

VPM DMA Writer

The VPM DMA Writer (VDW) writes data from the QPUs VPM-write-FIFO to the main memory after being set up to do so. The setup works similarly to the setup of the VCD, except for the VDW not being able to store data consecutively.

Level 2 Cache

The Level 2 Cache (L2C) is a shared cache between the ARM CPU and the VideoCore IV. It has a size of 128kB and is accessed by the VideoCore IV via the Texture Memory Lookup Units 2.2, the instruction cache, and the uniforms cache. From the GPU perspective, the L2C is read-only.

Texture and Memory Lookup Unit

Each of the 3 Slices has two Texture and Memory Lookup Units (TMUs), TMU0, and TMU1. TMU0 and TMU1 are swapped from the perspective of QPU2 and QPU3 so that when utilizing TMU0 only, just 2 QPUs actually share TMU0. For each of the four QPUs per Slice, a TMU holds a Vector FIFO of size four. Because of this FIFO, each QPU can request up to four vectors from memory, before reading the first one. The Texture and Memory Lookup Unit fetches data from main memory via a small internal cache per TMU, and the Level 2 Cache shared between all TMUs, so repeatedly reading the same values by the TMU is a lot faster than reading random data from main memory. The TMU is a Lookup Unit, which means it has no writing capability to main memory or any of the caches. Storing data to it is not possible apart from temporary storage in the FIFO by requesting data from the main memory. General-purpose data can be requested by writing a vector of addresses to the register TMUx_S with 'x' being TMU0 or TMU1.

3D Pipeline The Control List Executor, Primitive Tile Buffer, Primitive Setup Engine, and Front End Pipe are of solely use for graphics processing and are of no use for general-purpose computing as the data formats and automatic scheduling and pipeline feeding are too specific for general usage.

The QPUs

A Quad Processing Unit (QPU) is a 16-way SIMD vector processing unit. As the name suggests, it operates on four-element vectors, the so-called quads. The operations are multiplexed over four clock cycles to form 16-way SIMD.

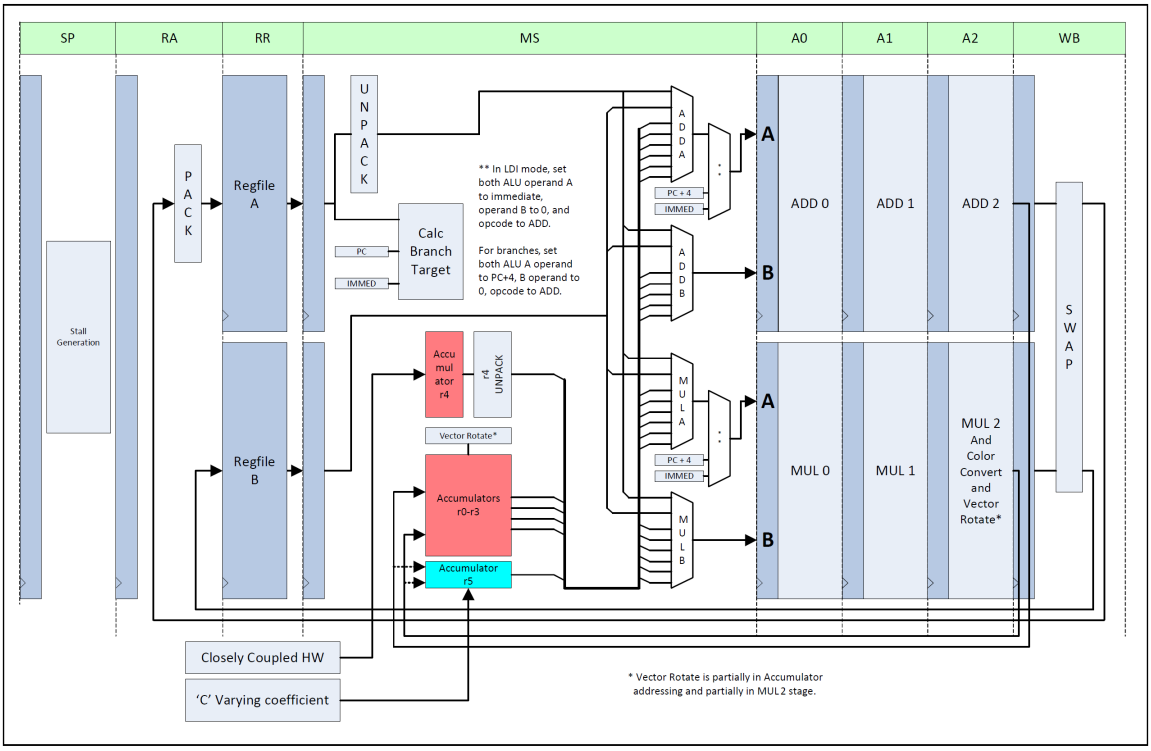

Figure 2.14: QPU Core Pipeline

Processor Registers Each QPU has two local address spaces comprised of 32 general-purpose physical registers, and 32 I/O and Special Function register each. These 32 I/O registers are not actually existing two times, but are actually a single virtual space of registers. The 32 I/O registers consist of:

- 4 general purpose accumulators (r0 - r3) with no latency.
- A read only Accumulator, that provides data from the Special Function Unit and the Texture Lookup Unit to the ALUs.
- Interface registers to the Special Function Unit, the Texture Memory Lookup Unit, the Vertex Pipe Memory, Tile Buffer and other graphics related registers.

Most of the I/O registers are read-only or have different functions mapped to them when comparing reading and writing. Another I/O register worth mentioning is the NOP-register, which is written to in every ALU instruction by one of both ALUs to prevent side effects since both ALUs are operating, even if just a single ALU is effectively processing data. After a write to a physical register of one of the register-files, one instruction of latency must be considered before reading from that register again. For intermediate results, a QPU can, however, write to one of four general-purpose accumulators that have no latency to be considered. The Special Function Unit provides results via Accumulator r4 three instructions after the data to be processed is written to any of the Special Function Unit registers.

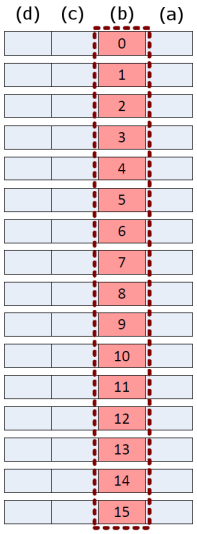

Figure 2.15: A lane of data in a QPU-register

Pack & Unpack Register File A supports unpacking and packing of four vectors of 8-bit lanes or two vectors of 16-bit data into a single 32-bit wide vector register. A lane is a vector of sub-words spread out over all 16 elements of a register, as shown in figure 2.15. The Accumulator r4 also supports packing, but no unpacking of data. Unpacking and packing is encoded into every single instruction. When packing for a write, or unpacking while reading, only the corresponding lane of data is changed, all other lanes remain as they were.

Arithmetic Logic Unit Each QPU has two separate, independent arithmetic logic units (ALU), one multiplication ALU, and one addition ALU. The add-ALU performs Integer and bitwise operations, while the mul-ALU performs floating point multiplications, Integer multiplications, and operations on packed 8bit data. Both ALUs perform operations at the same time, and both ALUs have their own operations encoded in each ALU instruction. While this theoretically doubles the performance of the VideoCore IV, utilizing both ALUs of each QPU requires incredible amounts of assembler optimizations and is only suitable for a limited amount of algorithms, due to both add- and mul-ALUs reading from the same registers. The ALUs either read one value from Register File A and one from Register File B, or they read one from Register File A and load a value called small-immediate. The small-immediate is encoded into the 6-bit field of the Register-address of Register File B. If the add-ALU is configured to write to Register File A, the mul-ALU automatically writes its result to register file B and vice versa. Both ALUs can be configured to write to different register addresses at the same time. A special note to the Integer multiplication in the mul-ALU; the operation masks the lower 24 bits of the input vectors, treating the upper 8 bits as zero.

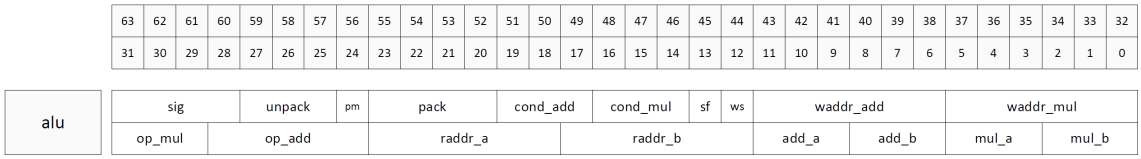

Figure 2.16: ALU Instructions

The mul_a, mul_b, add_a and add_b fields in the encoding describe the source of each operand a and b that can either be one of the two register-files or one of the accumulators. For writes, there is no mux, as all outputs are written to the register space A or B, even when writing to one of the Accumulators. See write address encoding 2.2.

Value	*Meaning*
0	Accumulator r0
1	Accumulator r1
2	Accumulator r2
3	Accumulator r3
4	Accumulator r4[a]
5	Accumulator r5*
6	Use value from register file A
7	Use value from register file B

Figure 2.17: Mux encoding

Both ALUs perform instructions on four vector elements per clock, hence the name Quad Processing Unit. It takes an ALU a total of four clocks to process a complete vector of 16 elements. In each of these four clocks, a quarter of the 16-element is fetched, and the operation started.

2.3 The QPULib

The QPULib is a C++ library for programming the VideoCore IV 3D GPU of the Raspberry Pi in the versions 1 - 3. As of December 2019, it remains in version 0.1.0. The library was first published in 2016 under the MIT license [Nay16].

2.3.1 Software Architecture

The QPULib effectively defines its own language. This Source Code (2.3.1) is stored as an array of statements at the beginning of compiling. This array gets then translated to an intermediate language which is effectively a list of *Instruction struct* that in the final step is encoded to an array of binary instructions (bytecode), that is fed to the instruction cache of the VideoCore IV 3D, or interpreted by the emulator 2.3.1.

Instruction	*opcode*	*Description*
nop	0	No operation
fadd	1	Floating point add
fsub	2	Floating point subtract
fmin	3	Floating point min
fmax	4	Floating point max
fminabs	5	Floating point min of absolute values
fmaxabs	6	Floating point max of absolute values
ftoi	7	Floating point to signed integer
itof	8	Signed integer to floating point
–	9–11	Reserved
add	12	Integer add
sub	13	Integer subtract
shr	14	Integer shift right
asr	15	Integer arithmetic shift right
ror	16	Integer rotate right
shl	17	Integer shift left
min	18	Integer min
max	19	Integer max
and	20	Bitwise AND
or	21	Bitwise OR
xor	22	Bitwise Exclusive OR
not	23	Bitwise NOT
clz	24	Count leading zeros
–	25–29	Reserved
v8adds	30	Add with saturation per 8-bit element
v8subs	31	Subtract with saturation per 8-bit element

Figure 2.18: add-ALU Operations

Data Types

Currently, the QPULib only implements two datatypes, Integers of 32bit length and Floats of 32-bit length. Both these types are vector types, and all their corresponding operations are always performed on all of the 16 vector elements. For the operations implemented in the VideoCore IV 3D see 2.4. Not all of the operations are accessible via the QPULib, though, as shown in figure 2.4. Additional operations, e.g., special functions from the SFU (2.2), are not implemented at the moment.

