GPU Accelerated Cryptography as an OS Service

Abstract. Graphics processing units (GPUs) have become popular devices for accelerating general purpose computing. In recent years there has been a surge in research involving the use of GPUs as cryptographic accelerators. Research has shown that contemporary GPU architectures can achieve higher throughput in the context of both symmetric and asymmetric key cryptography than a traditional CPU. Despite the existence of these new approaches, there remains no way for OS kernel services or userspace applications to make use of these implementations in a practical manner. To overcome this shortcoming, this paper investigates the integration of GPU accelerated cryptographic algorithms with an established service virtualisation layer within the Linux kernel, the OCF-Linux framework. This paper demonstrates that it is feasible to use a centralised kernel service to provide a standardised abstraction to GPU accelerated cryptographic functions for both kernelspace and userspace components.

1 Introduction

Symmetric key algorithms such as AES, DES, ARIA; symmetric key modes of operations; and asymmetric key algorithms such as RSA, DSA and ECC have recently been explored in the context of GPU acceleration [1–9]. It has been demonstrated that the GPU can act as an effective accelerator of symmetric key algorithms using sufficiently large buffers and of asymmetric key algorithms using a sufficient number of concurrent primitives. To use these implementations it is a requirement to interact with GPU specific interfaces such as the Nvidia CUDA API [10]. With the increasing number of GPU accelerated cryptographic algorithms, there is a need to provide an efficient and standardised operating system wide interface to these implementations. The OpenBSD Cryptographic Framework (OCF) [11] provides the basis of the solution to this problem.

The original OCF was developed for OpenBSD and has since been ported to FreeBSD [12], NetBSD and Linux [13]. It was created to provide uniform access to cryptographic accelerator functionality by hiding hardware specific details behind a standardised API. It provides access to this functionality for kernelspace services as well as normal userspace applications and APIs. For our investigation we use the Linux port of the OCF which currently supports the 2.6.26 Linux kernel. Although we do not directly use the native linux-crypto (Crypto API) project [14], which has in-built support for some crypto-cards, we note that the OCF acts as a wrapper for this library. We did not use linux-crypto for this work due to its current lack of support for asymmetric algorithms and the fledgling status of its userspace interface, however the main contributions in this paper are also relevant to this project.

The main contributions of this paper are, the effective integration of the GPU within the OCF model; the observation that the GPU interface is userspace only and the mechanisms used to allow it to be part of a kernel service; the introduction of a new memory management system within the OCF to allow efficient handling of memory transfers between multiple address spaces; and also an implementation of a general purpose multi-request batching scheme for asymmetric key requests with regard to the GPU. The only previous attempt to provide a form of uniform access to GPU crypto acceleration involved AES via an OpenSSL engine by Rosenberg et al. [15]. This implementation was applicable to userspace applications only and reported a 0 to 3% improvement over the CPU.

The motivation for this work is to provide a standard method of access to the latest GPU crypto acceleration work to all components within an operating system, with minimal loss of performance. This will allow application, kernel and driver developers to transparently include the GPU as part of their cryptographic solutions. We also observe that the GPU has a requirement of high work loads to achieve its peak performance. By using a centralised framework which is used for all system-wide cryptographic needs, we increase the likelihood of high occupancy on the GPU and thus its potential to act as an effective crypto-accelerator.

2 Background

2.1 OCF Background

Figure 1 shows a high level view of the OCF framework. The core component of the framework, the main "Crypto" layer, provides two APIs - the producer API for use by crypto-card device drivers and the consumer API for use by other kernel subsystems. An ioctl interface, which uses the /dev/crypto device file, provides a mechanism through which normal userspace applications can issue cryptographic requests. This interface is provided by the "Cryptodev" layer and uses the consumer API to pass on userspace requests to the Crypto layer. Devices drivers can register their support of various cryptographic algorithms with the Crypto layer. Cryptographic requests received directly by the Crypto layer or sent via the Cryptodev layer are matched with capable devices and issued to the corresponding device driver.

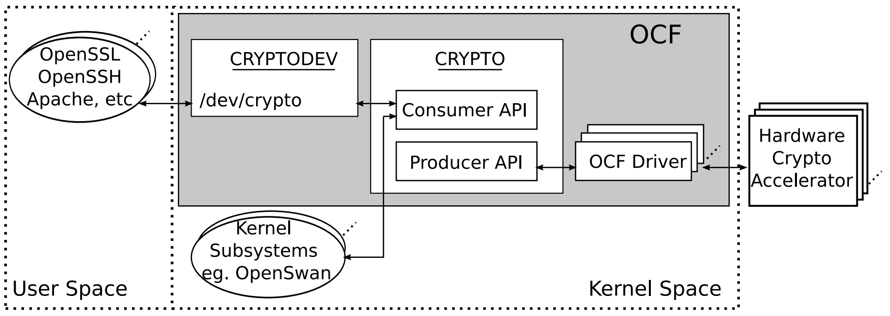

Fig. 1: Original OCF Architecture.

2.2 GPU Background

The GPU used in our implementations is Nvidia's 8800GTX (2006) which was the first DirectX 10 [16] compliant GPU released by Nvidia. It is Nvidia's first processor that supports the CUDA API [10]. The 8800GTX consists of 16 SIMD processors, called Streaming Multiprocessors (SM), each of which contain 8 ALUs. A single instruction is issued to an SM every 4 clock cycles, which is executed by all 8 ALUs. This creates an effective SIMD width of 32 operands for an SM. The code that runs on the GPU is referred to as a kernel. Via the CUDA API, the programmer can specify the number of threads that are required for execution on the GPU during a kernel call. These threads are grouped into programmer defined numbers of CUDA blocks, where each block of threads is guaranteed to run on a single SM. The number of threads per block is also programmer defined. Programmers should allocate threads in groups of 32, called a CUDA warp, to match the effective SIMD width mentioned above. A point of note relevant for this paper regards thread divergence. If any thread execution path diverges from the execution path of other threads within a CUDA warp, all the divergent code paths must be executed serially on the SM. An important note regarding GPU performance is its level of occupancy. This refers to the number of threads available for execution at any one time, and is important for hiding memory latency. It is desirable to have as high a level of occupancy as possible.

3 Integration of GPU and OCF

3.1 Overview

The OCF provides a standardised method for the integration of any cryptographic accelerator device driver using its producer API. This API allows a device driver to register itself and its supported algorithms with the OCF, making it a target for processing cryptographic requests. The device driver is responsible for registering four callback functions with the OCF, which are used for the set up and tear down of symmetric algorithm sessions and also for the processing of symmetric and asymmetric requests. We have created a GPU cryptographic driver, which fulfils the producer API requirements. The driver currently supports AES and modular exponentiation with CRT, suitable for RSA-1024. Supporting these two algorithms allows an analysis of the main issues arising from GPU integration with the OCF for both symmetric and asymmetric functions.

