An hybrid AES-256-GCM implementation for NEON CPU & CUDA GPU

Romain Dolbeau
Rennes, France
Email: romain@dolbeau.org

***Abstract*—This paper is a work-in-progress.**

This paper describes & evaluates a fast, hybrid implementation of the Advanced Encryption Standard with 256 bit keys (AES-256) block encryption in Galois/Counter Mode (GCM). The implementation is bit-compatible with the implemented standard in both the OpenSSL and Crypto++ libraries, while significantly (up to three times) faster for large amount of data. In this implementation, a fast AES encryption function written in CUDA is combined with a fast GCM hash function written in ARM NEON intrinsics. The two are combined to execute as asynchronously as possible to maximize throughput. The full code and latest version of this paper are available from http://www.dolbeau.name/dolbeau/crypto/.

Revision 1.52, 2014/11/05 07:37:59

***Index Terms*—AES, Galois/Counter Mode (GCM), Authenticated Encryption with Associated Data, AEAD, GPU, CUDA, high-performance, NEON, Kepler, Tegra K1**

I. Introduction

In today's world, encryption and authentication of data have become an important aspect of network communications. Speed in this domain is crucial, as the usually compute-intensive cryptographic algorithms can quickly become a bottleneck for communication speed. In this paper, we describe and evaluate a fast implementation of the standard AES-256-GCM algorithm for Authenticated-encryption with Associated-data (AEAD [1]), targeted at system featuring an ARM core with NEON SIMD instructions and a CUDA-enabled GPU. GPUs are highly parallel compute engines, and therefore quite suited to many data-parallel workload. Leveraging such parallel power help improves compute efficiency and raw speed.

AES[2][3] with 256 bit keys (the work in this paper can be leveraged to any key size, but the larger key are more compute-intensive) is a block cipher. GCM (described by McGrew & Viega in [4] with a NIST recommendation by Dworkin in [5]) is a mode of operation for block cipher to create an AEAD algorithm from a block cipher. The standard mode of operation of AES-256-GCM is such that all encrypted blocks can be computed in parallel (in a fashion similar to the counter (CTR) mode of operation [6]). The XOR process of the encrypted data and the encrypted block is also parallel, each block only depend on a single AES block. The GCM hash function itself is sequential on each encrypted-and-xored block during the encryption phase. Figure 1 represents the dependency chains (red arrows denoting a dependence from one block upon another) between the computation of AES blocks (in blue), the XORing between AES blocks and encrypted data (in purple) and the computation of GCM hash (in green). Additional Authentication Data (AAD) are only hashed, not encrypted, a process which must take place prior to the hashing of the encrypted data since the output of the hash function on the last block of AAD is an input to the hash function on the first block of encrypted data. The output of the encryption algorithm is all the XORed blocks (purple) and the output of the last GCM block (last green) as a hash value. During the decryption & validation phase, the parallel aspect of AES and the sequential aspect of GCM remain. However, since encrypted data is available immediately, the hash function is not dependant on the result of the decryption phase (as shown in figure 2). The output of the decryption algorithm is all the XORed blocks (purple) and the output of the last GCM block (last green) for validating the hash value.

Because of such parallelism, the global scheme used to implement AES-256-GCM is to compute the AES blocks in parallel on the GPU and the sequential GCM blocks on the CPU. AAD can always be hashed in parallel of the AES computation. AES data have to be streamed back from the GPU to the CPU before the GCM phase during encryption, but can also be done in parallel during decryption.

The structure of this paper is as follows. After mentioning related work, we first present a description of the

Fig. 1. Dependency chains between blocks in AES-GCM encryption

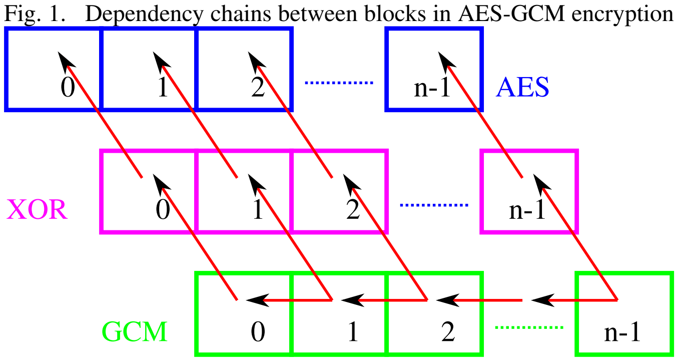
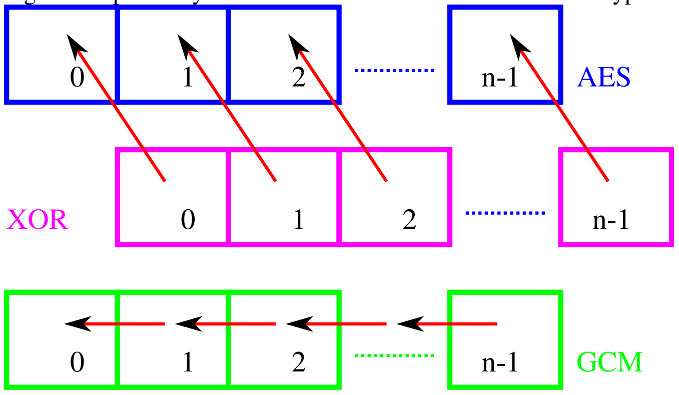


Fig. 2. Dependency chains between blocks in AES-GCM decryption



multiple AES implementation tested on the GPU. Then comes a brief description of the GCM implementation using NEON instruction. We finish with performance evaluation of the entire AES-256-GCM algorithm with or without AAD.

The full code and latest version of this paper are available from http://www.dolbeau.name/dolbeau/crypto/.

II. Related work

AES on GPU is nothing new, as it was already described in the book GPU Gems 3 [7] (chapter 36 is available online[1]) in 2007. The book already described the fact that the Electronic CodeBook (ECB) mode was unsafe, and that Cipher-Block Chaining (CBC) was not parallel. Multiple authors have tackled the issue of implementing AES on GPU since the release of the CUDA framework, among which Manavski [8], Bos et al. [9], Tomoiagă et al. [10][11], Li et al. [12], Iwai et al. [13], So-In et al. [14], and more.