Missing Datatypes In the previous section, we learned that the QPUs Register File A could handle packed data (see 2.2), and the mul-ALU can handle several operations on 8-bit datatypes 2.19. Especially an unsigned Integer of length 8-bit (Uint8) would make sense to be implemented for further flexibility for the user.

This could be either implemented as a packed data-type where four lanes of unsigned

Instruction	*opcode*	*Description*
nop	0	No operation
fmul	1	Floating point multiply
mul24	2	24 bit multiply
V8muld	3	Multiply two vectors of 4 8-bit values in the range [1.0, 0]
V8min	4	Return minimum value per 8-bit element
V8max	5	Return maximum value per 8-bit element
V8adds	6	Add with saturation per 8-bit element
V8subs	7	Subtract with saturation per 8-bit element

Figure 2.19: mul-ALU Operations

8-bit Integers stay packed and are operated on all lanes at once using the v8-instructions, or it could be implemented using the pack and unpack functionality of Register File A. The latter would then be interpreted as 32-bit Integers by the ALUs after unpacking to the ALUs and written back as 8-bit unsigned Integers using the pack-functionality. We have a more in-depth look into the issue in chapter 3.

		Operation Code	
Description	operator	Int	Float
Addition	+	add	fadd
Subtraction	−	sub	fsub
Multiplication	*	mul24	fmul
Minimum	min()	min	fmin
Maximum	max()	max	fmax
shift right	ushr()	shr	-
arithmetic shift right	>>	asr	-
rotate right	ror()	ror	-
shift left	<<	shl	-
bitwise AND	&	and	-
bitwise OR	|	or	-
bitwise XOR	^	xor	-
bitwise NOT	~	not	-

Table 2.4: Implemented Operators and their corresponding hardware-operation

Source Code

The Source code written by the user is C++ code that is easily distinguishable from code that is not run on the QPUs. Its look is defined by some macros that distinguish classic C++ 'if', 'else', and other conditional keywords from the conditionals executed on the QPUs. The QPULib's conditionals are shown in listing 2.3.

```
1 #define If(c)     If_(c); {
2 #define Else      } Else_(); {
3 #define End       } End_();
```

```
#define While(c) While_(c); {
#define Where(b) Where_(b); {
#define For(init, cond, inc) \
  { init;                    \
    For_(cond);              \
      inc;                   \
    ForBody_();
```

Listing 2.3: Conditional Macros

```
Int my_method(Int a, Int b) {
    Int c;
    Where(a > b)
        c = a - b;
    Else
        c = b - a;
    End
    return c;
}
```

Listing 2.4: Simple Source Code Example

The Source Code can combine code that runs on the QPUs and C++ code that runs during the runtime compilation for the QPUs. This can, for example, be used for loop unrolling, or dynamic allocation of arrays of vectors. Here in the second line, the array is taken care of by the C++ compiler, not the QPULib's compiler. The same applies to the 'for'-loops, for loop-unrolling, but not the 'For'-loops.

```
Int my_other_method(Int x) {
    Int v[5]; // array of 5 vectors, combined QPU and C++ code

    for(int i = 0; i < 5; i++) { // non-QPU code
        For(Int j = 0, j < x, j++) // QPU code
            v[i] = v[i] * 2; // combined QPU and C++ code
        End // QPU code
    } // non-QPU code

    for(int i = 1; i < 5; i++) { // non-QPU code
        v[0] = v[0] + v[i]; // combined QPU and C++ code
    } // non-QPU code

    return v[0]; // combined QPU and C++ code
}
```

Listing 2.5: Loop unrolling Example

Target Code

Target code is an intermediate code that can either be run by the emulator directly or compiled to bytecode by the QPULib's compiler 2.3.1 to run on the QPUs.

Emulator

An emulator built into the QPULib can be used for development and debug purposes. If the Emulator is set active by compiling without the $QPU = 1$ flag, the source code is not compiled into bytecode, but compilation stops at the intermediate Target Code, which is then

interpreted at runtime. For easy debugging and algorithm development in Emulator-mode, the QPULib offers print functions that can be written in the source code. The compiler ignores these if the $QPU = 1$ flag is set and the QPUs are invoked.

The Compiler The compiler compiles our Source Code at runtime to bytecode to be executed by the QPUs. While the compiler runs at the runtime of a C++ program, it still is technically a compiler and not an interpreter, because it finishes compiling to bytecode before running the program on the QPUs. The following list explains the major steps that are performed to form the final Target Code:

1. Straightforward translation to intermediate Target Code.
2. Insert code to terminate the program at the end.
3. Set up read- and write-instructions.
4. Compute a control flow graph and create jump labels.
5. Allocate a register for each variable.
6. Insert move-to-accumulator instructions where hardware constraints don't allow instructions directly with a read from a register.
7. Insert NOP (no operation) instructions where necessary, to satisfy hardware constraints of the VideoCore IV 3D architecture.
8. Replace branch-label with branch-target instructions.

Afterwards, the Target Code is encoded into an bytecode array of 32-bit words, that is stored an fetched from the instruction cache by the QPUs.

2.4 Related Work

Since AES is used that much, there are several implementations of AES on GPUs. P. Pauls implemented AES on the Raspberry Pi's VideoCore IV already using Assembly but could not find a performance benefit over the OpenSSL implementation on the very same platform, due to problems with accessing the VPM [Pau17]. In general, it is possible to speed up the parallelizable AES block cipher modes on GPUs. Some publications have already proven this. The exact speedup is, however, heavily dependent on the compared CPU and GPU and of course, the quality of the implementation. The fastest implementation today on a single GPU was written at the American University in Cairo, where they managed to achieve a throughput of up to 279.86 Gigabits per second (Gbps) in AES128 encryption on an Nvidia GTX 1080 using CUDA [AFD17]. Another example is the implementation of AES on an Nvidia GeForce 940MX, which is a mobile chip for use inside of laptops, again using the Compute Unified Device Architecture (CUDA) [WC19]. Compared to the Intel Core i5-7200U in the same machine, they saw a speedup of close to factor 30. A significant drawback of all AES implementations on highly parallelized machines is that only the CTR and ECB modes can be parallelized. There is no solution for running encryption of other

modes efficiently on GPUs.

The Raspberry Pi is a very popular device. It comes with no surprise that several libraries, especially for general-purpose computing on the Raspberry Pi's GPU, have been developed. There is an OpenCL implementation for the Raspberry Pi's VideoCore IV 3D GPU available [RP17]. It is called VC4CL and implements the OpenCL 1.2 standard. The project is split into a runtime-library, a compiler, and a standard-library, all specifically developed for the Raspberry Pi's VideoCore IV 3D GPU. Another framework for general-purpose GPU programming for the Raspberry Pi is the 'PyVideoCore' library [nin15]. It is written in Python and is a lower-level library, requiring more in-depth knowledge of the hardware compared to the QPULib.

2.5 Summary

This chapter gave an introduction to the Advanced Encryption Standard. We got an idea of how it works and learned that there are different modes. Some of these modes are not parallelizable for multiple blocks. These modes would be much slower on highly parallelized hardware and are therefore not suitable for implementation on GPUs. We also learned about the Raspberry Pi's VideoCore IV 3D GPU. It integrates 12 cores, the so-called QPUs. Each of them is a 16-way SIMD vector-processor. The hardware is complex, and most subsystems are not suitable for general-purpose use. As a high-level interface to it, we use the QPULib. Although it is not complete yet and lacks to address some crucial hardware features, the QPULib is usable and provides a straightforward entry into vector processing on the VideoCore IV 3D GPU. The development of several further features was started but has been paused since July of 2018.

3 Additions to the QPULib

The QPULib does not cover all hardware-implemented functionality of the VideoCore IV 3D. For general-purpose computing, broader hardware support could be useful. The following list is not complete but rather lists obvious, missing functionality that is implemented in hardware on the VideoCore IV 3D.

- The Special Function Unit (SFU) that can compute the mathematical functions; square root, reciprocal of a square root, logarithm and exponentiation.
- Features for pre-programming of the Vertex Pipe Memory (VPM) could automatically fill the VPM with data from the main memory.
- Operation codes for other datatypes than 32-bit Integers and 32-bit Floats, namely 8-bit unsigned Integers, 16-bit Integers and 16-bit Floats.
- Unpack and pack functionality that could make way for fast and easy access to sub-words of the 32-bit Vectors.

However, we focus on the features that are most useful for the implementation of AES. Since the AES does not rely on the functions provided by the Special Function Unit, we do not need to have the SFU-functions implemented. A pre-programmed setup of the VPM could be useful for filling the VPM and providing data for the next AES blocks to the QPUs automatically from the main memory. Unpacking and packing can speed up some steps in AES. However, because AES's word-size is an 8-bit wide, we chose the implementation of software support for 8-bit unsigned Integers.

3.1 Char

The word size of the AES-algorithm is a byte of 8 bit. All of the available datatypes in the QPULib are 32 bit wide. If we assign a single byte to each vector element of the QPU, the upper 24 bits are always zero. When loading data from the main memory to the QPUs, those 24 bit and therefore three-quarters of the available bandwidth are wasted with data we know is zero.

The VideoCore IV 3D architecture supports unpacking of 32-bit fields from Register File A for the ALUs and packing them back into a 32-bit field when writing back to Register File A. The packed data can be either interpreted as unsigned 8-bit Integers, signed 16-bit Integers or 16-bit Floats. Since there is no use for signed 16-bit values in AES, the focus lies on unsigned 8-bit Integers. The pack- and unpack-modes are encoded in reserved bit-fields in the instructions. Bits 59 to 57 select the unpack mode, which is only available in case of an ALU operation. Bits 55 to 52 select the pack-mode, and bit 56 is the pack/unpack-select bit. In case of using the unpack or pack functionality of Register File A, the pack/unpack

select bit has to be set to zero.

The unpack and pack functionality is useful in all cases where we are operating on packed 8-bit data, as shown in 2.2. Otherwise, masking out the unused bytes does require a masking-, and a shift- operation when reading or writing lanes to a register.

```
1    Int bytelane_b = (0x0000ff00 & data) >> 8;
```

Listing 3.1: Unpacking packed data without unpack functionality

Packing data back to a register is even more complicated and requires a total of 4 operations without using the pack functionality that Register File A provides.

```
1    Int my_register = x;
2    my_register = ((0x000000ff & data) << 8) | (my_register & 0xffff00ff);
```

Listing 3.2: Packing data without pack functionality

3.1.1 4x8-bit Vector Implementation

Interpreting the 32-bit words from the vector registers of the QPU as unsigned 8-bit Integers and performing the same operation on all of them is relatively straight forward. The QPUs implement the operations called v8adds, v8subs, and v8muld in the mul-ALU. These operations perform additions, subtractions, and multiplications on each of the four 8bit sub words per register element. v8min and v8max, which are also hardware-implemented, return a vector comprised of the minimum or maximum respectively per vector element sub-word.