The algorithms supported by the GPU driver are implemented using Nvidia's CUDA API. The CUDA interface is provided via a userspace runtime library and as such requires its usage to be from userspace processes. Unfortunately Nvidia do not provide any drivers which allow the direct control of their cards from within the kernel. This restriction forces all interactions with their cards to originate from userspace processes, making the provision of CUDA services from within the kernel a challenge. To overcome this restriction, we have split our GPU driver into two parts, a kernelspace driver and a userspace daemon.

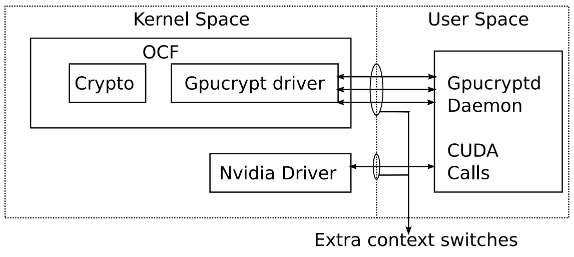

Fig. 2: OCF and GPU: High Level View - Different Address Space Problem.

Figure 2 shows a high level overview of the GPU driver integration into the OCF. It illustrates the separation of the GPU driver into the kernelspace part, **Gpucrypt**; and the userspace part, **Gpucryptd**. Gpucryptd follows the normal daemon convention, and as such runs as a high privilege background OS process. Gpucryptd is responsible for receiving cryptographic requests from Gpucrypt and processing them using Nvidia's userspace runtime API. A major disadvantage of this separation is the use of extra address spaces within the processing pipeline making data transfer more complex. Extra address spaces can result in a critical bottleneck in performance when processing requests unless memory is carefully managed. We explore this issue in full within the next topic. Another disadvantage of the driver separation is the introduction of two extra OS context switch points within the processing pipeline. This becomes more of an overhead when the number of requests increase, such as when request buffer sizes are small. Unfortunately there is no way to avoid these context switches, though as the GPU only suits cryptographic acceleration with large workloads and high arithmetic intensity we will see that the overhead has limited effect.

3.2 Memory Management

When using devices that handle high volumes of data transfer it is common practice to ask the driver to allocate the memory used in these transfers. This has the advantage that the driver knows what type of memory (contiguous/non-contiguous, zone location) suits the corresponding device for fast bus transfer. It is also common practice that allocated memory is shared between the driver and the calling process, either by driver allocation (mmap()) or by mapping userspace pages (get_user_pages()). If memory is not shared then userspace processes must undergo a copy of memory between user address space and kernel address space using the copy_from/to_user() kernel functions. Using an abstraction framework like the OCF, or the linux-crypto project, removes the direct line of communication required for standard driver requests for memory allocation, either by userspace processes or kernelspace subsystems.

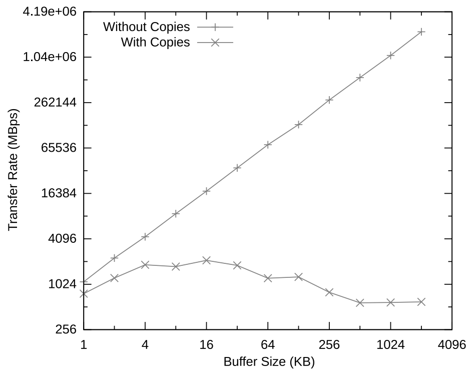

Fig. 3: Cryptodev Memory Management Overhead.

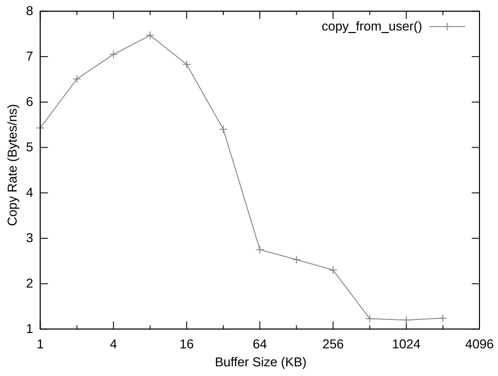

Fig. 4: Perf. of copy_from_user() Function.

Integration of the GPU with such a framework emphasises this deficiency due to two factors. First, the GPU requires large volumes of data for symmetric algorithms to reach its performance potential [1]. The larger the volumes of data, the worse the memory copy overhead. The OCF Cryptodev layer implements a copy from and to userspace policy for data transfer. Figure 3 shows the Cryptodev layer's performance with and without these copies as the buffer sizes increase. Comparing this to Figure 4 we can see that a memory copy looses efficiency as the buffer sizes increase. Second, one cannot give memory to the Nvidia driver and request it to be used for DMA acceleration, the memory must be requested from the driver. Thus, even using a direct IO approach (as in the new linux-crypto userspace API), where userspace pages are mapped in by the kernel on request, we must still perform a memory copy into and out of GPU DMA memory. Thus for any device which has DMA memory restrictions, it can be beneficial to have a mechanism for allowing the framework's drivers to manage their own memory.

We have added a new memory management system to the OCF which allows a consumer component (userspace application or kernelspace component) to share memory which is directly managed by the OCF drivers. This system allows the GPU driver to reduce the number of memory copies required during request processing to zero. Each memory allocation is recorded centrally by the OCF as a new memory mapping, which stores the driver's address (map_ptr) and the consumer component's address (app_ptr) of the allocated memory. map_ptr is the address the driver uses to refer to the allocated memory, which would normally be a kernelspace address space, however in the case of the GPU it will belong to the Gpucryptd daemon address space. The app_ptr is the address the consumer component uses to refer to the memory and will belong to a userspace processes' address space if the OCF was called via the Cryptodev interface, or otherwise belong to the kernel address space.

Figure 5 illustrates our implementation of the memory mappings for all consumer components. A separate mapping space, indexed by the current thread's thread group ID, is used to store all mappings for each consumer component (the ID is zero in the case of kernel consumer components). This ensures allocated memory can only be accessed by the process which requested the allocation. Within each space the mappings are grouped according to which device allocated the memory. When memory allocation fails due the lack of a device driver supporting the new memory management system, we still wish to avoid the memory copies performed by the Cryptodev layer. To solve this, when a memory allocation fails and is not tied to a specific device, the Cryptodev layer allocates its own memory for sharing with the userspace consumer, and as such maintains its own Cryptodev mappings. The following is a detailed account of how mappings are created, removed and used. The new memory management API is briefly listed in Appendix A.

