POLITÉCNICA
"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL

Degree in Computer Engineering

Technical University of Madrid

School of Computer Engineering

End of Term Project

GPU Accelerated AES

Research Center for Computational Simulation
Research Group on Quantum Information and Computation

Author: Jesús Martín Berlanga

Tutor: Dr. Jesús Martinez Mateo

MADRID, JUNE 2017

Abstract

An open-source AES GPU implementation has been developed from scratch with CUDA. The implementation is based in lookup tables and supports ECB, CTR, CBC*, and CFB* operation modes. Factors such as the parallelism level, the use of constant and shared memory, asynchronous page-locked IO, and overlapping between data transfers and calculations has been taking into account. The results show a speedup peak of 119% with CTR encryption against OpenSSL's CPU implementation with files larger than 60 MB (without AES-NI support) and a peak of 87% with CBC/CFB decryption of files larger than 125 MB. The maximum registered throughput is 237 MB/s, above drive buffer read rate of 208 MB/s (non-cached). Although it is expected to obtain performance yields around 55% with CTR encryption and 49% with CBC or CFB decryption, it is recommended to consider the use of a CPU implementation with AES-NI hardware until the current implementation is further analysed and, if possible, optimized.

Index terms— High Performance Computing (HPC), General Purpose Graphics Processing Unit (GPGPU), Advanced Encryption Standard (AES)

*only decryption

Resumen

Se ha realizado una implementación GPU de código abierto en CUDA del cifrador AES con tablas de búsqueda en los modos de operación ECB, CTR, CBC*, y CFB*. Se han tenido en cuenta distintos factores de rendimiento, entre ellos el nivel de paralelismo, el uso de memoria constante y compartida, la lectura asíncrona de ficheros en memoria no paginada, y el solapamiento de transferencias y cómputo. Se ha alcanzado una mejora máxima de 119 % para el cifrado en modo CTR con respecto a la implementación CPU de OpenSSL (sin AES-NI) y una mejora máxima del 87 % para el descifrado CBC/CFB. El mayor rendimiento registrado es de 237 MB/s, que esta por encima de la tasa de lectura con buffer del disco de 208 MB/s (sin cache). Aunque se pueden obtener mejoras del 55 % para el cifrado CTR con ficheros con tamaño mayor de 60 MB y del 49 % para descifrado CBC o CFB con ficheros mayores de 125 MB, se recomienda considerar el uso de una implementación CPU junto con hardware con soporte de la extensión AES-NI hasta que la implementación actual se analize en mas detalle y, si es posible, se mejore su optimización.

Palabras clave— Computación de Alto Rendimiento (HPC), Cómputos de Propósito General en Unidades de Procesamiento de Gráficos (GPGPU), Advanced Encryption Standard (AES)

*solo descrifrado

Preface

There is an increasing need for efficient solutions to the computationally intensive cryptography found in many specialized systems where the cipher usually act as a bottleneck that cannot sustain high data transference rate situations. For this reason, it is proposed to explore the viability of cipher algorithms in GPU architectures. In particular, we will measure the performance improvements of a GPU accelerated AES implementation. The benefits of using this computing platforms are clear when we have to process large amounts of data as occurs, for example, in an organization which has to deal with highly sensitive information and consequently requires to perform numerous ciphered backups. These organizations could be able to reduce costs and time by using a GPU cluster for backup purposes. We should neither forget about the proliferation of increasingly affordable high-end GPUs the end user could take advantage of with a GPU accelerated cryptography standard library. Due to the intrinsic intensity of operations during encryption or decryption the CPU may need to use all its resources to finish processing large files in a reasonable time. As a result, a GPU counterpart would be greatly convenient to exploit its unused capacity to solve the problem, even in less time, without saturating the main computing unit that could still be used for other tasks.

Contents

List of Figures

CHAPTER 1

Introduction

This paper explore different available alternatives regarding to the GPU acceleration of an Advanced Encryption Standard (AES) software implementation. The aim is to present if GPUs could be used to efficiently satisfy the high-throughput necesities when processing large rates of data. The reader will be thoroughly introduced in the area of GPU cipher acceleration so he can obtain a better understanding of the challenges and possibilities GPU hardware has to offer. For this reason, we will review the characteristics of symmetric ciphers, the impact of the block mode of operation for parallelization, and the advantages and disadvantages of diverse parallel programming models. We will discuss the specific details of a high-performance AES implementation along with its performance outcomes and limitations. The work here presented is materialized in *Paracrypt* [1], an open-source cryptographic library the reader is free to fork, improve, or use for his own interests, under the GNU General Public License v3.0 (GPLv3) [27].

1.1 Symmetric ciphers

Symmetric ciphers are those which use the same cryptographic key for encryption and decryption of a text. We will focus in symmetric ciphers because they are more suitable for large quantities of data. The encrypted output size of asymmetric ciphers is asymptotic, which means that the ciphertext take more disk space than the original plaintext. For large files this overhead is considerably high. Furthermore, their performance tend to be several times lower than the symmetric counterpart as they have to manipulate larger keys. Consequently, asymmetric ciphers are preferably used with small plaintexts. For instance, they are often used as a part of a hybrid cryptosystem [2] where the asymmetric cipher is used to encrypt symmetric encryption keys. In this way, we can obtain the best of both worlds: we can encrypt the major part of the data with an efficient symmetric algorithm at the same time we can use the asymmetric encryption for a safe key exchange process. Note that if S wants to secretly transmit a message to R with a symmetric cipher, first, both must previously know the shared secret key. Normally R does not know the key beforehand so we have to find a way to safely transmit this key.

1.1.1 Stream ciphers

Stream ciphers generate the protected output by combining the keystream, a sequentially generated pseudo-random key, with the plaintext. The seed from which all the keystream digits will be generated effectively serves as the key from which the receiver will be able to decrypt the message. Each successive keystream bit to be generated is dependent on the internal state, this is the same as stating that the generation is strictly sequential. As a result, it is hard to imagine how we could massively parallelize a conventional* stream algorithm with GPUs.

Nevertheless, these ciphers are used for their simplicity and easiness of implementation in hardware. Some ciphers are even designed to display some signs of parallelism, Trivium for example, can generate its keystream in chunks up to 64 bits [3]. Software can take advantage of this by making operations with 16, 32, or 64 bit integers but it cannot be further parallelized.

1.1.2 Block ciphers

On the other hand, block ciphers operate by applying the same transformation to each fixed size group of bits. This is parallel by definition because each block can be treated independently.

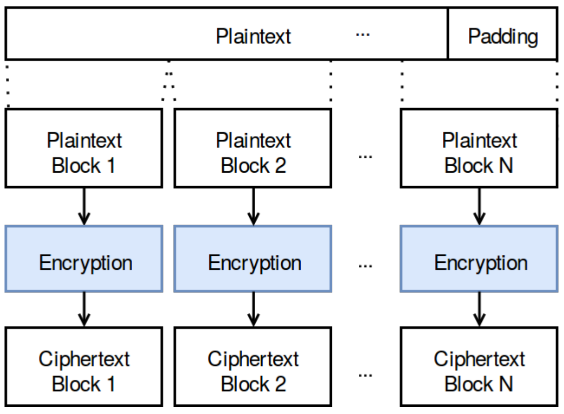

Figure 1.1: ECB encryption

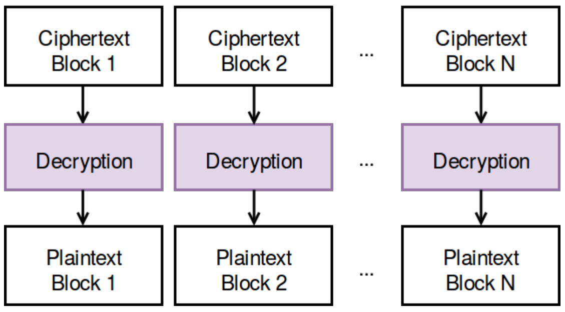

Figure 1.2: ECB decryption

Unfortunately, this raw mode of operation, the *Electronic Code-Book* (ECB), should be avoided [4, Sec. II.A] because it can lead to some security leaks: repetitions in the original text can be also found as repetitions in the ciphered text [5, Sec. 2.1]. An attacker could perform an statistical analysis and, for example, correlate information about passwords in a database.

The *Cipher Block Chaining* (CBC) mode mode tries to fix this problem by XORing the cipher input of each block with the output of the previous block. Although its the most

*Few stream ciphers support massive parallelization. CryptMT is one of those exceptions.

used mode of operation, it has one major drawback, its encryption is unparallelizable. Fortunately, the decryption can be simultaneously computed because the output of each decrypted block has to be XORed with the ciphertext of the previous block to which we already have access as can be seen in figure 1.4.

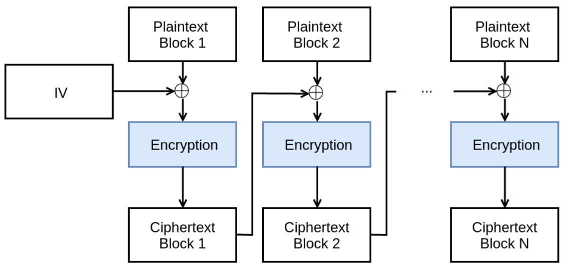

Figure 1.3: CBC encryption

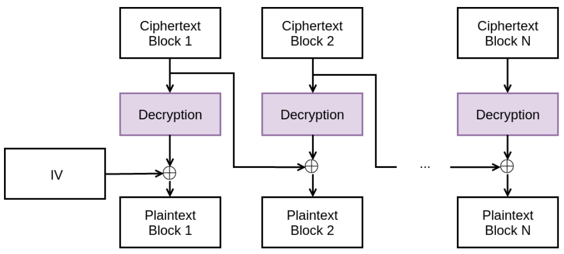

Figure 1.4: CBC decryption

To ensure the semantic security that avoid attackers to obtain information when reusing the same scheme with the same key, we have to introduce a random component that allow to obtain a different output when ciphering the same message with the same key. For this, most modes use what is called an initialization vector (IV) [5, Appx. C], a random starting block that will propagate its variability thorough the algorithm as happens with the CBC mode. This puts into perspective the insecurity of the ECB algorithm for which the IV would be useless [6] because each block is treated independently and thus, it wouldn't be propagated to the remaining blocks. In the CBC decryption mode the IV would not be able to propagate neither as it would only affect the first block, that's the reason why we cannot think about switching the XOR order if we are interested in speeding-up encryption instead of decryption. The *Propagating Cipher Block Chaining* (PCBC) mode is designed to also propagate small changes during decryption [7] yet at the expense of making it completely unparallelizable.

Considering that the plaintext will probably not be multiple of the block size, we need to define a padding scheme to ensure that ECB and CBC modes can correctly operate with fixed size divisions of the message [5, Appx. A]. It can be done by simply appending zeros to the end of the message but this could get us into problems with plaintexts that already contain zeros at the tail. It would be better to restrain ourselves to standardized padding schemes defined, for example, in the PKCS#7 [8, Sec. 10.3]. It should be noted that padding schemes always leak some information about the original length of the plaintext [9]. Whereas in some applications it might not be of importance to publish the original plaintext length, in others it may be critical to avoid size-based correlation attacks. For this reason, we can go further and apply a random size padding [10] that could increase the message length, up to a certain limit, by several blocks.

Complex solutions such as the *Ciphertext Stealing* (CTS) [11] can also solve the padding problem but there are better known alternatives that can get rid of these complications: The *Cipher Feedback* (CFB) mode is very similar to the CBC mode but does not need padding because a block output is obtained by XORing the plaintext with the cipher output of the previous block and thus the plaintext is never used for the input of the block cipher encryption (the input for the fist block is the IV). The CFB decryption can also be parallelized as happened with the CBC mode.

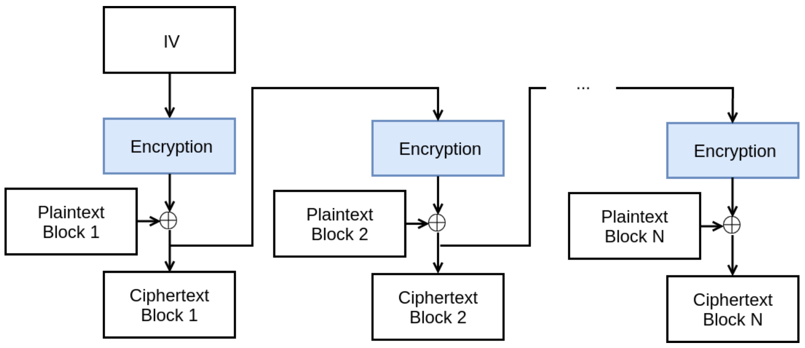

Figure 1.5: CFB encryption

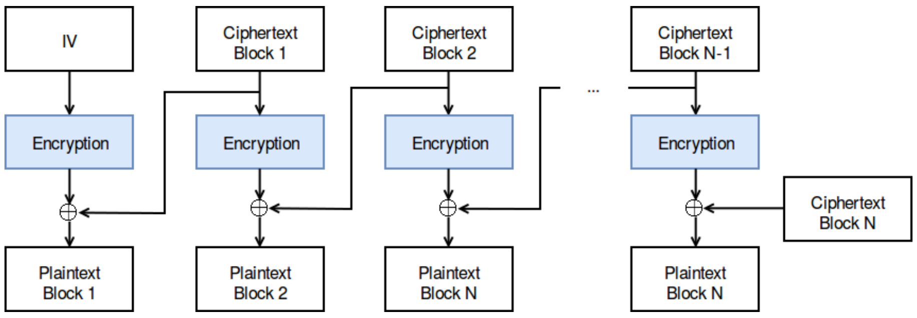

Figure 1.6: CFB decryption

The *Counter* mode (CTR) neither have to deal with padding as it does not feed the block cipher with the plaintext, instead, a generated keystream is ciphered (as happens in a synchronous stream cipher) and XORed with the plaintext. The keystream can be composed of any deterministic sequence. The most simple and popular approach is to cipher each block with an incremental counter, hence the name of the mode. The algorithm is considered to be secure even if the counter sequence is public. The great advantage of this mode is its parallelism support for both encryption and decryption. In this case we do not have an IV because blocks do not produce feedback between them. Alternatively, a nonce is to be used [5, Appx. B]. The nonce is an acknowledged randomly generated block-size number that is combined (addition, xor, or any other suitable operation) with each sequence value to act as a arbitrary offset for the counter. Finally, note in fig. 1.7 how the CTR mode uses the block cipher in encryption mode. In the CTR mode we can cipher and decipher with the same encryption code.

When dealing with large datasets, an important characteristic to take into account is the ability to randomly access information by decrypting a portion of the ciphertext without needing to decrypt the whole file. If an algorithm is completely paralellizable at decryption it means that each block can be decrypted individually. For this reason, those algorithms - ECB, CBC, CFB, and CTR - support random read access.

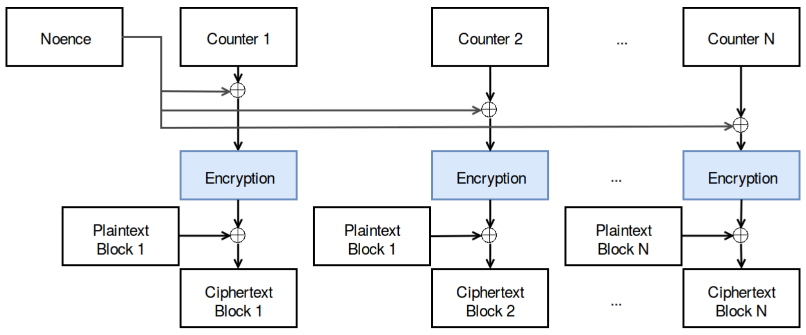

Figure 1.7: CTR encryption

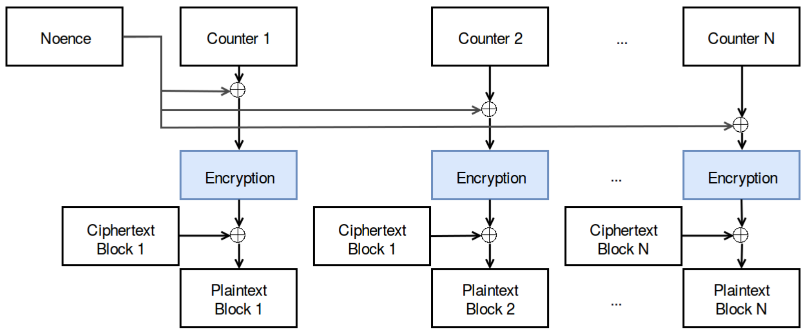

Figure 1.8: CTR decryption

1.1.3 Advanced Encryption Standard

The Advanced Encryption Standard (AES) [19] is a subset of the Rijndael cipher specified by the National Institute of Standards and Technology (NIST). The AES uses a fixed block size of 128 bits (16 bytes) whereas, in contrast, Rijndael specifies a set of ciphers with different keys and block sizes. The AES can use a key of 128, 192, and 256 bits that are expanded into round keys of 128 bits. Using a bigger key size increases security because more transformation rounds are performed during encryption. Each round/cycle features a series of steps with both substitutions and permutations. The AES is fast in both hardware and software with the use of lookup tables and has become the most popular and widely accepted symmetric-key cryptographic algorithm.