AES in GCM mode has not received such an extensive body of work. Schönberger & Fuß [15] have implemented it using the same distribution of work between CPU and GPU. However, there is no precise description of the GPU or CPU code used. Additionally, the Intel CPU used did not support Intel instructions dedicated to AES, which achieve both high-performance and constant-time encryption (thus avoiding timings attacks when running AES on the CPU).

It seems most if not all published work involving GPU compare high-end GPU or even dedicated Tesla[2] accelerators to high-end desktop or server processors. In this work, we compare a embedded/mobile CPU to the same CPU accelerated by its built-in mobile GPU, the Tegra K1[3].

To ensure reliability of the code, every implementation benchmarked is validated against both the OpenSSL and Crypto++ libraries. In the GCM case, the interface used is that of the *supercop*[11] benchmark for AEAD algorithms for encryption and decryption.

III. AES in CUDA

A. Introduction

Rather than describe the specific implementation that seems optimal on the GPU included in the Tegra K1[3], we describe here every approach considered for every step of the algorithm. The C code uses the preprocessor to build the kernel functions, thus allowing every possible combinations of approach to be built with minimal code duplication.

NVidia GPUs supporting CUDA come in many flavors, with wildly different architectures. Such architectures are described by NVidia by “Compute Capabilities”, a set of specifications that is common to several model of GPUs (currently described in Appendix G[4] of the Cuda C Programming Guide). More information about the CUDA programming model[5] can be found starting at NVidia official page[6].

The original set of hardware supporting CUDA has Compute Capabilities 1.0 (CC1.0), such as the Tesla C870 and GeForce 8800GTX. This was quickly update to CC1.1 in the GeForce 8800GT, and later to CC1.3 which introduced double-precision floating-point support. Currently, three main families are available: CC2.0 and CC2.1 (code-name “Fermi”), CC3.0, CC3.2 and

[1] http://http.developer.nvidia.com/GPUGems3/gpugems3_ch36.html

[2] http://www.nvidia.com/object/tesla-supercomputing-solutions.html

[3] http://www.nvidia.com/object/tegra-k1-processor.html

[4] http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

[5] http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model

[6] http://www.nvidia.com/object/cuda_home_new.html

CC3.5 (code-name "Kepler"[16]) and CC5.0 (code-name "Maxwell"). Despite similar numbers, the specific CC is quite an important consideration. For instance, CC2.0 has large amount of double-precision hardware and can emit one instruction per cycle per scheduler, while 2.1 has a low amount of double-precision hardware and emit two instructions per cycle per scheduler. In the Kepler family that is the basis for the Tegra K1 GPU, CC3.0 is limited to 63 32-bits wide register per CUDA "thread" (same as CC2.x), while CC3.2 and CC3.5 can use up to 255 registers per "thread". CC3.0 also lacks the "funnel shifter" found in CC3.2 and CC3.5, an useful feature for cryptography since it allows single-instruction rotation of 32-bits registers.

It is therefore quite conceivable that the "best" implementation will be very architecture-specific or even GPU-specific, due to the varying tradeoffs in available instructions, available registers, memory bandwidth, memory sensitivity to access patterns[7], inter-"thread" shuffling of data[8], latency and bandwidth of the "shared" memory[9], the type and cost of bank conflicts in said "shared" memory, and so on.

The standard AES description [3] involves four steps for each AES round: *SubBytes* (commonly known by the data structure name S-Box), *ShiftRows*, *MixColumns* and *AddRoundKey*, all of which have been largely studied for optimizations on CPU such as by the AES inventors themselves, Bertoni et al. [17] and others, including implementors. In GCM mode, since the input data is not used as input to the block cipher but XORed with the output of the cipher, an extra *XORing* step after all AES rounds is to be considered. Also, as GPU have specific requirements for efficient "global" memory accesses, GPU implementation must also considers the *Loading* of inputs and *Storing* of results steps after the rounds. The following subsections describe the steps and the implementation opportunities on GPU.

A common optimization in CUDA (or with OpenCL) is to tweak the block size (and so the grid size). In the current implementations, only a block size of 256 CUDA "threads" per CUDA "block" is considered. The reason is the presence of several tables of 256 entries, which are trivially loaded in "shared" memory when using 256 threads. Lower number of threads would require multiple loads, whereas a large number of threads would require conditionals. Studying the effect of changing the block size is nonetheless of interest as a lower number of blocks would put less pressure on "shared" memory. That is not currently an issue for our implementations on Fermi (where an occupancy of 1.000 can be achieved with up to 8 KiB of "shared" memory per 256-threads block), but could be a small issue on Kepler (where an occupancy of 1.000 can be achieved with only up to 6 KiB of "shared" memory per 256-threads block). Studying the effect of changing the block size should be part of our future work.

B. Loading

The loading of data when using AES for direct encryption is at the beginning of the cipher, as the data themselves are encrypted. For mode such as CTR and GCM, the loading is only necessary immediately prior to the XORing step, as only a nonce or input vector (IV) and a counter are encrypted.

In both case, each AES block of 16 bytes has to be loaded. Here comes the first consideration: how many CUDA "threads" are going to be used to implement the block cipher. The most obvious answer is one, leading to 16 consecutive bytes of data being loaded for each thread. Two or four threads are also possible, sharing the workload but requiring inter-threads data exchange during the AES rounds. All three opportunities have been tested, although most of the test have been done with the seemingly more efficient first solution of one thread per block.