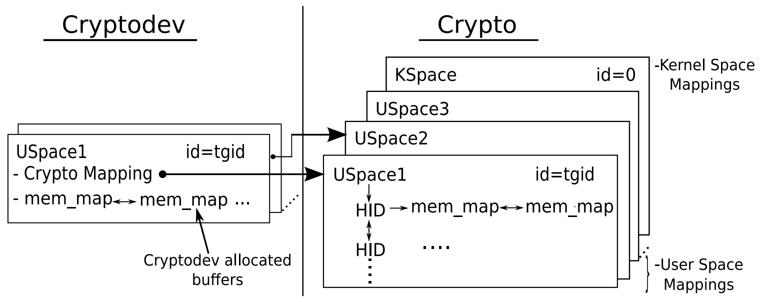

Fig. 5: Crypto and Cryptodev Layers:Memory Mappings Internal Structure.

Memory map creation:

Userspace, allocation request: a userspace process makes a request for memory via a new ioctl supported by the Cryptodev layer. This allocation request can suggest a specific device to carry out the allocation or allow the OCF to choose a device and report the used device back to the consumer. The standard mmap() system call is not used as it cannot support device specification in this way. The OCF relays the allocation request to the device driver, which performs the allocation and returns the underlying memory pages and map_ptr to the OCF. The OCF takes these pages and manually calls the internal kernel version of mmap(), which generates a new virtual memory area (VMA) for the userspace process. We use the returned pages from the device driver to map to this VMA, and return the VMA start address to the userspace process. In the case where no device can be found, the Cryptodev layer allocates its own kernel memory, and maps this to the calling process' VMA. We illustrate an example of a userspace process allocation request to the GPU in Figure 6. We can see six cases of context switch between user and kernel mode; four of them involve the added trip out of and back into the kernel due to userspace CUDA calls as mentioned previously. Here we have highlighted context switch 1 and 2. The above explains the mechanism for the elimination of a memory copy at context switch 1. We discuss the GPU driver part of the new memory management system which eliminates the copy at context switch 2 in Section 3.3.

Userspace, fork: on fork a child process will have shared access to OCF allocated memory. Allocated memory returned by the OCF is set as always shared. Supporting private memory by implementing a copy-on-write procedure or executing a new allocation for each fork were deemed unnecessary considering that a child process can allocate its own memory. We share the memory between parent and child by overriding the VMA's vm_open() kernel function, which is called by the kernel when a new reference is created, such as on fork(). When this function is called, we create a new Cryptodev or Crypto memory mapping, within a new mapping space representing the process' new address space. Note that light-weight processes automatically share allocated memory as the VMAs of the processes are shared.

Kernelspace, allocation request: a process in kernel mode, or a kernel component, request new memory directly from the Crypto layer using a new crypto_alloc() function. This processes the request as above, selecting an appropriate device. However, instead of dealing with a userspace process' VMA, the OCF returns a kernelspace pointer which references the new memory. The memory returned is not necessarily within the kernel address space, in the case of the GPU it belongs to the Gpucryptd address space. Thus, the OCF selectively performs a vmap() to map the memory into the kernel virtual address space if necessary.

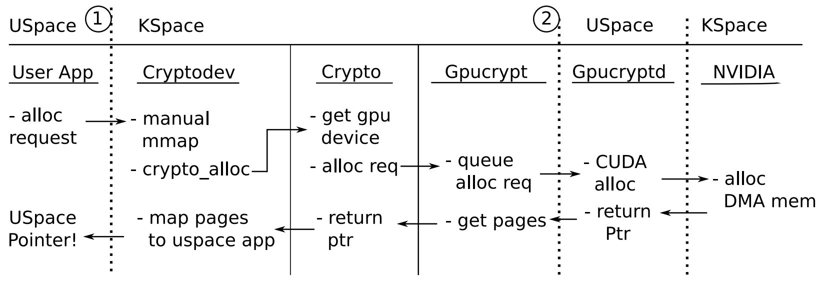

Fig. 6: New OCF Memory Alloc: Cryptodev to GPU.

Memory map removal:

Userspace: a userspace process can free OCF allocated memory by executing the munmap() Unix command or by terminating. On such an event the kernel calls the vm_close()

kernel function for the allocated memory's VMA, which we have overwritten. If this process is the last to hold an open reference to the shared memory we issue a free command to the Crypto layer for device allocated memory or the Cryptodev layer for Cryptodev allocated memory. After the memory is freed the memory map is removed from the appropriate Crypto or Cryptodev memory map spaces.

Kernelspace: kernelspace components issue a free directly to the Crypto layer via the crypto_free() command. Unlike the Cryptodev free process above, the OCF must ensure that there exists a valid mapping for the kernelspace pointer for the specified device. If found, a free command is issued to the device driver and on return the memory mapping is removed from the kernel mapping space, see "KSpace" in Figure 5.

Memory map translation:

Userspace: all cryptographic requests received via the Cryptodev interface are scanned for pointers which may require translation. First, the Crytpo layer is called to find a mapping which matches the userspace pointer (app_ptr) and the device specified within the request. This is done by finding the mapping space corresponding to the calling process using its thread group ID. Once the space is found we scan for a matching map within the corresponding device list of memory mappings. If a match is found, the device address (map_ptr) recovered replaces the userspace pointer within the cryptographic request and can be used directly by the device without any copies taking place. If no match is found, we repeat a similar process for any Cryptodev allocated memory. If no match is still found we default to the original OCF behaviour of using kmalloc() and the copy_from/to_user() kernel functions to copy the userspace buffers into the new kernelspace buffers. Figure 7 gives a brief illustration of the interactions between the Crypto and Cryptodev layer during this translation. The original OCF behaviour should possibly be upgraded to use the get_user_pages() call, as in the linux-crypto project, to default to using direct IO thereby eliminating the need for copies within the Cryptodev layer. The request is always tagged internally to ensure the device driver can detect if a request pointer is a native device address or a normal kernelspace address.

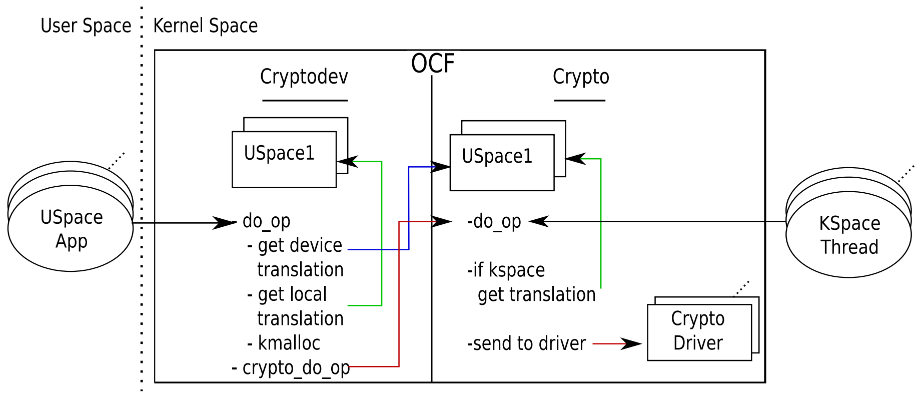

Fig. 7: Crypto and Cryptodev Layers: Memory Mapping Translation Process.