1.2 High performance platforms

Traditionally, GPU programming has required the use of languages specialized in shaders and textures [12] typically used for computer graphics. However, new platforms such as OpenCL and CUDA are targeting General-Purpose Computing on Graphics Processing Units (GPGPU) and support arithmetic operation with integers which make them attractive for the parallelization of cryptographic algorithms.

Whilst OpenCL can provide a higher degree of portability targeting any many-core GPU or CPU architecture it also impedes to explicitly target hardware peculiarities. OpenCL portability makes it an ideal tool for heterogeneous computing but CUDA permit to develop the source code taking into account specific architectural details which, for most applications, permit to reach higher performances [47]. Furthermore, even though CUDA is proprietary to NVIDIA and can only be used with NVIDIA GPUs, it has more advanced documentation, a bigger community of developers, more libraries, and more advanced compilers, tools and profilers. We choose CUDA because its both more attractive for development and can typically achieve higher performances than OpenCL.

On the CPU side, we have to mention the *Advanced Encryption Standard Intruction Set* (AES-NI), an extension of Intel and AMD *x86* architecture designed to accelerate AES encryption and decryption. AES-NI is available in OpenSSL, a very popular library available for most Unix operative systems whose core implement multiple cryptographic functions.

CHAPTER 2

Implementations

By using the OpenSSL CPU AES implementation as a reference it has been possible to stablish the foundations of a completely new (CUDA) GPU cryptographic library, Paracrypt. The code of the key schedule sequential algorithm has been reutilized but (everything else, including) the core of the software has been designed completely from scratch to suit GPU parallel computing. The software can perform AES cryptographic operations at different levels of parallelism with different memory configurations (usage of constant memory), support asynchronous input/output operations with page-locked memory, and supports different modes of operation (ECB and CTR encryption and decryption, and CBC and CFB decryption) with multiple streams and devices. The *Boost* library [25] is used for logs [25, Ch. 1], parsing of program options [25, Ch. 21], threads and synchronization [25, Ch. 38], and unitary tests [25, Part IV]. In addition, *Valgrind* [26] has been used to detect and fix memory leaks. The software is prepared to be compiled in Linux as a shared library or as a binary tool for its use in the command line. The software can also be compiled in development, debug or release modes: the development mode prints a more detailed and extended trace of the execution of the program than the debug mode and it is useful to find hidden bugs. Naturally, these prints and asserts can significantly reduce the performance of the program and are bypassed in the release mode which also enables optimization flags in the *nvcc* compiler.

2.1 Thread scheme

To strive for a high occupancy we first calculate the fixed block size (threads per block) that we will use to launch kernels and fully utilize the GPU hardware threading resources. As it is described in the *Achieved Occupancy* chapter at the *NVIDIA® Nsight^{TM} Application Development Environment for Heterogeneous Platforms User Guide* [16]:

> ”There is a maximum number of warps which can be concurrently active on a Streaming Multiprocessor (SM), as listed in the Programming Guide’s table of compute capabilities. Occupancy is defined as the ratio of active warps on an SM to the maximum number of active warps supported by the SM.”

We can calculate the numbers of threads per block that will permit to create blocks that will fit perfectly in the device and utilize the maximum number of active warps and blocks by dividing the maximum number of resident threads per multiprocessor by the maximum number of resident blocks per multiprocessor [23]. The maximum number of resident threads per multiprocessor is defined as the number of maximum warps per multiprocessor per the warp size.

$$\text{block size} = \frac{\text{max. resident threads per SM}}{\text{max. resident blocks per SM}} = \frac{\text{max. resident warps per SM} \cdot \text{warp size}}{\text{max. resident blocks per SM}}$$

In CUDA, the maximum number threads per multiprocessor can be obtained at run-time with a call to *cudaGetDeviceProperties()*. The maximum number of resident blocks per SM, however, has to be hard-coded to match table 2.1 [14, Appx. G, Table 14]. In the *Paracrypt* implementation, the number of threads per thread blocks to be used for encryption and decryption is calculated when instantiating a new object of type *CUDACipherDevice*.

```
paracrypt::CUDACipherDevice::CUDACipherDevice(int device)
{
    cudaGetDeviceProperties(&(this->devProp), device);

    ...

    int M = this->devProp.major;
    int m = this->devProp.minor;
    if (M <= 2) {
        this->maxBlocksPerSM = 8;
    } else if (M <= 3 && m <= 7) {
        this->maxBlocksPerSM = 16;
    } else {
        this->maxBlocksPerSM = 32;
    }

    ...

    this->nThreadsPerThreadBlock = this->devProp.maxThreadsPerMultiProcessor / this->maxBlocksPerSM;
}
```

Table 2.1: Compute capabilities

	Compute Capability										
Technical Specifications	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2
Maximum number of concurrent kernels	16		4	32				16	128	32	16
Maximum number of threads per block	1024										
Max. resident threads per multiprocessor	1536	2048									
Max. resident warps per multiprocessor	48	64									
Warp size	32										
Max. resident blocks per multiprocessor	8	16				32					

Finally, before the kernel is called we have to calculate the grid size given the number of plaintext blocks (or ciphertexts for decryption) [23] for a particular operation. If we have previously established that the device will use 192 threads per block and we will use an implementation that uses four threads to cipher each input block then, to cipher for example 97 blocks, the grid size would be $3 = \lceil\frac{97}{48}\rceil$ since each thread-block of 192 threads can cipher a maximum of $192/4 = 48$ input-blocks. In this last example, the two first blocks of the grid would compute the first 96 input-blocks and a third block of threads would have to be included to compute the last input-block. We have to be careful and include a conditional check in the CUDA kernels in such a way that we do not do any unwanted operation with threads that do not have any input-data assigned. If a *CUDACipherDevice* object has been already instantiated we can easily obtain the grid size and threads per block and run a kernel as is shown in the following fragment of code:

```
int paracrypt::CudaEcbAES4B::encrypt(const unsigned char in[],
                const unsigned char out[],
                int n_blocks)
{
    ...

    // 4B implementation - 4 threads per block
    int gridSize = this->getDevice()->getGridSize(n_blocks, 4);
    int threadsPerBlock = this->getDevice()->getThreadsPerThreadBlock();

    ...

    // call kernel with correct grid and block size
    __cuda_ecb_aes_4b_encrypt__<<<gridSize,threadsPerBlock>>>(...)

    ...
}
```

2.2 Data transfers & computation

The GPU has its own memory hierarchy and, for this reason, cryptographic data has to be read from the input file in the host, loaded in memory, and then transferred to the GPU before performing any computation. Host-device transfers are costly and should be one of the main optimization concerns. Data read from the input file is stored in a buffer before being transferred to the GPU. By default this buffer would be host-paginated if it is allocated using *malloc()*, but CUDA *cudaHostAlloc()* function can be used to allocate pinned memory (also known as page-locked memory). This type of memory is directly available in RAM and does not have the pagination overhead. As a result, pinned memory permit to achieve higher transfer bandwidths [29].

Ideally, we want to allocate as many pinned memory as RAM is available in order to reduce the penalty associated with per-operation overheads, fitting the entire file in RAM if possible. However, over-allocation of pinned memory can substantially reduce the amount of physical memory stagnating the operative system and other processes and leading to poorer performances. In our implementation, the *Pinned* class (see fig. 2.1) allow IO objects to reserve pinned memory without trying to allocate more RAM than the available in the system or leaving the system without a reasonable amount of RAM. In an OpenCL implementation we should had also verified that we were not trying to allocate more page-locked memory with *mlock()* [28, Sec. 3] (in Linux) than the permitted by *RLIMIT_MEMLOCK* [28, Sec. 2] limit, but in a CUDA implementation the *cudaHostAlloc()* function is not subjected to this limitation. In our code we also take into account the amount of global memory available in the system GPU devices as well as the input file size to avoid allocating more RAM that it is actually needed to reach a minimum number of host-device transfers. Additionally, the cost of calls to cudaHostAlloc() for progressively bigger quantities of memory is increasingly higher meanwhile the doubtful benefits of using larger staging areas are probably lower than the allocation cost itself. For this reason, our application can be configured with a limit for the staging area between the host and the GPUs.

Furthermore, transfers time can be masked by overlapping memory copies and computing [30]. The overlapping can be achieved by launching multiple kernels in parallel, each one would process a fraction of file available at the pinned buffer. The *Launcher* class (see fig. 2.1) search all available GPUs in the system and their number of streams, then creates one *CudaAES* cipher per GPU-stream (the copy constructors permit multiple ciphers in the same GPU device to share the same copy of keys and tables), and finally uses a *BlockIO* object to iteratively read chunks of data to be operated with each subsequent *CudaAES* cipher. The *Launcher* object can then wait for operations to finish in the order those where issued with *checkWait()*, or alternatively, actively scan (busy-wait) all ciphers with *checkWait()* until any of them report to have finished with the encryption or decryption operation.

When a operation has finished, the modified chunk is written to the output file with the *BlockIO* object and, if we have yet not reached the last chunk, another chunk is read with the IO object. The *BlockIO* object makes sure so that the launcher does not have to worry about padding or bytes offset when using the random access functionality (see sec. 2.7): zeros are appended and removed for CTR mode and PKCS#7 is used in CBC, CFB, and ECB modes.

A simple solution for IO operations would be to access to the input file each time the read function is called and wait the operation to complete. However, by doing this, we would have to stop the progress of the application not being able to do any useful operations until the blocking call finishes. This is the reason why it has been decided to implement an asynchronous approach (see fig. 2.2). The asynchronous approach uses threads and synchronization methods (locks) from the *Boost* library: a reader thread is continuously reading chunks from the input file and placing them in a queue so that when the *Launcher* call the read method they can be directly extracted from the queue and there are no interruptions to perform IO communications (unless the queue is empty when the Launcher would have to wait until more chunks are available). The is another writer that perform write operations asynchronously when the *Launcher* places the chunks to be written in a output queue without performing any IO operation himself.

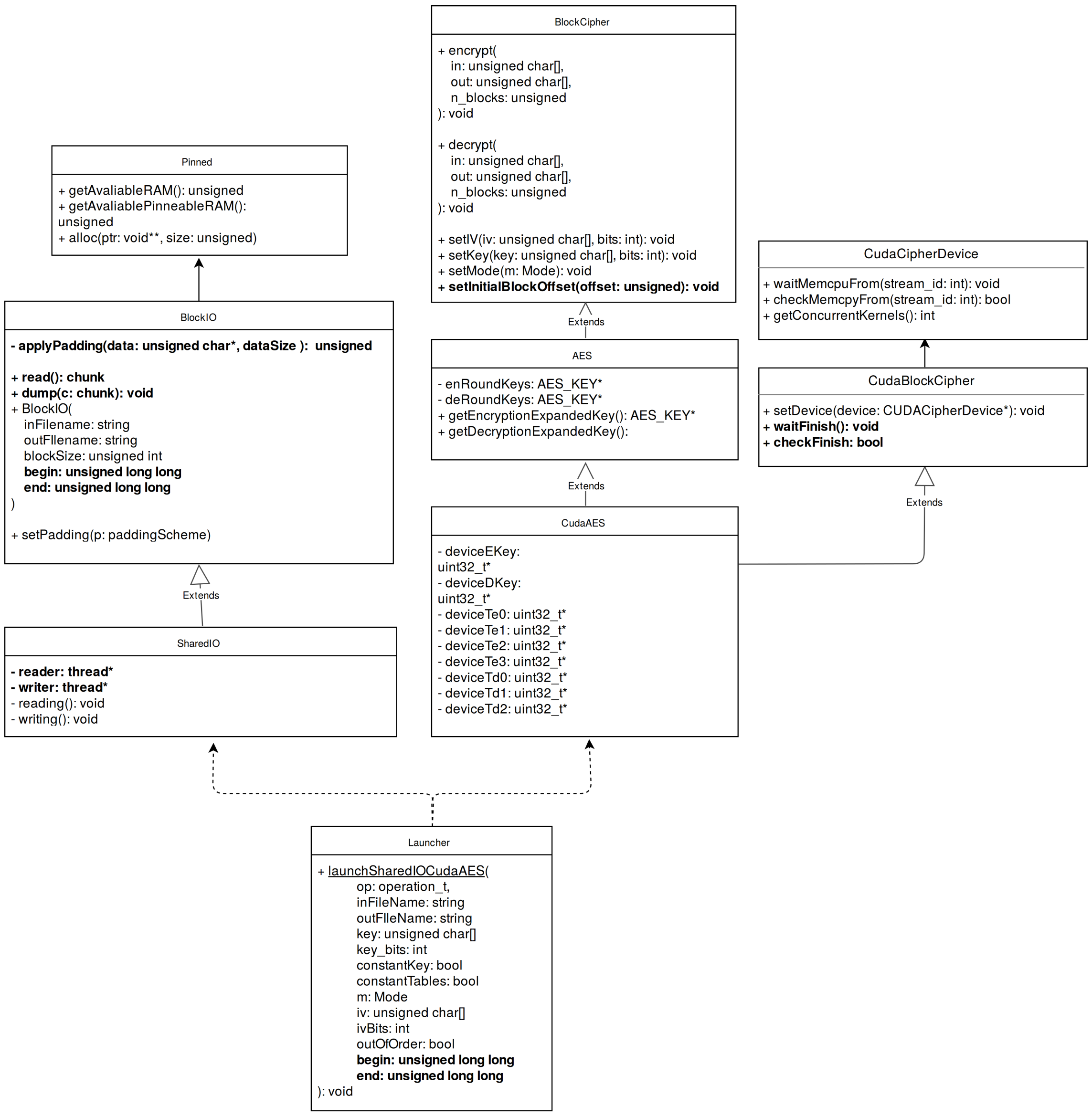
BlockCipher
+ encrypt(in: unsigned char[], out: unsigned char[], n_blocks: unsigned): void
+ decrypt(in: unsigned char[], out: unsigned char[], n_blocks: unsigned): void
+ setIV(iv: unsigned char[], bits: int): void
+ setKey(key: unsigned char[], bits: int): void
+ setMode(m: Mode): void
+ setInitialBlockOffset(offset: unsigned): void
Pinned
+ getAvaliableRAM(): unsigned
+ getAvaliablePinneableRAM(): unsigned
+ alloc(ptr: void**, size: unsigned)
CudaCipherDevice
+ waitMemcpuFrom(stream_id: int): void
+ checkMemcpyFrom(stream_id: int): bool
+ getConcurrentKernels(): int
BlockIO
- applyPadding(data: unsigned char*, dataSize): unsigned
+ read(): chunk
+ dump(c: chunk): void
+ BlockIO(inFilename: string outFIlename: string blockSize: unsigned int begin: unsigned long long end: unsigned long long)
+ setPadding(p: paddingScheme)
Extends
AES
- enRoundKeys: AES_KEY*
- deRoundKeys: AES_KEY*
+ getEncryptionExpandedKey(): AES_KEY*
+ getDecryptionExpandedKey():
CudaBlockCipher
+ setDevice(device: CUDACipherDevice*): void
+ waitFinish(): void
+ checkFinish: bool
Extends
Extends
CudaAES
- deviceEKey: uint32_t*
- deviceDKey: uint32_t*
- deviceTe0: uint32_t*
- deviceTe1: uint32_t*
- deviceTe2: uint32_t*
- deviceTe3: uint32_t*
- deviceTd0: uint32_t*
- deviceTd1: uint32_t*
- deviceTd2: uint32_t*
Extends
SharedIO
- reader: thread*
- writer: thread*
- reading(): void
- writing(): void
Launcher
+ launchSharedIOCudaAES(op: operation_t, inFileName: string outFIleName: string key: unsigned char[] key_bits: int constantKey: bool constantTables: bool m: Mode iv: unsigned char[] ivBits: int outOfOrder: bool begin: unsigned long long end: unsigned long long): void