Vector element	sub-word				
	A	B	C	D	total
0	byte	byte	byte	byte	32 bit
1	byte	byte	byte	byte	32 bit
2	byte	byte	byte	byte	32 bit
3	byte	byte	byte	byte	32 bit
4	byte	byte	byte	byte	32 bit
5	byte	byte	byte	byte	32 bit
6	byte	byte	byte	byte	32 bit
7	byte	byte	byte	byte	32 bit
8	byte	byte	byte	byte	32 bit
9	byte	byte	byte	byte	32 bit
10	byte	byte	byte	byte	32 bit
11	byte	byte	byte	byte	32 bit
12	byte	byte	byte	byte	32 bit
13	byte	byte	byte	byte	32 bit
14	byte	byte	byte	byte	32 bit
15	byte	byte	byte	byte	32 bit

Table 3.1: Interpretation of a vector for v8 operations

The v8-operations have not been made use of in the original implementation of the library. Either a new datatype or new operators have to be implemented for the source code syntax,

and the translation to target code and instructions have to be fitted. To avoid confusion with the 'Int'-datatype, which was already implemented, a new 'Char'-datatype is introduced. The main difference to the 'Int'-implementation are the different ALU-operations encoded for the same operator; these are shown in table 3.2. A new C++ 'struct', similar to that of the 'Int'-datatype was implemented, the only difference being the different operators. Next, some new op-codes are introduced, next to the ones already existing in the QPULib, and the corresponding switch-cases updated. As a final step, the Emulator had to be extended by the newly introduced operation codes. This was straightforward because the actual emulation of the operations is just packaged in a switch-case block.

Description	Operator	Operation Code		
		Int	Float	Char (4*8-bit)
Addition	+	add	fadd	v8adds
Subtraction	−	sub	fsub	v8subs
Multiplication	∗	mul24	fmul	v8muld
Minimum	$min()$	min	fmin	v8min
Maximum	$max()$	max	fmax	v8max
shift right	$ushr()$	shr	-	-
arithmetic shift right	>>	asr	-	-
rotate right	$ror()$	ror	-	-
shift left	<<	shl	-	-
bitwise AND	&	and	-	and
bitwise OR	|	or	-	or
bitwise XOR	^	xor	-	xor
bitwise NOT	~	not	-	not

Table 3.2: Operators for each Datatype

3.1.2 Concept Packing and Unpacking 8-bit Vectors

The idea of this datatype is using the unpack functionality of Regfile A or Accumulator R4 to load a vector consisting of a single lane of 8-bit sub-words into the ALUs instead of the whole 32-bit words from each element. For this datatype, two new methods would have to be deeply implemented into the QPULib. The following lines of code show exemplary usage of them. In the first example, the second-least-significant lane from data is unpacked and stored in the new variable 'bytelane_b'. Unpacking of unsigned 8-bit Integers, without a following 'pack', should always write to an Integer for data integrity.

```
1    Char data = x;
2    Int bytelane_b = unpack(data, SUBWORD_B);
```

Listing 3.3: Unpacking data using unpack functionality

In the following second example, an Integer, where only the lower 8-bit are filled, is packed to a new variable. Packing should always write to the packed datatype 'Char', introduced in 3.1.1.

```
1    Int data = x;
2    Char my_register;
```

```
3     pack(my_register, SUBWORD_B, data);
```

Listing 3.4: Packing data using pack functionality

When a write to a register occurs, the data has to be packed back into a given sub-word again. Packing also is only possible to Register File A. Register File B does not support the packing or unpacking of vectors, so special attention has to be directed to optimizing register allocation for each variable that is unpacked from or packed. If an operand is stored in Register File B, a move to Regfile A has to be performed first, and both operands have to be loaded from Regfile A in the case of an operation on two lanes of sub-words that are directly loaded. In this case, the accumulator must be used, and two consecutive loads from Regfile A have to be performed. The first one is unpacking to the ALUs without data alteration and storing it to one of the accumulators. The following operation must unpack the second operand from Regfile A and the load the first operand from the accumulator we previously wrote to.

The following line of code would be translated exemplarily to assembly similar to the pseudocode in algorithm 5. It would unpack two lanes of unsigned 8-bit Integers, perform a multiplication, and write them back to a new variable called 'product'. Note that in this example, no packing functionality is used.

```
1 Int product = unpack(x, SUBWORD_C) * unpack(y, SUBWORD_C);
```

Listing 3.5: Multiplication of two sub-words

Algorithm 5 Unpacking of two registers for an ALU instruction

Input: *Vector x*
Vector y
Output: *product of x and y*
move x to a register of Regfile A
unpack subword C of y and store it in accumulator 1
unpack subword C of x and multiply with y, store in variable product
return *product*

When several consecutive operations are done on the same data, the accumulator can be used and filled with unpacked data ready for the next operation. Writing and reading from accumulators is faster than using the physical register files since a write to a register of Register File A or B can not be followed by a read from the same register in the next instruction [Bro13, 37].

Implementation Complexity The QPULib is a quite complex construct, so implementing the unpack- and pack-functionality is not entirely straightforward. Unpack- and pack-flags would have to be carried all the way from Source Code to Target Code, and to the encoded instructions. Register allocation, and specially optimized register allocation, to actually see a performance gain, would require alterations to deeply nested functions. Even the Emulator would need some adaptations.

3.2 Summary

The VideoCore IV 3D GPU has hardware-support for operations for 8-bit datatypes and the packing and unpacking of 8-bit lanes. Unfortunately, the QPULib didn't provide access to those features yet. It is still unfinished at this point, and some branches for the development of further features exist in the GitHub-repository [Nay16]. We implemented a new datatype called 'Char', that operates on 64 unsigned 8-bit Integers at per instruction. The concept for unpacking and packing of lanes was evaluated but exceeded the scope of this work.

4 AES Implementation

In this chapter we are focusing on the actual implementation of the Advanced Encryption Standard on the VideoCore IV 3D using the QPULib. At first, we focus on the implementation without the additions to the QPULib described in chapter 3. Later, in section 4.3, an optimized version of the algorithm, operating on four AES state matrices at once, is implemented.

4.1 Parallelization Concept

There are two ways to parellelize AES. One way is to operate on all bytes of a state matrix at once. The other is to operate several state matrices at once. The VideoCore IV 3D can do both at the same time. Each block can be put in a single 16-element vector and operated on with a single instruction. This is called Single-Instruction-Multiple-Data (SIMD). Since the VideoCoreIV has 12 QPUs that can execute instructions simultaneously, 12 vectors can be operated on at any given moment. In 2.2 we showed which AES-modes are parallelizable in encryption and decryption. This graphic is to be interpreted as following: In every mode every state matrix can be operated on as a 16-element vector with SIMD, but parellelising several blocks at once is impossible when each block depends on the previous block. In general the operations on each state matrix are the same, but not every element experiences the same operations in a scalar AES algorithm. For example in Shift Rows the first row is not shifted at all, while all others are, but all by different values. Another example is the Galois Multiplication used in Mix Columns, where each byte takes different branches depending on its own value and the value of the other operand. Where necessary the QPUs can write the result of an ALU operation back to a subset of the vectors elements using the conditional flags. Where a low number of elements are configured to be written to using the conditional flags, a lot of the compute capability of the QPU is wasted.

4.2 First AES Implementation

Here we want to have a look of what an implementation of a naive approach to implementing AES on the VideoCore IV using the QPULib looks like. We are going to use the QPULib as found on github at the time of the beginning of this thesis. For this we just take 16 Integers from main memory. These were filled with one byte of the state matrix beforehand each. This byte sits at the LSB side of the Integer.

4.2.1 Key Expansion

In AES the same key is used to encrypt all data blocks and thus only ever has to be expanded once during initialisation by the key schedule. The expanded key can be stored in main memory and accessed by either a direct memory load or through the Texture and

Vector Element index	32-bit Integer			
	MS byte	byte	byte	LS byte
0	0	0	0	state[0]
1	0	0	0	state[1]
2	0	0	0	state[2]
3	0	0	0	state[3]
4	0	0	0	state[4]
5	0	0	0	state[5]
6	0	0	0	state[6]
7	0	0	0	state[7]
8	0	0	0	state[8]
9	0	0	0	state[9]
10	0	0	0	state[10]
11	0	0	0	state[11]
12	0	0	0	state[12]
13	0	0	0	state[13]
14	0	0	0	state[14]
15	0	0	0	state[15]

Table 4.1: A register filled with a single dimension representation of the state matrix

Memory Lookup Unit. Or it can be stored in the QPUs registers and accessed extremely fast from there. Since the key can be stored and is not altered, the key schedule is not critical for performance and can be done on the CPU for simplicity.

4.2.2 Add Round Key

Each round a different 16-element part of the expanded key is used to bitwise exlusive or the each of the 16 state elements respectively. When expanding the key, we made sure the key is in the same format as the state matrix, using only the least significant 8 bit of a 32-bit Integer. Using SIMD instructions the exclusive or is a single operation, preceded by fetching the correct part of the key from main memory.

```
1 Int add_round_key(Int state, Ptr<Int> key_pointer) {
2   Int key = *(key_pointer + (round << 4)); // round * 16
3   state = state ^ key;
4   return state;
5 }
```

Listing 4.1: shift rows code

4.2.3 Sub Bytes

Values that are read repeatedly through the TMU will stay in the Level 2 Cache of the VideoCore IV as long as the value is not altered. The level 2 cache can not be written to, so values that have changed will have to be read from the much slower main memory again. In each round 16 values are read from the S-Box, one for each byte. In the case of a 128-bit key this leads to 176 bytes read from the S-Box for each AES-block processed, meaning there are 11 times more memory accesses from substitutioning bytes than there are reading blocks of plaintext. With greater key lengths come more memory accesses per AES-block. Luckily the S-Box is a constant and will be in the Level 2 Cache completely after just a few processed AES-blocks. Again, the SBox is formatted to only fill the least significant 8 bit of 32-bit Integers. We are always fetching 32-bit words from memory with each access. Since each byte needs to be substituted by the value at its own values position in the SBoxs one-dimensional representation we are using, we can simply add a each value from the state matrix to a vector of pointers all pointing to the 0-element of the SBox. This newly formed vector now points to the addresses where the corresponding substitution value lies. A new vector is then gathered by the TMU and returned into the TMUs FIFO for the requesting QPU.

```
/**
 * @param state         the state matrix one-dimensional in a vector
 * @param s_box_pointer uniform pointer to s_box[0]
 */
Int gpu_sub_bytes(Int state, Ptr<Int> s_box_pointer) {
    gather(s_box_pointer + state);
  receive(state);
  return state;
}
```

Listing 4.2: Sub Bytes implementation

4.2.4 Shift Rows

In Shift Rows the last 3 rows of a state matrix are to be shifted to the left by one, two and three bytes respectively as described in 2.1.2. Since our state matrix is represented one dimensional in the vector and we got three different operands, these need to be represented in SIMD instructions as well. The elements that are in the correct position after each of the different 3 rotations performed, need to written to be set in place by writing to a set of state vector elements. This is done by setting the conditional flags set using the *Where* keyword.

```
Int gpu_shift_rows(Int state) {
  Int shift = ((index() * 5) & 0x0F) - index(); // A helping vector with
    generic values indicate where a value needs to be set after rotating the
    vector.
  Int old_state = state; // The original state before shifting

  Where((shift == -12) || (shift == 4))
    state = rotate(old_state, 12);
  End
  Where((shift & 0x0F) == 8)
    state = rotate(old_state, 8);
  End
  Where((shift == -4) || (shift == 12))
    state = rotate(old_state, 4);
  End

  return state;
}
```