Kernelspace: cryptographic requests received undergo a similar procedure as above, except only the kernel mapping space is searched and only the Crypto memory mappings are searched. Considering that, as kernel mode processes are trusted, we provide the ability for these processes to translate the allocated memory before the request is sent to the Crypto layer. This allows the kernel processes to use native device driver pointers in their requests with tagging thus avoiding translation overhead.

Exisiting consumers: care has been taken to ensure existing consumers can continue to use the OCF with minimal impact to performance.

3.3 GPU Driver and Daemon

Here we discuss in detail the GPU driver component within the OCF and in particular its separation into a kernel driver and a userspace daemon. As previously mentioned this separation is necessary due to the requirement of using a userspace API to communicate with Nvidia's GPUs. If Nvidia provided kernel level access to its device drivers this separation could be avoided, as would the extra kernel to user mode context switches. Figure 8 illustrates both the Gpucrypt driver and Gpucryptd daemon components and an overview of how they cooperate to fulfil the requests delivered by the OCF. The main parts of this figure are discussed below.

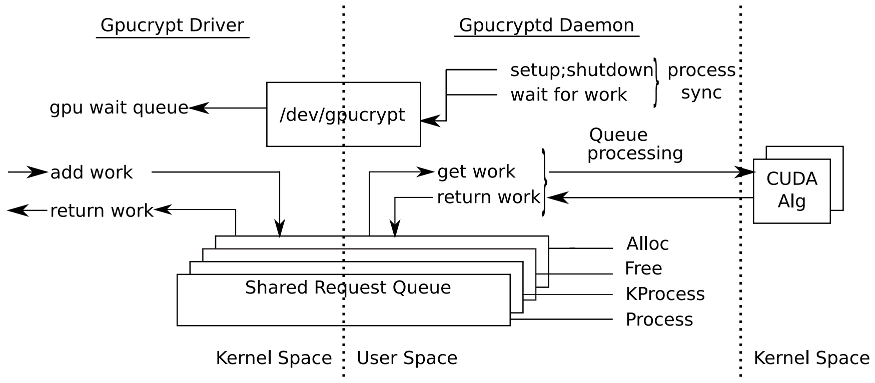

Fig. 8: GPU Driver Gpucrypt and GPU Daemon Gpucryptd.

/dev/gpucrypt: for the purpose of providing a communications channel between the two components we have created a new OS character device file called /dev/gpucrypt. On OCF startup, the Gpucrypt driver module is initialised and connects itself with the /dev/gpucrypt device file. The Gpucryptd component can subsequently open this device file and communicate with Gpucrypt via ioctls. These ioctls are used for initial handshake of Gpucryptd with Gpucrypt when the daemon sets up shared buffers for use in request processing. It also uses the interface to send a "ready for work" and "shutdown" signals. When the Gpucrypt driver receives these it correspondingly registers and unregisters with the OCF Crypto layer. The /dev/gpucrypt device is most intensively used to coordinate the processing of cryptographic and memory requests. When no work is available on the requests queues, the Gpucryptd daemon calls the driver to passively wait for more work by putting itself to sleep. Thus, whenever work is received from the OCF the driver calls wake on the daemon process' wait queue. Whenever work is finished and requires returning to the driver, the daemon uses an ioctl to signal that the work is finished, to remove the work from the queue and to call the Crypto layer for request return. The ioctls are listed and detailed in Appendix B.

Processing Requests: the Gpucrypt driver implements four shared request queues, one for each type of OCF request supported: symmetric, asymmetric, alloc and free requests. These queues are allocated by the Gpucryptd daemon at startup and memory mapped into the Gpucrypt driver, thus allowing efficient transmission of request data. When the Gpucrypt driver receives a requests from the OCF, it copies all the necessary instructions into the relevant queue. All pointers used in the requests at this stage have undergone address translation, and the addresses used within the queue are from the Gpucryptd daemon address space. Thus, the Gpucryptd daemon does not have to worry about address mapping, it can treat all pointers as native in a normal manner.

Cryptographic Requests: the Gpucrypt driver is designed to be multi-threaded and asynchronous for cryptographic requests, helping to increase the concurrency of requests on the process queues. The advantage of using separate queues for each request type is that, apart from simplifying queue management, the Gpucryptd daemon can maximise the opportunity for batching requests. The GPU benefits from large amounts of work, thus if multiple threads are issuing requests, system wide throughput can benefit from the batching of multiple requests before they are delivered to the GPU. If a single queue were used, it would increase the complexity of maximising batching of cryptographic requests with the interference of memory requests, which could unnecessarily reduce the size of a batch of cryptographic requests. Request batching is discussed in its own section later.

Memory Requests: as with standard memory allocation and free operations, we have implemented these as blocking requests. Apart from blocking the consumer thread, memory requests do not block any other request from being processed within the OCF. Figure 6, which served as an example of an OCF alloc request, can now be discussed in the context of Gpucrypt and Gpucryptd. To service an allocation request, the Gpucrypt driver first puts the allocation details on the shared alloc request queue. The Gpucryptd daemon processes this by executing the cudaMallocHost function call, which allocates pinned DMA accelerated memory. The returned address is placed back on the shared request queue, which is then used by the Gpucrypt driver to access the underlying pages. On initialisation of the Gpucryptd daemon, it registers with the Gpucrypt driver its internal task kernel pointer. This is used to retrieve access to the daemon's underlying virtual memory areas and pages. A note should be made that the virtual memory area used to reference the CUDA allocated pages is flagged with VM_IO. Device driver programmers commonly use this flag to prevent memory from being included in core dumps, however it also has the effect of treating the memory area as backed by non system RAM. For IO mapped memory it is necessary to restrict access to the underlying pages as they don't exist, however in our experience, CUDA only returns RAM backed memory. We must temporarily disable this flag in order to retrieve the underlying pages, though we take the precaution of acquiring the Gpucryptd's memory map semaphore during this period. Our experience is that this technique has successfully returned the underlying pages to the Crypto layer in all of our tests.