Figure 2.1: Paracrypt UML class diagram

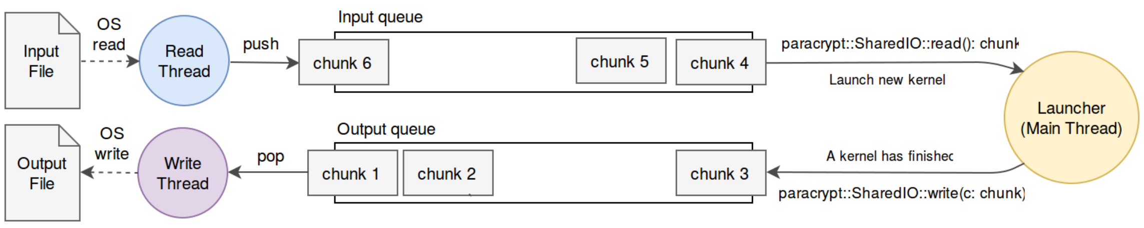

Figure 2.2: Asynchronous input/output

2.3 High-level AES description

AES uses a symmetric key of 128, 192, or 256 bits for both encryption and decryption. Although the same input key is used for encryption and decryption, the key is expanded into new several round keys of the same length, 128 bits, using a key expansion algorithm that will generate different round keys from encryption and decryption. There is one 128 bit round key for each round plus one additional round key for an initial round key step. The number of rounds is determined by the input key size, 10 transformation rounds for a 128 bit key, 12 transformation rounds for a 192 bit size key, and 14 transformation rounds for a 256 bit key. Once the rounds keys has been computed we can use the following template for our implementation (the transformation applied in each round is explained in the following section 2.4).

```
// expand key and call the encryption/decryption function
void aes_encrypt(uint128_t blocks[], int nBlocks, uint128_t encryptionRoundKeys[], int keyBits) {
   for(each block in blocks) {
      uint128_t state = block;

      // Initial round key - combine key with block of the round
      state = state ^ encryptionRoundKeys[0];

      for(int i = 1; i <= 10; i++) {
         TRANSFORMATION_ROUND(state,encryptionRoundKeys[i]);
      }

      // Additional rounds for 192 or 256 bit keys
      if(keyBits >= 192) {
         TRANSFORMATION_ROUND(state,encryptionRoundKeys[11]);
         if(keyBits == 256) {
            TRANSFORMATION_ROUND(state,encryptionRoundKeys[12]);
         }
      }

      // Final round
      FINAL_ROUND();

      // Update result in-place
      block = state;
   }
}
```

2.4 T-tables AES implementation

The cipher state, a 4x4 bytes matrix, is updated in each round by columns: S_0, S_1, S_2, S_3

$$\begin{array}{cccc} S_0 & S_1 & S_2 & S_3 \\ \left(\begin{array}{c} a_{00} \\ a_{10} \\ a_{20} \\ a_{30} \end{array}\right. & \begin{array}{c} a_{01} \\ a_{11} \\ a_{21} \\ a_{31} \end{array} & \begin{array}{c} a_{02} \\ a_{12} \\ a_{22} \\ a_{32} \end{array} & \left.\begin{array}{c} a_{03} \\ a_{13} \\ a_{23} \\ a_{33} \end{array}\right) \end{array}$$

As the Rijndael authors explain in the AES proposal, with 32 bits processors all the steps applied in a round transformation can be performed with four lookup tables [22, Sec. 5.2]: T_0, T_1, T_2, T_3. This allows a much faster implementation that only needs four table lookups and four XORs per column. The T-tables implementation, for example, avoids to directly compute the *MixColumns* step, which performs a Galois multiplication with a constant polynomial and that, by itself, would require far more operations than a whole T-table round.

$$\begin{aligned}
S'_0 &= T_0[a_{00}] \oplus T_1[a_{11}] \oplus T_2[a_{22}] \oplus T_3[a_{33}] \oplus k_0 \\
S'_1 &= T_0[a_{01}] \oplus T_1[a_{12}] \oplus T_2[a_{23}] \oplus T_3[a_{30}] \oplus k_1 \\
S'_2 &= T_0[a_{02}] \oplus T_1[a_{13}] \oplus T_2[a_{20}] \oplus T_3[a_{31}] \oplus k_2 \\
S'_3 &= T_0[a_{03}] \oplus T_1[a_{10}] \oplus T_2[a_{21}] \oplus T_3[a_{32}] \oplus k_3
\end{aligned}$$

To obtain the updated state (S'_0, S'_1, S'_2, S'_3), each 256-entry table is accessed with one of the state bytes (a_{ij}) and the 32 bit results are XORed. Finally, each column is XORed with one of the four 32 bit words in which the 128 bit round key is divided.

CPU cryptography libraries (normally) provide support for both little-endian and big-endian machines. For this reason, all the data manipulated during encryption or decryption is defined in machine order and stored in 32 bit unsigned integers: This includes the round keys and the T-tables. The input data is a string however, a sequence of bytes in big-endian order. As a result, OpenSSL for instance, needs to pre-process each input block with *GETU32*, a macro that convert four bytes from big-endian order to machine order. Finally, the results are converted back to big-endian order using *PUTU32* over the four words of each AES block. On the contrary, on a CUDA implementation we can safely bypass this conversion and its overhead because we know beforehand that NVIDIA GPUs use a little-endian byte order [14, Ch. 4] - at the beginning of the *CUDA C Programming Guide* chapter 4 is stated:

> "The NVIDIA GPU architecture uses a little-endian representation."

Paracrypt implementations copy the key rounds, the T-tables, and the blocks data to the GPU device in a big-endian order without doing any conversion to the input/output data; loading big-endian data into the GPU little-endian registers is not a problem because we only use XOR logic operations in the GPU code - we do not execute multiplications, divisions, or any other type of numerical operation with integers. The inverse cipher operations used for decryption can be implemented with another set of tables in the same way the previously explained tables are used for encryption. The tables have been obtained from OpenSSL source code [24] (*aes-core.c*) and adapted for our purposes.

Considering that the tables and round keys will not suffer any modifications during the execution of encryption/decryption kernels it reasonable to determine that constant memory is a suitable storage for them. Constant memory is a read-only memory accessed through constant cache. Constant memory is intended to be broadcast [31] to all threads in a warp reducing the required memory bandwidth when all threads in a warp access the same value [32] [14, Sec. 5.3.2]. Since the round keys are accessed at the same time by all threads within a warp it is safe to think we can obtain a boost in performance by using constant memory for them. Contrariwise, threads access the T-Tables in different locations and the usage of constant memory for them could be inefficient.

2.5 Key schedule

AES key schedule algorithm is heavily sequential and would not benefit from a GPU parallel implementation. Therefore, we have re-utilized OpenSSL modules for the generation of the round keys. The source code for the *AES_set_encrypt_key()* and *AES_set_decrypt_key()* functions that generate the round keys can be found at the *aes_core.c* [24] file in OpenSSL GitHub repository. The resultant key rounds values are stored in machine order so they have to be converted to big-endian order with *endian.h* standard header and the *htobe32()* function before copying them to the GPU.

2.6 Parallelism

The AES block cipher can be implemented with different levels of parallelism. On one hand, at block-level parallelism each thread in the grid is responsible for the encryption/decryption of an individual block. Each thread only need to have access to the state of its own block. In consequence, the state can remain stored in registers throughout the execution and there is no need for synchronization between threads, a huge advantage over further parallelism where a block is processed by more than one thread. In the latter later case, each column (4B parallelism - 4 threads per block) or each two columns (2B parallelism - 2 threads per block) of the state could be processed by a thread at the expense of using synchronization in conjunction with shared memory. Finally, a last implementation uses 16 threads per block (16B parallelism): each thread access to the T-tables and the round key at a byte level to compute an individual byte (a'_{ij}) of each new state.

The 16B implementation is expected to yield far superior results than the others when the input data is long enough to ensure that all GPU threads are being used. On the other hand, 8B, 4B, and 1B implementations offer progressively a higher occupancy with less data and would accordingly offer better results with smaller files. The 16B implementation is simpler, easier to understand, and more suitable for our purposes. In any case, all parallel versions have been implemented in CUDA so we can further explain their differences and experimentally analyze their performances.

2.6.1 16B parallelism

In this implementation each thread matches a block and, as a result, each 128-bit block of data can be indexed with the the CUDA thread number. Each block consist of four 32-bit words so we multiply the block index per four to point to the first of the 32 bit words. Then, the block words one, two, three, and four can be accessed with $d[p]$, $d[p+1]$, $d[p+2]$, $d[p+3]$, respectively.

```
__global__ void __cuda_ecb_aes_16b_encrypt__(
      int n,      // Number of blocks we have to encrypt
      uint32_t* d, // pointer to (4*n) 32-bit words of data
      uint32_t* k, // round keys
      int key_bits,
      uint32_t* T0,
      uint32_t* T1,
      uint32_t* T2,
      uint32_t* T3
   )

   // data block index
   int bi = ((blockIdx.x * blockDim.x) + threadIdx.x);
   int p = bi*4;

   ...

   // save results
   d[p] = s0;
   d[p+1] = s1;
   d[p+2] = s2;
   d[p+3] = s3;
}
```

The columns cannot be updated in-place because, due to the *ShiftRows* operation, the original columns are needed to compute each of the four new columns. For this reason we use eight 32-bit integers instead of four to store the state. We interchangeably use $s0$, $s1$, $s2$, $s3$ and $t0$, $t1$, $t2$, $t3$ to store the results of each new round and calculate the next one. The code used to update the first column of the state in each round can be found below:

```
   uint32_t s0,s1,s2,s3,t0,t1,t2,t3;

   ...

   // We cannot overwrite s0 because
   //  we need s0 to calculate the other
   //  three of columns of the state!
   t0 =
      T0[(s0      ) & 0xff] ^ // T0[a00] ^
      T1[(s1 >> 8) & 0xff] ^ // T1[a11] ^
      T2[(s2 >> 16) & 0xff] ^ // T2[a22] ^
      T3[(s3 >> 24)      ] ^ // T3[a33] ^
      k[round_number*4+0];
   t1 = ...
   t2 = ...
   t3 = ...
```

If we want to access each byte of the state using pointers instead of using logic operators we first obtain a byte pointer ($s0p$, $s1p$, $s2p$, $s3p$, ...) to the first element of each state column and then we can access, for example, to the first, second, third, and four byte of the first state column with $s0p[0]$, $s0p[1]$, $s0p[2]$, $s0p[3]$, ..., respectively. Below is the code to execute two consecutive AES rounds using these pointers:

```
uint8_t* s0p = (uint8_t*) &s0;
uint8_t* s1p = (uint8_t*) &s1;
uint8_t* s2p = (uint8_t*) &s2;
uint8_t* s3p = (uint8_t*) &s3;
uint8_t* t0p = (uint8_t*) &t0;
uint8_t* t1p = (uint8_t*) &t1;
uint8_t* t2p = (uint8_t*) &t2;
uint8_t* t3p = (uint8_t*) &t3;

// initial round
...

// round 1
t0 = T0[s0p[0]] ^ T1[s1p[1]] ^ T2[s2p[2]] ^ T3[s3p[3]] ^ k[4];
t1 = T0[s1p[0]] ^ T1[s2p[1]] ^ T2[s3p[2]] ^ T3[s0p[3]] ^ k[5];
t2 = T0[s2p[0]] ^ T1[s3p[1]] ^ T2[s0p[2]] ^ T3[s1p[3]] ^ k[6];
t3 = T0[s3p[0]] ^ T1[s0p[1]] ^ T2[s1p[2]] ^ T3[s2p[3]] ^ k[7];

// round 2
s0 = T0[t0p[0]] ^ T1[t1p[1]] ^ T2[t2p[2]] ^ T3[t3p[3]] ^ k[8];
s1 = T0[t1p[0]] ^ T1[t2p[1]] ^ T2[t3p[2]] ^ T3[t0p[3]] ^ k[9];
s2 = T0[t2p[0]] ^ T1[t3p[1]] ^ T2[t0p[2]] ^ T3[t1p[3]] ^ k[10];
s3 = T0[t3p[0]] ^ T1[t0p[1]] ^ T2[t1p[2]] ^ T3[t2p[3]] ^ k[11];
```

The OpenSSL implementation uses logic operators to access individual bytes in the state. For example, the second byte (out of four) in the 32 bit word (*uint32_t*) *s0* can be access with a *shift* and an *AND* operation: "*(s0* $\gg$ 8*) & 0xff*". In the CPU these two basic operations can be less costly than using a load instruction with a pointer and a byte index/offset. In the GPU, however, even though integer operations are natively supported in hardware they have a small throughput compared to 32-bit floating point operations (table 2.2) [14, Sec. 5.4.1] for which the GPU prepared for. In compute capability 3.0, for instance, although 32-bit bitwise AND operations have a throughput of 160 operations per clock cycle per multiprocessor, integer shift operations have an inferior throughput of 32 compared with the throughput of 192 for 32-bit floting point *add* and *multiply* operations. Consequently, it is reasonable to think that, in the GPU, indexed access to the bytes of a 32-bit word can have similar, if not better, performance than the access using a shift and logic operators. Both versions have been implemented, and can be chosen at run-time, for 16B, 8B, and 4B parallelism.

