GipherFS: a GPU-accelerated ciphered file system

José Lourenço
Instituto Superior Técnico
Portugal
jose.lourenco@ist.utl.pt

ABSTRACT

Information security is a major concern in computer systems and its importance has increased in recent years. However, cryptographic algorithms are typically computationally demanding and can place a heavy burden on machines' resources, especially those without accelerators, thus limiting the overall performance in systems with several users running concurrent applications. On the other side, Graphics Processor Units (GPUs) have evolved from specialized computer graphics accelerators to semi-ubiquitous general-purpose co-processors [1]. This paper focuses on improving the security and performance of the file system by offloading the computation of the cryptographic algorithms, used in the protection of the file's data, to the GPU, typically underused by the system. For this purpose, a protection layer was added to the *ext4* file system using a virtual file system based on FUSE [2] with built-in confidentiality mechanisms, using the GPU as a cryptographic co-processor. The obtained experimental results suggest that the proposed solution improves the file protection process throughput by 3.5 times, while also offloading the CPU for other tasks.

CCS CONCEPTS

• **Security and privacy** → **Systems security** → **File system security**; **Security and privacy** → **Cryptography** → **Symmetric cryptography and hash functions** → Block and stream ciphers

KEYWORDS

Confidentiality, File System, Cryptography, AES, CUDA, FUSE

1 INTRODUCTION

Security awareness has been growing in computer users and one of the major concerns is the confidentiality of the users' data. The most typical approach to achieve this, and one of the most practical ones is to encrypt the data, particularly that which is stored in the hard drive. The basic principle of encryption is to transform data into an ineligible form to an adversary. This transformation, especially when dealing with large amounts of data, is done by symmetrical encryption algorithms, such as the Advanced Encryption Standard (AES) [3]. These algorithms are based on a secret value called the secret key, which must be known in order to encrypt and decrypt the data. While being a relatively efficient encryption approach, when large amounts of data need to be accessed and protected the cost of this process can be significant, thus additionally overloading the Central Processing Unit (CPU) resources. Even with multi-core CPUs and the introduction of specialized dedicated instructions at the hardware level, the impact of cryptographic primitives on a system's CPU load can still be quite noticeable. This is especially the case when a system's CPU load is already very high and there are little computational resources left available. In these situations, security mechanisms, such as data confidentiality provided by the AES encryption algorithm, tend to be overlooked and considered expendable in favor of the system's functional requirements.

On the other hand, Graphics Processing Units (GPUs) have become pervasive in most computational systems from servers and desktops, up to mobile phones, and even some embedded devices. In recent years the GPU computational power has increased significantly, mainly due to the high requirements of graphical computer applications. However, this computing power is currently not restricted to graphical computation. Thanks to the introduction of general purpose programming environments [4][5], other types of applications can now leverage the GPU compute resources, which has made it a very popular parallel computing platform, especially in the scientific community. Motivated by this fact, the work proposed in this paper particularly focuses on exploiting the computational power of the GPU to enhance the file system with cryptographic protection mechanisms, with the goal of not only providing faster encryption, but also offloading the CPU resources.

In brief, this paper proposes a solution to the problem of building a secure system that provides confidentiality of user file data, by using the AES algorithm, the standard for data encryption, while minimizing the penalty induced to the overall system performance, caused by the additional load on a machine's CPU. The approach proposed in this paper focuses on offloading the computationally heavy operations required by AES from the CPU to the seldom used and computationally powerful GPU. This relocation drastically reduces the impact of the AES cryptographic primitives on the CPU, apart from the required execution control of GPU kernel launches and data transfers. At the same time, it aims at making the most of the usually underused GPU capabilities to support this widely adopted data encryption standard.

Other approaches have been proposed for solving this problem using similar approaches, in diverse settings [6][7]. Our experiments further support the validity of this approach, and results of read and write throughputs on the order of 580 MB/s and 95 MB/s, respectively, are presented.

2 STATE OF THE ART

Using the GPU as a cryptographic accelerator is not a new endeavor. The first efforts to efficiently implement the AES encryption algorithm in the GPU [8] date back to the time when the GPU was still a dedicated graphics accelerator, which could only be programmed using graphical APIs and was very limited by the graphics pipeline. This was neither an easy task nor did it provide great performance benefits. The advent of general-purpose computation on the GPU (GPGPU) [9] made it possible for a wider range of scientists to exploit it as a cryptographic accelerator. Early CUDA based implementations of AES [10] present solutions up to 20 times faster than equivalent versions running on the commodity hardware of the time. More recently, speedups in the order of 50 have been reported [11] achieving bandwidths as high as 60 Gbps on state of the art GPUs.

The GPU has also been used in the context of computer file systems for diverse applications, such as searching through a file tree [12] or ensuring data integrity by accelerating a number of computationally intensive primitives based on hashing [13][14]. Deeper integration of the GPU into the operating system has also been explored in order to provide a further generalization of its execution model [15] and allow GPU programs to leverage typical operating system mechanisms, such as scheduling and memory management [16].

The use of GPU for accelerating file encryption mechanisms has also been proposed before, both for replacing the typical CPU implementation of encryption algorithms found in widely used encryption suites, such as *TrueCrypt*, and for accelerating the cryptographic primitives used by secure file systems. Some of these efforts are based on extending the operating system mechanisms, in order to better support the integration of the GPU as a cryptographic accelerator [16]. On the other hand, some scientific studies are predominantly based on user level constructs to achieve a similar functionality, typically through the use and extension of user level file systems [2].