Request Order: maintaining separate shared queues has advantages as stated above, however it has a disadvantage of not automatically preserving the original request ordering between the differing types of requests. This can cause faults when memory requests are run out of order with respect to cryptographic requests. This can be solved by either using a single request queue, or by using read-write semaphores within the OCF. We have used a read-write semaphore for each mapping space (i.e. one per consumer process) within the OCF and found the solution to give minimal overhead. Each cryptographic request is responsible for acquiring a read-write semaphore for reading if a memory translation has occurred and releasing the semaphore on request exit. Each memory request must acquire a read-write semaphore for writing, which ensures the memory request is the only request for the consumer process within the OCF pipeline. This ensures that any translations which were valid at the start of the processing of request, remain so until the end. The use of read-write semaphores allows multiple cryptographic requests which use a memory translation to exist concurrently within the OCF pipeline. Also if no memory translation is used, e.g. legacy consumer processes, then no semaphores are used as in the original OCF.

Driver Removal: a driver can be removed at any time, and thus we must deal with the case of allocated memory when such an event occurs. Requests can be migrated to another device by the OCF and thus memory allocated for one device can be sent to another device. The Cryptodev layer sees this event as a failed translation and defaults to copying the memory from the userspace process, thus the requests will continue to proceed, however at a slower pace. To avoid this slow down the consumer process must monitor the requests for a change in device used and if a change occurs the OCF allocated memory should be freed and allocated again by the new device. If no OCF allocated memory is used then no action is required. Note that even though the device may have freed the memory, its pages are kept alive due to the consumer process' reference. Requests sent directly to the Crypto layer will also fail the translation stage and the memory will be treated as a standard kernelspace memory pointer. Again the kernel consumer thread must monitor requests for changes in the device used and reallocate memory when this occurs.

3.4 Security

We look at each of the changes made to the OCF in terms of their security implications. The new functionality of requesting memory from the OCF requires requires that all memory returned is automatically zeroed to protect from leaking information. The use of memory translation bypasses the need for the copy_to/from_user() kernel functions. These kernel functions perform important validation, ensuring that the addresses are part of the calling process' address space. We ensure that this validation is maintained by only searching for translations within a mapping space which is indexed by the thread group ID. This combined with the fact that the mappings within the space only contain userspace pointers which are generated by the kernel on behalf of the process, ensure that any match found during translation are valid userspace addresses for that process. During translation we also check the size of the buffers specified within the cryptographic requests, to ensure no buffer overflow will occur. Regarding the Gpucrypt driver, it must be ensured that the /dev/gpucrypt file is accessible by the root user only. If this is not the case, then any userspace programme may connect to the Gpucrypt driver and receive OCF cryptographic requests.

4 Concurrent Request Processing

4.1 Asymmetric Request Batching

The batching of requests allows for an increase in system wide throughput by permitting the combination of separate requests to be sent to and processed by the GPU. The types of requests that comprise a batch and the preprocessing that can be done to this batch can have significant effects on performance. General purpose symmetric key batching on the GPU has been discussed in detail within the paper [1], so we will not go into this further in this paper. However, to date there has been no treatment of general purpose batching of distinct asymmetric key requests on the GPU. Considering the OCF presents the opportunity to batch requests for delivery to the GPU we investigate the possibility of asymmetric request batching here.

Single Request: currently the OCF does not support an efficient method of executing more than one asymmetric cryptographic operation within a single request. The framework provides the ability to chain multiple requests with a link list, however this translates to a multi-request batching problem. To achieve good performance for single request processing we extend the OCF interface to permit the request's input buffers to contain multiple instances of its input vectors. This permits requests, e.g. in relation to modular exponentiation, to contain multiple bases for each exponent/modulus pair.

General Purpose Request Batching: we have based our asymmetric key implementation on the serial radix algorithm for CRT modular exponentiation suitable for RSA-1024 presented within the paper [9]. This involves spawning a new CUDA thread to handle the exponentiation of each base. As there can be multiple bases per request, and one thread per base, we require a mechanism which allows each thread to dynamically discover its request data. We must also take into consideration that the base, modulus and exponent for each operation is split into two, due to the CRT technique [17]. This involves using the prime factors P and Q of the modulus N to generate smaller pairs of bases, mod P and Q, and smaller pairs of exponents, mod $P-1$ and $Q-1$. We can then separately calculate the resultant smaller modular exponentiation within the residue number system $\{P, Q\}$ and recombine at the end to produce the final result.

Figure 9 illustrates the mechanism used to direct the threads to their corresponding data. As in [9], we direct all odd numbered CUDA blocks to P related data and all even CUDA blocks to Q related data. The base data is configured in a manner that each CUDA thread can simply scale their global thread ID to find the offset of their base data. During the preprocessing stage (discussed next), we generate a message to request index, labelled Msg2ReqIndex. This index is used to translate the message number, i.e. the base number within the full batch of requests, to the request number. The request number is used to generate a offset into the modulus, exponent and related per request data. In the figure we can see that the modulus, exponent and related data is split into two groups. This allows a simple conditional addition of a single offset to the request offset to direct a thread to the P or Q related data depending on whether the CUDA block is odd or even.

Request Preprocessing: the Gpucryptd daemon can have access to multiple asymmetric requests at a single time. The GPU's processing performance of these requests can depend greatly on the order in which they appear within the GPU buffers. Concerning an efficient modular exponentiation implementation, the code path taken is largely dependent on the exponent. When the GPU executes modular exponentiations with different exponents within

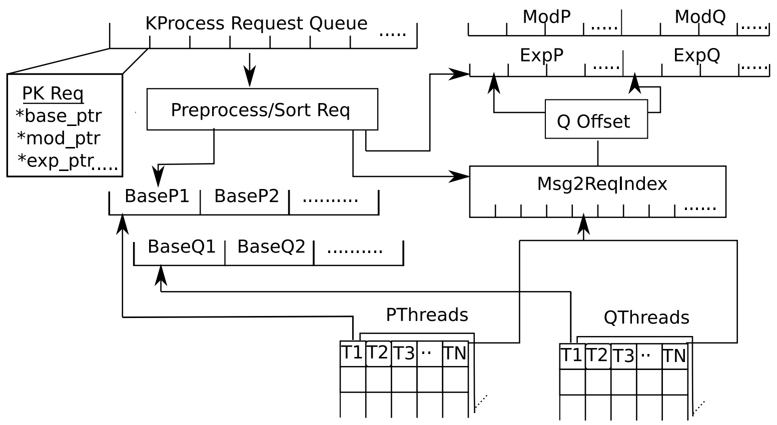

Fig. 9: Mechanism for Processing Multiple Distinct Asymmetric Key Requests.

the same CUDA warp, we experience thread divergence and a cost is incurred. This is due to having to execute the separate code paths serially rather than concurrently, see Section 2.2. The more varied the exponents within a warp, the higher the warp cost, and thus the higher the CUDA block cost, up to a limit. We have measured the different warp costs for a GPU execution of a modular exponentiation in various divergent scenarios and the cost ranges between 1 and 2.5, where 1 is equal to the minimum run time of a non-divergent modular exponentiation. Ideally we would be able to efficiently take any array of varying sized and keyed requests and reorder them to derive the minimum total cost, or runtime.