Table 2.2: Arithmetic instructions

	Compute Capability								
Arithmetic instruction & throughput	2.0	2.1	3.0, 3.2	3.5, 3.7	5.0, 5.2	5.3	6.0	6.1	6.2
32-bit floating-point add, multiply, multiply-add	32	48	192	192	128	128	64	128	128
32-bit integer shift	16	16	32	64	64	64	32	64	64
32-bit bitwise AND, OR, XOR	32	48	160	160	128	128	64	128	128

2.6.2 8B and 4B parallelism

In this version each pair of threads (8B parallelism), or each four threads (4B parallelism), share the same state. We declare the state as external shared memory and allocate enough memory for the execution before the kernel is launched. Below is the code for two rounds of a 4B version, note how *_syncthreads()* is used before the beginning of each round and conditional brackets are used to split the code corresponding to each one of the threads that work with the same block.

```
extern __shared__ uint32_t state[];
// initialize state variables (s0p, s1p, s2p, s3p, ...)
// to point to shared memory

// ti = ... // thread id: 0, 1, 2, or 3

// initial round
...

// round 1
__syncthreads();
if(valid_thread && ti == 0) {
    t0[sti] = T0[s0p[0]] ^ T1[s1p[1]] ^ T2[s2p[2]] ^ T3[s3p[3]] ^ k[4];
}
else if(valid_thread && ti == 1) {
    t1[sti] = T0[s1p[0]] ^ T1[s2p[1]] ^ T2[s3p[2]] ^ T3[s0p[3]] ^ k[5];
}
else if(valid_thread && ti == 2) {
    t2[sti] = T0[s2p[0]] ^ T1[s3p[1]] ^ T2[s0p[2]] ^ T3[s1p[3]] ^ k[6];
}
else if(valid_thread && ti == 3) {
    t3[sti] = T0[s3p[0]] ^ T1[s0p[1]] ^ T2[s1p[2]] ^ T3[s2p[3]] ^ k[7];
}

// round 2
__syncthreads();
if(valid_thread && ti == 0) {
    s0[sti] = T0[t0p[0]] ^ T1[t1p[1]] ^ T2[t2p[2]] ^ T3[t3p[3]] ^ k[8];
}
else if(valid_thread && ti == 1) {
    s1[sti] = T0[t1p[0]] ^ T1[t2p[1]] ^ T2[t3p[2]] ^ T3[t0p[3]] ^ k[9];
}
if(valid_thread && ti == 2) {
    s2[sti] = T0[t2p[0]] ^ T1[t3p[1]] ^ T2[t0p[2]] ^ T3[t1p[3]] ^ k[10];
}
if(valid_thread && ti == 3) {
    s3[sti] = T0[t3p[0]] ^ T1[t0p[1]] ^ T2[t1p[2]] ^ T3[t2p[3]] ^ k[11];
}
```

2.6.3 1B parallelism

This version is similar to a 4B/8B implementation that access each byte of the state with a byte pointer. This implementation only operates with bytes. Only 8 bits of the 32 bits registers are used for computation, a waste of potential which is expected to give poor results compared to the previous implementations. In this case, we not only access the state with byte pointers $s0p[0]$, $s0p[1]$, $s0p[2]$, $s0p[3]$, ... but also access to the tables T_n and key k with a byte pointer (*uint8_t**).

```
__global__ void __cuda_aes_1b_encrypt__(
        ...
        uint8_t* k, // i-th entry bytes from k [i*4+0] to k [i*4+15]
        uint8_t* T0, // i-th entry bytes from T0[i*4+0] to T0[i*4+15]
        uint8_t* T1, // i-th entry bytes from T1[i*4+0] to T1[i*4+15]
        uint8_t* T2, // i-th entry bytes from T2[i*4+0] to T2[i*4+15]
        uint8_t* T3  // i-th entry bytes from T3[i*4+0] to T3[i*4+15]
){
   ...
   /* S0 = ... */
   if(valid_thread && ti == 0) { // compute the first byte out of four of s0
      s0p[0] = T0[s0p[0]*4+0] ^ T1[s1p[1]*4+0] ^ T2[s2p[2]*4+0] ^ T3[s3p[3]*4+0] ^
           k[4*((round_number)*4)+0];
   }
   else if(valid_thread && ti == 1) { // compute the second byte out of four of s0
      s0p[1] = T0[s0p[0]*4+1] ^ T1[s1p[1]*4+1] ^ T2[s2p[2]*4+1] ^ T3[s3p[3]*4+1] ^
           k[4*((round_number)*4)+1];
   }
   else if(valid_thread && ti == 2) { // compute the third byte out of four of s0
      s0p[2] = T0[s0p[0]*4+2] ^ T1[s1p[1]*4+2] ^ T2[s2p[2]*4+2] ^ T3[s3p[3]*4+2] ^
           k[4*((round_number)*4)+2];
   }
   else if(valid_thread && ti == 3) { // compute the fourth byte out of four of s0
      s0p[3] = T0[s0p[0]*4+3] ^ T1[s1p[1]*4+3] ^ T2[s2p[2]*4+3] ^ T3[s3p[3]*4+3] ^
           k[4*((round_number)*4)+3];
   }
   /* S1 = ... */
   /* S2 = ... */
   /* S3 = ... */
```

2.7 Random access

As it is explained in sec. 1.1.2, it is possible to decipher specific regions of the ciphertext without having to decipher it entirely. Paracrypt permit the user to select the region of bytes that he wants to decrypt, if the user is dealing with big files and it is only interested in reading parts of the file contents this functionality can save him considerable amounts of time. The feature is implemented in the *BlockIO* class. The class constructor can take two optional arguments, *begin* and *end* (see fig. 2.1), to indicate the indexes of the first and last bytes the user wants to retrieve. Then, when the read method is called, a the IO object start by returning a chunk whose first block is the entire block (bytes preceding the begin byte are not left out) that contain the begin byte. The same thing happens when the last chunk is returned, the last block that contains the end byte is returned entirely so that the cipher that work with entire blocks can operate successfully. Finally, when write method is called, the class is intelligent enough to only write to the resultant decrypted file the bytes the user has selected, discarding others. Below (fig. 2.3) the reader can find an example where only bytes from 23 to 32 (included) are decrypted from a total of 64 bytes: to obtain these 12 bytes only two (blocks 2 and 3) out of four blocks (from 1 to 4) have to be decrypted.

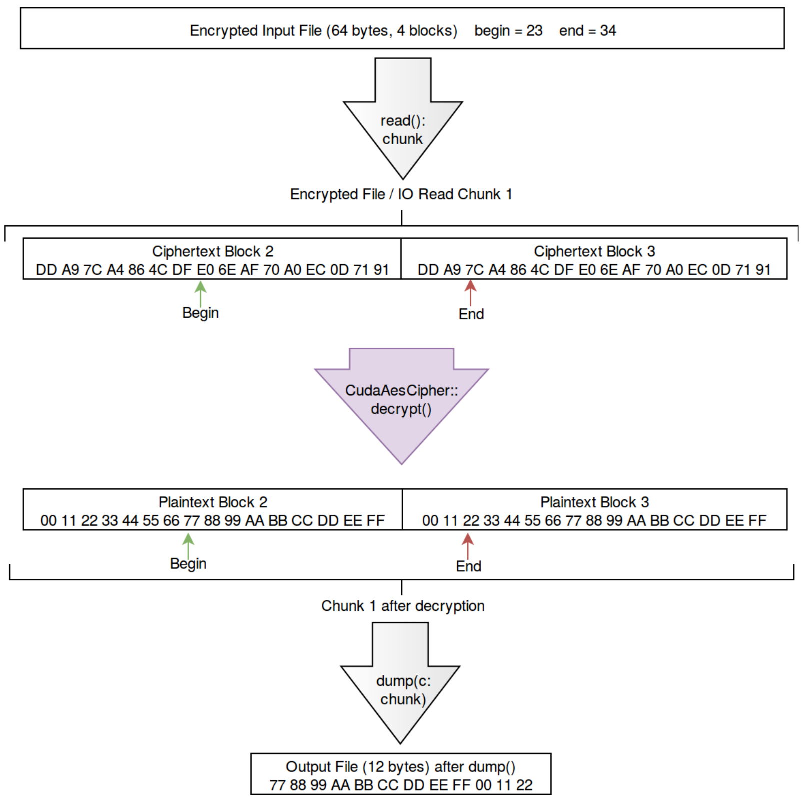
Encrypted Input File (64 bytes, 4 blocks) begin = 23 end = 34
read(): chunk
Encrypted File / IO Read Chunk 1
Ciphertext Block 2
DD A9 7C A4 86 4C DF E0 6E AF 70 A0 EC 0D 71 91
Ciphertext Block 3
DD A9 7C A4 86 4C DF E0 6E AF 70 A0 EC 0D 71 91
Begin
End
CudaAesCipher:: decrypt()
Plaintext Block 2
00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF
Plaintext Block 3
00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF
Begin
End
Chunk 1 after decryption
dump(c: chunk)
Output File (12 bytes) after dump()
77 88 99 AA BB CC DD EE FF 00 11 22

Figure 2.3: Random access example

2.8 Modes of operation

In addition to the basic ECB mode, CBC, CFB, and CTR modes have been implemented. Since CBC and CFB encryption is not parallelizable (sec. 1.1.2), only decryption is supported for these two modes. CBC and CFB decryption has been tested to decipher data previously ciphered with OpenSSL. CTR mode support both encryption and decryption and only need (in the same way CFB only needs) the encryption kernels and encryption tables to work properly (tag. ref.).

2.8.1 CBC

To implement the CBC mode we include an additional last step in the original kernel in which we XOR each block with the previous block or with the initialization vector for the first block as is described in the CBC decryption diagram (fig. 1.4). The problem is that, since data is processed in-place for efficiency purposes, the state i (in local memory) cannot be overwritten until threads that compute the CBC step for the next block $i+1$ have also finished. The *__syncthreads()* CUDA function can only be used to synchronize threads in the same block so that they not overwrite each other data before each one has finished reading the previous block $i-1$. There is no form of synchronization between blocks that does not affect performance (e.g. using heavy synchronization methods such as *cudaDeviceSynchronize()*), for this reason, when we copy the blocks of data to the GPU we also transfer an additional read-only copy of each last cipher-block (these copies are referred as neighbors in the source code) processed in each block of threads. By doing this we provide "untouched" versions of these blocks of data that are assured to not have suffered any modifications by the time they are accessed from another thread of blocks. Figure 2.4 illustrates this concept for a grid of dimension *4x4* and 16B parallelism (one thread per cipher-block).

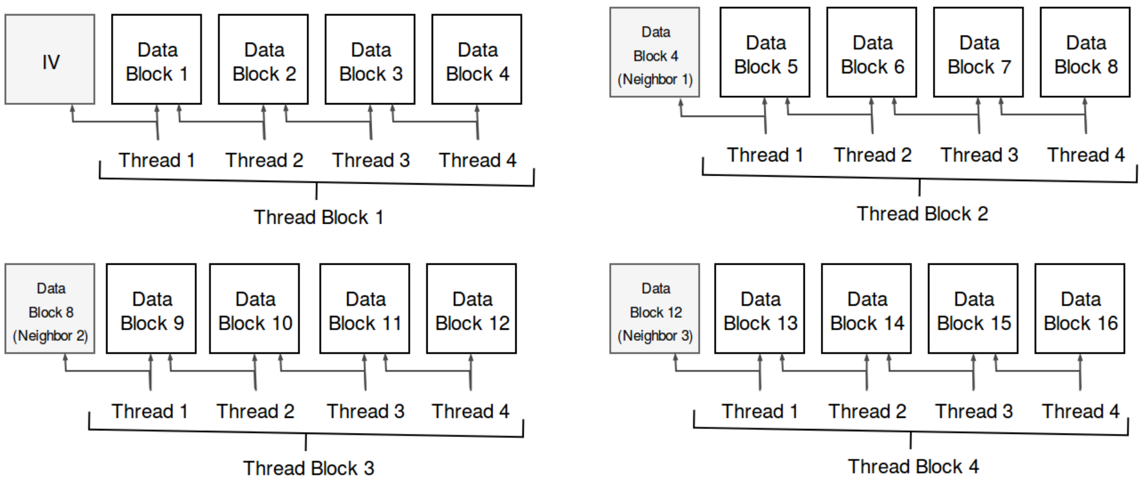

Figure 2.4: Chain/feedback modes and neighbors

The following section of code for the CBC mode follows the previous diagram. The CBC section is included just before results are saved in global memory. The first thread in the grid (in the first thread in the first thread-block: *bi == 0*) uses the initialization vector, the first thread of each subsequent block uses a neighbor, and other threads access to the previous block that is processed by another thread of its same thread-block. Note how we include new *m*, *iv*, and *neigh* parameters to detect which mode of operation is being used and, in case of CBC and CFB, access to the neighbor blocks.

```
__global__ void __cuda_aes_16b_decrypt__(
       const paracrypt::BlockCipher::Mode m,
     unsigned int n,
     uint32_t* d,
     uint32_t* neigh,
     uint32_t* iv,
     uint32_t* k,
     const int key_bits,
     uint32_t* T0,
     uint32_t* T1,
     uint32_t* T2,
     uint32_t* T3,
     uint8_t* T4
   )
{
  uint32_t bi = ((blockIdx.x * blockDim.x) + threadIdx.x);
  ...
  if(m == paracrypt::BlockCipher::CBC) {
    uint32_t c0,c1,c2,c3;
    if(bi == 0) {
    // there is no previous block - use initialization vector
      c0 = iv[0];
      c1 = iv[1];
      c2 = iv[2];
      c3 = iv[3];
    } else {
      // previous block
      if(threadIdx.x == 0) {
        // previous cipher-block is in another
        //  thread-block so we cannot __syncthreads()
        //  to ensure the data we access is not
        //  already overwritten
        int np = (blockIdx.x*4)-4;
        c0 = neigh[np ];
        c1 = neigh[np+1];
        c2 = neigh[np+2];
        c3 = neigh[np+3];
      }
      else {
        c0 = d[p-4];
        c1 = d[p-3];
        c2 = d[p-2];
        c3 = d[p-1];
      }
    }
    s0 ^= c0;
    s1 ^= c1;
    s2 ^= c2;
    s3 ^= c3;

    // sync. before saving result
    //  and overwriting data
    __syncthreads();
  }

  if(bi < n) {
    // save result
    d[p] = s0;
    d[p+1] = s1;
    d[p+2] = s2;
    d[p+3] = s3;
  }
}
```

2.8.2 CFB

The code for the CFB mode is very similar to the CBC section with some slight differences. The CFB method uses the encryption cipher function so in this case the section is located in the *__cuda_aes_encrypt__* kernel. In addition, as seen in fig. 1.6, the encryption input is the previous cipher-text so we have to create a first CFB section for the initial round step where the previous cipher-text block is combined with the first round-key. The IV and neighbors are used in the same way as in the CBC mode. Finally we include an additional XOR with the ciphertext before data is saved to global memory.

```
__global__ void __cuda_aes_16b_encrypt__(
        const paracrypt::BlockCipher::Mode m,
        unsigned int n,
        uint32_t* d,
        uint32_t* neigh,
        uint32_t* iv,
        uint32_t* k,
        const int key_bits,
        uint32_t* T0,
        uint32_t* T1,
        uint32_t* T2,
        uint32_t* T3
      )
{
    uint32_t bi = ((blockIdx.x * blockDim.x) + threadIdx.x); // block index
    ...
    if(m == paracrypt::BlockCipher::CFB) {
        if(bi == 0) {
            s0 = iv[0] ^ k[0];
            s1 = iv[1] ^ k[1];
            s2 = iv[2] ^ k[2];
            s3 = iv[3] ^ k[3];
        }
        else {
            if(threadIdx.x == 0) {
                int np = (blockIdx.x*4)-4;
                s0 = neigh[np  ] ^ k[0];
                s1 = neigh[np+1] ^ k[1];
                s2 = neigh[np+2] ^ k[2];
                s3 = neigh[np+3] ^ k[3];
            }
            else {
                s0 = d[p-4] ^ k[0];
                s1 = d[p-3] ^ k[1];
                s2 = d[p-2] ^ k[2];
                s3 = d[p-1] ^ k[3];
            }
        }
    }
    else {
        // ECB initial round
    }
    ...   // Encryption rounds
    if(m == paracrypt::BlockCipher::CFB){
        s0 ^= d[p  ];
        s1 ^= d[p+1];
        s2 ^= d[p+2];
        s3 ^= d[p+3];
    }
    d[p]   = s0;
    d[p+1] = s1;
    d[p+2] = s2;
    d[p+3] = s3;
}
```

2.8.3 CTR

In the counter mode we add an additional section at the beginning of the encryption cipher function as a substitution for the initial AES round. Rather than combining the first-round expanded key with data, it is combined with a counter. For performance reasons, instead of an incrementing 128-bit counter we use a 32-bit counter defined by the cipher-block index that is XORed with each of the four 32-bit words of the expanded key. For security reasons, the counter its also combined with the initialization vector words. Finally another section at the end of the encryption function has to be included as happens with the CFB mode, we can reuse the CFB code adding a check in the conditional if.

```
__global__ void __cuda_aes_16b_encrypt__(
      const paracrypt::BlockCipher::Mode m,
      unsigned int n,
      uint32_t* d,
      uint32_t* neigh,
      uint32_t* iv,
      uint32_t* k,
      const int key_bits,
      uint32_t* T0,
      uint32_t* T1,
      uint32_t* T2,
      uint32_t* T3
    )
{
   uint32_t bi = ((blockIdx.x * blockDim.x) + threadIdx.x); // block index
   ...
   if(m == paracrypt::BlockCipher::CTR) {
      s0 = bi ^ iv[0] ^ k[0];
      s1 = bi ^ iv[1] ^ k[1];
      s2 = bi ^ iv[2] ^ k[2];
      s3 = bi ^ iv[3] ^ k[3];
   }
   ...
   if(
      m == paracrypt::BlockCipher::CFB ||
      m == paracrypt::BlockCipher::CTR
   ){
      s0 ^= d[p ];
      s1 ^= d[p+1];
      s2 ^= d[p+2];
      s3 ^= d[p+3];
   }
   d[p] = s0;
   d[p+1] = s1;
   d[p+2] = s2;
   d[p+3] = s3;
}
```

We also include offset variable for the counter for data that cannot be processed in a single kernel execution (or for data processed with multiple kernels in parallel), for example, when the input file is bigger than the staging area. The offset variable permit to consequent kernels to begin at the correct offset (the cipher-block index) and not at the counter zero.

```
__global__ void __cuda_aes_16b_encrypt__(
      const paracrypt::BlockCipher::Mode m,
      unsigned int n,
      uint32_t offset, // <------------------
      uint32_t* d,
      uint32_t* neigh,
      uint32_t* iv,
      uint32_t* k,
      const int key_bits,
      uint32_t* T0,
      uint32_t* T1,
      uint32_t* T2,
      uint32_t* T3
    )
{
  uint32_t bi = ((blockIdx.x * blockDim.x) + threadIdx.x); // block index
  ...
  if(m == paracrypt::BlockCipher::CTR) {
    uint32_t global_bi = offset+bi;
    s0 = global_bi ^ iv[0] ^ k[0];
    s1 = global_bi ^ iv[1] ^ k[1];
    s2 = global_bi ^ iv[2] ^ k[2];
    s3 = global_bi ^ iv[3] ^ k[3];
  }
  ...
}
```

2.8.4 Concurrent streams

The data partition in ECB mode is straightforward: chunks of the input file are read, assigned to a GPU stream to be encrypted or decrypted, and then written at the same file offset they had been read. For the CTR mode the *Launcher* has only to set the value of the counter offset to the block offset of each read chunk. However, the procedure for CBC and CFB modes is slightly more complex: for the computation of each i plaintext it is needed to access both actual i and previous $i-1$ ciphertext blocks. A CBC/CFB cipher (fig. 2.5) can be decomposed in multiple ciphers to form a chain where the IV of each subsequent decryption is the last block assigned to his predecessor as is illustrated in fig 2.6. The Launcher class is able to form this chain when it detects CBC or CFB modes are being used.