3 GPU-BASED SECURE FILE SYSTEM FRAMEWORK

Herein, a pragmatic solution is proposed to speedup the ciphering procedures required to ensure protection of the file system using a GPU. The proposed approach is built on top of a fully functional physical file system by incorporating the automatic and transparent encryption and decryption of the accessed files using the GPU as a cryptographic engine. Rather than relying on low-level Operating System (OS) calls, the proposed solution considers the use of FUSE [2], a user interface allowing to create a Virtual File System in user space, which is enriched with CUDA support, thus allowing to integrate specific GPU cryptographic kernels also in the user level. As such, this approach allows to focus on the system's fundamental goal of adding cryptographic security as an additional layer of data processing and leverage the functionality already provided by stable and widely adopted general-purpose file systems. As such, one of the main advantages of the proposed system is achieved, namely the ability for this solution to be deployed by non-privileged users, without the need to install OS kernel level modules. Thus, along with the resulting transparent usage, the proposed solution can also be easily and widely adopted.

3.1 System overview

The proposed solution consists of three processing steps, as illustrated in the following text for the write operation. In the first step, the I/O operations, mapped to the virtual file system, are intercepted by FUSE, in particular the *write()* system call. After this interception step, FUSE calls the GPU encryption kernel and it transfers all necessary data received from the write call itself. To conclude, the GPU encrypted data is transferred back to the host (system main memory) and written into the actual underlying physical file system by performing the *write()* system call to a physically mapped address. The corresponding physical address is defined by the virtual to physical file system mapping. In case of a read operation the similar three-step procedure is followed, with only exception that in the second step the encrypted data is retrieved from the actual physical file system and the GPU kernel is called to decrypt this same data. These steps are illustrated in Figure 1, for both *read()* and *write()* systems calls.

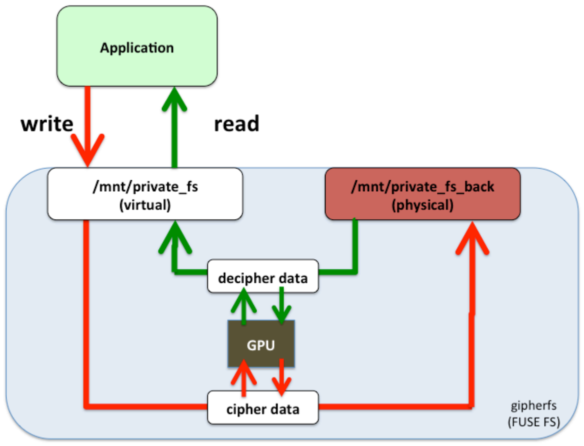


Figure 1: File System's *read* and *write* operations.

Although the interception of FUSE may impose some additional delays, the major limiting factor for the overall system's performance lies in the encryption and decryption of the data. These cryptographic operations can be performed either by the CPU or by a cryptographic co-processor, in this case the GPU. In what concerns the encryption process, the AES algorithm was used, since it is a widely accepted *de facto* standard, for which several implementations are available. In regard to the encryption itself, 128 bit keys and the ECB (Electronic Codebook) cipher mode were selected.

Besides the data transformation functions, the encryption and decryption process requires an additional component for setting up and initializing all the AES parameters, particularly the secret key. It is important to note that only by providing the same secret key between calls will it be possible to gain access to previously created files. An additional detail about the ciphering process is that AES is a block cipher, therefore in ECB mode it can only handle data blocks of 16 bytes. As such, it is necessary to first apply padding to the input data, in this case using *PKCS#7* [17].

3.2 GPU based version

The approach considered to speedup the cryptographic operations relies on the GPU as a cryptographic co-processor, where the encryption and decryption procedures are offloaded, after the data is intercepted by FUSE, as depicted by Figure 2.

As in any typical CUDA-based GPU program, the implementation of AES on the GPU requires the three following steps:

1. Transfer data from the host (CPU) to the GPU
2. Process the data on the GPU
3. Transfer resulting data back from the GPU to the host

The data transfer steps are an obvious source of execution overheads, nevertheless required in order to take advantage of the GPU computational power. The trade-off between this additional cost and the performance benefits obtained from implementing AES on the GPU is one of the main aspects of a successful GPU implementation, as evaluated in the next section in order to determine the optimal amount of data to be processed in each GPU kernel invocation. The GPU AES code used in the proposed system is open-source and obtained from [18], which resembles the implementation described in [11].

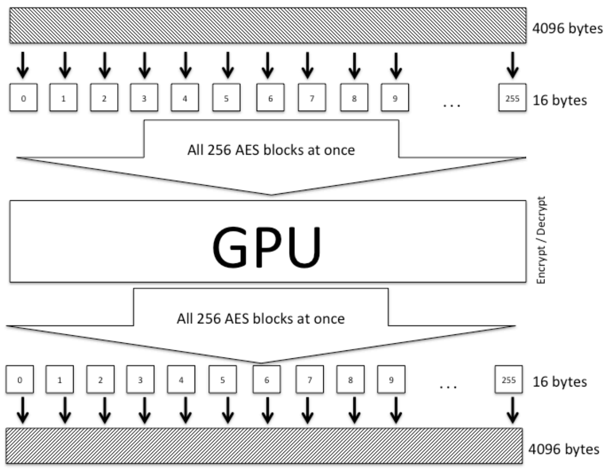


Figure 2: Processing 4096 bytes in the GPU

Due to its block nature, AES must handle data in a fixed-size, aligned data blocks of 16 bytes each. In order to simplify the logic required to correctly handle these AES blocks data is processed in multiples of 16 bytes.

To properly evaluate the performance gains that can be achieved with the proposed GPU-based approach, a CPU-based version was also developed, where the CPU is used for the cryptographic processing. In this version, designated as *cipherfs*, the GPU is not used in our system for cryptographic operations. *cipherfs* is identical to the GPU version (herein designated as *gipherfs)*, but implemented using only the CPU.