We can draw a loose analogy between this problem and the perfect packing version of the 2-dimensional strip packing problem [18]. If we let the cost of each CUDA block become the height of an object, the width of the object is 1 and the width of the container is the number of available SMs, then we wish to minimise the height of the container holding all the objects. The analogy is not exact as we also have the added complexity that the height of each object, i.e. the block cost, can vary depending on how requests are ordered. To find the optimal solution to this is computationally impractical. However, we can use heuristics to arrive at a reasonable solution. If we first consider that the block cost increases whenever an exponent changes within the array of requests, we should sort all requests according to their exponent, thus creating a list of non-divergent groups of requests. We perform this sort by an approximation, using only the first integer of the exponent, giving a good accuracy/efficiency trade off. We label this approach as “1 pass”.

We do not have control over the order in which the Nvidia driver chooses its CUDA blocks for execution when an SM becomes free, however it is reasonable to assume it follows a first fit approach, i.e. whenever an SM is free it takes the next lowest block by ID and assigns it to the SM. A reasonable close to optimal approach to solving the strip packing problem is to use the first fit descending heuristic. We follow this heuristic by sorting the non-divergent groups in increasing order of the number of operations within each group. This ensures that the most costly CUDA blocks occur in the lower block IDs. We call this approach “2 pass” as it involves the 1 pass sort above and an extra sort.

Both the 1 and 2 pass techniques are contrasted with no sorting (0 pass) in various scenarios in Figure 10. The scenarios are run outside of the OCF as they concern asymmetric key batching on the GPU in general and not just in the context of the framework. The tests consisted of sending a multiple requests to the GPU for concurrent execution. The size of each request within each test was randomly chosen in a guided manner. The “Large” tests restricted the sizes of the requests (the number of bases per request, typically 100-300) to be high, the “Small” tests contained only small requests (typically 1-10) and the “Mixed” tests contained a random mixture of large and small request sizes. Each test was run with a varying probability for each request to be followed by a request with the same exponent and modulus, i.e. the same key. This is labelled “Collision Probability”, with 1 meaning all requests are using the same key and 512 meaning a 1 in 512 chance of two requests chosen at random from the test having the same key. This collision probability simulates a multithreaded environment sending requests to the OCF with differing numbers of system wide keys.

Figure 10 analyses the relative performance of the 0, 1 and 2 pass techniques within each scenario. It can be seen that the 0 pass approach underperforms in all scenarios. The 1 and 2 pass techniques mostly perform the same with a slight overhead noticeable for the 2 pass approach. However, the 2 pass approach substantially outperforms the 1 pass approach when the collision probability is low and there is a mix of request sizes. The performance improvement of 2 pass at a collision probability of 1/512 is 24%. Small requests are more costly than large requests as the rate of change of the exponent is higher. These small messages when mixed randomly between lower cost large requests, form a layer of thinly distributed costly warps. It is beneficial to move these costly small requests into a small number of high cost blocks (especially as the block cost converges on a limit), thus increasing the number of low cost blocks which can run concurrently and finish while the high costs blocks complete. We recommend the use of this 2 pass approach due to its better performance in this scenario and relatively small overhead in the general case.

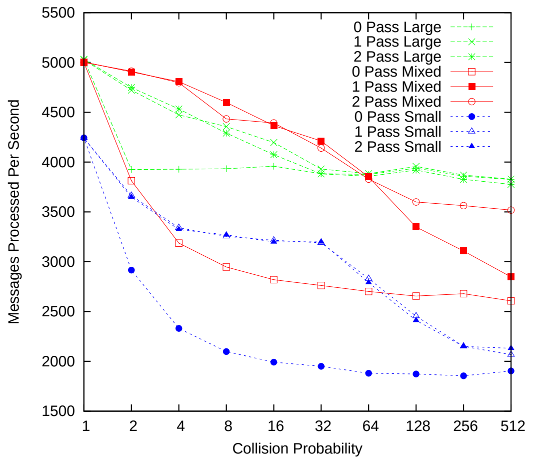

Fig. 10: Comparison of Pre-processing techniques for RSA-1024 Request Batching.

4.2 Request Pipelining

In the scenario where we have multiple cryptographic requests outstanding on the Gpucryptd daemon queue, we have the opportunity to split the processing and return of requests into two concurrent operations. The CUDA API allows for the execution of a kernel on the

GPU asynchronously. The Gpucryptd daemon permits both asymmetric and symmetric algorithms to retrieve more requests from the queue without returning. The daemon also supports callbacks to return completed requests. Thus, when implementing a cryptographic algorithm for the GPU, it is straight forward to overlap the return of previously completed requests with the execution of the next requests. We present the effects of pipelining in Section 5.1.

5 Performance

5.1 Symmetric Key Performance

To analyse the overhead of using the OCF for symmetric key performance, we use an AES implementation based on that presented in [1]. Figure 11 shows the performance of AES when operating on different sized buffers. We can see the non-OCF version of the implementation, labelled as "Standalone", performs comparably to [1]. We compare this standalone version to four other tests. Two tests were performed using normal userspace processes to initiate the requests and thus go via the Cryptodev layer of the OCF, labelled as "Cryptodev with/without MM". The remaining two tests were performed using a kernel thread which initiated the requests directly via the Crypto layer of the OCF, labelled "Crypto with/without MM". The "with MM" and "without MM" tags, refer to variants of the tests whereby we either include our new memory management system or use the original OCF memory management respectively.

We can see that the two tests which use the OCF and the new memory management system, perform with a small overhead compared to the standalone version. Based on the cryptodev interface, the average percentage overhead of using the OCF is 3.4%, with a range of 9.3% for the smallest request buffers through to 0.2% for the largest buffers. The spread in overhead percentage is due to the smaller request buffers requiring more calls through the OCF to perform the same amount of processing compared with larger request buffers. Although it cannot be seen here, there is a slight advantage to executing the cryptographic requests from the kernel as the Cryptodev layer overhead is removed.