The 16 consecutive bytes per thread is an issue for optimal memory performance on GPU, as it is a suboptimal memory access pattern. A straightforward implementation of each thread loading each of the needed 32 bits in sequence would create a non-consecutive access pattern. It is better for consecutive "threads" to access consecutive elements in "global" memory, a process known as "coalescing". Three implementations are considered for this step:

1) The straightforward implementation. "Global" memory access is not very efficient, but has no additional resource requirements;
2) Reorganizing memory accesses using the "shared" memory. This is most common way of handling such data pattern in CUDA. "Global" memory access is very efficient, but a large amount of "shared" memory is required (4 KibiBytes for a

[7]http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#global-memory-2-x, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#global-memory-3-0

[8]http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-shuffle-functions

[9]http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory-2-x, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory-3-0

256 threads block) along with writing and reading to said "shared memory";

3) Reorganizing memory accesses using the inter-"thread" data shuffling available in "Kepler" (CC3.X) and later architecture. "Global" memory access is very efficient. While there is an overhead using the shuffling functions, no extra "shared" memory is required in this case. However, this is not usable on "Fermi" and earlier architectures.[10]

C. SubBytes (a.k.a. S-Box)

The substitution step replace each byte in the 16 bytes state of AES by the value in a 256 bytes look-up table, the AES S-Box. While the data inside the S-Box can be computed, the process is extremely computationally complex and it is not practically feasible to avoid the table look-ups during the algorithm, unless dedicated instructions exist in the hardware such as Intel's AES-NI. Such look-ups in memory cause issue even for security in addition to performance, as the cache behavior might allow timings attacks (e.g. Bernstein in [18]).

In practice, on GPU, the most efficient way for such random access to a small table is to pre-load the entire table in "shared" memory. This is far from perfect: it is conceivable that "shared" memory bank conflicts have timing issue that could be exploited, and those conflicts can be measured by hardware counter that also could be leveraged against the implementation.

We considered several possible implementations:

1) Only the standard AES S-Box is exploited. This can be either by using a standard 256 bytes table, or using a 1024 bytes tables putting each 8 bits value in its own 32 bits word - either could be better or worse than each other due to memory banking.
2) In addition to the standard S-Box, one to four 1024 bytes Forward Table are loaded in shared memory. Such Forward Table are described in paragraph 5.1.2 of [2], and combine the *SubBytes*, *ShiftRows* and *MixColumns* step in look-ups in the tables and full 32-bit XOR operation. As the last three tables are rotation of the first, is is possible to only use one or two tables combined with extra rotations. Those rotations are of 8, 16 or 24 bits and therefore can be implemented either with a true rotation (or failing that two shifts and a (X)OR) or a byte permutation instruction.

[10]Our code contains substitution functions using the "shared" memory to use this implementation on "Fermi"-class GPU, effectively creating an alternative implementation trade-offs between size and number of load/stores for the "shared" memory case.

D. ShiftRows

This diffusion step only rotates each of the 4 lines of 4 bytes by full bytes. This is seldom explicitly implemented, as it can easily be merged with the previous (when implementing the substitution, the looked-up bytes are inserted in the rotated position instead of the starting position) or the next (by changing the read index). Obviously, it is not used when the Forward Tables are used.

E. MixColumns

This step is another diffusion step, but this time implementation is more complex, as in theory it involves multiplications in $GF(2^8)$. In practice, as the matrix used only contains ones, twos and threes, the multiplication can be implemented by shifting, XORing and overflow testing.

This step can be merged in the Forward Table. If it isn't, then there is several ways to implement it.

1) 8 bits operations. Each byte is recomputed independently after extraction, and reinserted. This will include the *ShiftRows* step in the extraction part.
2) 32 bits operations. The $GF(2^8)$ multiplications and additions (XOR) are made by group of four inside 32-bits registers. The *ShiftRows* step is done first (unless merged in the *SubBytes* step), followed by times two multiplications, followed by the required additions. Finally the data are reorganized into the proper structure prior to the *AddRoundKey* step.
3) 32 bits operations on reorganized data. Same principle, but the data are not reorganized at the end, and the input is expected in the same "diagonalized" format. This requires to pre-"diagonalize" the subkeys (but only once for all AES blocks), but saves permutations for every AES rounds.

F. AddRoundKey

This step is simply XORing the round subkey with the current data. Even for the "diagonalized" *MixColumns* step which requires a specific format for the subkeys, this is just a XOR operation.

G. XORing

Another extremely simple step, used only in CTR and GCM mode. Input data and the output of the cipher are xored together. However, computing this on the

GPU requires the data on the GPU, using a potentially expensive `cudaMemcpy` API call. It is possible instead to not send the data to the GPU and computes the XORing along with the GCM computation on the CPU instead.

H. *Storing*

This is to store in "global" memory either the results of the ciphers or of the XORing of the cipher output with the input data. This is the same issue, with the same solutions, as the *Loading* step.

IV. GCM IN NEON

The Galois/Counter Mode (GCM)[4][5] is a relatively expensive operation in software, due to its reliance on multiplication in the Galois field $GF(2^{128})$. Since performance of GCM is critical to many cryptographic libraries, many implementations exists. On Intel and compatible processor, the instruction `pclmulqdq` is a very efficient building block as described by Gueron in [19]. Implementations exploiting the instructions when available are available in libraries such as OpenSSL [20] or Crypto++ [21], and along many more cryptographic implementations in the *supercop*[11] benchmark, part of eBACS [22].

For our particular implementation, we used not assembly code but built-in functions from the compiler[12] to access specific single-instruction multiple-data (SIMD, a.k.a. "vector") instructions from the NEON extension to the ARM architecture (whose suitability for high-performance cryptography was studied by Bernstein et al. in [23]). Rather than reinvent the wheel, we implemented the efficient algorithm invented for this very purpose by Câmara et al. in [24]. Specifically, we reimplemented Algorithm 6 and used it as a building block for the full GCM hash.

Tables 3 and 4 show performance values for several GCM implementations for various hardware, hashing 4 MiB and 128 MiB respectively. The first halves or each table are in a synthetic benchmark, the second halves are AES-256-GCM with no encrypted data (only associated data, so AES is not involved). ref. impl. is the C-only reference implementation from *supercop*[11]. The three pclmul lines are implementations using Intel's `pclmulqdq` instructions, with three level of unrolling. Unrolling allows for a more efficient implementation of GCM by factorizing some computations, at the cost of additional pre-computations (details are available in [25] and [19]). The last two are Crypto++ and OpenSSL. The hardware includes the Cortex A15 core from Jetson TK1; Core i7-920 ("Nehalem", without the `pclmulqdq` instruction); Xeon X5650 ("Westmere", which introduced the cryptographic instructions); and Core i5-4570S ("Haswell"). As can be seen in the table, our implementations are competitive on all supported platforms. OpenSSL still has an edge on the "Haswell" architecture. Our NEON implementation is faster by a wide margin than both reference implementations on the Cortex A15[13].