Listing 4.3: Shift Rows implementation

4.2.5 Mix Columns

In Mix Columns, each column of the state matrix is to be multiplied by a matrix in the Galois Field $GF(2^8)$, as explained in 2.1.2. To do the matrix multiplication, we first need to form four vectors where we then can efficiently multiply with a state matrix on all 16 elements at once. For that also the matrix has to be in a corresponding form. For optimization purposes, the MDS matrix is written to a register once and then rotated 3 times to match the corresponding column of the state each step. The four products of the multiplications are then added in $GF(2^8)$ with exclusive or operations.

```
Int mix_columns(Int state, Int mix_column_vector) {
  Int v0 = state;
  Int v1 = rotate(state, 1);
  Int v2 = rotate(state, 2);
  Int v3 = v1;
  Int rot14 = rotate(state, 14);
  Int rot15 = rotate(state, 15);
  Int x = index() & 0x03;

  Where(x == 0x00)
    v1 = rot15;
    v2 = rot14;
    v3 = rotate(state, 13);
  End
  Where(x == 0x01)
    v0 = v1;
    v1 = state;
    v2 = rot15;
    v3 = rot14;
  End
  Where(x == 0x02)
    v0 = v2;
    v2 = state;
    v3 = rot15;
  End
```

```
  Where(x == 0x03)
    v3 = v1;
    v0 = rotate(state, 3);
    v1 = v2;
    v2 = v3; // ex v1
    v3 = state;
  End

  v0 = galois_mul(mix_column_vector, v0);
  v1 = galois_mul(rotate(mix_column_vector, 1), v1);
  v2 = galois_mul(rotate(mix_column_vector, 2), v2);
  v3 = galois_mul(rotate(mix_column_vector, 3), v3);

  return (v0 ^ v1 ^ v2 ^ v3);
}
```

Listing 4.4: Mix Columns implementation

```
uint8_t galois_mul(uint8_t a, uint8_t b) {
  uint8_t p = 0; // p as the product of the multiplication

  while (b ) {
    if (b & 1) { // If b odd , then add corresponding a to p
      p = p ^ a; // In GF (2^8) , addition is XOR
    }
    if(a & 0x80 ) { // If a >= 128 it will overflow when shifted left , so
    reduce
      a = (a << 1) ^ 0x11b ; // XOR with primitive polynomial x^8 + x^4 + x^3
    + x + 1
    } else {
      a <<= 1; // Multiply a by 2
    }
    b >>= 1; // Divide b by 2
  }
  return p;
}
```

Listing 4.5: Galois Multiplication on CPU

The Galois-multiplication shifts the parameter b right by one bit, dividing it by 2, eight times, but the result gets updated only if b is greater than zero. Since the Galois-multiplication is commutative, the parameters can be swapped with no effect on the result. In the AES algorithm, the the value of the second parameter in the Galois-multiplication is never bigger than 3 which is $0b11$ binary and uses only two bits, resulting in maximum two right shifts by one bit before becoming zero, indicating the multiplication is complete. The algorithm can be shortened to two right shifts of b, and two result updates dependent on parameter a. The instructions and computing time can be cut down to less than 1/6 of the original algorithm. See fig. 7.3

```
Int galois_mul_optimized(Int b, Int a) {
  Int p = 0;

  Where((b & 0x01) != 0) // if b odd then add corresponding a to r. When b is
    uniform, no 'Where' needed.
    p = p ^ a; // In GF(2 exp 8), addition is XOR
```

```
6   End
7
8   // No 'Where(b > 0)' necessary, since when arguments are 2 or 3, b is 0
      after the same number of iterations everywhere, namely after 2 iterations.
9   Int oldA = a;
10  a = a << 1; // multiply by 2
11
12  Where((oldA & 0x80) != 0) // if a >= 128 it will overflow when shifted left,
      so reduce
13    a = (oldA << 1) ^ 0x11b; // XOR with primitive polynomial x^8 + x^4 + x^3
     + x + 1
14  End
15
16  b = b >> 1; // divide by 2
17
18  Where((b & 0x01) != 0)
19    p = p ^ a;
20  End
21
22  return p;
23 }
```

Listing 4.6: For values 1, 2, and 3 optimized Galois Multiplication on QPU

4.2.6 Complete Algorithm

```
1  void gpu_aes128_single_block(Ptr<Int> state_ptr, Ptr<Int> keyPtr, Ptr<Int>
     sub_box_ptr, Ptr<Int> mix_column_ptr) {
2
3     /* Caution state_ptr is uniform pointer to first vector element
4    * This is on purpose to avoid subtracting index().
5    */
6     Int state = *state_ptr;
7     Int mix_column_vec = *mix_column_ptr; // a helping vector comprised of the
       MDS-matrix used in AES
8
9     state = gpu_add_round_key(state, *key_ptr);
10
11    For(Int round = 1, round < 10, round = round + 1) //
12        state = gpu_sbox(sub_box_ptr, state);
13        state = gpu_shift_rows(state);
14        state = gpu_mix_columns(state, mix_column_vec);
15        state = gpu_add_round_key(state, *(key_ptr + (round << 4)));
16    End
17
18    state = gpu_sbox(sub_box_ptr, state);
19    state = gpu_shift_rows(state);
20    state = gpu_add_round_key(state, *(key_ptr + 160));
21
22    store(state, state_ptr + index()); // store block to main memory
23 }
```

Listing 4.7: AES128 GPU algorithm

AES-Block	Vector Element	32-bit Integer sub-word LS a	b	c	MS d
0	0	$state_{0,3}$	$state_{0,2}$	$state_{0,1}$	$state_{0,0}$
	1	$state_{0,7}$	$state_{0,6}$	$state_{0,5}$	$state_{0,4}$
	2	$state_{0,11}$	$state_{0,10}$	$state_{0,9}$	$state_{0,8}$
	3	$state_{0,15}$	$state_{0,14}$	$state_{0,13}$	$state_{0,12}$
1	4	$state_{1,3}$	$state_{1,2}$	$state_{1,1}$	$state_{1,0}$
	5	$state_{1,7}$	$state_{1,6}$	$state_{1,5}$	$state_{1,4}$
	6	$state_{1,11}$	$state_{1,10}$	$state_{1,9}$	$state_{1,8}$
	7	$state_{1,15}$	$state_{1,14}$	$state_{1,13}$	$state_{1,12}$
2	8	$state_{2,3}$	$state_{2,2}$	$state_{2,1}$	$state_{2,0}$
	9	$state_{2,7}$	$state_{2,6}$	$state_{2,5}$	$state_{2,4}$
	10	$state_{2,11}$	$state_{2,10}$	$state_{2,9}$	$state_{2,8}$
	11	$state_{2,15}$	$state_{2,14}$	$state_{2,13}$	$state_{2,12}$
3	12	$state_{3,3}$	$state_{3,2}$	$state_{3,1}$	$state_{3,0}$
	13	$state_{3,7}$	$state_{3,6}$	$state_{3,5}$	$state_{3,4}$
	14	$state_{3,11}$	$state_{3,10}$	$state_{3,9}$	$state_{3,8}$
	15	$state_{3,15}$	$state_{3,14}$	$state_{3,13}$	$state_{3,12}$

Table 4.2: A register filled with 4 AES Blocks interpreted vertically

4.2.7 Summary

We have managed to implement the algorithm using SIMD instructions. Up to this point, the upper 24 bits of each vector element has been zero, and only the lower 8 bits are filled with data. Also the Byte order of the state in our registers has caused a hard to read and slow implementation, making a lot of slow rotations and setting of conditional flags necessary. These are all operations, where we don't actually manipulate the bytes, and are, therefore, seen as overhead.

4.3 AES Implementation Using Packed Data

With the vector interpreted as a horizontal element with the bytes of each vector element interpreted as being vertically aligned, we can interpret the state matrix as being in the byte order intended by the Rijndael algorithm 2.1. A horizontal rotation is now a vector rotation and a vertical rotation is an Integer rotation. Using the QPULib, we can rotate a vector

```
1 Int vector;
2 rotate(vector, 1); // horizontal rotation right by one element
```

Listing 4.8: Horizontal rotation

```
1 Int vector;
```

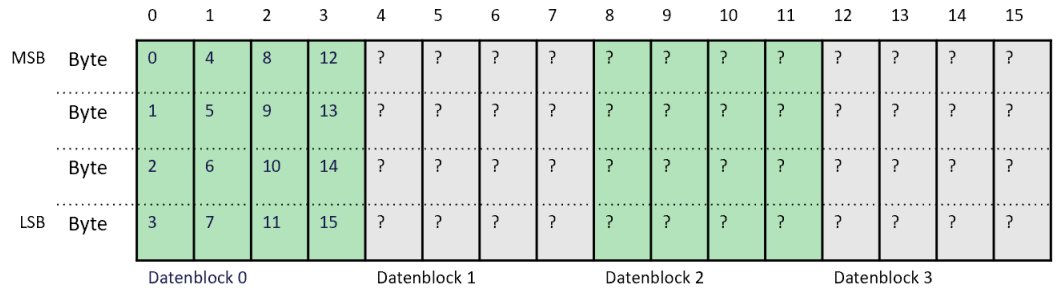

Figure 4.1: Single AES block interpreted horizontally

```
ror(vector, 8); // vertical rotation down by one Byte
```

Listing 4.9: Vertical rotation

The horizontal rotation comes in very handy with Mix Columns,

4.3.1 The Key

We can reuse our previous implementation of the Key Expansion, see 4.2.1. Since our state matrix is now aligned differently, the key also has to be aligned correctly for the Add Round Key step. Instead of expanding the key into an Integer array where the upper 24 bits of each 32-bit wide Integer is always zero, we are now expanding the key into an array of 8-bit wide unsigned chars. The first 16 Byte of the key is now in the same byte order and alignment as the state matrix. For each round, the same key has to be added to all four state matrices in our vector. To achieve this, we can use the Texture Memory Lookup Unit and replicate the key lying in four consecutive 32-bit wide fields four times into a single vector.

```
Int add_round_key(Int state, Int round, Ptr<Int> key_pointer) {
    Int key;
    gather(key_pointer + (round << 2) + (index() & 0x03)); // replicate key of
     round four times
    receive(key);
    return state ^ key;
}
```

Listing 4.10: Load key for a defined round

4.3.2 Sub Bytes

We filled the S-Box the same way as in our previous, naive implementation. The upper 24 bits are always zero. This leaves the 256-Byte S-Box with a footprint of 1024 Bytes. This is, however, necessary for gathering the data efficiently. We can now unpack each of the four 8-bit lanes, and add the values to a uniform base pointer that points to the zero-element of the S-Box. This leaves us with vectors of address pointers to the correct positions of each substitution Byte. Now we can call gather with these vectors and receive the substituted bytes into a temporary variable, from which we can pack the lanes back into the state vector using masks.

```
Int subBytes(Int state, Ptr<Int> sBox) {
    gather(sBox + (state & 0xff)); // lane 0
    gather(sBox + shr((state & 0xff00), 8)); // lane 1
    gather(sBox + shr((state & 0xff0000), 16)); // lane 2
    gather(sBox + shr((state & 0xff000000), 24)); // lane 3 (msb)
    // TMU FIFO full
    Int lane; // temporary variable for receiving without data loss
    receive(state); // lane 0 (lsb) set, upper 24-bits are zero at this point

    receive(lane); // receive lane 1
    state = state | (lane << 8); // lane 1 set

    receive(lane); // receive lane 2
    state = state | (lane << 16); // lane 2 set

    receive(lane); // receive lane 3
    // TMU FIFO empty
    return (state | (lane << 24)); // lane 3 (msb) set
}
```

Listing 4.11: Sub Bytes in lanes

Optimization potential The total number of required instructions amounts to 16 for the gathers, including unpacking of the state matrix. Another 16 instructions of receiving and packing the data back into the state vector are needed. In between, especially if the S-Box is not in cache completely yet, latency has to be considered to collect the data placed into accumulator r4 by the TMU upon receiving from the TMUs FIFO. A solution for this would be to utilize the packing and unpacking functionality of Register File A. For the gathers for the lanes, a total of only four instructions, one per gather, would be necessary. The writes to the state vector could see a theoretical decrease of instructions to a total of eight. For each lane, one to receive the substituted Bytes to accumulator r4, and another instruction to pack them into the state vector.