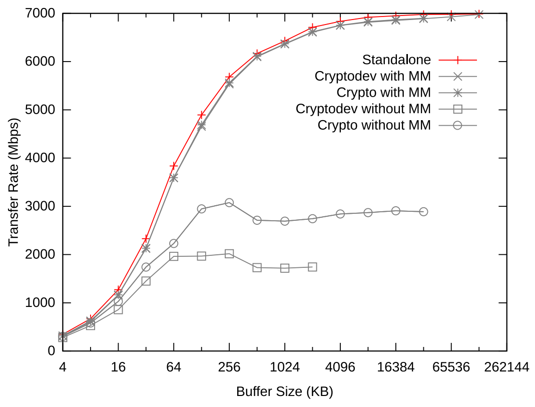

Fig. 11: OCF+GPU: AES Perf.

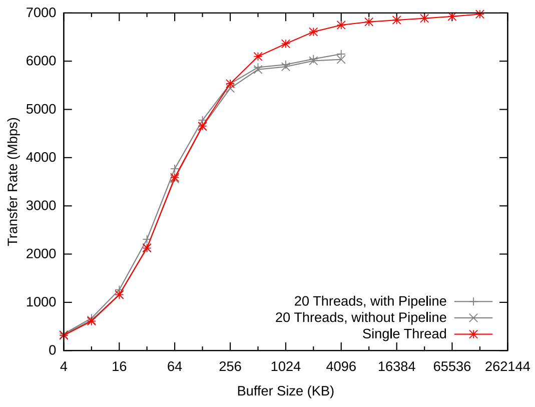

Fig. 12: OCF+GPU: Concurrency and Pipelining Perf.

The two tests which are performed without the new memory management system experience a substantial reduction in performance as the buffer sizes increase. This is due to having to perform extra memory copies for each transition between address spaces. The shape of the graphs can be understood when compared to Figure 4. The reason for "Crypto without MM" outperforming "Cryptodev without MM", is that the direct calls to the Crypto layer from kernelspace eliminates one of the address space transitions, thus reducing the number of copies performed. The reasons for the Crytpo and Cryptodev without MM tests being more limited in buffer sizes sampled is due to default limitations on vmap() and Cryptodev respectively. These can be bypassed, though it results in little difference.

Figure 12 is used to investigate both the effects of multithreaded symmetric key requests and pipelining as discussed in Section 4.2. The "Single Thread" test is the same as "Standalone" above, involving multiple iterations of symmetric key requests with varying buffer sizes. The multithreaded tests consist of executing the same amount of operations as the Single Thread test, however they are split across 20 threads. One of the test runs with the previously mentioned pipeline and can be seen to slightly out perform at smaller buffer sizes. It achieves this efficiency from asynchronously returning request results, thus hiding (or partially hiding) the return cost. Both of the multithreaded tests taper prematurely in performance as the buffer size increases, this is presumed due to the inefficiencies inherent in using larger buffers within a cached based system.

5.2 Asymmetric Key Performance

For our tests of asymmetric key performance we used an implementation based on the modular exponentiation approach for RSA-1024 presented in [9]. Figure 13 shows a comparison of running a standalone version of this implementation and using the implementation via the OCF Cryptodev layer from a userspace process. We can see here that there is no discernible difference in performance, in fact it is difficult to see there are two plotted graphs in the figure. This is due to the high arithmetic intensity inherent in the modular exponentiation algorithm and thus the OCF overhead is relatively very small. The average percentage overhead of using the OCF via the Cryptodev interface compared to the standalone version is 0.4%, with a range of 0.1% for the smallest number of messages to 0.6% for the largest. A related point is that we have performed these tests with and without the new memory management system and also with and without pipelining as in the symmetric key tests above. The results were indistinguishable from the standalone version due to the small overhead associated with data transfer through the OCF compared to the work done on the GPU.

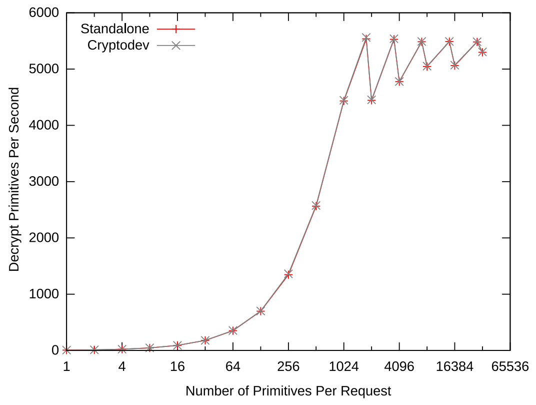

Fig. 13: OCF+GPU: RSA-1024 Single Thread Perf.

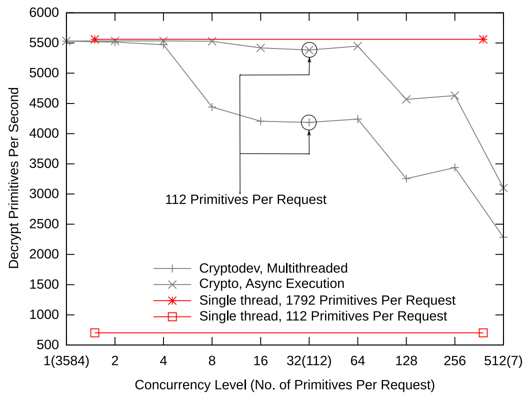

Fig. 14: OCF+GPU: RSA-1024 Concurrency Perf.

Figure 14 illustrates the behaviour of the OCF when processing multiple asymmetric key requests with the same key concurrently. We achieve concurrency by using multiple threads via the Cryptodev interface, as it is a blocking interface there is no other way a userspace thread can achieve concurrency. We also test concurrency via direct kernel calls to the Crypto layer, which permits asynchronous request execution and thus we can have multiple outstanding requests at one time. The Concurrency Level label refers to either the number of threads (Cryptodev test) or the number of concurrent requests sent asynchronously (Crypto test). All tests, both concurrent and single threaded illustrated perform the same total number of asymmetric operations, hence as the concurrency increases the number of messages per request decreases (shown in brackets on the x-axis). We have highlighted the performance improvement when processing requests size of 128 primitives concurrently versus serially. This improvement is due to the use of batching as described in Section 4.1. The main cause of the performance degradation when processing requests concurrently, is the inability to maintain an occupied GPU. In the tests, as the concurrency increases the request sizes decrease, thus the OS has a harder time to deliver sufficient numbers of requests to the queue for batching. The reason the Crypto test outperforms the Cryptodev test is that the OS does not have to reschedule processes as frequently to deliver the same amount of data to the GPU.