Fig. 3. GCM performance in MB/s for 4 MiB

Implementation	Cortex A15	i7-920	X5650	i5-4570S
ref. impl.	4.78	8.05	8.40	10.9
pclmul no unr.			735	1330
pclmul unroll 4			1420	3660
pclmul unroll 8			1430	5000
NEON	176			
(pclmul unr. 8)			1420	2670
(NEON)	176			
(Crypto++)	135	727	1490	1590
(OpenSSL)	69.3	368	1460	2810

Fig. 4. GCM performance in MB/s for 128 MiB

Implementation	Cortex A15	i7-920	X5650	i5-4570S
ref. impl.	4.78	8.00	8.36	10.9
pclmul no unr.			729	1330
pclmul unroll 4			1390	3660
pclmul unroll 8			1400	5170
NEON	174			
(pclmul unro 8)			1410	5180
(NEON)	181			
(Crypto++)	135	724	1440	3260
(OpenSSL)	69.5	368	1420	5830

V. HYBRID GCM MODE

The interface to use our GCM implementation is from the *supercop*[11] benchmark. Two functions are implemented, one for encryption and generation of the hash value, and one for decryption and validation.

A. *Encryption*

The general principle has been outlined in the introduction. Specifically, the sequence of computation is the following:

[11]http://bench.cr.yp.to/supercop.html

[12]The *supercop*[11] benchmark includes both assembly and higher-level implementations using built-in functions (a.k.a "intrinsics") of many algorithms, showing that assembly coding is not required for high performance. However, the performance is then heavily reliant on the compiler, since the compiler is responsible for e.g. instructions scheduling and registers allocation.

[13]However, the OpenSSL implementation is faster than ours when running on the previous generation Cortex A8 core, for which it was originally designed.

- AES subkeys are generated on the CPU. If the GPU implementation requires "diagonalization" of the key, then this is also done on the CPU;
- Only full CUDA blocks are used, each dealing with 4096 bytes of input data (256 AES block). The size of the compute grid is computed a this point, along with how much data will be left for the CPU to encrypt;
- The required data (subkeys, nonce, counter, and possibly the input data if the XORing step is done on the GPU) are sent to the GPU memory;
- Kernel is launched asynchronously on the GPU to compute AES blocks inside GPU memory;
- All the leftovers data not encrypted by the GPU are now encrypted and XORed on the CPU (while the GPU kernel is running);
- All the AAD are hashed on the CPU (while the GPU kernel is running);
- Encrypted data are recovered from GPU to CPU;
- Encrypted data & final GCM block are hashed on the CPU; hashing will also handle XORing if that was not done on the GPU.

We call this implementation "single-call". Clearly, that process is suboptimal, since no encrypted data are hashed while the GPU encryption is going on. An alternative implementation called "by chunks" can do this by splitting the GPU kernel launch into multiple "chunks". Each successive "chunk" can then be recovered as soon as they are finished (using CUDA "events" for GPU-CPU synchronization), thus allowing pipelining of AES and GCM computations. Along with the choice of which compute engine (CPU or GPU) does the XORing, this creates four different implementations. Of course, any of the possible kernels described in section III can be used for the GPU computations.

B. Decryption

The decryption process is quite similar to the encryption, but with more opportunities for parallelism.

- AES subkeys are generated on the CPU.
- Only full CUDA blocks are used, each dealing with 4096 bytes of input data (256 AES block). The size of the compute grid is computed a this point, along with how much data will be left for the CPU to encrypt (GCM does not require AES decryption, only encryption of blocks);
- The required data (subkeys, nonce, counter, input data) are sent to the GPU memory;
- Kernel is launched asynchronously on the GPU to compute AES blocks inside GPU memory;
- All data are hashed (AAD then encrypted then final block) on the CPU (while the GPU kernel is running);
- If the hash is validated, decrypted data are recovered from GPU to CPU. Otherwise, the decrypted data in GPU and CPU memory are zeroed instead.

At the moment, only a single variant of the decryption process is implemented.

VI. Performance of AES-256 in CTR and GCM mode

Although GCM is the goal, AES in CTR mode uses exactly the same building blocks to create the GPU kernels. Note that the Tegra K1 system (a Jetson TK1) shows performance instability when benchmarking a large number of kernels, unless the frequency of the GPU is set (by default the GPU is in power-saving mode and frequency can change, the procedure is explained in [26]) . So first we study the AES kernels in CTR mode on a GPU based on the "Fermi" architecture. Our host for the "Fermi" card unfortunately does not support `pclmulqdq`, so only CTR mode is tested in that environment.

A. GeForce GT 620 "Fermi"

This is a low-end card of Compute Capabilities 2.1 (details in appendix VII). As a low-power desktop cards, it does not exhibit any peculiarities in performance due to power or thermal constraints. It produces highly reproducible results from one run to another, which is convenient for benchmarking a large number of kernels. The code as of this writing benchmarks 220 different AES kernels based on the one-block-per-thread approach. In every case, XOR is made on the GPU, this aspect will be studied in the actual GCM code. The *Loading* and *Storing* stage do not use the inter-"threads" shuffles, as they are not available on the Fermi architecture (they represent an extra 55 tests). These tests are made to evaluate the relative performances of the various solutions that can be used to implement AES in CUDA.

Table 5 is an extract from the full results. The columns are in the order GPU time in milliseconds (sort key), kernel name (with the prefix `aes_ctr_cuda_` removed), number of registers used, amount of "shared" memory used, and occupancy. The number were obtained via the command-line profiler, they are not the exact number seen from the CPU. However, they are more accurate to compare in-GPU kernel performances.