4.3.3 Shift Rows

When interpreting a QPUs vector as being horizontal, the rotation of each row is just a vector rotation. With the rotation, the state matrices are flowing over to their neighboring state matrices, as shown in figure 4.2.

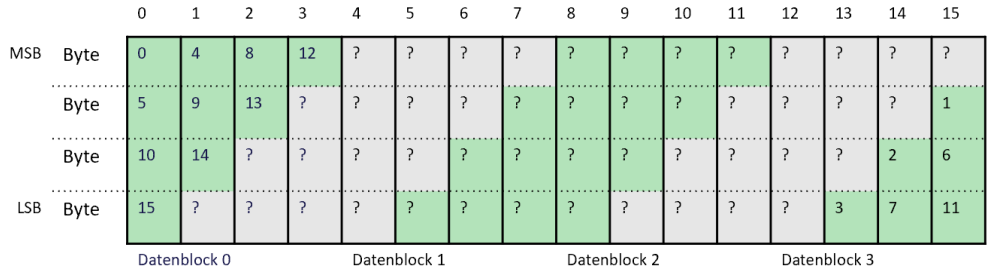

Figure 4.2: State vector with 4 AES blocks after Shift Rows without a counter rotation

A counter-rotation has to be performed to accommodate the overflows, and the vector has to be updated where an overflow happened. In 4.12 line 5, 11 and 17 perform the first rotation of for each lane. Line 7, 13, and 19 perform the counter rotations. Line 9, 15, and 21 write the rotated lanes back to a temporary vector, effectively packing it lane by lane. 4.3 shows the first state matrix in a vector after shift rows.

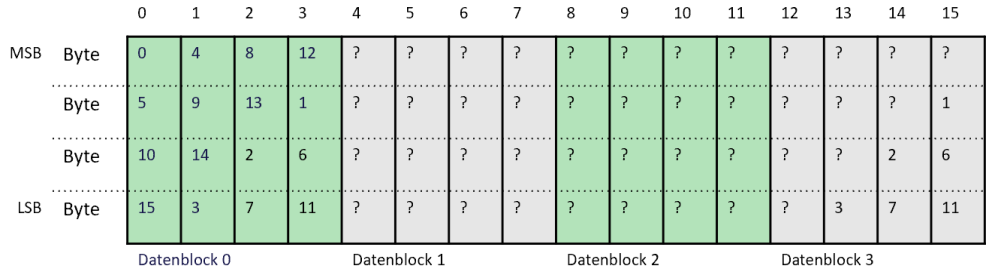

Figure 4.3: State vector with 4 AES blocks after Shift Rows

```
1  Int shiftRows(Int state) {
2      Int rotHelper = index() & 0x03;
3      Int tmpState = state & 0xff;
4
5      Int tmpRotation = rotate(state & 0xff00, 15);
6      Where(rotHelper == 3)
7          tmpRotation = rotate(tmpRotation, 4);
8      End
9      tmpState = tmpState | tmpRotation;
10
11     tmpRotation = rotate(state & 0xff0000, 14);
12     Where(rotHelper > 1)
13         tmpRotation = rotate(tmpRotation, 4);
14     End
15     tmpState = tmpState | tmpRotation;
16
17     tmpRotation = rotate(state & 0xff000000, 13);
18     Where(rotHelper > 0)
19         tmpRotation = rotate(tmpRotation, 4);
20     End
21     return tmpState | tmpRotation;
22 }
```

Listing 4.12: Shift Rows in lanes

4.3.4 Mix Columns

Compared to our previous, naive implementation of Mix Columns, the 8-bit implementation is significantly shorter. Here, all Integers are rotated up by a Bytes width, using an Integer rotation right by 8 bit, and then multiplied in $GF(2^8)$ by 2, rotated again by to align the next Byte and multiplied by 3. In the next two rotations, the state would need to be multiplied by 1 both times, but a Multiplication by 1 does not alter the value in $GF(2^8)$. At last, the additions of the intermediate products, as necessary in a matrix multiplication are performed by performing exclusive-or operations. This way, each column of the state, respectively, in each element of the vector, is multiplied by the MDS matrix. To speed

up the computationally expensive Galois-multiplication, we had to make sure the second operand to the Galois-multiplication is a uniform vector. Because exclusive-or operations are commutative, we were able to align the Integer rotations and the matrix. This way, conditional execution of operations can be reduced to a minimum. 4.3.4
2.1.2

```
1 Int mix_columns(Int state) {
2     return xtime(state) ^ galois_mul64Byte3(ror(state, 8)) ^ ror(state, 16) ^
      ror(state, 24);
3 }
```

Listing 4.13: GPU implementation of Mix Columns

Galois Multiplication The Galois Multiplication can be executed on all 64 Bytes per state vector at once. To accomplish this and speed it up at the same time, we need a uniform and small operand. Luckily, the Mix Columns step can be restructured to provide both thanks to the addition in $GF(2^8)$ being commutative. Now that one of the operands is known to be either 2 or 3, we can produce two highly optimized algorithms. There is just a small problem that needed to be solved first. The other operand can't be controlled, since each of the states Bytes is random, from the methods point of view. Still, we can't get rid of one conditional operation: the reduction by the irreducible polynomial 0x11b. But we can mask out the bytes where this operation does not apply. This is made possible by the newly introduced 8-bit based instruction "muld". We take the most significant bit (MSB) from each of the Bytes and replicate it across the Byte by multiplying it by 253. The mul-ALU interprets both operands of the muld-operation as being in the range of 0.0 to 1.0. See 3.2 and 2.19. The value interpreted by the muld-operation can be approximated by dividing the actual value by 255. In line 4 of listing 4.14, the multiplication of the high bit, with a value of $128/255 = 0.5$ by $253/255 = 0,99$ results in a replication of the high across the Byte. This can now be used as a mask for writing back results of the reduction by the irreducible polynomial only where the mask applies.

```
1 Int gpu_xtime(Int state) { // Galois multiplication by 2
2     Char a = toChar(toInt(toChar(state) & (char)0x7F) << 1);
3     Char mask = toChar(state) & (char)0x80;
4     return toInt(a ^ ((char)0x1B & (mask | muld(mask, (char)253))));
5 }
6
7 Int galois_mul64Byte3(Int state) { // galois multiplication by 3
8     return state ^ gpu_xtime(state);
9 }
```

Listing 4.14: Shift Rows in lanes

Complete Algorithm In the end, we bring all the steps together while following the specification exactly to form an algorithm similar to the example in 2.1. In between, several gathers through the TMU are performed to pre-fetch data as early as possible.

```
void gpu_enc_kernel_function(Ptr<Int> base_state_ptr, Ptr<Int> key_ptr, Ptr<Int> sub_box_ptr, Int aes_rounds, Int n) {
    Int key;
    Int state;
    Ptr<Int> state_ptr = base_state_ptr + index() + (me() << 4);
    Int increment = numQPUs() << 4;
    gather(state_ptr);

    For (Int i = 0, i < n, i = i + increment)
      gather(key_ptr + (index() & 0x03)); // gather the first Round Key from the main memory / Level 2 Cache
      receive(state);
        receive(key);

        state = state ^ key; // Add Round Key

        For(Int round = 1, round < aes_rounds, round = round + 1)
            state = sub_bytes(state, sub_box_ptr); // Sub Bytes
          gather(key_ptr + (round << 2) + (index() & 0x03)); // gather the next Round Key from the main memory / Level 2 Cache
            state = mix_columns(shift_rows(state)); // Shift Rows and immediately Mix Columns to avoid writing to a register
          receive(key); // receive Key from TMU FIFO
            state = state ^ key; // Add Round Key
        End

        state = sub_bytes(state, sub_box_ptr); // Sub Bytes
      gather(key_ptr + (aes_rounds << 2) + (index() & 0x03)); // gather the last Round Key from the main memory / Level 2 Cache
      gather(state_ptr + i + increment); // already gather the next block for encryption
      state = shift_rows(state); // Shift Rows
      receive(key); // receive Key from TMU FIFO
        store(state ^ key, state_ptr + i); // Add Round Key and store the encrypted block to the main memory
    End
    receive(state); // empty the FIFO
    return;
}
```

Listing 4.15: The complete algorithm for AES encryption

4.3.5 Decryption

For decryption, all steps are in reversed order. All steps also reverse their corresponding steps of encryption. The inverted methods can be found in the appendix 7.

```
void gpu_dec_kernel_function(Ptr<Int> base_state_ptr, Ptr<Int> key_ptr, Ptr<Int> sub_box_ptr, Int aes_rounds, Int n) {
    Int key;
    Int state;
    Ptr<Int> state_ptr = base_state_ptr + index() + (me() << 4);
    Int increment = numQPUs() << 4;
    gather(state_ptr); //


    For (Int i = 0, i < n, i = i + increment)
        gather(key_ptr + (aes_rounds << 2) + (index() & 0x03));
        receive(state);
        receive(key);

        state = state ^ key; // Add Round Key

        For(Int round = aes_rounds-1, round > 0, round = round - 1)
            state = inv_shift_rows(state); // Shift Rows inverse
            state = sub_bytes(state, sub_box_ptr); // Sub Bytes inverse
            gather(key_ptr + (round << 2) + (index() & 0x03));
            receive(key);
            state = state ^ key; // Add Round Key
            state = inverse_mix_columns(state); // Mix Columns inverse
        End

        state = inv_shift_rows(state); // Shift Rows inverse
        state = sub_bytes(state, sub_box_ptr); // Sub Bytes inverse
        gather(key_ptr + (index() & 0x03));
        gather(state_ptr + i + increment);
        receive(key);
        store(state ^ key, state_ptr + i); // Add Round Key and store decrypted block
    End
    receive(state);
    return;
}
```

Listing 4.16: The complete algorithm for AES decryption

4.4 Summary

In this chapter, we implemented two different encryption algorithms, the former operating on 32-bit Integers and the latter operating on packed 8-bit Integers, making use of just one unsigned 8-bit Integer instruction. The first implementation was slow and wasted three-quarters of its used bandwidth. But it was the first proof that AES can indeed be implemented on the Raspberry Pi's GPU using the QPULib. The latter implementation had a different approach and packed four whole state-blocks per register. The QPUs hardware implementation of 8-bit datatypes wasn't as necessary as expected for that. Only the 'v8muld'-operation provides a significant advantage over the existing operations for 32-bit Integers in our implementation. The 'v8muld'-operation is used in the performance-critical

Galois-multiplication, where its usage resulted in a significant decrease of instructions for the function. We also implemented a decryption-algorithm, that reuses several functions of the encryption-algorithm. All of the steps of AES are of the same complexity for decryption as for encryption, except for the greater values used in the inverted Mix Columns function, compared to the forward Mix Columns function. Because of that, it takes the decryption a higher amount of instructions to terminate, which must results in a lower throughput if the algorithm is not memory-bound.