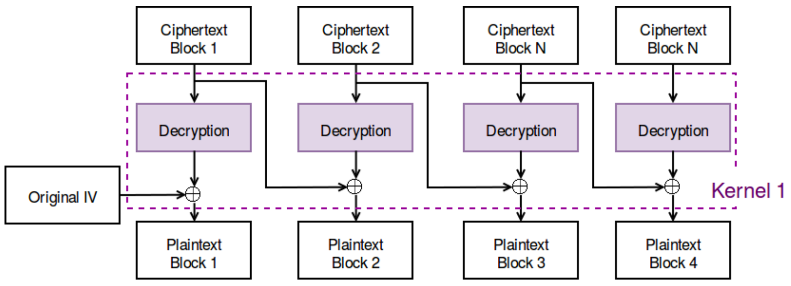

Figure 2.5: Single cipher

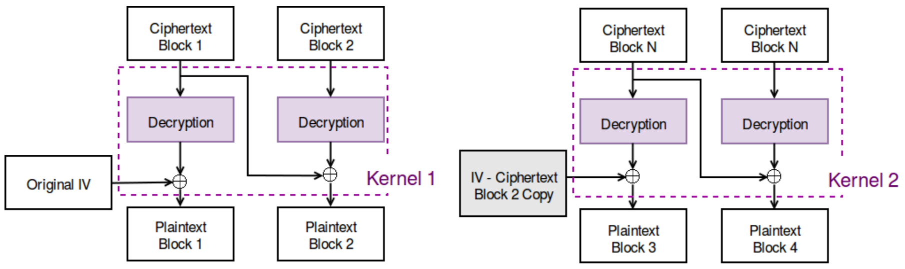

Figure 2.6: Equivalent chain of ciphers

CHAPTER 3

Performances

Performance results have been generated with a *Bash* script that creates files with different sizes up to 4 GB and then executes different configurations for encryption and decryption with Paracrypt and OpenSSL command line utilities. Real time is measured in nanoseconds and averaged for multiple executions in order to obtain more accurate results. Increasingly small files are averaged more times to assure they run for a total time comparable to the time ran by bigger files. The output generated by the Bash script is a set of raw files containing in each row the number of bytes encrypted or decrypted and the real time it took to complete the operation. A Python script is used to read the raw files, plot the results with *matplotlib*, and generate *.csv* comparative tables. Performances correspond to a Linux system with a GeForce GTX 780 and a solid state drive (details available in appendix A).

3.1 Paralellism

It has been surprising to discover that the level of parallelism with which AES has been implemented has not been an important factor in performance contrary to a maximum improvement of 1307% from a 16B implementation against a 1B implementation described in *AES encryption implementation on CUDA GPU and its analysis* by Keisuke Iwai et al [33]. As can be seen in fig. 3.1 the 1B implementation that was expected to work much worse than the others (sec. 2.6) only does so for files smaller than 30 MB. We thought that 1B implementation higher level of occupancy (more threads) could result in better performance with lower files, but this does not seem to be the case as we have obtained results proving the contrary. In the 200-to-250 MB/s performance peak the results seem to indicate, as we initially thought, that the performance decreases as the level of parallelism increases as a consequence of having to use more synchronization. However, for files larger than 125 MB the performance differences are negligible, probably because there are more important critical factors and bottlenecks that hide level of parallelism possible performance differences (most likely the drive act as a bottleneck).

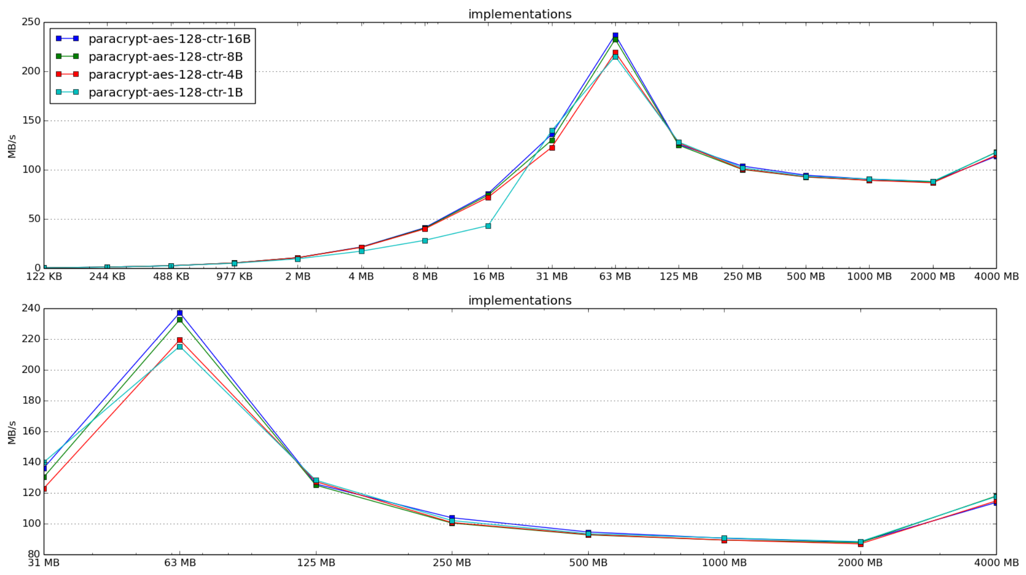

Figure 3.1: Performance comparison between AES levels of paralelism

It has been possible to overlap data transferences and computation in ECB and CTR modes as can be seen in fig. 3.3 where 16 streams are used (our GPU supports 16 kernels in parllel). We have used the NVTX library to mark in the *NVIDIA Visual Profiler* the time consumed in IO operations by the read and write threads (marked in green at fig. 3.4). However as can be seen in figure 3.4, the same level of overlapping can be achieved with only four threads. It is very difficult to overlap data with more than four streams [44, pp. 24], for this reason, our implementation use a limit of four streams per GPU by default: figure shows how 4 streams are enough to obtain a slight improvement in performance, 6%, against 1 stream (no overlap) and doesn't fall too behind the 7% average performance improvement with more streams.

MB/s per file size	4000 MB	2000 MB	1000 MB	500 MB	250 MB	125 MB	63 MB	31 MB	16 MB	average	delta 1-stream
1-stream	110.42	85.27	85.09	88.02	101.91	123.1	205.73	127.58	73.56	111.18	0 %
2-stream	118.02	87.47	90.58	93.26	99.78	122.05	233.12	134.14	75.55	117.10	**5.32 %**
4-stream	116.73	88.24	90.11	91.68	97.37	129.75	235.95	134.22	76.11	117.79	**5.94 %**
8-stream	115.69	87.52	89.41	93.14	104.41	129.82	240.18	135.49	76	119.07	**7.09 %**
unlimited-streams	117.5	87.7	90.73	94.06	106.11	130.43	236.07	135	76.4	119.33	**7.32 %**

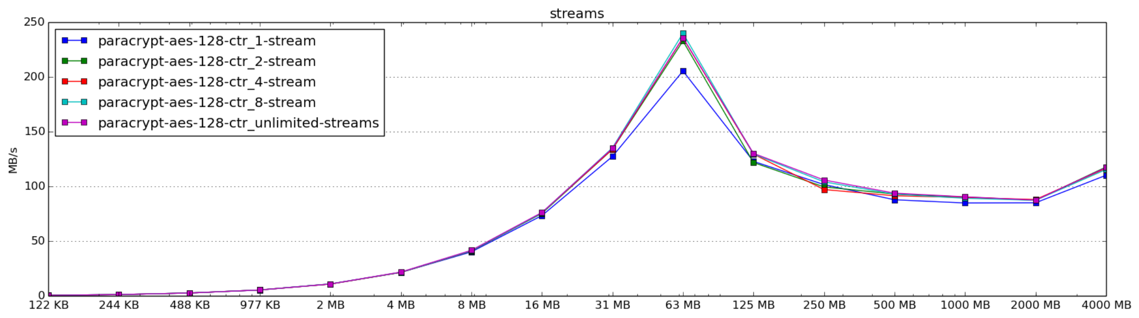

Figure 3.2: Performance using different number of streams

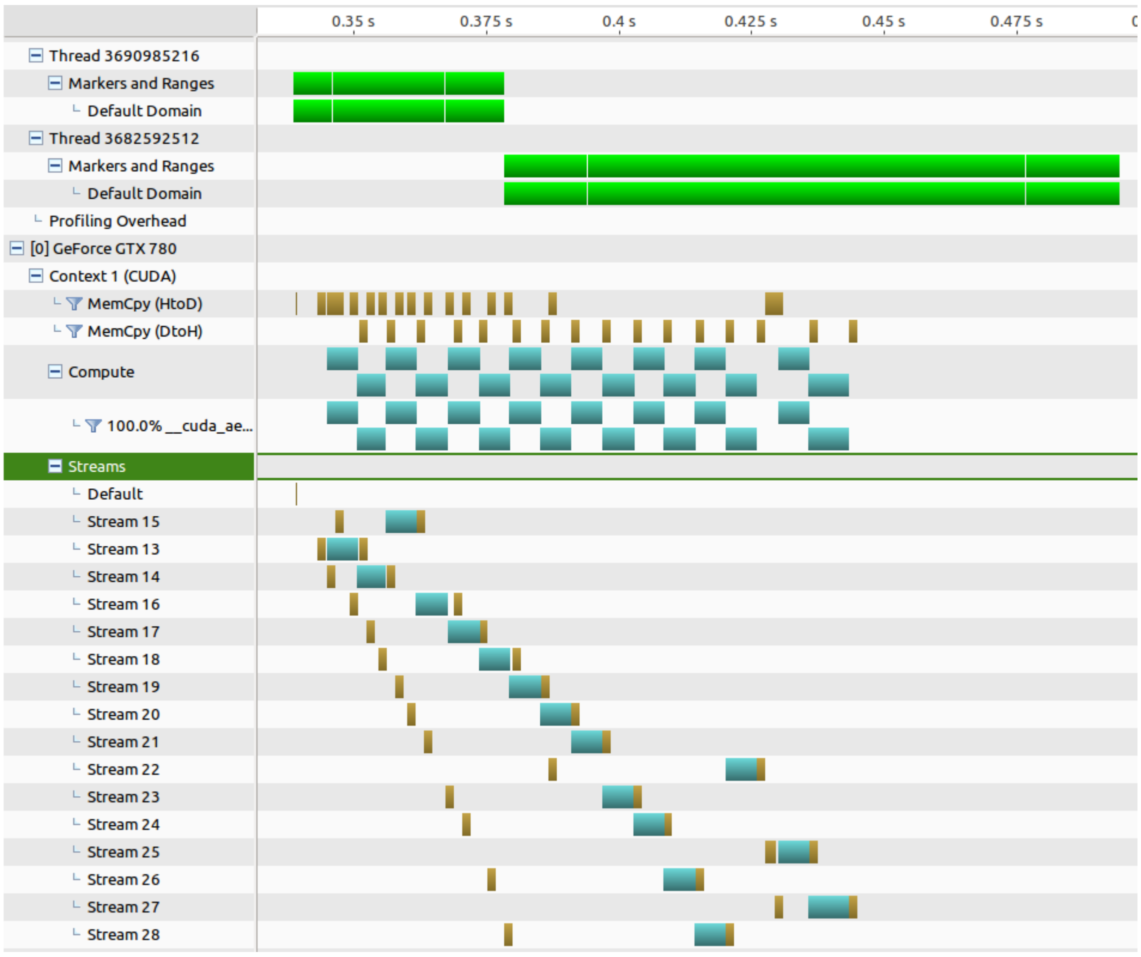

Figure 3.3: Parallelism with sixteen streams

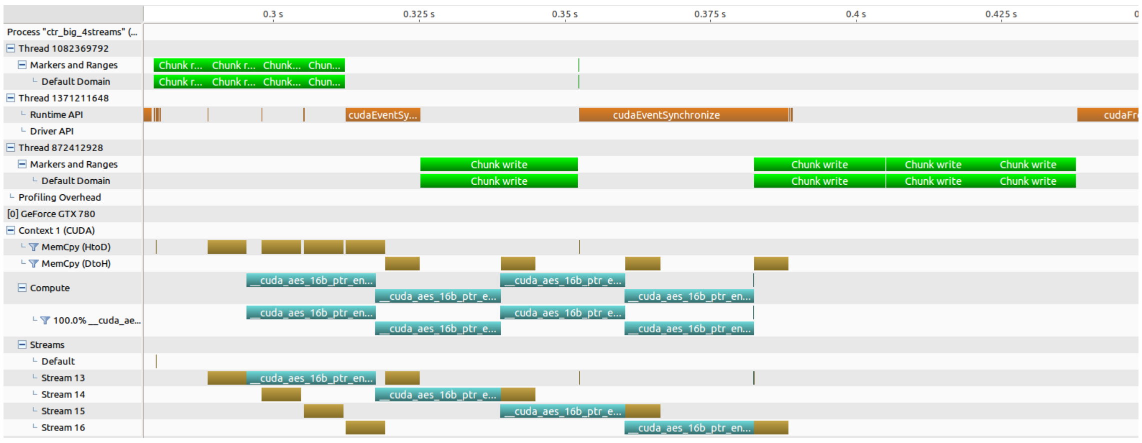

Figure 3.4: Parallelism with four streams

When we operate with small files overlapping does not occur. In fig. 3.5, for example, it is shown how data transferences do not appear at the same time kernels computation appears in the different streams for a 1.6 KB file.

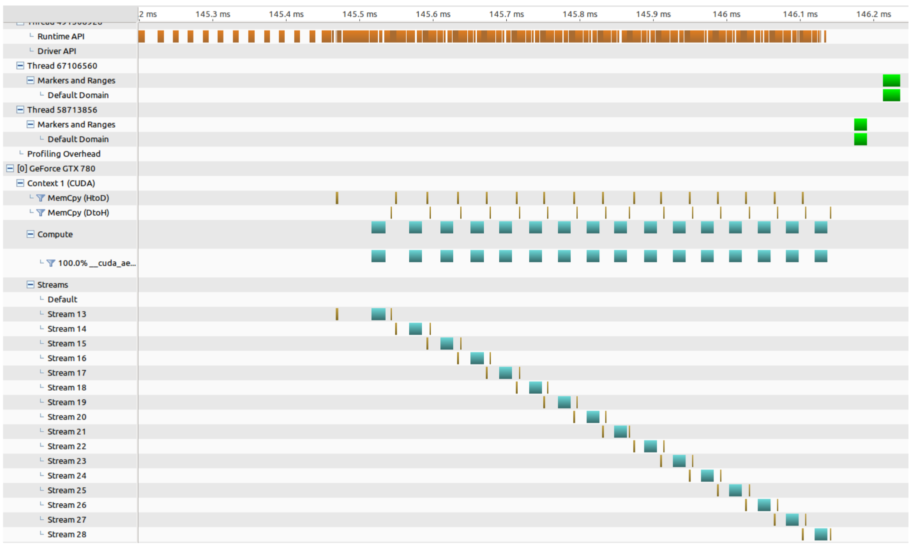

Figure 3.5: Profile of a small file

After profiling CBC and CFB modes (fig. 3.6) we have discovered that the use of separated memory transferences operations for the initialization vectors and neighbors has resulted in the breakage of overlapping. See section 4.1.1 to discover how this problem could be fixed.