Fig. 5. AES-CTR kernels on GT620 (extracts)

Time (ms)	Kernel	regs	shared	occup.
122629.375	BTB32SRDIAGKEY0_PRMT_8nocoalnocoal	20	520	1.000
123738.594	BTB32SRDIAGKEY0_PRMT_8coalcoal	22	4616	0.833
125767.617	BTB32DIAGKEY0_PRMT_8coalcoal	23	4616	0.833
125792.547	BTB32SRDIAGKEY0_PRMT_8coalnocoal	21	4616	0.833
126205.664	BTB32DIAGKEY0_PRMT_8nocoalcoal	21	4616	0.833
…	…	…	…	…
140236.516	FT_INT4_PRMT_8coalcoal	24	8712	0.833
140317.219	FT_SEQ4_PRMT_8coalcoal	23	8712	0.833
…	…	…	…	…
141986.375	FT_INT1_PRMT_8coalnocoal	22	5640	0.833
142098.938	FT_SEQ1_PRMT_8nocoalnocoal	21	1544	0.833
…	…	…	…	…

The fastest implementation - by a very short margin - is the kernel cryptically named “BTB32SRDIAGKEY0_PRMT_8nocoalnocoal”. This is not using any Forward Tables (FTs-using kernels are denoted by starting with `FT_` and the number 1, 2 or 4 prior to the `_PRMT` token, indicating the number of tables). This particular kernel uses no explicit coalescing for either the *Loading* or *Storing* step, uses a 8-bits S-Box table, uses permutation instruction rather than C code for the very last round (which does not include the *MixColumns* step), merge the *ShiftRows* step into the *SubBytes* step, and uses 32-bits operation with pre-“diagonalized” subkeys for the *MixColumns* step. Note that all implementations and the macro to build them are available in the code.

The second fastest is the same, but using “shared” memory-based memory coalescing. As can be seen, this second kernel uses more shared memory than the first per block. This is actually not a problem in this case. As can be seen in Appendix A: GeForce GT 620, this GPU has a limit of 1536 threads per multiprocessor (SM), 48 KiB of “shared” memory per SM, and at most 32768 registers available per SM. Since all kernels above uses 256 threads per block, at most 6 blocks can run simultaneously: an occupancy of 1.000 in CUDA terms. As long as each block uses less than 8 KiB, “shared” memory is not a limit. However, for 1536 threads, only 21 registers can be used in each thread for full occupancy (in practice the limit is 20 as the granularity of allocation is not one). Since the second kernel needs 22 registers versus 20 for the first, occupancy drops to $5/6$ or 0.833.

The third fastest is the same as the second, but the *ShiftRows* is no longer merged with the *SubBytes* step. Instead, it uses three explicit permutation to achieve the same result. The fourth is the second with no coalescing for the store - so the benefits is lost for stores, but the cost of allocating and addressing the shared memory is still here, hence the lower speed (higher time) than the second-fastest kernel.

The table goes on and on, mostly with the expected results. Forward Tables are not more efficient than computations on GPU, since they cause many costly “shared” memory bank conflicts. The number of tables has little influence, since even with four tables and coalescing the 48 KiB shared memory still allows up to 0.833 occupancy, the level to which register pressure usually limits kernels anyway (a recurring issue with Fermi-based hardware). The cost of rotation associated with a lower number of tables is also very low thanks to the single-instruction byte permutation. 32-bits computation are much faster for the *MixColumns* step than 8-bits. Memory coalescing does not offer a significant advantage when accessing the GPU memory, since the original pattern is cache-friendly, implementing coalescing induces overheads, and AES is very compute-intensive anyway. However, the story is extremely different if the memory is not the GPU memory, but page-locked memory (“pinned” memory) on the host - accesses over the PCI express bus in this case requires coalescing to avoid an extremely large performance penalty, as shown by results in figure 6 where the top two kernels have coalesced *Loading* and the last two do not. Coalescing in *Loading* (enabled in the first and third entry) is much less dramatic in performance.

Fig. 6. AES-CTR kernels on GT620 in “pinned” memory

Time (ms)	Kernel	regs	shared	occup.
123811.875	BTB32SRDIAGKEY0_PRMT_8coalcoal	22	4616	0.833
132751.109	BTB32SRDIAGKEY0_PRMT_8nocoalcoal	22	4616	0.833
1091020.625	BTB32SRDIAGKEY0_PRMT_8coalnocoal	21	4616	0.833
1094071.625	BTB32SRDIAGKEY0_PRMT_8nocoalnocoal	20	520	1.000

B. Tegra K1 “Kepler”

The Tegra TK1 System-on-Chip (SoC) includes four Cortex A15 cores, a low-power companion core (the Linux kernel only reports four usable cores), and a “Kepler” GPU described in Appendix B: Tegra K1. The SoC is designed for mobile device, and includes many power-saving features. Unfortunately, and perhaps because of those power-saving features, the performance observed when benchmarking kernels is not reliable. That is, the same binary using the same data might show different performance (different time measured by the command-line profiler) for the same kernel, with variation by a factor of over 2. It is therefore necessary to first fix the GPU frequency before benchmarking (see [26]). The speed was set at the maximum supported frequency of 852 MHz for the GPU (gbus) and 924 MHz for the memory (emc).

1) AES in CTR mode: The "Kepler" architecture can reach full occupancy with a higher number of registers per threads, since it has 65536 registers for up to 2048 threads per multiprocessors. This can be seen in the partial results in table 7, where kernels with up to 32 registers have an occupancy of one. However, some kernels will be limited by the shared memory, such as "FT_SEQ2_PRMT_8coalcoal" which uses more than the maximum of 6 KiB per blocks when using 256 threads per blocks (this kernel needs two forward tables and the coalescing buffer in shared memory). Unlike on the Fermi architecture, the fastest kernels use coalescing (using shared memory).

As with the GT620, kernels can work in host memory rather than the GPU memory, results are shown in table 8. However, unlike a discrete graphic card, the memory is physically the same for the host and the GPU in the Tegra K1 (as in most system-on-chip). From this sharing, one could hope for a more efficient implementation. In practice, results are mixed: while the worst-case scenario isn't not as bad as it was over the PCI bus in the GT620, the best-case scenario is more than twice as slow as it was in GPU memory. Also, it seems coalescing the loads is more important than coalescing the stores in this case.