5 Evaluation

In this chapter, we test our implementations for performance. For comparison, we put up the OpenSSL implementation of AES running on the Raspberry Pi's ARM cores. All tests, unless otherwise stated, have been conducted on a Raspberry Pi 2 Model B running Raspbian Stretch. The implementations have been checked for correctness by encrypting using our engine and decrypting using OpenSSL's internal AES engine using OpenSSL 1.1.0. Decryption has also been checked by encrypting using OpenSSL's engine and decrypting using ours. All tests show results for AES-128 in ECB mode, as this is the only mode implemented in our engine.

5.1 Test Preparation

For testing, we built an OpenSSL engine that implements the VideoCore IV 3D AES implementation 4.3. We can now encrypt data with the command shown in 5.1.

```
$ sudo openssl enc -engine `pwd`/QPUAESEngine.so AES-128-ECB
```

Listing 5.1: Encrypt using our OpenSSL engine

First, we benchmark OpenSSL's internal AES engine, that we can call by the following command:

```
$ sudo openssl speed -engine `pwd`/QPUAESEngine.so -evp AES-128-ECB -elapsed
```

Listing 5.2: Benchmark our GPU OpenSSL engine

We can also use our engine for comparison with OpenSSL's benchmarking tool 'speed' using the following command:

```
$ openssl speed -evp AES-128-ECB
```

Listing 5.3: Benchmark OpenSSL's internal engine

To measure the CPU overhead of compiling the QPULib's Source Code (See 2.3.1) of our AES kernel function, we put timers around the compile-function that calls the Kernel's constructor. To measure the kernel time of the AES implementation on the QPUs we measured in the same way.

```
auto start = high_resolution_clock::now();
auto aes_kernel = compile(encryption_aes_kernel); // calls the QPULib's Kernel
    constructor
auto stop = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(stop-start);
```

Listing 5.4: Benchmark our GPU OpenSSL engine

The OpenSSL 'speed'-benchmark only measures data batches of up to 16384 Bytes. While testing, we found that the QPUs invocation-overhead still takes a big hit on performance at that batch size. To be able to compare batch-sizes larger than that directly, we introduced

a factor-constant that replicates the encryption data a given number of times and performs and writes it back to the output repeatedly before terminating. The absolute batch-size encrypted by the QPUs can be accurately simulated this way up to the size of the QPU's shared memory.

While testing, we also found that a quite big chunk of the encryption time in our implementation is spent copying memory from the input into the shared memory between CPU and the GPU. To show what the core algorithm is capable of, we subtracted the overhead of copying memory and calculated the throughput, therefore, without any other overhead than invocation alone.

5.2 Performance

Encryption

OpenSSL's benchmark of its internal engine yields results of 19068 kB/s for blockwise processing of 16 Bytes, one AES block, and an 18.6% faster 22446 kB/s for processing 1024 blocks per batch on the test platform. The difference in performance for the different batch sizes can be explained with overhead from invoking the algorithm more often for the same amount of data processed when using smaller batches. In between, the overhead becomes less and less significant to the complete process. Because we measured with the internal OpenSSL benchmark, values that are not provided by the benchmark are not available in our result table. For larger data batches of more than 2^{14} (16384) Bytes, we don't expect any significant speedup over the biggest tested batch size.

Data length (Bytes)	**OpenSSL**	**1 QPU**	**12 QPUs**	**12 QPUs** (kernel time)
2^4	19068	65	66	153
2^6	21435	261	265	615
2^8	22250	819	1034	2415
2^{10}	22393	1533	3715	8063
2^{12}	na	2298	10307	18618
2^{14}	22446	2635	13912	27769
2^{16}	na	2699	16896	31890
2^{18}	na	2808	21483	32576
2^{20}	na	2833	22220	33043
2^{22}	na	2840	23150	33279
2^{24}	na	2863	24309	33391
2^{26}	na	na	24993	33467
2^{27}	na	na	25087	33481

Table 5.1: Encryption throughput comparison. Results in kB/s

In the second result column for encryption, we tested throughput for a single QPU. For that, we had set the number of QPUs to one by adding the line $kernel.setNumQPUs(1);$. The throughput starts with a merely 65 kB/s for the smallest batch size of 2^4 (16) Bytes. At this point, OpenSSL's implementation is roughly 324 times faster than ours. With bigger batch sizes, the overhead of invoking the kernel becomes less significant, and a gradual speedup can be observed up to 2863 kB/s. Already at 2^{12} (4096) Bytes, the single QPU is

at ˜80% of its maximum observed performance at 2^{24} Bytes. Results for batch sizes larger than 2^{24} are not available due to a kernel time of more than 10 seconds, which is not allowed by the VideoCore IV 3D.

Next up is the column for 12 QPUs doing the encryption. At 2^4 (16) Bytes processed, no significant speedup over the single QPU can be observed. For the larger batches, the 12 QPUs ramp-up significantly faster than the single QPU from roughly 2^{10} Bytes of batch size, but are far away from a theoretically perfect scaling factor of 12 over the single QPU at this point. At 2^{12} Bytes, we are only at roughly 41% of the maximum observed throughput of 25087 kB/s at 2^2 Bytes. The difference of ˜80% of its maximum speed for the single QPU at 2^{12} Bytes of batch size, compared to the ˜41% for the 12 QPUs at the same batch-size, is suspected to be caused by more caches and pipes having to be filled up. At the maximum tested batch-size, a scaling factor of ˜8.7 over the single QPU can be observed. This leads to the conclusion, that once all pipes and caches are filled, the algorithm scales well with the number of available QPUs.

To show how much time is spent outside the kernel, we include a further column, where kernel time only was measured, and the throughput, that is shown, calculated from it. Here we can see that a minimum of 25% of the engine's runtime is spent outside the kernel function that is running on the QPUs. Most of the overhead implicitly shown here comes from copying the plaintext to the shared memory between CPU and GPU, and copying it back to the output after the kernel terminated.

Overclocking the GPU The base frequency of the Raspberry Pi's GPU is 250 MHz, but it can be easily overclocked to 500 MHz without any instability issues. To do so, the line "qpu_freq=500" was added to "/boot/config.txt", and the Raspberry Pi was rebooted. With an overclocked Raspberry Pi, the maximum throughput we measured was 38567 kB/s at a batch-size of 2^{26} Bytes, including overhead from copying data to and fro the shared memory. Excluding overhead, with only kernel time measured to calculate the throughput, a massive 66274 kB/s was measured, again at a batch-size of 2^{26} Bytes. However, while the performance was better throughout the range, it varied widely from run to run due to throttling. Interestingly, the GPU does not change clock speeds while a program is running on it.

Decryption

Again, OpenSSL's engine is very fast, as one can see in the first column of our results in table 5.2. Even from the smallest batch-size on, it reaches ˜76% of the maximum performance we measured at a batch-size of 2^{14} Bytes.

For our decryption implementation, the runtime of Shift Rows, the round-key addition, and the Byte substitution is the same as for encryption. Mix Columns, however, takes more instructions to terminate in comparison to the encryption algorithm. The second operand for decryption is greater in decryption and will, therefore, take more steps, as explained in 2.1.2. The algorithm to perform the Galois multiplication for decryption has also not been optimized and computes the result of each lane (2.15) individually. This leads to a substantial slowdown compared to our encryption algorithm.

The column showing results for a single QPU, shows similar behavior to its encryption counterpart, just slower, due to the unoptimized algorithm. At a batch-size of 2^{12} Bytes, it hits ˜97% of its maximum performance. The 12-QPU column also shows similar behavior to its encryption-counterpart, again being slower, in any case. It ramps up a bit quicker than its counterpart, already performing at ˜69% of its maximum performance at a batch-size of 2^{12}. The relatively smaller overhead, compared to the complexity of the algorithm, can explain the quicker ramp-up to maximum throughput. While the overhead stays the same for encryption and decryption, the complexity of the algorithm is greater for decryption. Also, the slow memory accesses stay the same compared to the encryption algorithm. Together with the increased complexity, this leads to better scaling over more QPUs, coming in at 11.35, and just missing the optimum scaling factor of 12. Of course, a faster algorithm would be preferred over a better scaling factor.

Data length (Bytes)	**OpenSSL**	**1 QPU**	**12 QPUs**
2^4	17083	38	37
2^6	20629	149	143
2^8	21825	229	566
2^{10}	22146	303	1320
2^{12}	na	332	2667
2^{14}	22249	339	3391
2^{16}	na	340	3582
2^{18}	na	342	3794
2^{20}	na	342	3857
2^{22}	na	na	3860
2^{24}	na	na	3883

Table 5.2: Unoptimized decryption throughput comparison including overhead

Decryption on Overclocked GPU Similarly to encryption, we tested decryption with an overclocked Raspberry Pi 2 Model B. Here, the maximum throughput nearly doubled, maxing out at 7512 kB/s the maximum tested batch-size of 2^{26} Bytes, including overhead from copying memory around. Without the overhead, a maximum throughput of 8180 kB/s was observed.

5.3 Problems

In the evaluation phase, some problems with our implementation arose.

Overhead of the QPULib

The kernel is compiled at runtime, which makes up a significant overhead each time the algorithm is set up. Compiling the AES encryption implementation to bytecode at runtime took a minimum of 88ms, and compiling the unoptimized decryption algorithm a minimum of 288 milliseconds on a Raspberry Pi 2 Model B. The minimum invocation and termination time of the compiled kernel, performing encryption on just a single block of 16 Bytes, came in at minimum of 97 microseconds in case of encryption and a minimum of 270 microseconds

in case of decryption. This leaves the implementation impractical for processing of small batches of data.

Raspberry Pi 4

The Raspberry Pi 4 is incompatible to the QPULib and our implementation due to its VideoCore VI GPU.

6 Conclusion

The main goal of the thesis was to evaluate if AES on the Raspberry Pi's GPU can be practical. We managed to implement hardware-accelerated AES on the Raspberry Pi's GPU using the especially for the VideoCore IV 3D designed QPULib-library. The first implementation was correct, but not particularly fast or practical, since it wasted most of the bandwidth and also took four times more space in the shared memory than necessary.