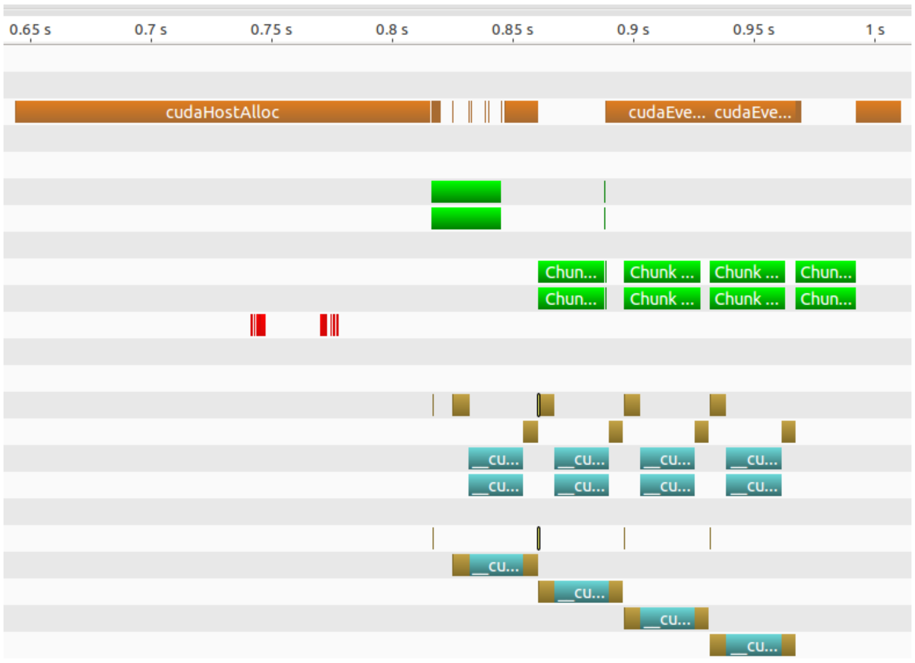

Figure 3.6: CBC breakage of overlapping

Finally, the results show a decrease of performance (fig. 3.13) when operations are finished out-of-order (sec. 2.2). The reason for these results probably stems from the fact that operations have to be issued in a determined order so that the queue scheduler can overlap operations. The issue order is very important and it's alteration could generate regions of the time-line without overlapping. In any case, the performance differences are almost non-existent. The in-order implementation preferred for its simplicity and because does not need to use busy-wait in the host.

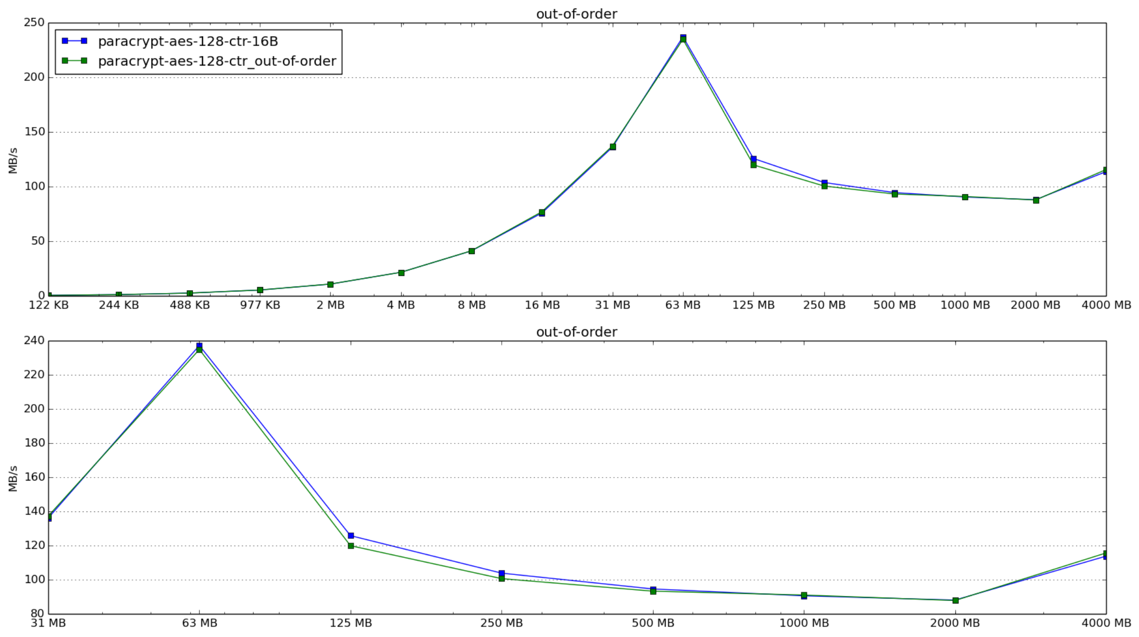

Figure 3.7: Performance of out-of-order operations

3.2 Memory

Changes in memory configurations does not show any drastic effects in performance results neither. We have run the experimental tests to compare the use of constant memory 3.8 for the cipher round key and lookup tables in 16B, 8B, 4B, and 1B implementations and obtained very similar results. Keisuke Iwai et al. results showed a 50% improvement when using shared memory for the T-tables instead of constant memory in a 16B implementation. In an attempt to improve Paracrypt performance results an alternative version of Paracrypt that load the T-tables into shared memory has been developed but, as seen in fig. 3.9, no extraordinary improvements have been perceived (in fact, worse performance have been registered with bigger files) maybe because as Qinjian Li et al. [39] explain in its paper, frequent random access (tables are accessed with bytes from the state that do not follow any order) to the the lookup tables stored in shared memory cause too many bank conflicts.

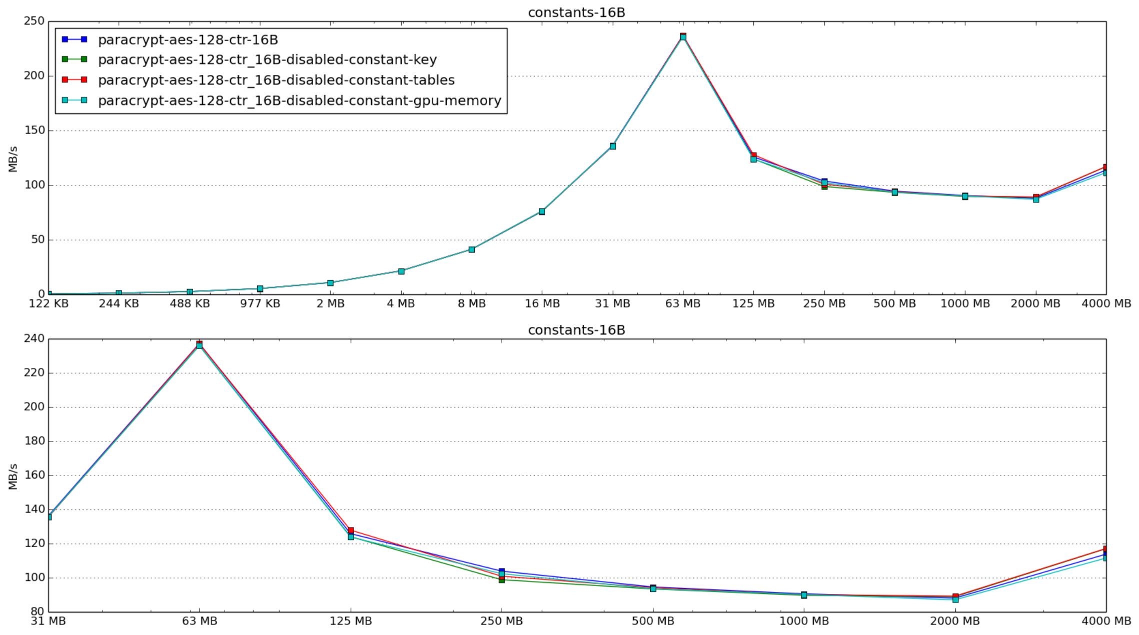

Figure 3.8: Impact of constant memory in performance

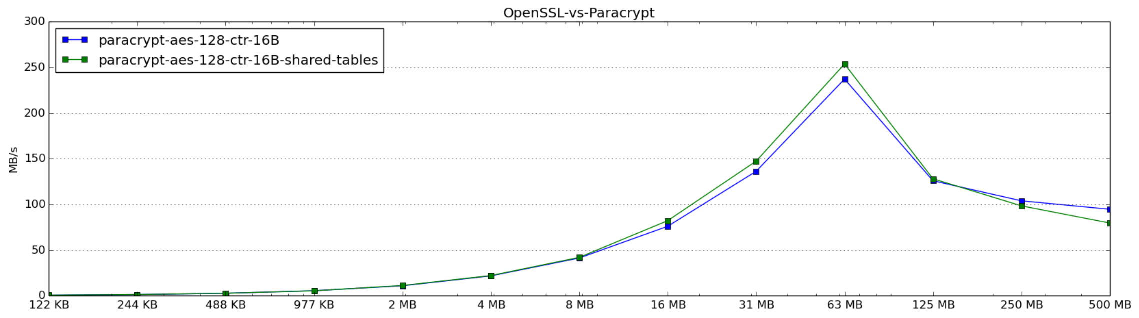

Figure 3.9: Performance of tables stored in shared memory

There are no available CUDA functions to copy data to the device shared memory so data is loaded to shared variables at the beginning of the kernels. Each thread in a block of threads is responsible of loading a word of the tables from global memory to shared memory, then *__syncthreads()* is used to ensure all words are available to all the threads in the grid.

```
__shared__ uint32_t sT0[256]; // <--- tables in shared memory
__shared__ uint32_t sT1[256]; // <---
__shared__ uint32_t sT2[256]; // <---
__shared__ uint32_t sT3[256]; // <---
int n_threads_loading;
if(blockDim.x < 256*4) {
   n_threads_loading = blockDim.x;
} else {
   n_threads_loading = 256*4;
}
if(threadIdx.x < n_threads_loading) {
    int loadsPerThread = (256*4) / n_threads_loading;
    int beginWord = loadsPerThread*threadIdx.x;
    for(int i = 0; i < loadsPerThread; i++) {
       int wi = i+beginWord;
       if(wi < 256) {
          sT0[wi] = T0_32bits[wi];
       } else if(wi < 256*2) {
          sT1[wi%256] = T1[wi%256];
       } else if(wi < 256*3) {
          sT2[wi%256] = T2[wi%256];
       } else if(wi < 256*4) {
          sT3[wi%256] = T3[wi%256];
       }
    }
}
__syncthreads();
```

Qinjian Li et al. indicate in his paper [39] that the use of page-locked memory can result in a poor performance due to the additional time spent in its allocation:

> "although the AES encryption and decryption make significant performance advance, the bandwidth of PCI-E bus and page-lock memory allocation cost are vital limitations. It makes the throughput of encryption and decryption greatly reduced. Even overlapping techniques used, this problem can't be solved satisfactorily."

We have implemented another alternative version where paginated memory is allocated with *malloc()* standard function to analyze the results. Fig. show how the extra cost of pinned memory allocation is surpassed by higher bandwidth performance benefits when processing larger files.

However the profiler shows that the initialization of the runtime API can take almost the same time required for data transferences and computation (fig. 3.12 and fig. 3.10). In the CUDA manual is explained how the runtime API only starts its inilizializiation with the first call to a CUDA runtime function [15, Sec. 4.29], we have attempted to use a dummy *malloc()* at the beginning of the program but have neither obtained performance improvements (fig 3.11): perhaps it would be possible to initialize the API library in a separated host thread.

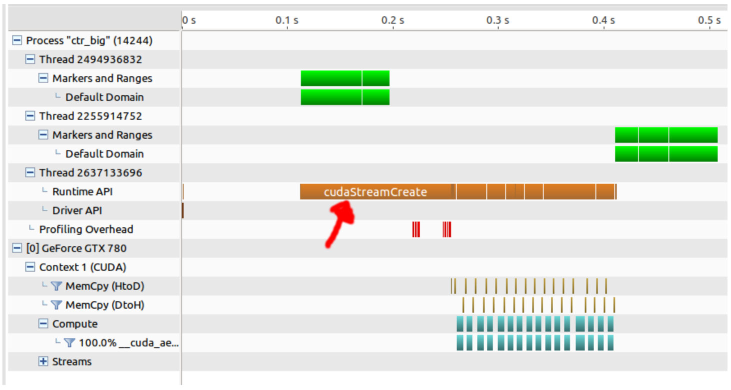

Figure 3.10: Runtime initialization (paginated memory)

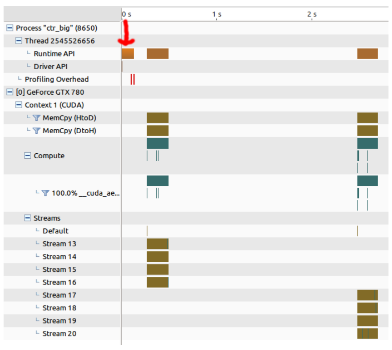

Figure 3.11: Dummy malloc for runtime initialization

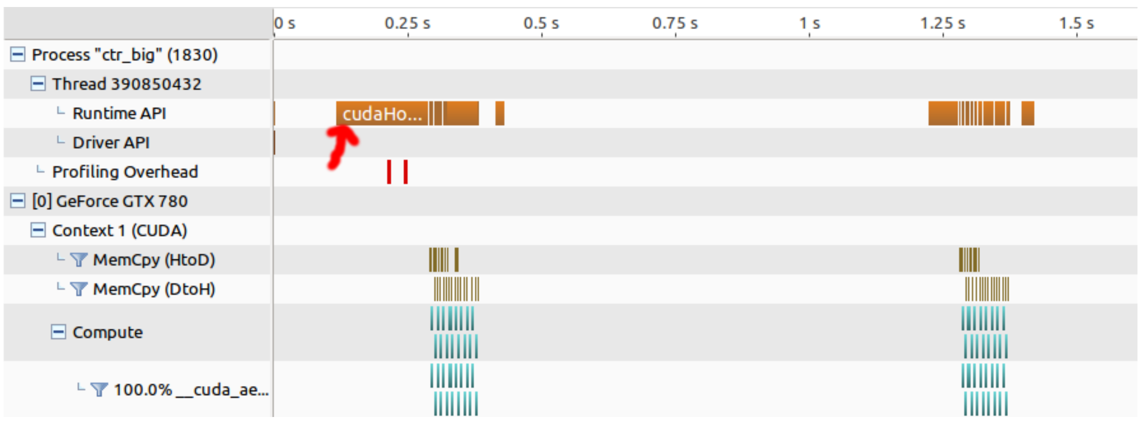

Figure 3.12: Runtime initialization (page locked-memory)

The size of the staging area results also show that allocating both too much (*ulimited-staging*) or too few (*1MB-staging*) memory lead to slower performances. For this reason, our implementations use a staging area of 8 MB by default - this size has shown to produce the higher throughput.

MB/s per file size	4 GB	2 GB	1 GB	500 MB	250 MB	125 MB	63 MB	average
1MB-staging	113.34	85.87	85.67	91.19	97.45	130.37	200.12	114.85
2MB-staging	116.38	86.32	88.89	90.58	94.11	116.39	221.26	116.27
8MB-staging	118.28	88.12	90.65	93.84	104.55	124.91	240.05	**122.91**
32MB-staging	117.28	88.24	90.02	93.68	102.07	125.2	234.84	121.61
unlimited-staging	115.87	86.19	88.83	93.45	103.65	126.79	219.67	119.20

Finally, the use of logic operators to access individual bytes in the state (sec. 2.6.1) haven't shown almost any difference in performance although 16B and 4B implementations perform negligibly better with logic operators.