In order to optimize the use of the CPU processing resources, the implementation of the AES cryptographic primitives was extracted from the *wolfSSL* [19] cryptographic library implemented in the C programming language. This implementation takes advantage of AES-NI (Advanced Encryption Standard New Instructions), the x86 instruction set architecture extension AES acceleration [20], where the data blocks are sequentially processed in dedicated AES instructions.

3.2.1 Maximizing payload length: FUSE big-writes mode

The operations making up the AES encryption algorithm can be categorized as compute-intensive as they consist of the application of several rounds of data transformation functions to the same piece of data. However, the overall performance of the AES encryption algorithm will only be able to benefit from the relocation of its computation from the CPU to the GPU if it is able to compensate for the penalty of transferring the input and output data to and from the GPU. This can only be achieved by maximizing the amount of data processed by each GPU invocation.

The amount of data processed by *gipherfs* in each GPU invocation is determined by the length of FUSE's *read* and *write* operations payload. The payload ultimately depends of how the client application uses the file system API, but FUSE also plays a major role in it by modifying the original request and handling it in smaller data chunks in the case of the *write* operation. By default, FUSE handles the payload of incoming write operations in data chunks of at most 4 KB. This maps to a single CUDA thread block composed of 256 threads, and is not sufficient to compensate for the memory transfer overhead.

FUSE provides an alternative mode, which makes it possible to overcome this limitation and increase the maximum payload of *write* operations. The mode is called *big-writes* and increases the maximum *write* operation payload by a factor of 16, from 4 KB to 64 KB. Section 4.2 shows that this amount of data is already sufficient to compensate the overheads incurred by the memory transfers, greatly boosting the performance of *gipherfs*.

3.2.2 Improving device memory management

Some of the most expensive operations in any GPU program implementation are the ones related to memory management and transfer. Memory transfers to and from the GPU are perhaps the most notable and introduce a great deal of overhead. These can actually completely render the approach of using the GPU for accelerating compute intensive tasks ineffective by incurring an excessive amount of overhead over doing the computation directly on the CPU. Even though recent GPU architectures already support overlapping of memory transfers and kernel execution, it is still not possible to avoid them completely.

There are however some operations whose cost can be minimized, or even completely eliminated, by employing more efficient memory management mechanisms. This is the case of device memory allocation. A more detailed description of a typical memory transfer operation between the host and the GPU consists of the following steps, assuming data is already available at the host:

1. Allocate device memory
2. Copy data from the host to the device
3. Copy data from the device to the host
4. Free device memory

Steps two and three cannot be avoided at all, since memory must be transferred to the device for processing and back to the host for collecting results. Depending on the problem at hand however, steps one and four can be avoided by reusing previously allocated memory on the device. However, this mechanism can only be applied to systems where the required amount of allocated memory on the device can be estimated with a high degree of confidence. Otherwise the program will either fail due to lack of memory or the reallocation of device memory will render the mechanism ineffective.

This is the case of *gipherfs* where the maximum amount of required allocated memory on the device is limited by the maximum amount of data FUSE tries to read or write, i.e. decipher or cipher, simultaneously on the GPU. These can be determined by analyzing how FUSE works. In respect to writes, and even though *gipherfs* uses FUSE in multithreaded mode, only a single write operation is handled by FUSE at any single time. This is due to a lock on the *fuse* kernel module, which serializes write operations. As a result the maximum amount of memory required by a write operation will be the maximum payload handled by a single *write* operation, which is 64 KB in FUSE *big-writes* mode plus 16 bytes of potential padding, generated whenever write operations extend the previous file size. On the other hand, it is possible to determine that at most four read operations can execute concurrently, each capable of handling a maximum payload of 128 KB. Padding need not be considered in this case, since it is not generated by read operations.

gipherfs implements an efficient memory allocation mechanism by pre-allocating and reusing a pool of device memory blocks, sized according the previously mentioned observations. When the system starts, 64 128 KB blocks are pre-allocated on the device. Even though, at most 5 concurrent operations may be executing on the device according to the observations, blocks are over allocated in order to prevent starvation in the case the system actually tries to allocate more memory on the device. Every time there is a need to allocate memory on the device, a pointer to one of the available device memory blocks is obtained from the pool, instead of performing the actual memory allocation. Conversely, after kernel execution, instead of freeing the block from device memory, the no longer required pointer to the device memory is returned to the pool. In the case more than 64 blocks are required at the same time, which is not expected, the thread requiring the block will wait until one is available. The performance improvements provided by this mechanism are demonstrated in Section 4.3, where a comparison is made between the system with and without this mechanism in place.

4 EXPERIMENTAL RESULTS

In order to evaluate the impact of the proposed solution and possible tradeoffs, this section presents the obtained experimental results for the proposed GPU-based systems, as well as two other system variants. In particular, the experimental results were obtained for: 1) the virtual file system without added protection, in order to evaluate the impact of introducing the cryptographic protection layer, designated as *plainfs*; 2) the proposed systems, but deploying the security layer using only the CPU, designated as *cipherfs*; 3) and finally the complete proposed system with the protection layer supported by the GPU, designated as *gipherfs*. The effectiveness of the optimizations enabling an efficient integration of the GPU computing mechanisms are analyzed in detail by considering different modes of *gipherfs* with and without them.

This section provides evidence that the introduction of GPU-accelerated cryptographic mechanisms into the operation of a Virtual File System without security features only reduces its performance by a half in the worst case, and that relocating the computation required by this kind of mechanism from the CPU to the GPU can boost its performance by a factor of as much as 3.5 times.