Fig. 7. AES-CTR kernels on Tegra K1 (extracts)

Time (ms)	Kernel	regs	shared	occup.
105252.977	BTB32SRDIAGKEY0_PRMT_8coalcoal	30	4612	1.000
105481.469	BTB32SRDIAGKEY0_C_8coalcoal	30	4612	1.000
108210.227	BTB32DIAGKEY0_PRMT_8coalcoal	31	4612	1.000
109460.227	BTB32DIAGKEY0_C_8coalcoal	31	4612	1.000
113932.891	BTB32DIAGKEY0_PRMT_8coalnocoal	32	4612	1.000
…	…	…	…	…
211367.859	BTB0_PRMT8AS32_8coalcoal	34	4612	0.750
213734.531	BTB0_PRMT8AS32_8coalnocoal	35	4612	0.750
…	…	…	…	…
220478.938	FT_INT1_C_8coalcoal	37	5636	0.750
221382.859	FT_SEQ2_PRMT_8coalcoal	32	6660	0.875
…	…	…	…	…

Fig. 8. AES-CTR kernels on Tegra K1 in "pinned" memory

Time (ms)	Kernel	regs	shared	occup.
244830.016	BTB32SRDIAGKEY0_PRMT_8coalcoal	30	4612	1.000
261703.688	BTB32SRDIAGKEY0_PRMT_8coalshufcoalshuf	35	516	0.750
277638.094	BTB32SRDIAGKEY0_PRMT_8coalnocoal	32	4612	1.000
278203.781	BTB32SRDIAGKEY0_PRMT_8coalshufnocoal	31	516	1.000
290254.031	BTB32SRDIAGKEY0_PRMT_8nocoalnocoal	29	516	1.000
529112.625	BTB32SRDIAGKEY0_PRMT_8nocoalcoalshuf	34	516	0.750
566771.188	BTB32SRDIAGKEY0_PRMT_8nocoalcoal	34	4612	0.750

2) AES in GCM mode: A summary of performance for the four implementation of encryption described in section V-A, plotting the speed against the size of the data encrypted, can be seen in figure 10. The figure also includes the two reference libraries. The decryption speed is shown in figure **??**.

3) Single-call GCM, no AAD: This is the trivial variant, doing all of AES on GPU in one call. As can be seen in the figures, both encryption and decryption are faster than the OpenSSL library (itself faster than Crypto++) at sizes of 64 KiB and larger. Despite doing an identical amount of work, decryption is faster as it can fully overlap AES and GCM as explained in section V.

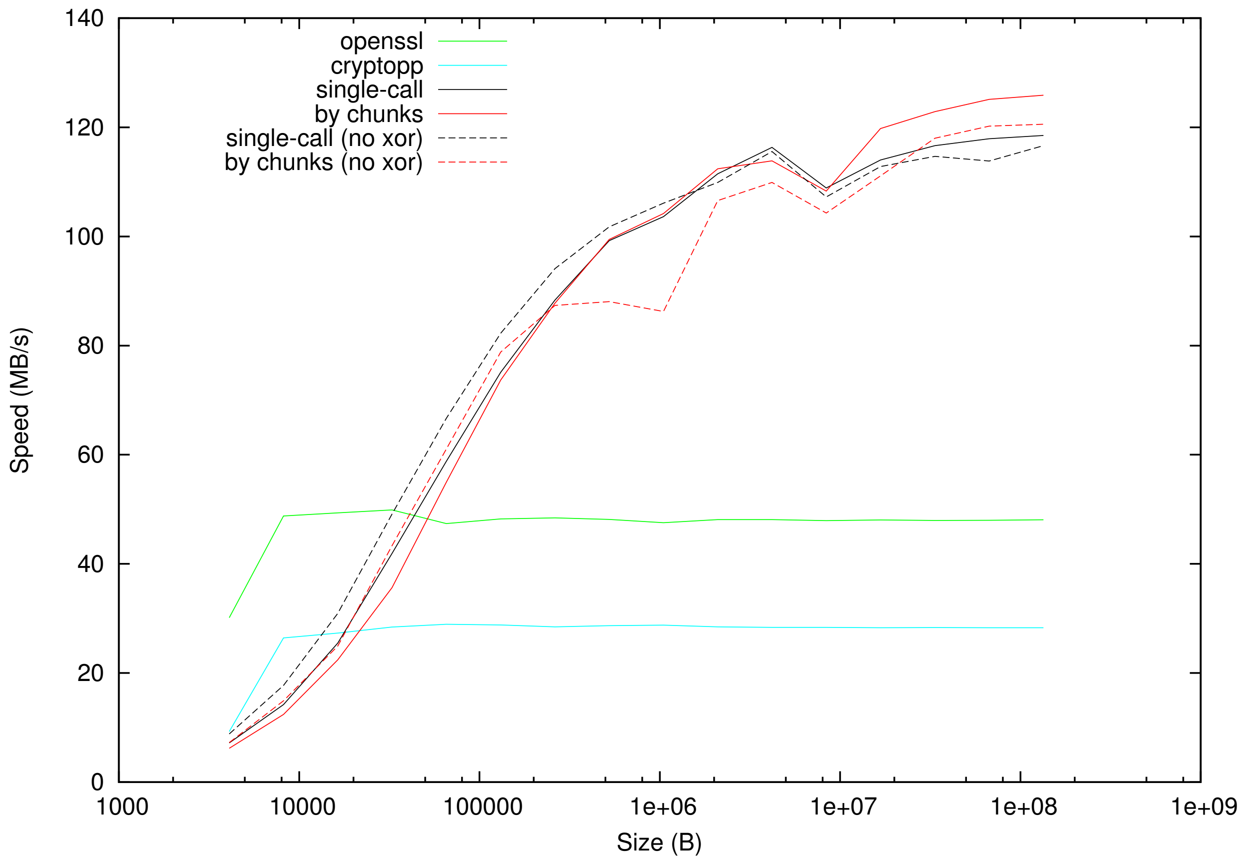
4) By chunks GCM, no AAD: This variant tries to overlap AES and GCM in encryption by cutting data into "chunks". Specifically, for this run, the code tries to cut the data in 8 "chunks" of at least 512 KiB and at most 8 MiB. Any size of 512 KiB or less will only use one chunk; from 1024 KiB to 2 MiB from 2 to 4 chunks of 512 KiB; from 4 MiB to 64 MiB 8 chunks of 512 KiB to 8 MiB ; and 128 MiB uses 16 chunks of 8 MiB. This seems to be a good compromise, though a full exploration of the parameter space was not performed. As can be seen, the performance improves significantly for encryption of 2 MiB and more. Lower sizes are impacted by the higher overheads of this implementation. Decryption performance is the same, except for measurement errors.