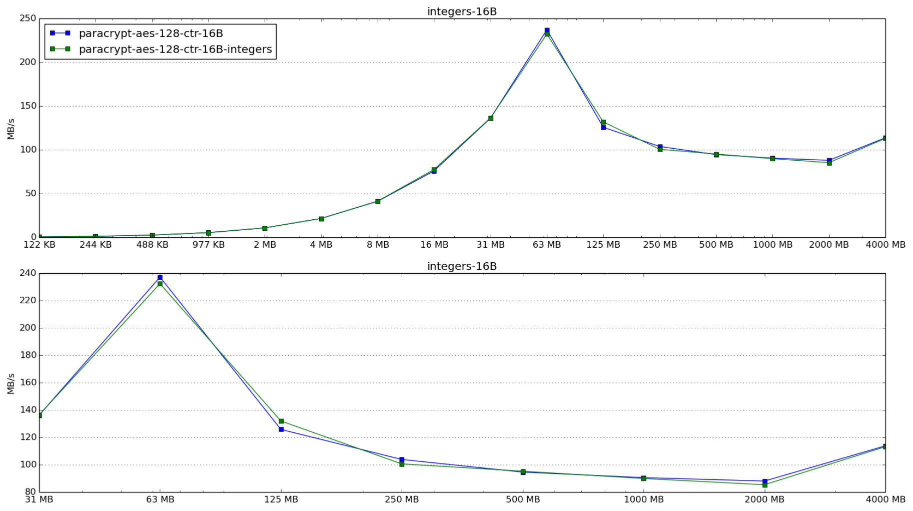

Figure 3.13: Performance of logic operators to access individual bytes in the state (16B)

3.3 Comparison made against OpenSSL

Encryption (16B parallelism without integers, without shared memory for T-Boxes, and with enabled pinned memory) have shown an average speedup 1.55 for files with sizes greater than 63 MB against OpenSSL CPU implementation without AES-NI support. Encryption of a 4GB file has registered the maximum speedup of 2.19.

MB/s per file size	4000 MB	2000 MB	1000 MB	500 MB	250 MB	125 MB	63 MB	average
openssl-aes-128-ctr	52.09	68.01	66.5	61.93	69.68	85.24	148.48	78.84
paracrypt-aes-128-ctr-16B	113.98	88.23	90.74	94.76	104.02	125.99	237.35	122.15
speedup	**118.81 %**	29.73 %	36.45 %	53.01 %	49.28 %	47.80 %	59.85 %	**54.92 %**

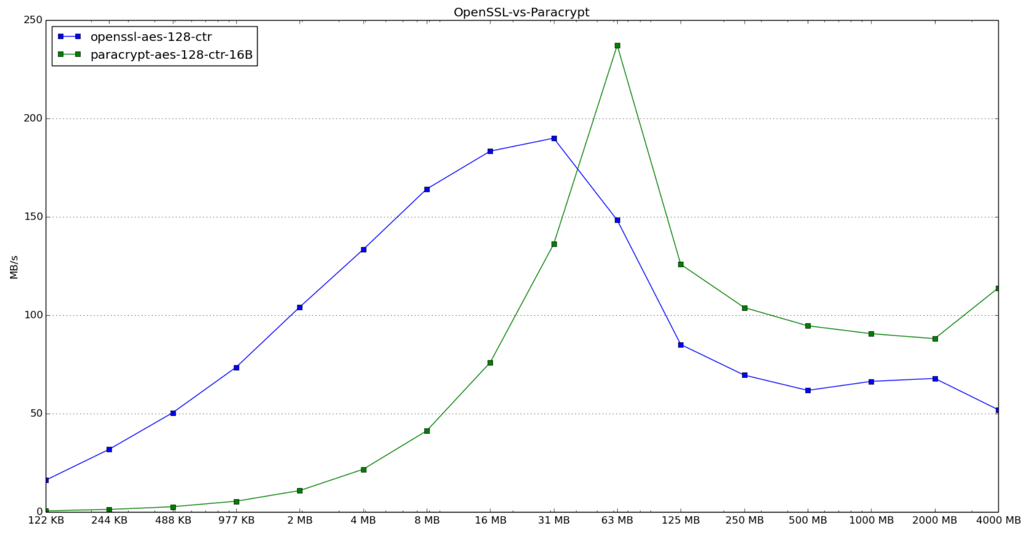

Figure 3.14: OpenSSL vs Pararypt encryption

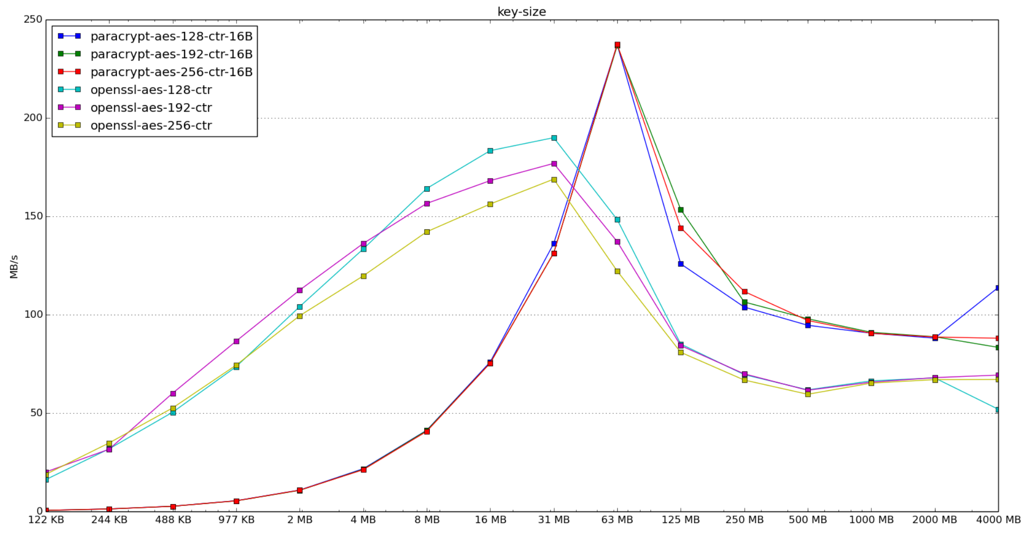

Figure 3.15: OpenSSL and Paracrypt performance with different key sizes

Decryption (16B parallelism without integers, without shared memory for T-Boxes, and with enabled pinned memory) has performed slightly worse than encryption. Decryption modes have shown an average speedup 1.5 for files with sizes greater than 125 MB against OpenSSL CPU implementation without AES-NI support. Decryption of a 4GB file has registered the maximum speedup of 1.87.

MB/s per file size	4000 MB	2000 MB	1000 MB	500 MB	250 MB	125 MB	average
paracrypt-aes-128-cbc	82.32	83.68	84.44	83.71	84.48	80.27	83.15
paracrypt-aes-128-cfb	81.91	84.76	84.43	82.29	81.75	80.89	82.67
openssl-aes-128-cbc-decryption	45.26	67.75	54.02	50.49	54.77	67.38	56.61
openssl-aes-128-cfb-decryption	42.65	62.46	53.11	48.57	54.68	63.46	54.15
Paracrypt average decryption	82.115	84.22	84.435	83	83.115	80.58	82.91
Openssl average decryption	43.955	65.105	53.565	49.53	54.725	65.42	55.38
decryption speedup	**86.81 %**	29.36 %	57.63 %	67.57 %	51.87 %	23.17 %	**49.70 %**

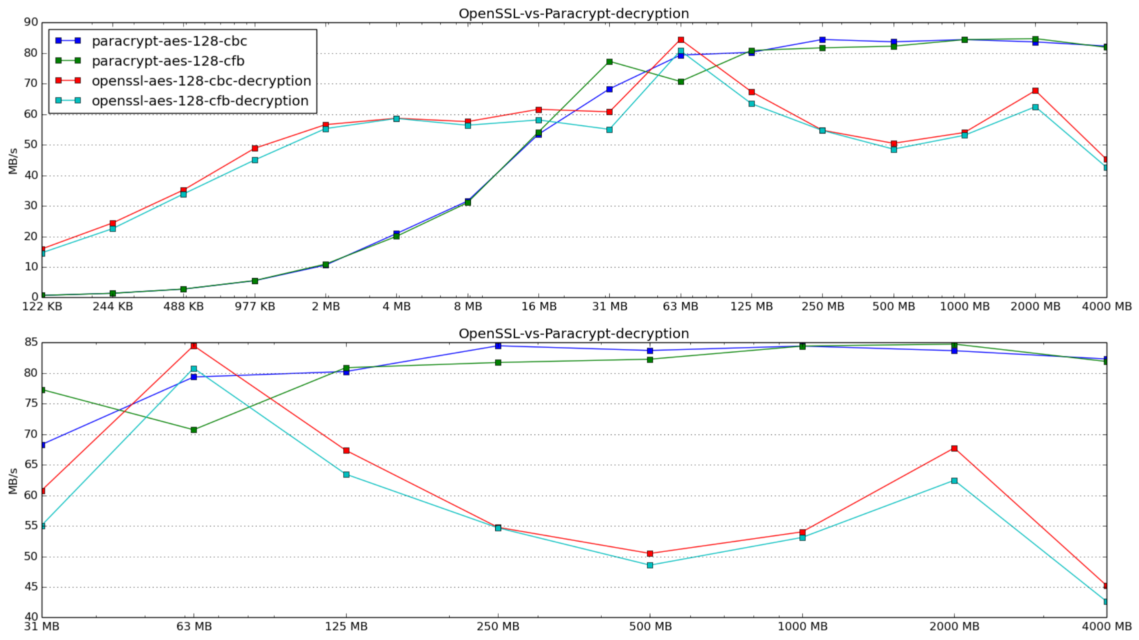

Figure 3.16: OpenSSL vs Pararypt decryption

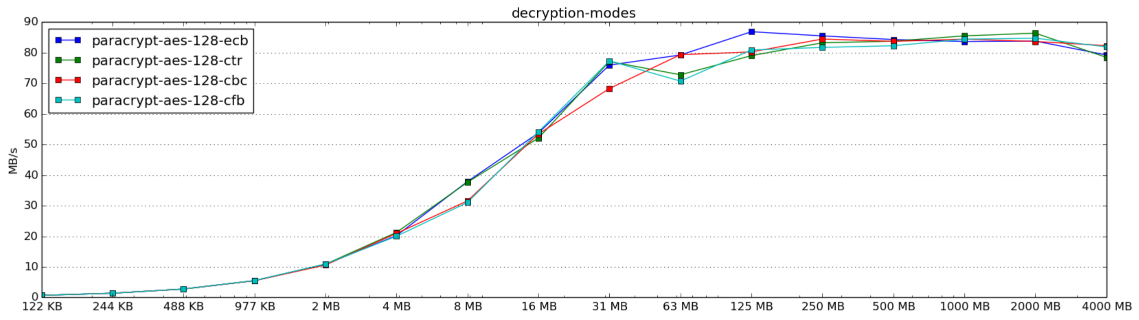

Figure 3.17: Performance of decryption modes

CHAPTER 4

Conclusions

In the past, previous attempts were made to implement the AES algorithm in GPUs: before GPGPU models were adopted, Takeshi from SEGA Corporation implemented AES encryption and decryption in the GPU with the use of shading programming languages and OpenGL extensions that yielded slight improvements, about 1.7 times, against the CPU-based OpenSSL implementation. In contrast, the use of CUDA, a general purpose GPU programming language, has made easier the development process of Paracrypt. Our implementation, however, with a maximum encryption speed of 237.35 MB/s need further testing to determine if the drive with buffered read speed of 208.27 MB/s is acting as a bottleneck or further optimization is needed to be up to the mark with previous CUDA implementations that have reached speeds up to 7.5 GB/s [39]. The aim of the implementation was to provide the high-demands of throughput that some specialized system require. Our intentions were to use the implementation in conjunction with fast quantum key distribution (QKD) [48] but we have to seriously question whether the use of Paracrypt is a viable solution since CPU hardware with AES-NI support can yield comparable speedups of 38% for encryption and 37.5% for decryption [43] and be cheaper to acquire. Further analysis and optimization of our implementation should be made to reach the level of performance expected from recent CUDA implementations (table 4.1).

Table 4.1: Performance of previous works

Reference	Device	Language	Troughput	Year
Cook et al. [34]	GeForce3 Ti200	OpenGL	191 KB/s	2005
T. Yamanouchi [12]	GeForce 8800 GTS	OpenGL	93.5 MB/s	2007
Harrison et al. [35]	Geforce 7900GT	DirectX9	109 MB/s	2007
Manavski [36]	GeForce 8800 GTX	CUDA	1 GB/s	2007
Harrison et al. [40]	NVIDIA G80	CUDA	864 MB/s	2008
Di Biagio et. al. [37]	GeForce 8800 GT	CUDA	1.56 GB/s	2009
Nishikawa et. al. [38]	GeForce GTX 285	CUDA	781 MB/s	2010
Chonglei Mei et. al. [41]	Geforce 9200M	CUDA	800 MB/s	2010
Keisuke Iwai et.al. [33]	Geforce GTX285	CUDA	4.375 GB/s	2010
Nishikawa et. al. [38]	Tesla C2050	CUDA	6.25 GB/s	2011
Qinjian Li et al. [39]	Tesla C2050	CUDA	7.5 GB/s	2012
J. Martin [1]	**GeForce GTX 780**	**CUDA**	**237 MB/s**	**2017**

4.1 Future work

4.1.1 Implementation improvements

The problems encountered during the CBC and CFB profiling (sec. 3.1) can be solved by batching the, currently separated, IV, neighbors, and plaintext/ciphertext transfer operations into a single transfer to the GPU. This eliminates the per-transfer overhead and ensures that the hardware queues can correctly overlap data and transferences so that kernel concurrency does not break. Additionally, the amount of data transferred between the host and the GPU can be minimized by directly performing the neighbor copy operations in GPU code to store them in shared memory instead of generating the neighbours in the host and then copying to the GPU.

Furthermore, as we explained in sec. 3.1, the out-of-order haven't showed satisfactory results. However, the out-of-order functionality still could be improved by using CUDA callback functions to know in which of the streams an operation has finished instead of busy-waiting. In any case, it is not recommended to continue the work in this area because, as we have explained (sec. 3.1), the way and order in which operations are issued into the streams is highly important to obtain satisfactory concurrency results.

Finally, in our implementation only depth-first (fig. 4.1) issue order has been implemented for kernel concurrency. However, another type of order, breadth-first (fig. 4.2), is also possible to be implemented. The issue order can make possible to overlap both host-to-device (HD) and device-to-host (DH) transfers with kernels execution [44, pp. 21-22] depending on the particular GPU architecture the software is being run on. For this reason, it is advisable to permit the user select in the program options the order in which operations are enqueued.

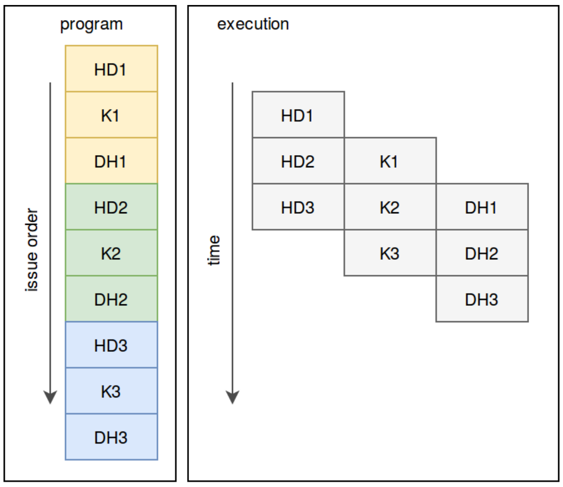

Figure 4.1: Depth-first issue order

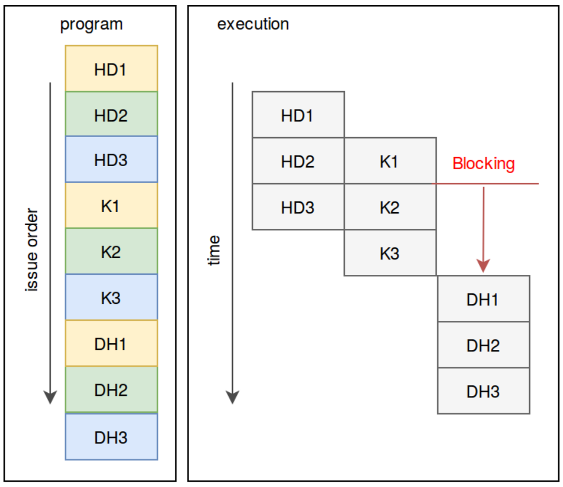

Figure 4.2: Breadth-first issue order

Our implementation also lacks from a key derivation algorithm.