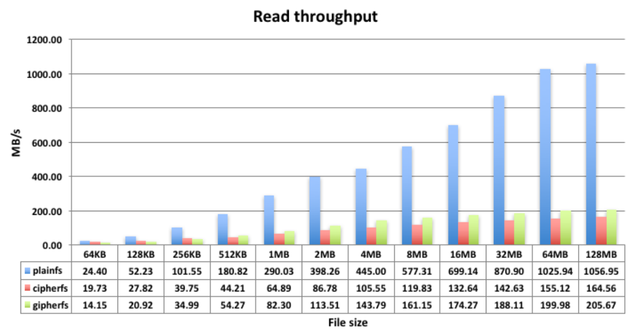
All experimental results were obtained in a computing platform consisting of an Intel Core i7-5960X CPU (3 GHz) with 32GB of main memory (DRAM) and an NVIDIA GeForce GTX 980 GPU, powered by NVIDIA's Maxwell GPU computing architecture.

4.1 Direct integration of GPU AES acceleration

The evaluation of the system starts by measuring the performance of a straightforward implementation of *gipherfs*, which consists of the system's first original implementation and lacks the optimizations described in Section 3.

Figures 3 and 4 present the bandwidth, in MB/s, obtained when copying a file into and out of a directory mounted on the three different Virtual File Systems: *plainfs*, *cipherfs* and *gipherfs*, the latter consisting of a straightforward implementation without any optimizations.

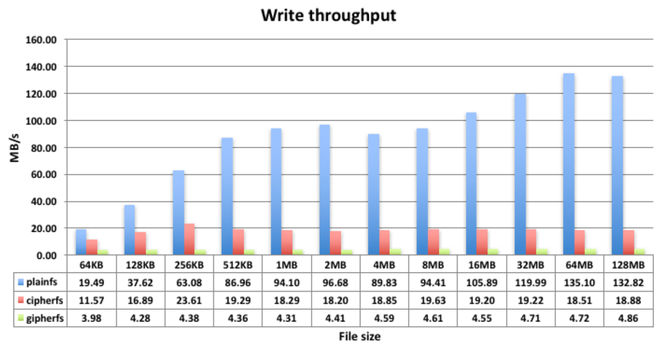
Figure 3 presents the results obtained when copying a file out of file system, and can be used to determine the file system's read bandwidth. Figure 4 measures the opposite operation, copying a file into a directory mounted in the file system, and can thus be used to determine the file system's write bandwidth. The results were obtained my measuring the time taken to perform a copy operation, using the *cp* bash command, of files ranging from 64 KB to 128 MB. The transfer time was averaged over 100 iterations of the copy operation in each direction, in order to obtain a stabilized value, which was used to compute the average bandwidth.



	64KB	128KB	256KB	512KB	1MB	2MB	4MB	8MB	16MB	32MB	64MB	128MB
🟦 plainfs	24.40	52.23	101.55	180.82	290.03	398.26	445.00	577.31	699.14	870.90	1025.94	1056.95
🟥 cipherfs	19.73	27.82	39.75	44.21	64.89	86.78	105.55	119.83	132.64	142.63	155.12	164.56
🟩 gipherfs	14.15	20.92	34.99	54.27	82.30	113.51	143.79	161.15	174.27	188.11	199.98	205.67

Figure 3: Naive *gipherfs*: read throughput

The obtained results match the expected performance of each of the file systems. *plainfs* is the fastest by a large margin in both directions, about 6 and 20 times for read and write operations, respectively. This is obvious given the absence of any data transformation applied to its file data. The discrepancy becomes even more apparent for write operations, which are serialized and limited to process at most 4 KB at a time by FUSE in this mode.



	64KB	128KB	256KB	512KB	1MB	2MB	4MB	8MB	16MB	32MB	64MB	128MB
🟦 plainfs	19.49	37.62	63.08	86.96	94.10	96.68	89.83	94.41	105.89	119.99	135.10	132.82
🟥 cipherfs	11.57	16.89	23.61	19.29	18.29	18.20	18.85	19.63	19.20	19.22	18.51	18.88
🟩 gipherfs	3.98	4.28	4.38	4.36	4.31	4.41	4.59	4.61	4.55	4.71	4.72	4.86

Figure 4: Naive *gipherfs*: write throughput

Focusing on the performance of *cipherfs* and *gipherfs*, it can be noted that both have similar read bandwidth and that *gipherfs* is actually slightly faster for bigger files. This is explained by the benefits of processing bigger chunks of data at a time in the GPU. As files grow, the amount of data processed by FUSE in each read operation also increases, to a maximum of 128 KB per operation. This amount of data is already enough to compensate for the GPU memory transfer overheads incurred in each GPU invocation. As such, moving the computation of AES to the GPU already pays off in this situation, even if only by a small margin. By opposition, the limitations imposed by FUSE on write operations especially hurt the performance of *gipherfs*. It is almost 5 times slower than *cipherfs* due to the fact that processing at most 4 KB at a time in each GPU invocation is not enough to compensate for the memory transfer overheads.

Another interesting property is that only *plainfs* benefits from processing bigger files, evidenced by the constant increase in its bandwidth as file size increases, for both reads and writes. The same doesn't happen for both *gipherfs* and *cipherfs*. This can be explained by *plainfs* not being limited by any data transformation computation, as is the case of the other two systems. The increased bandwidth of *plainfs* means that the data processing time isn't proportional to the total amount of data processed and that it benefits from processing more data. *gipherfs* and *cipherfs* however are limited by the AES transformations, which also limit their maximum bandwidth.