6 Conclusions

We have seen that the GPU can be effectively integrated into the OCF with careful design of a driver consisting of a kernelspace OCF driver and a userspace daemon. The paper shows that there is an average overhead of 3.4% when using the OCF for AES over a standalone implementation. In the context of RSA-1024 we see that there is a very low 0.3% average overhead when compared to a standalone version. A new memory management system within the OCF was shown to be critical to maintaining this performance for symmetric key operations. Without its use we see a drop in performance of over 50% when using the OCF's kernelspace Crypto interface, and over a 70% drop when the OCF's userspace Cryptodev interface is used.

We presented a new general purpose mechanism for processing multiple asymmetric key requests on the GPU and found that the preprocessing of mixed key requests is crucial to maintaining performance. We have also shown the effectivness of integrating this mechanism as part of the OCF and its use within multithreaded and asynchronous scenarios. The most important factor regarding performance in these scenarios is the ability of the OS to schedule multiple threads efficiently so as to provide enough work for the GPU to reach peak performance. We conclude that GPU accelerated cryptographic functions can be made available in a uniform manner to all OS components, both in-kernel and userspace, without excessive overhead.

APPENDIX A: OCF New Memory Management Interface

CRYPTO Interface:

crypto_alloc(): pass in the size and optionally the device ID for allocation. Returns a kernelspace pointer and the device which actually performed the allocation. If no device driver supports the new memory management system, or the specified device driver does not, then

null is returned.
crypto_free(): pass in a pointer returned by crypto_alloc().
crypto_translate(): pass in a pointer returned by crypto_alloc(). Returns a device space pointer if a mapping is found.

CRYPTODEV IOCTL Interface:
CIOCALLOC: this takes in an allocation request structure as a parameter, which specifies the requested buffer size and suggested device ID. The same structure is used to return the userspace pointer and actual device used for the allocation.

APPENDIX B: Gpucrypt IOCTL Interface:

GPU_REGISTER_*_REQ_BUF: this is a series of ioctls which Gpucryptd driver executes on startup to register shared request queues for each type of OCF request supported by the GPU driver.*
GPU_READY: after the request buffers are created, shared and initialised and all state is ready for operation, the Gpucryptd daemon registers itself as ready for work with the Gpucrypt driver. The driver, on receipt of this ioctl, issues a register command to the OCF to inform it that it is ready to start receiving requests.
GPU_WAIT_FOR_WORK: the Gpucryptd daemon process cycles through the request queues, continually processing any available work. When there are no more requests to process, rather than continually scanning it calls the Gpucrypt driver to wait for work using this ioctl. On receipt of this request the Gpucrypt sleeps the calling process on a kernel wait queue. When work is subsequently received from the OCF, the Gpucryptd is woken by calling wake on this wait queue, thus releasing Gpucryptd to finish the rest of the ioctl and return to userspace to process the new work.
GPU_RETURN_*_REQ: on finishing of a request, the Gpucryptd daemon uses this series of ioctls to deliver the work back to the OCF. This ioctl calls the OCF crypto_done() function which can either process the registered callback function for the request immediately or allow the OCF return queue kernel thread to do so later. One note worth mentioning is that an application which has a long callback function may resist setting the cryptographic request to execute an immediate callback as the callback is normally run in interrupt context. However, when the GPU is used, crypto_done is called from within process context, specifically the Gpucryptd context, and thus long callback functions are less problematic and thus the use of the separate return queue kernel thread can be avoided.*
GPU_SHUTDOWN: this ioctl is called on shutdown, which in turn unregisters the Gpucrypt driver from the OCF.
*The series stated above refer to ALLOC, FREE, KPROCESS (asymmetric request processing) and PROCESS (symmetric request processing).

References

1. O. Harrison and J. Waldron, "Practical Symmetric Key Cryptography on Modern Graphics Hardware". 17th USENIX Security Symposium. San Jose, CA. July 28 - August 1, 2008.
2. J. Yang and J. Goodman, "Symmetric Key Cryptography on Modern Graphics Hardware". ASIACRYPT 2007. Kuching, Malaysia. December 2-6, 2007.

3. Y. Yeom, Y. Cho and M. Yung, "High-Speed Implementations of Block Cipher ARIA Using Graphics Processing Units". 2008 International Conference on Multimedia and Ubiquitous Engineering. Busan, Korea. April 24-26, 2008.
4. S.A. Manavski, "CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography". IEEE International Conference on Signal Processing and Communications, 2007. ICSPC 2007. Dubai. November 24-27, 2007.
5. O. Harrison and J. Waldron, "AES Encryption Implementation and Analysis on Commodity Graphics Processing Units". CHES 2007. Vienna, Austria. September 10-13, 2007.
6. R. Szerwinski and T. Güneysu, "Exploiting the Power of GPUs for Asymmetric Cryptography". 10th International Workshop on Cryptographic Hardware and Embedded Systems. Washington DC, USA. August 10-13, 2008.
7. A. Moss, D. Page and N.P. Smart, "Toward Acceleration of RSA Using 3D Graphics Hardware". Eleventh IMA International Conference on Cryptography and Coding. Cirencester, UK. December 18-20, 2007.
8. S. Fleissner, "GPU-Accelerated Montgomery Exponentiation". Computational Science ICCS 2007, 7th International Conference. Beijing, China. May 27-30, 2007.
9. O. Harrison and J. Waldron, "Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware". Africacrypt 2009. Gammarth, Tunisia. June 21-25, 2009.
10. Nvidia Corporation, "CUDA", `http://developer.nvidia.com/object/cuda.html`.
11. A. Keromytis, J. Wright, and T. de Raadt, "The Design of the OpenBSD Cryptographic Framework. USENIX Annual Technical Conference 2003. San Antonio, Texas, USA. June 9-14, 2003.
12. S. J. Leffler, "Cryptographic device support for FreeBSD". Usenix, BSD Conference 2003. San Mateo, California, USA. September 8-12, 2003.
13. "OCF-Linux Project Homepage" - `http://ocf-linux.sourceforge.net/`.
14. "linux-crypto (Crypto API)" - `http://mail.nl.linux.org/linux-crypto/`.
15. U. Rosenberg. "Using Graphic Processing Unit in Block Cipher Calculations". University of Tartu. `http://math.ut.ee/ uraes/openssl-gpu/`.
16. Microsoft, "Direct X Technology", `http://msdn.microsoft.com/directx/`.
17. J-J. Quisquater and C. Couvreur, "Fast Decipherment Algorithm for RSA Public-Key Cryptosystem". Electronics Letters, Vol. 18, No. 21, Pages 905-907, 1982.
18. M. C. Riffa, X. Bonnairea and B. Neveub, "A revision of recent approaches for two-dimensional strip-packing problems". Engineering Applications of Artificial Intelligence. Volume 22, Issues 4-5, June 2009, Pages 823-827.