4.1.2 Additional tests

Despite the fact that we have implemented asynchronous IO operations in the belief they would be beneficial for performance (sec. 2.2), additional tests have yet to be performed against a simple blocking IO version to contrast that this functionality is effective. It ought be possible to select the desired type of IO in the command line tool program options. Moreover, our implementation has been developed to support multi-GPU cryptography but hasn't been yet tested in systems with multiple available GPU devices.

Finally, development branches where shared memory and paginable staging memory are implemented should be incorporated in the main development branch to add support in the command line tool for the usage of shared memory for the look-up tables or the usage of paginable memory for the staging area. By doing this, it will be easier to do further performance tests and determine how to improve the current implementation. **The SSD drive of our machine seem to be acting as a bottleneck so it is important to perform more tests in environments that can supply higher rates of data while benchmarking how much PCI-e bandwidth is being used to ascertain if higher speedups against a CPU version are possible with our implementation.**

4.1.3 Other topics

Below is a list of other ways in which our work could be continued:

- An CPU-GPU hibrid implementation that combines both architectures for a higher throughput [45]. The implementation can make use of the OpenSSL library which has support for AES-NI to cipher part of the input with the CPU.
- Add support for GCM Authenticated Encryption (see next page). The galois field multiplication is optimized for CPU and could be reutilized from Crypto++ GCM source code [46].
- An OpenCL implementation of AES.
- Workload distribution and implementation for multi-GPU heterogeneus enviroments that, for example, use both AMD and NVidia GPUs.
- Portability concerns: Make sure only standard headers compatible across diferent OS are being used. Create a build configuration for Windows, favour the use of Boost portable C++ source libraries against OS specific headers and libraries and test the successful compilation of Paracrypt in different systems.
- Extend the support to other ciphers.
- Explore the possibility of brute-force attacks with the GPU in both block and stream ciphers. Even though if the stream cipher keystream generation algorithm is not paralellizable a brute force attack is possible by generating many keystreams and trying to find a meaningful decryption.

Authenticated Encryption

Authenticated Encryption with Associated Data (AEAD), or simply *Authenticated Encryption* (AE), provides not only data confidentiality but also data integrity and authenticity for which the receiver will be able to validate the source of the message as well as confirm that data is not corrupted and has not been modified by a third party. AEAD provide authentication with only one pass, saving code and most likely computation. For this reason, AEAD is a more efficient and secure alternative to generic compositions where authentication is provided separately [20]. AEAD encrypted data is accompanied with an additional message authentication code (MAC) resultant of combining a cipher algorithm and a authentication code algorithm. The *Counter with CBC-MAC* (CCM) scheme, for example, combine the CTR mode with a cipher block chaining message authentication code (CBC-MAC) that is calculated by encrypting the CTR output in CBC mode with a zero initialization vector and keeping the last block output. However, before the *Galois/Counter Mode* (GCM) appeared none of these AEAD algorithms (considering only those free from intellectual property) could be parallelized, not meeting the criteria for a solution capable of sustaining high data rates [21]. The GCM combines the CTR mode with universal hashing (a hash function that guarantees a low number of collisions) over a binary Galois field and can be efficiently implemented in software using table-driven operations. While the CTR mode is parallelizable at a block level, the Galois hash function used for authentication is not. Nonetheless, the Galois authentication code can be computed incrementally and each Galois field multiplication operation can be performed in parallel at a bit level, which permits to keep up with a parallel CTR mode.

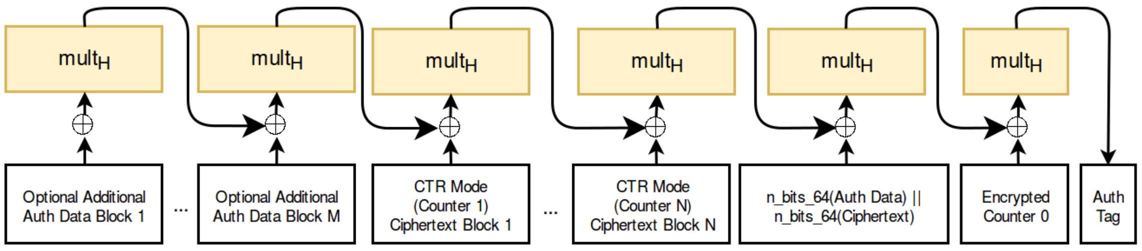
mult_H
mult_H
mult_H
mult_H
mult_H
mult_H
Optional Additional Auth Data Block 1
...
Optional Additional Auth Data Block M
CTR Mode (Counter 1) Ciphertext Block 1
...
CTR Mode (Counter N) Ciphertext Block N
n_bits_64(Auth Data) || n_bits_64(Ciphertext)
Encrypted Counter 0
Auth Tag

Figure 4.3: GCM authentication

The GCM authentication tag is calculated by incrementally XORing Galois field multiplications $mult_H$ of the hash key H with each ciphertext resultant of using the PTR mode starting at the counter sequence 1. The algorithm requires to encrypt two additional blocks with the cipher block: one to calculate the hash key, a string result of encrypting a 128 bit zero string block $H = E(K, 0^{128})$, and another to encrypt the nonce and XOR the result in the final step of the authentication algorithm.

Bibliography

[1] J. Martín. (June 20, 2017). *Paracrypt* [Online]. Available: https://paracrypt.bdevel.org

[2] P. Kuppuswamy and S. Q. Y. Al-Khalidi, "Hybrid Encryption/Decryption Technique Using New Public Key and Symmetric Key Algorithm," *MIS Review*, Vol. 19, No. 2, pp. 1-13, Mar., 2014. DOI: 10.6131/MISR.2014.1902.01

[3] C. De Canniére and B. Preneel. *Trivium Specifications* [Online]. Available: http://www.ecrypt.eu.org/stream/p3ciphers/trivium/trivium_p3.pdf

[4] D. Blazhevski et al., "Modes of Operation of the AES Algorithmm," in *The 10th Conference for Informatics and Information Technology*, Skopje, 2013.

[5] M. Dworkin, *Recommendation for Block Cipher Modes of Operation: Methods and Techniques*, 2001 Ed., Gaithersburg: NIST, 2001. DOI: 10.6028/NIST.SP.800-38A

[6] P. Rogaway, "Evaluation of Some Blockcipher Modes of Operation," University of California, Davis, CA, 2011, p. 30.

[7] R. Oppliger, "Block Ciphers: Modes of Operation," in *Contemporary Cryptography*, 2nd Ed., Norwood: Artech House, 2011, ch. 6, sec. 2.3, p. 175.

[8] *PKCS #7: Cryptographic Message Syntax*, RFC 2315, March 1998.

[9] C. Wang et al., "Low information leakage random padding scheme for block encryption," *Journal of Discrete Mathematical Sciences and Cryptography*, vol. 11, no. 4, pp. 385–391, Jun, 2008. DOI: 10.1080/09720529.2008.10698191

[10] DI Management Services. (30 March, 2016). *Using Padding in Encryption: Using random padding to disguise the size of the message* [Online]. Available: http://www.di-mgt.com.au/cryptopad.html#randomdisguise

[11] M. Dworkin, *Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode*, Gaithersburg: NIST, 2010. DOI: 10.6028/NIST.SP.800-38A-Add

[12] T. Yamanouchi, "AES Encryption and Decryption on the GPU," in *GPU Gems 3*. NVIDIA, ch. 36.

[13] NVIDIA. (January 12, 2017). *CUDA Toolkit Documentation* (v8.0) [Online]. Available: https://docs.nvidia.com/cuda/index.html

[14] NVIDIA. (March 20, 2017). *CUDA C Programming Guide* (v8.0.61) [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide

[15] NVIDIA. (June 2, 2017). *NVIDIA CUDA Runtime API* (v8.0) [Online]. Available: http://docs.nvidia.com/cuda/cuda-runtime-api/index.html#axzz4j0gLolBp

[16] NVIDIA. *NVIDIA® Nsight™ Application Development Environment for Heterogeneous Platforms, Visual Studio Edition 5.2 User Guide: Achieved Occupancy* (Rev. 1.0.170410) [Online]. Available: http://docs.nvidia.com/gameworks/content/developertools/desktop/nsight/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm

[17] N. Wilt, “Integer Support,” in *The CUDA Handbook*. Upper Saddle River, NJ: Adison-Wesley, 2013, ch. 8, sec. 2.

[18] E. Niewiadomska-Szynkiewicza et al., “A Hybrid CPU/GPU Cluster for Encryption and Decryption of Large Amounts of Data,” *Journal of Telecommunications and Information Technology*, no. 3. 2012.

[19] *Advanced Encryption Standard (AES)*, FIPS 197, November 26, 2001. DOI: 10.6028/NIST.FIPS.197

[20] J. Black. *Authenticated Encryption* [Online]. Available: https://www.cs.colorado.edu/~jrblack/papers/ae.pdf

[21] D. A. McGrew and J. Viega. *The Galois/Counter Mode of Operation (GCM)* [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf

[22] J. Daemen and V. Rijmen. *AES Proposal: Rijndael* [Online]. Available: http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf

[23] C Luo. *Side-Channel Power Analysis of a GPU AES Implementation* [Online]. Sec C. Available: http://www1.ece.neu.edu/~saoni/files/Chao_ICCD_2015.pdf

[24] The OpenSSL Project Authors. *OpenSSL 1.1.1-dev* [Online]. 2016. GitHub repository. Available: https://github.com/openssl/openssl/blob/master/crypto/aes/aes_core.c

[25] B. Dawes et al. *Boost c++ libraries* [Online]. Available: http://www.boost.org/

[26] Valgrind™ Developers. *Valgrind Documentation* [Online]. Ch. 8. Available: http://valgrind.org/docs/manual/mc-manual.html

[27] Free Software Foundation. *GNU General Public License* (29 June 2007, v3) [Online]. Available: https://www.gnu.org/licenses/gpl-3.0.en.html

[28] Oracle. *Oracle Solaris Reference Manuals* (v11.2) [Online]. Available: https://docs.oracle.com/cd/E36784_01/

[29] M. Harris. NVIDIA Corporation. *How to Optimize Data Transfers in CUDA C/C++* (4 December 2012) [Online]: Available: `https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/`

[30] M. Harris. NVIDIA Corporation. *How to Overlap Data Transfers in CUDA C/C++* (13 December 2012) [Online]: Available: `https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/`

[31] D. Cyca. Acceleware. *Constant Cache vs. Read-Only Cache - Part 1* (29 August 2013) [Online]: Available: `http://www.acceleware.com/blog/constant-cache-vs-read-only-cache`

[32] J. McKennon. Microway. *GPU Memory Types – Performance Comparison* (6 August 2013) [Online]: Available: `https://www.microway.com/hpc-tech-tips/gpu-memory-types-performance-comparison/`

[33] K. Iwai et al., "AES encryption implementation on CUDA GPU and its analysis," in *First International Conference on Networking and Computing*, Higashi, Hiroshima, Japan, Nov. 2010. DOI 10.1109/IC-NC.2010.49

[34] D. L. Cook et al., "Cryptographics: Secret key cryptography using graphics cards," in *In RSA Conference, Cryptographer's Track (CT-RSA)*, 2005, pp. 334–350.

[35] O. Harrison et al., "Aes encryption implementation and analysis on commodity graphics processing units," in *9th Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2007, pp. 209–226.

[36] S. A. Manavski, "Cuda compatible gpu as an efficient hardware accelerator for aes cryptography," in *IEEE International Conference on Signal Processing and Communication, IC-SPC*, 2007, pp. 65–66.

[37] A. D. Biagio, "Design of a parallel aes for graphics hardware using the cuda framework," *Parallel and Distributed Processing Symposium, International*, pp. 1–8, 2009.

[38] N. Nishikawa et al., "Granularity optimization method for aes encryption implementation on cuda (in Japanese)," in *IEICE technical report. VLSI Design Technologies (VLD2009-69)*, Kanagawa, Japan, Jan. 2010, pp. 107–112

[39] Q. Li et al., "Implementation and Analysis of AES Encryption on GPU," Northwestern Polytechnical University, China, 2012. Available: `http://www.comp.hkbu.edu.hk/~chxw/papers/AES_2012.pdf`

[40] O. Harrison et al., "Practical Symmetric Key Cryptography on Modern Graphics Hardware," Proc. of the *17th conference on Security symposium*. San Jose, CA, 2008, pp. 195-209.

[41] C. Mei et al., "CUDA-based AES parallelization with fine-tuned GPU memory utilization," Proc. of 2010 *IEEE International Symposium on Parallel Distributed Processing Workshops and PhD Forum (IPDPSW)*, IEEE press, 2010, pp.1-7.

[42] N.Nishikawa et al., “High-Performance Sysmmetric Block Ciphers on CUDA,” Proc. of 2011 *Second International Conference on Networking and Computing (ICNC)*, 2011, pp.221-227.

[43] A. Basu et al. *Intel® AES-NI Performance Testing on Linux*/Java* Stack* (1 June 2012) [Online]. Available: https://software.intel.com/en-us/articles/intel-aes-ni-performance-testing-on-linuxjava-stack

[44] S. Rennich, NVIDIA. *CUDA C/C++Streams and Concurrency* [Online]. Available: http://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

[45] Ewa Niewiadomska-Szynkiewicza, Michał Marksa, Jarosław Janturab, and Mikołaj Podbielski, *A Hybrid CPU/GPU Cluster for Encryption and Decryption of Large Amounts of Data*, Warsaw University of Technology, Poland Research and Academic Computer Network (NASK). Disponible en línea: http://yadda.icm.edu.pl/yadda/element/bwmeta1.element.baztech-article-BATA-0017-0004/c/httpwww_itl_waw_plczasopismajtit2012332.pdf.

[46] *Crypto++: Free C++ class library of cryptographic schemes* (v5.6.5) [Online]. Available: https://www.cryptopp.com/docs/ref/gcm_8cpp_source.html

[47] J. Fang et al., “A comprehensive performance comparison of CUDA and OpenCL,” in *International Conference on Parallel Processing (ICPP)*, September 2011, pp.216-225. DOI: 10.1109/ICPP.2011.45

[48] J. Martinez-Mateo et al., “Key Reconciliation for High Performance Quantum Key Distribution,” *Scientific Reports*, vol. 3, no. 1576, April 2013. DOI 10.1038/srep01576

A. Target machine specifications

1. OS: Ubuntu 14.04.4 LTS 64 bits

   kernel: Linux 3.13.0-57-generic (x86_64)
2. CPU: 8x Intel(R) Core(TM) i7 CPU 960 @ 3.20GHz
3. RAM: 18488 MB
4. GPU: GeForce GTX 780

   CUDA Capability 3.5

   CUDA Cores: 2304

   Total Memory: 3072 MB

   Memory Interface: 384-bit

   Bus Type: PCI Express x16 Gen2

   NVIDIA Driver Version: 361.93.02
5. Drive: KINGSTON SV300S37A120G

   Model Family: SandForce Driven SSDs

   User Capacity: 120 GB

   Rotation Rate: Solid State Drive

   Timed buffered disk reads*: 626 MB in 3.01 seconds = 208.27 MB/s
6. Compilers:

   g++ version 4.8.4 (Ubuntu 4.8.4-2ubuntu1 14.04.3)

* from hdparm manual:

> "Buffered disk read displays the speed of reading through the buffer cache to the disk without any prior caching of data. This measurement is an indication of how fast the drive can sustain sequential data reads under Linux, without any filesystem overhead. To ensure accurate measurements, the buffer cache is flushed."