4.2 Enabling *big-writes*

The straightforward integration of the GPU accelerated version of AES yields rather disappointing results when compared with a typical implementation using only the CPU. Even though it provides equivalent read performance, it has a much lower write bandwidth. This can be explained by how FUSE handles write operations by default, in which payloads are split into data chunks of at most 4 KB data chunks, processed serially. This mode of operation definitely doesn't fit well the GPU programming paradigm, which favors compute intensive tasks and is highly penalized by I/O operations. The limited amount of data processed by each call to the GPU, resulting in an excessive amount of data transfers between the host and the GPU, severely hurts the write bandwidth of *gipherfs*, as evidenced by Figure 4.

It seems plausible that increasing the amount of data processed by each write operation, and consequently by each GPU invocation, would greatly benefit the write performance of *gipherfs*. In order to determine the potential benefits of increasing the size of the FUSE write payload, described in Section 3.2.1, the bandwidth of the isolated AES encryption operation was benchmarked on the CPU and on the GPU. For this purpose, the impact of increasing the amount of data processed by each individual call was measured. The experience consisted of processing an 8 MB file in increasingly sized data chunks, ranging from 2 to 512 KB, both in the CPU and on the GPU and comparing the obtained bandwidth.

Figure 5 shows that the average processing time of each data block increases with its size in the CPU version, while it stays relatively constant in the GPU. More specifically, as the block size doubles, also does the CPU processing time. This can be explained by the sequential nature of the CPU AES encryption implementation, in which each AES block is processed serially, one at a time. By opposition, the GPU AES encryption implementation processes all AES blocks concurrently, thus incurring little overhead when processing a larger amount of data. Actually, increasing the amount of data processed in a single GPU call 256 times, from 2 to 512 KB, only increases processing time by a factor of 2. Moreover, Figure 5 shows that processing at least 64 KB of data in each GPU call already makes the GPU implementation of the AES encryption operation more efficient than that of the CPU.

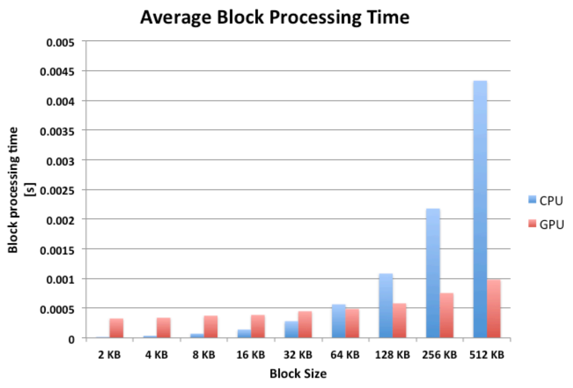


Figure 5: AES encryption benchmark: average block processing time

Figure 6 presents the total time taken by each implementation to process the complete 8 MB file. The results are consistent with the average block processing time, and show that the GPU performance is highly penalized when the file is processed in small data chunks. The CPU version on the contrary is unaffected by this variation, as its total processing time remains constant. Again, the GPU implementation is able to beat the CPU when the block size is at least 64 KB long. Figure 7 presents an alternative perspective of the complete file processing performance, based on the average bandwidth.

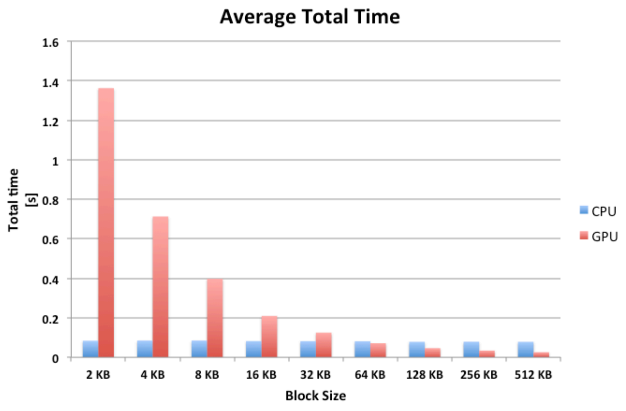


Figure 6: AES encryption benchmark: average total processing time

These results are consistent with the ones presented in Figure 4 and further sustain the argument for the poor write performance of the straightforward implementation of *gipherfs*. Recalling that FUSE splits write operations in chunks of 4 KB by default, the low write bandwidth of *gipherfs* becomes evident by comparing the bandwidth of the CPU and GPU versions of the AES encryption operation for 4 KB data chunks in Figure 7. The benchmark results further sustain the hypothesis formulated in section 3.2.1 that increasing the amount of data processed in each FUSE write operation would benefit the performance of *gipherfs*.

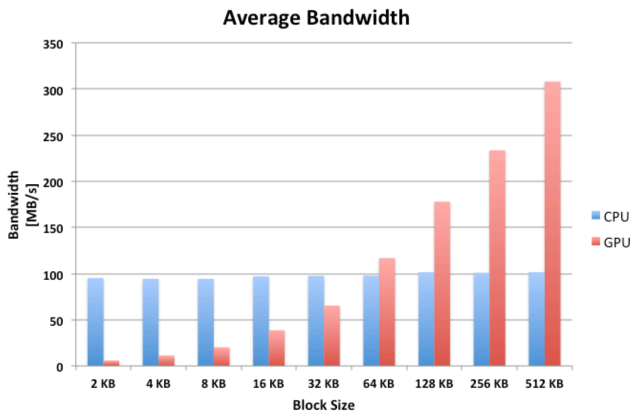
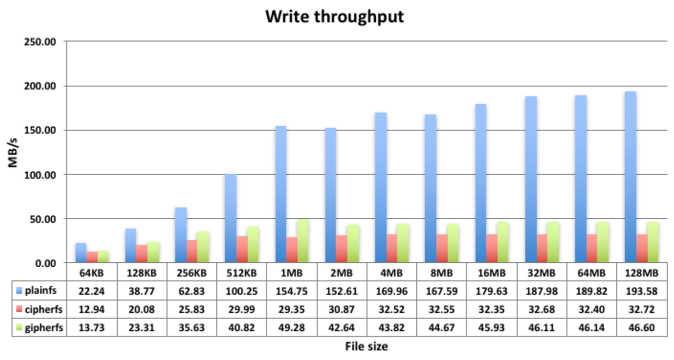