5) Effects of AAD: Adding an identical amount of Additional Authenticated Data requires a significant amount of CPU work, since only GCM is applied to AAD. When the GPU runs at full speed, it is capable of doing AES in counter mode at a faster rate (over a gigabyte per seconds) than a single core can do GCM (at around 180 megabyte per second). Adding a large amount of AAD makes the computation even more limited by the ability of a core to do GCM. Since there is always some GCM to do while AES in running, this results in similar speed for encryption and decryption, and negates the advantage of using "chunks" for AES.

VII. Conclusion & future work

In this paper, we introduce first an extensive study of possible AES implementations using CUDA. We then introduce a full hybrid implementation of AES-256 in GCM mode built on top of our high-performance implementation of AES and the best published implementation of GCM in NEON. We show that this hybrid implementation is significantly faster than two state-of-the-art CPU implementations in two widely used libraries, with or without Additional Authenticated Data, for both encryption & decryption, as long as enough data has to be encrypted to justify the overhead of exploiting the GPU.

Fig. 9. Performance of encryption with no AAD



As mentioned, changing the "block" size might have an influence on AES performance and should be studied. Using more than one CUDA "threads" to compute each AES block should also be more thoroughly evaluated. Optimal "Chunk" size could also be studied, along with the effect of using "pinned" memory on the host to allow asynchronous transfers. Encrypting a large amount of data with a single key in a single call is not the most commonly benchmarked aspect of AEAD encryption, where small data size is the norm. Future work will include throughput measurement for smaller data sizes.

REFERENCES

[1] P. Rogaway, "Authenticated-encryption with associated-data," in Proceedings of the 9th ACM Conference on Computer and Communications Security, ser. CCS '02. New York, NY, USA: ACM, 2002, pp. 98–107. [Online]. Available: http://doi.acm.org/10.1145/586110.586125

[2] J. Daemen and V. Rijmen, "AES proposal: Rijndael," 1998. [Online]. Available: http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf

[3] N. I. of Standards and Technology, "Advanced encryption standard," NIST FIPS PUB 197, 2001. [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[4] D. A. McGrew and J. Viega, "The galois/counter mode of operation (GCM)," NIST Modes Operation Symmetric Key Block Ciphers, 2005. [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-revised-spec.pdf

[5] M. J. Dworkin, "Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC," Gaithersburg, MD, United States, Tech. Rep., 2007.

[6] H. Lipmaa, P. Rogaway, and D. Wagner, "CTR-mode encryption."

[7] H. Nguyen, Gpu Gems 3, 1st ed. Addison-Wesley Professional, 2007.

[8] S. Manavski, "Cuda compatible gpu as an efficient hardware accelerator for aes cryptography," in Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on, Nov 2007, pp. 65–68.

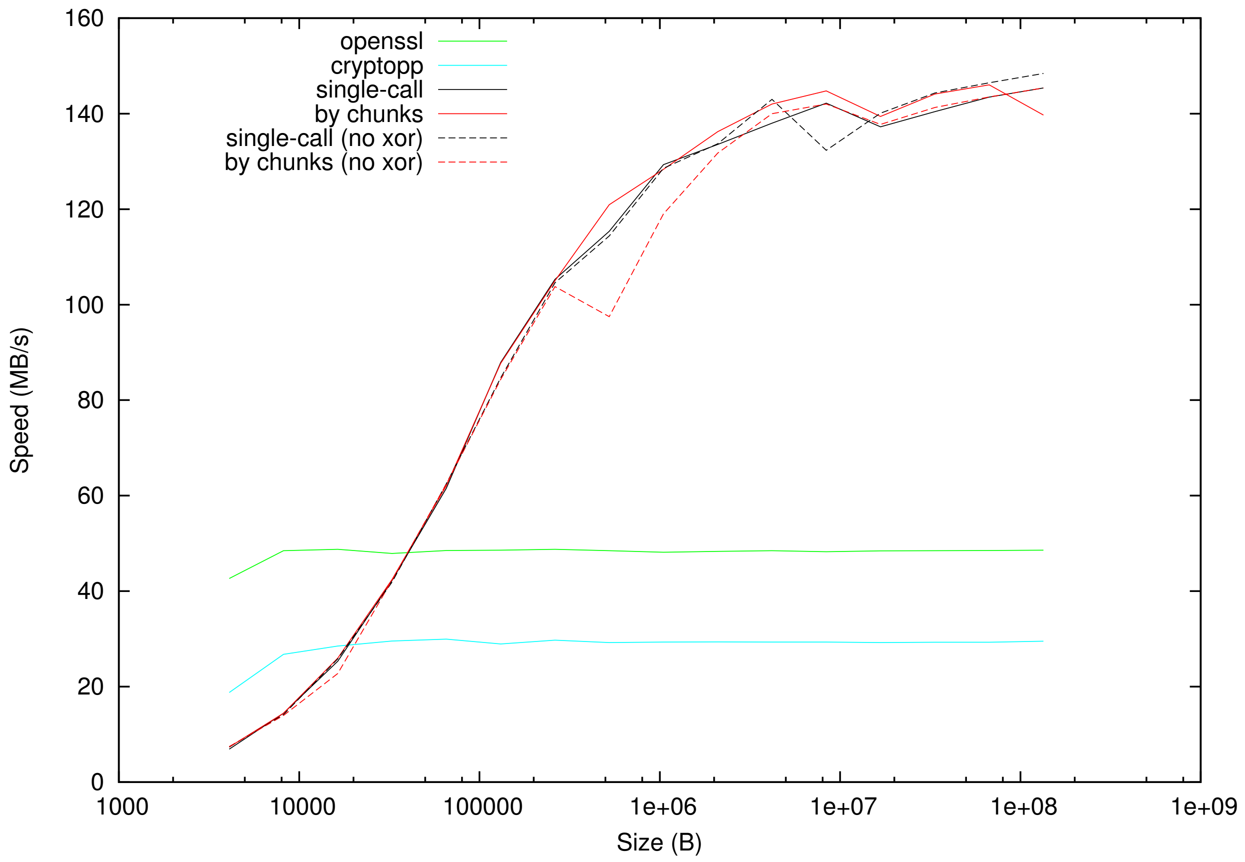
[9] J. W. Bos, D. A. Osvik, and D. Stefan, "Fast implementations of AES on various platforms," IACR Cryptology ePrint Archive, vol. 2009, p. 501, 2009. [Online]. Available: http://eprint.iacr.org/2009/501

[10] D.-R. Tomoiagă and M. Stratulat, "AES on GPU using CUDA."