We extended the QPULib by the datatype 'Char' for optimizations of our implementation. This datatype operates on all 64 Bytes that fit into a 16-element vector of 32-bit wide elements at once, making it effectively 64-way SIMD for some operations. The 'Char'-implementation can not only be used for AES, but for all kinds of other Byte-based algorithms that are parallelizable as well.

Another, faster version was then implemented, using the extensions to the QPULib, that we previously made. The second algorithm is faster and does not use any more bandwidth to the main memory than necessary. It is also less impractical since it also does not waste space in the memory shared between GPU and CPU anymore. Copying memory is faster too because no write-stride is any longer necessary. The algorithm has been checked for correctness using OpenSSL and can thus be used in real-world implementations. While the implementation takes the workload from the CPU, it does not provide any significant speedup. The performance was held back by CPU overhead for copying memory mostly. Unfortunately, there is no know way around that. The algorithm itself, however, is well suited for a SIMD implementation. Further optimizations, as described in 6, could speed up the implementation and possibly even make it practical.

Future Work

To make the implementation faster and, therefore, more practical, several more measures can be taken.

Manual optimization The QPULib offers a high-level interface to the VideoCore IV 3D. While the generated bytecode seems to be generally good and reliable, a manually optimized algorithm could be faster by making use of unimplemented operators, side effects, and more flexible register allocation.

Pack and Unpack functionality for Vectors Implement the concept of individual 8-bit vectors shown in 3.1.2. Using the pack and unpack functionality would result in a significant speedup of the evaluated algorithm. An example of the potential is described here 4.3.2.

Automatically filled pipeline The Vertex Pipe Memory offers pre-programmed reads from the main memory. These offer a potential speed up to the algorithm due to latency for each read of an AES plaintext block from the main memory. Each read of the next state block is predictable and can, therefore, be pre-programmed. This would also reduce the computational overhead for setting up reads via the TMU, which is already heavily utilized by repeated fast memory lookups for the Sub Bytes step and the key-addition.

Performance Counters As a final optimization measure to the AES algorithm on the QPUs, Performance Counters could be used. Performance Counters are hardware implemented in the QPUs and offer a set of counters that can be read as a vector by the QPU. A total of 16 out of 30 available Performance Counters can be set up and read at different times while the program runs on the QPUs. Results can then be interpreted by the programmer and optimizations evaluated accordingly. See figure 7.2 for implemented counters.

AES Modes Currently, our implementation only features the AES-128-ECB Mode 2.2, a mode that is not used very often because of its security flaws. For modes that are not parallelizable, this would come with a very significant performance hit, however. In these modes, paired with an implementation similar to ours, a hardware register would only be filled up to a quarter, leaving three-quarters of the resources of a QPU unused, and would, therefore, be precisely four times slower than our single-QPU benchmarking results shown in table 5.1.

Speedup decryption With little work, the Galois multiplication used in our decryption algorithm can be optimized and sped up. This could result in a decryption throughput close to our encryption throughput.

7 Appendix

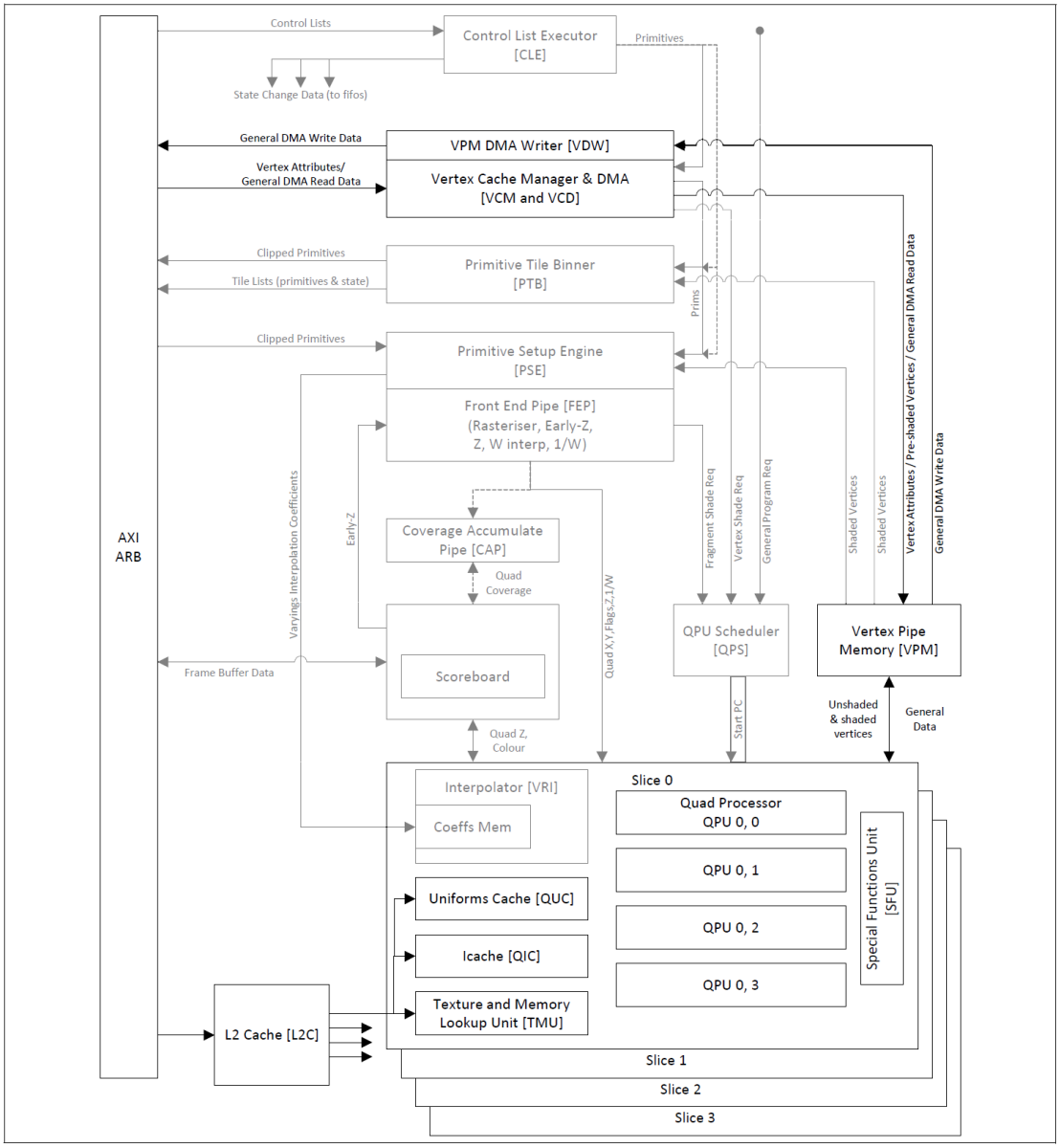

Figure 7.1: VideoCore IV 3D archtitecture overview, graphics only hardware is greyed out.

Performance Counter Count Source IDs	
Count ID	***Count Description***
0	FEP Valid primitives that result in no rendered pixels, for all rendered tiles
1	FEP Valid primitives for all rendered tiles. (primitives may be counted in more than one tile)
2	FEP Early-Z/Near/Far clipped quads
3	FEP Valid quads
4	TLB Quads with no pixels passing the stencil test
5	TLB Quads with no pixels passing the Z and stencil tests
6	TLB Quads with any pixels passing the Z and stencil tests
7	TLB Quads with all pixels having zero coverage
8	TLB Quads with any pixels having non-zero coverage
9	TLB Quads with valid pixels written to color buffer
10	PTB Primitives discarded by being outside the viewport
11	PTB Primitives that need clipping
12	PSE Primitives that are discarded because they are reversed
13	QPU Total idle clock cycles for all QPUs
14	QPU Total clock cycles for all QPUs doing vertex/coordinate shading
15	QPU Total clock cycles for all QPUs doing fragment shading
16	QPU Total clock cycles for all QPUs executing valid instructions
17	QPU Total clock cycles for all QPUs stalled waiting for TMUs
18	QPU Total clock cycles for all QPUs stalled waiting for Scoreboard
19	QPU Total clock cycles for all QPUs stalled waiting for Varyings
20	QPU Total instruction cache hits for all slices
21	QPU Total instruction cache misses for all slices
22	QPU Total uniforms cache hits for all slices
23	QPU Total uniforms cache misses for all slices
24	TMU Total texture quads processed
25	TMU Total texture cache misses (number of fetches from memory/L2cache)
26	VPM Total clock cycles VDW is stalled waiting for VPM access
27	VPM Total clock cycles VCD is stalled waiting for VPM access
28	L2C Total Level 2 cache hits
29	L2C Total Level 2 cache misses

Figure 7.2: Available Performance Counters on the Raspberry Pi's GPU to measure various performance critical aspects of a program run on it

Performance measurements

numQPUs	no-pre-fetch, blocking stores	pre-fetch, no wait for store
1	270 ms	73 ms
2	185 ms	63 ms
3	185 ms	63 ms
4	185 ms	63 ms
5	185 ms	63 ms
6	184 ms	63 ms
7	184 ms	63 ms
8	184 ms	63 ms
9	184 ms	63 ms
10	184 ms	63 ms
11	185 ms	63 ms
12	184 ms	63 ms

Table 7.1: Comparing memory performance of a simple 'a + b = c' loop iterated over 1.6 million values. The first result column shows blocking loads and blocking stores, which stall the QPUs until completed before performing the next instruction. The second result column shows times for the same algorithm without waiting for stores to main memory to be completed, and with a pre-fetch of one value via the TMU.

Active QPUs	Clock high load via VPM	Clock high load via TMU
1	591ms	350 ms
2	345ms	175 ms
3	234ms	116 ms
4	205ms	87 ms
5	202ms	70 ms
6	201ms	64 ms
7	203ms	64 ms
8	202ms	64 ms
9	202ms	64 ms
10	202ms	64 ms
11	202ms	64 ms
12	202ms	64 ms

Table 7.2: Comparison of an unoptimized galois multiplication with blocking load and store operations via VPM versus loads via the TMU and non-blocking stores. Results show times for processing of 1.6M values in milliseconds.