Figure 7: AES encryption benchmark: average bandwidth

Figure 8 presents the performance of an optimized version of all Virtual File Systems, using FUSE in *big-writes* mode. In this mode, each write operation is able to process at most 64 KB of data at a time, instead of the default maximum of 4 KB. By comparing these results with the ones presented in Figure 4 it becomes clear that all iterations benefit with the introduction of *big-writes* mode. *plainfs* and *cipherfs* write bandwidth increases by a factor of about 0.5 and 2 times respectively. However, *gipherfs* is the one that benefits the most. Its write bandwidth is boosted by a factor of 10 for big files, and brings it to par with the performance of *cipherfs* for both read and write operations, with writes actually being consistently more efficient in *gipherfs*.



	64KB	128KB	256KB	512KB	1MB	2MB	4MB	8MB	16MB	32MB	64MB	128MB
🟦 plainfs	22.24	38.77	62.83	100.25	154.75	152.61	169.96	167.59	179.63	187.98	189.82	193.58
🟥 cipherfs	12.94	20.08	25.83	29.99	29.35	30.87	32.52	32.55	32.35	32.68	32.40	32.72
🟩 gipherfs	13.73	23.31	35.63	40.82	49.28	42.64	43.82	44.67	45.93	46.11	46.14	46.60

Figure 8: Increasing *gipherfs* write payload: write throughput

4.3 Improving device memory management

The last GPU integration optimization proposed in Section 3.2.3 aims at minimizing the memory management overheads required by GPU programs. More specifically, eliminating the device memory allocation and deallocation steps. As in the previous experimental evaluation, a specific benchmark was developed to evaluate the potential benefits of this optimization. For this purpose, all the components making up a complete call to the GPU were analyzed in detail, so as to determine their relative cost. The analysis was again based on the encryption of an 8 MB file, processed in blocks of size ranging from 2 to 512 KB.

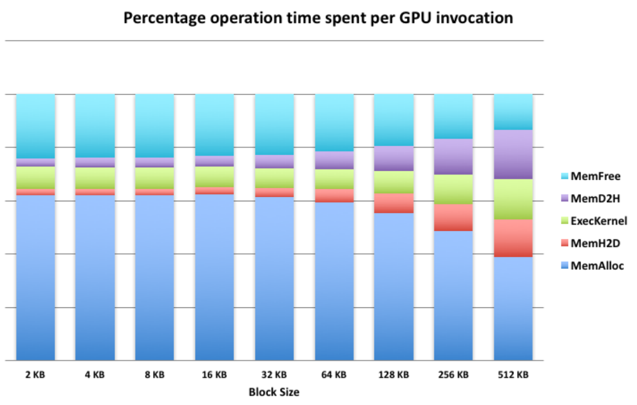
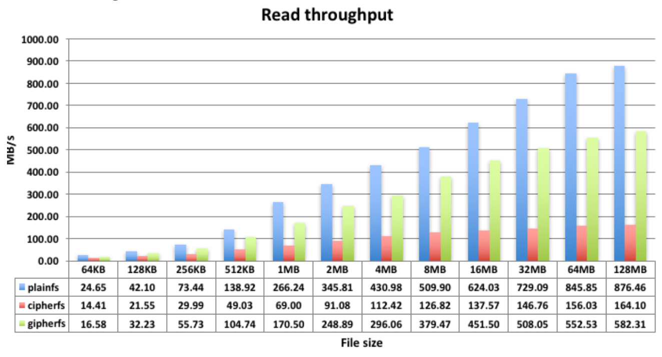


Figure 9: GPU memory operations benchmark: relative cost

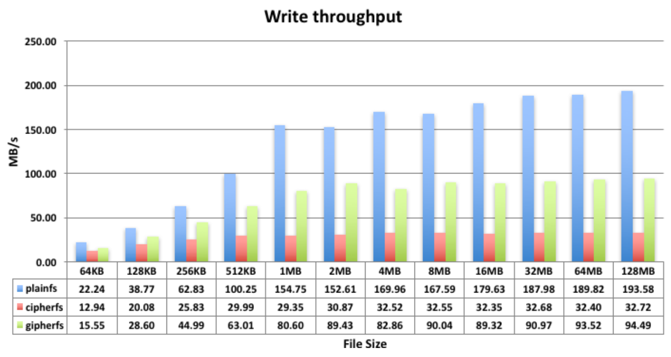
Figure 9 presents the relative time spent: allocating memory on the GPU (MemAlloc); transferring memory from the host to the GPU (MemH2D); executing the AES encryption kernel on the GPU (ExecKernel); transferring the results from the GPU to the host (MemD2H); and releasing the GPU memory (MemFree). The results show that most of the time is spent allocating memory on the device, followed by releasing it, especially for smaller block sizes. As the block size increases, memory transfers and kernel execution amortize the cost of these operations. However, they always represent at least 50% of total GPU execution time. The percentages for 64 and 128 KB block sizes are especially relevant, as these are the maximum payloads handled by FUSE's write and read operations respectively, using *big-writes* mode. In this setting, device memory allocation and release make up about 80% of the total GPU execution time. These results support the expected benefits introduced by eliminating the memory allocation and release steps in GPU invocations by pre-allocating and reusing device memory buffers between GPU invocations.



	64KB	128KB	256KB	512KB	1MB	2MB	4MB	8MB	16MB	32MB	64MB	128MB
🟦 plainfs	24.65	42.10	73.44	138.92	266.24	345.81	430.98	509.90	624.03	729.09	845.85	876.46
🟥 cipherfs	14.41	21.55	29.99	49.03	69.00	91.08	112.42	126.82	137.57	146.76	156.03	164.10
🟩 gipherfs	16.58	32.23	55.73	104.74	170.50	248.89	296.06	379.47	451.50	508.05	552.53	582.31

Figure 10: Improved device memory management in *gipherfs*: read throughput

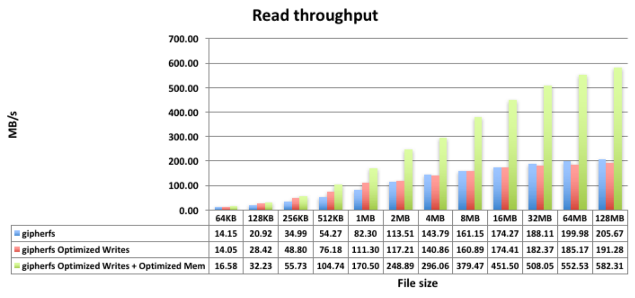
Figures 10 and 11 present a final comparison between the three iterations of the system, using a version of *gipherfs* implementing both of the optimizations described in Section 3. Focusing first on the benefits enabled by avoiding device memory allocation, *gipherfs* performance is further boosted by a factor of 2 for both read and write operations. This comes as a result of eliminating the operations responsible for about 80% of the total GPU execution time, as evidenced by Figure 9.



	64KB	128KB	256KB	512KB	1MB	2MB	4MB	8MB	16MB	32MB	64MB	128MB
🟦 plainfs	22.24	38.77	62.83	100.25	154.75	152.61	169.96	167.59	179.63	187.98	189.82	193.58
🟥 cipherfs	12.94	20.08	25.83	29.99	29.35	30.87	32.52	32.55	32.35	32.68	32.40	32.72
🟩 gipherfs	15.55	28.60	44.99	63.01	80.60	89.43	82.86	90.04	89.32	90.97	93.52	94.49

Figure 11: Improved device memory management in *gipherfs*: write throughput

Moreover, the final version of *gipherfs* outperforms *cipherfs* by a factor of 3.5 and 3 for read and write operations, respectively. This provides evidence of the benefits of moving the computation of the AES algorithm from the CPU to the GPU, especially when large amounts of data are processed. Finally, the comparison of *plainfs* and *gipherfs* can be used to argue that introducing GPU-accelerated cryptographic mechanisms into the operation of file systems only reduces its performance by a half, in the worst case.

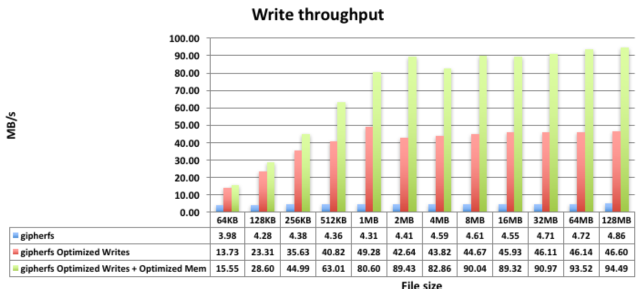


	64KB	128KB	256KB	512KB	1MB	2MB	4MB	8MB	16MB	32MB	64MB	128MB
🟦 gipherfs	14.15	20.92	34.99	54.27	82.30	113.51	143.79	161.15	174.27	188.11	199.98	205.67
🟥 gipherfs Optimized Writes	14.05	28.42	48.80	76.18	111.30	117.21	140.86	160.89	174.41	182.37	185.17	191.28
🟩 gipherfs Optimized Writes + Optimized Mem	16.58	32.23	55.73	104.74	170.50	248.89	296.06	379.47	451.50	508.05	552.53	582.31

Figure 12: *giphefs* optimizations: impact on read throughput

Figures 12 and 13 summarize the benefits introduced by each optimization to gipherfs in order to highlight them. Due to the concurrent nature of FUSE read operations, the read bandwidth of gipherfs is consistently higher than its write bandwidth. Reads don't actually benefit from the introduction of big-writes mode, as it affects only the write operation, but the improved device memory management boosts its performance by a factor of 3. On the other hand gipherfs write operations benefit the most from the introduction of big-writes mode, which increases its throughput by a factor of more than 10, from 4 to 46 MB/s. Despite also benefiting from the improved device memory management, it does so only by a factor of 2, approximately.

This further supports the relevance of the optimizations introduced into the system, which benefit each of its main I/O operations differently but in a balanced manner.



	64KB	128KB	256KB	512KB	1MB	2MB	4MB	8MB	16MB	32MB	64MB	128MB
🟦 gipherfs	3.98	4.28	4.38	4.36	4.31	4.41	4.59	4.61	4.55	4.71	4.72	4.86
🟥 gipherfs Optimized Writes	13.73	23.31	35.63	40.82	49.28	42.64	43.82	44.67	45.93	46.11	46.14	46.60
🟩 gipherfs Optimized Writes + Optimized Mem	15.55	28.60	44.99	63.01	80.60	89.43	82.86	90.04	89.32	90.97	93.52	94.49

Figure 13: *giphefs* optimizations: impact on write throughput

4.3 State-of-the-art analysis

In [7], the authors present a very similar approach to the one presented in this paper. However, the approach proposed herein differs in several fundamental aspects when compared to this state-of-the-art approach. In particular, the work proposed in [7] aims at extending a fully functional FUSE file system by replacing EncFS's openSSL CPU based implementation of AES by its GPU counter-part version. Although certain obstacles when introducing the GPU-accelerated version of AES are briefly mentioned (mainly, *EncFS* unsuitability to the GPU programming model), the solution proposed in [7] lacks the ability of fine-tuning certain implementation aspects of the file system, which are crucial for improving the overall system performance. On the other hand, the herein proposed GPU-based secure file system solution is developed from scratch, thus it allows for tweaking all key aspects relevant to maximize the GPU performance, i.e., by taking into account all GPU implementation specificities (as elaborated in the previous sections). It is also worth to emphasize that the CPU openSSL version of AES used in [7] for baseline comparison with the GPU implementation does not exploit AES-NI like extensions, which are nowadays commonly available in modern CPUs.