Active QPUs	original algor.	original algor. parameters switched	optimized algor.
1	162	66	25 ms
2	81	33	12 ms
3	54	22	8 ms
4	40	16	6 ms
5	32	13	5 ms
6	27	11	4 ms
7	23	9	3 ms
8	20	8	3 ms
9	18	9	2 ms
10	16	6	2 ms
11	14	6	2 ms
12	13	6	2 ms

Table 7.3: Unknown operands implementation of the Galois multiplication versus optimized algorithm for values that occur in AES encryption. Results show times for processing of 1.0M values, clock high

numQPUs	time in milliseconds	throughput in kB/s
1 QPUs	1683 ms	594 kB/s
2 QPUs	842 ms	1187 kB/s
3 QPUs	562 ms	1779 kB/s
4 QPUs	421 ms	2375 kB/s
5 QPUs	337 ms	2967 kB/s
6 QPUs	281 ms	3558 kB/s
7 QPUs	240 ms	4166 kB/s
8 QPUs	210 ms	4761 kB/s
9 QPUs	188 ms	5319 kB/s
10 QPUs	168 ms	5952 kB/s
11 QPUs	153 ms	6535 kB/s
12 QPUs	140 ms	7142 kB/s

Table 7.4: AES-128-ECB encryption with the first implementation shown in approach to 4.2. Here, we measured kernel time only, overhead is not factored in. Throughput is calculated from timings. Scaling is near perfection with increasing number of QPUs, suggesting no memory bottleneck at all. 1.0M values, clock high

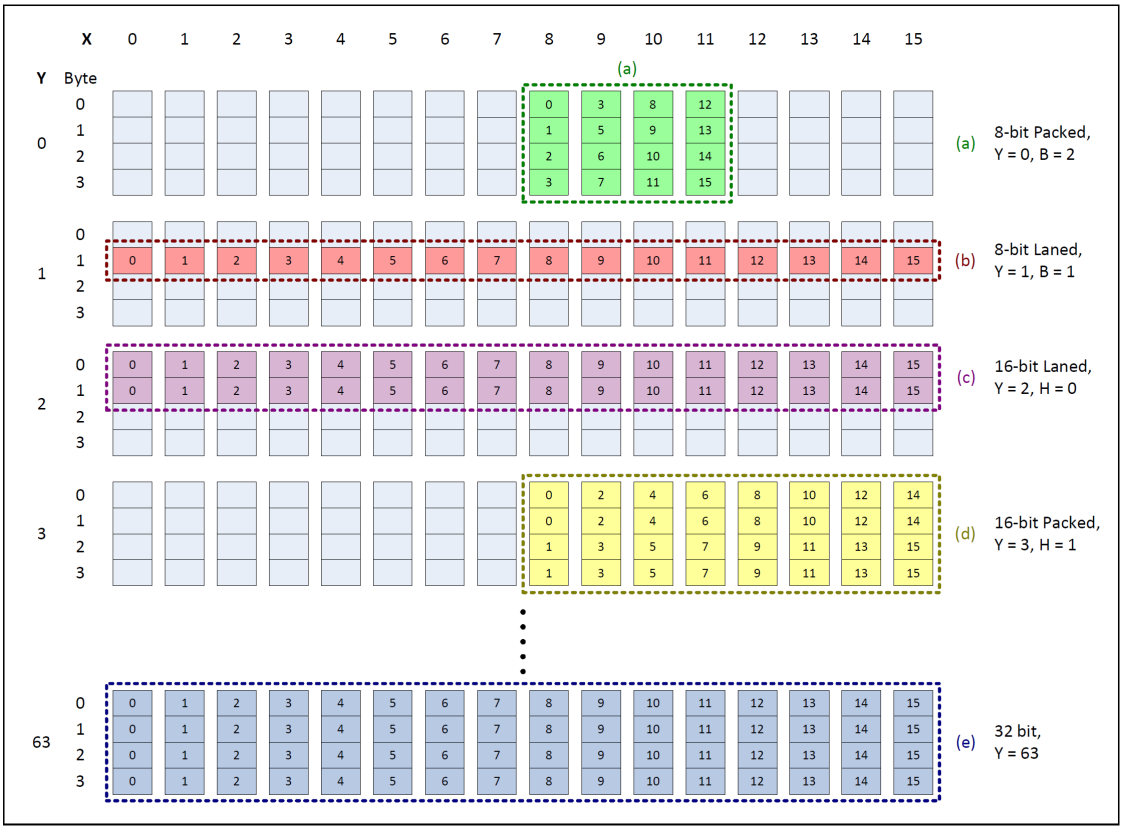

Figure 7.3: Horizontal access mode to the VPM

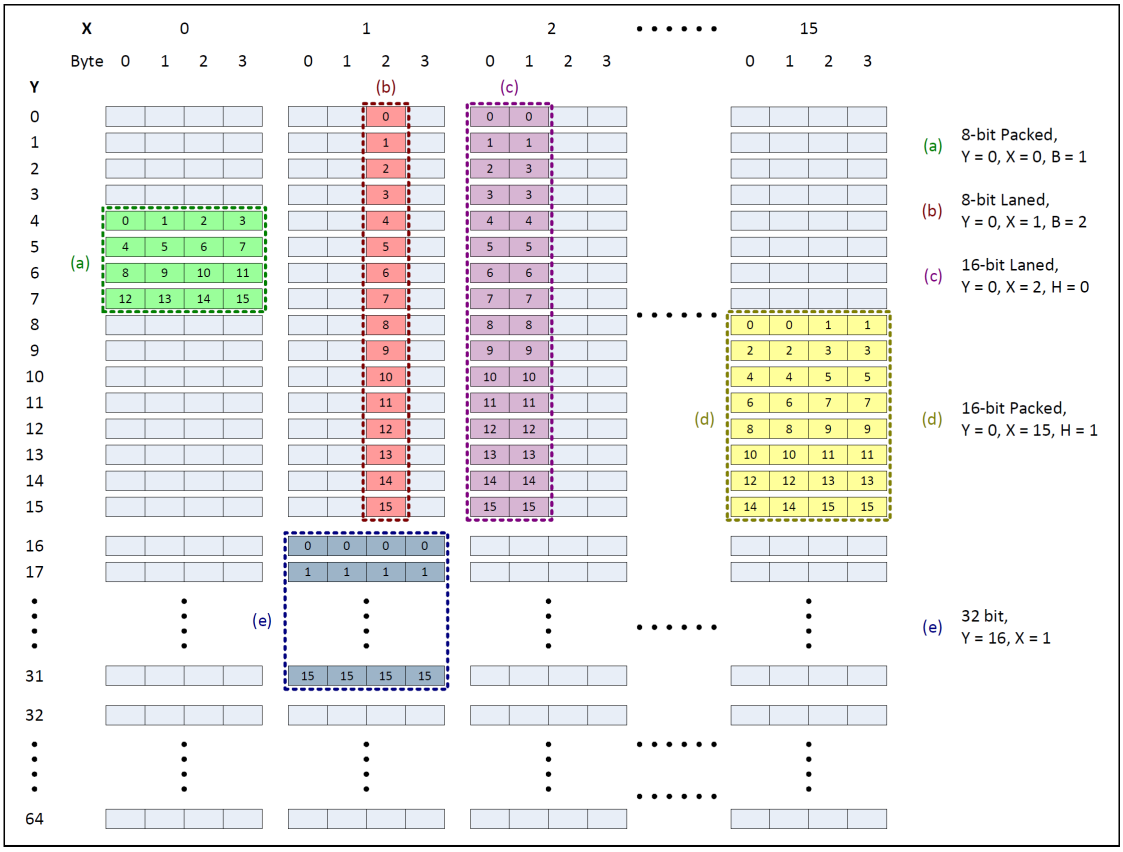

Figure 7.4: Vertical access mode to the VPM

```
Int inv_shift_rows(Int state) { // CORRECT for rotated
    Int rotHelper = index() & 0x03;
    Int tmpState = state & 0xff; // row 0 set

    Int tmpRotation = rotate(state & 0xff00, 1);
    Where(rotHelper == 0)
        tmpRotation = rotate(state & 0xff00, 13);
    End
    tmpState = tmpState | tmpRotation;

    tmpRotation = rotate(state & 0xff0000, 2);
    Where(rotHelper < 2)
        tmpRotation = rotate(state & 0xff0000, 14);
    End
    tmpState = tmpState | tmpRotation;

    tmpRotation = rotate(state & 0xff000000, 3);
    Where(rotHelper < 3)
        tmpRotation = rotate(state & 0xff000000, 15);
    End
    return tmpState | tmpRotation;
}
```

Listing 7.1: The Shift Rows step inverted for decryption has the same complexity as the non-inversed function used for encryption.

```
Int inverse_mix_columns(Int state) {
    return    galois_multiplication_per_lane(state, 0x0E)
            ^ galois_multiplication_per_lane(ror(state, 8), 0x0B)
            ^ galois_multiplication_per_lane(ror(state, 16), 0x0D)
            ^ galois_multiplication_per_lane(ror(state, 24), 0x09);
}
```

Listing 7.2: The inverted Mix Columns step for decryption. To make use of the fast Accumulators, we avoid writing intermediate results to variables.

```
Int galois_multiplication_per_lane(Int state, Int orig_b) {
    Int result_state = 0; // temporary state

    For(Int i = 0, i < 4, i = i+1) // for each lane

        Int lane = (state & (0xFF << (i << 3))) >> (i << 3); // unpacking of
    the lane
        Int old_a;
        Int b = orig_b; // load second operand
        Int p = 0;

        for(int j = 0; j < 4; j++) { // max 4 bits set, so we never need a
    full Galois-multiplication iterating over 8 bits,
            but just 4
            Where((b & 1) == 1) // if low bit is set
              p = p ^ lane; // update result
          End

            old_a = lane; // save high bits before shifting them out of
    context
            lane = lane << 1; //
            Where((old_a & 0x80) == 0x80) // high_bit_set true
               lane = lane ^ 0x11B; // reduce with polynom; delete overflow
            End

            b = b >> 1; // shift operand b to the right, deviding it by 2
        }
        result_state = result_state | (p << (i << 3)); // write result back to
     lane
    End
    return result_state;
}
```

Listing 7.3: The GPU implementation of Mix Columns used for decryption. Operand b is never greater than 0x0E, so we never need more than four iterations. This implementation of the Galois-multiplication is rather slow, as shown in the Evaluation chapter.

List of Figures

Bibliography

[AFD17] Abdelrahman, Ahmed A. ; Fouad, Mohamed M. ; Dahshan, Hisham: High Performance CUDA AES Implementation: A Quantitative Performance Analysis Approach. (2017). https://www.researchgate.net/publication/317475198_High_Performance_CUDA_AES_Implementation_A_Quantitative_Performance_Analysis_Approach

[Bro13] Broadcom (Hrsg.): *VideoCore® IV 3D Architecture Reference Guide*. Broadcom, September 2013. https://docs.broadcom.com/docs-and-downloads/docs/support/videocore/VideoCoreIV-AG100-R.pdf

[Com07] Commons, Wikimedia: *Block cipher mode of operation*. https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation. Version: 2007

[DR02] Daemen, Joan ; Rijmen, Vincent: *The Design of Rijndael: AES - The Advanced Encryption Standard*. 2002

[Haa08] Haan, Laurent: *Advanced Encryption Standard (AES)*. http://codeplanet.eu/tutorials/cpp/51-advanced-encryption-standard.html. Version: Februar 2008. – Online; accessed 7-December-2019

[Nay16] Naylor, Matthew: *QPULib*. https://github.com/mn416/QPULib, 2016. – Online; accessed 7-December-2019

[nin15] *PyVideoCore*. https://github.com/nineties/py-videocore, 2015. – Online; accessed 7-December-2019

[Pau17] Pauls, Paul: Parallelization of AES on Raspberry Pi GPU in Assembly. (2017), Juli. http://mnm-team.org/pub/Fopras/paul17/PDF-Version/paul17.pdf

[RP17] Raspberry Pi, Entwicklung einer OpenCL-Implementierung für die VideoCore IV GPU d.: *VC4CL*. https://github.com/doe300/VC4CL, 2017. – Online; accessed 7-December-2019

[WC19] Wang, Canhui ; Chu, Xiaowen: GPU Accelerated AES Algorithm. In: *CoRR* abs/1902.05234 (2019). http://arxiv.org/abs/1902.05234