5 CONCLUSIONS

This paper presents a secure file system, using the AES standard encryption algorithm to provide file confidentiality in an efficient and transparent manner. A Virtual File System extended with cryptographic mechanisms accelerated by the GPU is implemented using its high performance capabilities for executing the cryptographic primitives of AES, speeding up execution and increasing throughput. Using the GPU as a cryptographic co-processor maximizes its utility and minimizes the impact on the CPU and on the overall system performance. Moreover, integrating the mechanism into the operation of a Virtual File System facilitates its adoption by not requiring explicit user intervention.

The obtained results confirm the advantages and merit of this approach by showing a consistent increase in throughput, specially for big files. The system is able to increase write and read throughputs by a factor of 3 and 3,5 respectively, when compared with an equivalent CPU-based implementation. Moreover, the results also show that introducing the encryption primitives accelerated by the GPU only reduces throughput by a half on the worst case.

6 FUTURE WORK

While good results are obtained, several directions could be followed in the future work. In terms of the system's security, a better key management mechanism and a more secure mode of operation, such as CTR or XTS, for the AES encryption algorithm may be used, increasing the system's security, without compromising its performance. Additionally, other security concerns such as data integrity and authentication may also be introduced into the system, using a similar GPU based approach. In terms of performance, more advanced CUDA mechanisms, such as CUDA streams, may be leveraged in order to further increase the overall system's throughput. FUSE's limitations, namely the serialization imposed on write operations could also be addressed in order to further improve the system's write throughput.

REFERENCES

[1] Shimpi, Anand Lal; Wilson, Derek *NVIDIA's GeForce 8800 (G80): GPUs Re-architected for DirectX 10*, 2006
[2] Sourceforge. File systems using FUSE, *http://sourceforge.net/apps/mediawiki /fuse/index.php?title=Fil-eSystems*, 2010
[3] National Institute of Standards and Technology (NIST). *FIPS-197: Advanced Encryption Standard*, November, *http://www.itl.nist.gov/fipspubs/*, 2001
[4] Khronos Group, *OpenCL Specification 1.1*, *http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf*
[5] NVIDIA. CUDA 4.0. *http://developer.nvidia.com/cuda-toolkit-40*, 2011
[6] Weibin Sun, Robert Ricci, and Matthew L. Curry, *GPUstore: harnessing GPU computing for storage systems in the OS kernel*, In *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR '12)*, ACM, New York, USA, Article 9, 12 pages, 2012
[7] Eimot, Magne, *Offloading an encrypted user space file system on Graphical Processing Units*, Master Thesis, University of Oslo, Department of Informatics, Oslo, Norway, 2009
[8] Harrison, O., Waldron, J., *AES encryption implementation and analysis on commodity graphics processing units.* In *Paillier, P., Verbauwhede, I. (eds.) Cryptographic Hardware and Embedded Systems—CHES 2007*, Lecture Notes in Computer Science, vol. 4727, pp. 209–226. Springer, Heidelberg, 2007
[9] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, T. Purcell, *A survey of general-purpose computation on graphics hardware*, Comput. Graph. Forum, vol. 26, no. 1, pp. 80-113, 2007
[10] S. Manavski et al., *Cuda compatible gpu as an efficient hardware accelerator for aes cryptography*, IEEE Int. Conf. on Signal Processing C& Communications, pp. 65-68, 2007
[11] Q. Li, C. Zhong, K. Zhao, X. Mei, and X. Chu, *Implementation and analysis of aes encryption on gpu*, In *14th IEEE International Conference on High Performance Computing and Communication 9th IEEE International Conference on Embedded Software and Systems*, HPCC-ICESS, pp. 843-848, 2012
[12] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel, *GPUfs: integrating a file system with GPUs*, SIGPLAN Not. 48, 485-498, 2013
[13] Abdullah Gharaibeh, Samer Al-Kiswany, Sathish Gopalakrishnan, and Matei Ripeanu, *A GPU accelerated storage system*, In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10). ACM, New York, USA, 167-178, 2010
[14] Samer Al-Kiswany, Abdullah Gharaibeh, Elizeu Santos-Neto, George Yuan, and Matei Ripeanu, *StoreGPU: exploiting graphics processing units to accelerate distributed storage systems*, In *Proceedings of the 17th international symposium on High performance distributed computing (HPDC '08)*, ACM, New York, USA, 165-174, 2008

[15] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt, *Gdev: first-class GPU resource management in the operating system.* In *Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC'12)*, USENIX Association, Berkeley, CA, USA, 37-37, 2012

[16] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel, *PTask: operating system abstractions to manage GPUs as compute devices*, In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, ACM, New York, NY, 2011

[17] *PKCS #7*, *Cryptographic Message Syntax Standard*, RSA Laboratories, Version 1.5, Nov 1993

[18] *https://github.com/nablahero/cuda_aes*

[19] *https://www.wolfssl.com*

[20] Shay Gueron, *Intel Advanced Encryption Standard (AES) New Instructions Set*, Intel Architecture Group, Israel Development Center, Revision 3.01, *https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf*, 2012