[11] ——, "AES algorithm adapted on GPU using CUDA for small data and large data volume encryption," International Journal of Applied Mathematics and Informatics, vol. 5, 2011.

[12] Q. Li, C. Zhong, K. Zhao, X. Mei, and X. Chu, "Implementation and analysis of aes encryption on gpu," in High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems

Fig. 10. Performance of decryption with no AAD



(HPCC-ICESS), 2012 IEEE 14th International Conference on, June 2012, pp. 843–848.

[13] K. Iwai, N. Nishikawa, and T. Kurokawa, "Acceleration of aes encryption on cuda gpu," International Journal of Networking and Computing, vol. 2, no. 1, pp. 131–145, 2012. [Online]. Available: http://www.ijnc.org/index.php/ijnc/article/view/38

[14] C. So-In, S. Poolsanguan, C. Poonriboon, K. Rujirakul, and C. Phudphut, "Performance evaluation of parallel AES implementations over CUDA GPU framework," International Journal of Digital Content Technology and its Application, vol. 7, pp. 501–511, 2013.

[15] G. Schönberger and J. Fuß, "GPU-assisted AES encryption using GCM," in Proceedings of the 12th IFIP TC 6/TC 11 International Conference on Communications and Multimedia Security, ser. CMS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 178–185. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24712-5_16

[16] NVIDIA, "Kepler GK110 whitepaper," 2012. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[17] G. Bertoni, L. Breveglieri, P. Fragneto, M. Macchetti, and S. Marchesin, "Efficient software implementation of aes on 32-bit platforms," in Cryptographic Hardware and Embedded Systems - CHES 2002, ser. Lecture Notes in Computer Science, B. Kaliski, . Ko, and C. Paar, Eds. Springer Berlin Heidelberg, 2003, vol. 2523, pp. 159–171. [Online]. Available: http://dx.doi.org/10.1007/3-540-36400-5_13

[18] D. J. Bernstein, "Cache-timing attacks on aes," Tech. Rep., 2005. [Online]. Available: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[19] S. Gueron and M. E. Kounavis, "Intel® carry-less multiplication instruction and its usage for computing the GCM mode," 2010. [Online]. Available: https://software.intel.com/en-us/articles/intel-carry-less-multiplication-instruction-and-its-usage-for-computing-the-g

[20] The OpenSSL Project, "OpenSSL: The open source toolkit for SSL/TLS," April 2003, www.openssl.org. [Online]. Available: http://www.openssl.org/

[21] W. Dai, "Crypto++®library," 2010. [Online]. Available: http://www.cryptopp.com/

[22] D. J. Bernstein and T. Lange, "eBACS: ECRYPT benchmarking of cryptographic systems," 2009. [Online]. Available: http://bench.cr.yp.to

[23] D. J. Bernstein and P. Schwabe, "NEON crypto," in Cryptographic Hardware and Embedded Systems - CHES 2012, ser. Lecture Notes in Computer Science, E. Prouff and P. Schaumont, Eds. Springer Berlin Heidelberg, 2012, vol. 7428, pp. 320–339. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33027-8_19

[24] D. Câmara, C. P. Gouvêa, J. López, and R. Dahab, "Fast software polynomial multiplication on ARM processors using the NEON engine," in Security Engineering and Intelligence Informatics, ser. Lecture Notes in Computer Science, A. Cuzzocrea, C. Kittl, D. Simos, E. Weippl, and L. Xu, Eds. Springer Berlin

Heidelberg, 2013, vol. 8128, pp. 137–154. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40588-4_10

[25] K. Jankowski and P. Laurent, “Packed aes-gcm algorithm suitable for aes/pclmulqdq instructions,” Computers, IEEE Transactions on, vol. 60, no. 1, pp. 135–138, Jan 2011.

[26] “elinux.org: Jetson/performance, controlling gpu performance,” 2014. [Online]. Available: http://elinux.org/Jetson/Performance#Controlling_GPU_performance

Appendix A: GeForce GT 620

```
CUDA Device Query (Runtime API) version (CUDART static linking)
 Detected 1 CUDA Capable device(s)
 Device 0: "GeForce GT 620"
   CUDA Driver Version / Runtime Version          6.5 / 6.5
   CUDA Capability Major/Minor version number:    2.1
   Total amount of global memory:                 1023 MBytes
          (1072889856 bytes)
   ( 2) Multiprocessors, ( 48) CUDA Cores/MP:     96 CUDA Cores
   GPU Clock rate:                                1400 MHz (1.40 GHz)
   Memory Clock rate:                             535 Mhz
   Memory Bus Width:                              64-bit
   L2 Cache Size:                                 131072 bytes
   Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D
          =(65536, 65535), 3D=(2048, 2048, 2048)
   Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048
          layers
   Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384),
          2048 layers
   Total amount of constant memory:               65536 bytes
   Total amount of shared memory per block:       49152 bytes
   Total number of registers available per block: 32768
   Warp size:                                     32
   Maximum number of threads per multiprocessor:  1536
   Maximum number of threads per block:           1024
   Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
   Max dimension size of a grid size    (x,y,z): (65535, 65535, 65535)
   Maximum memory pitch:                          2147483647 bytes
   Texture alignment:                             512 bytes
   Concurrent copy and kernel execution:          Yes with 1 copy
          engine(s)
   Run time limit on kernels:                     Yes
   Integrated GPU sharing Host Memory:            No
   Support host page-locked memory mapping:       Yes
   Alignment requirement for Surfaces:            Yes
   Device has ECC support:                        Disabled
   Device supports Unified Addressing (UVA):      Yes
   Device PCI Bus ID / PCI location ID:           1 / 0
   Compute Mode:
      < Default (multiple host threads can use ::cudaSetDevice() with
          device simultaneously) >

 deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 6.5, CUDA
       Runtime Version = 6.5, NumDevs = 1, Device0 = GeForce GT 620
 Result = PASS


 [CUDA Bandwidth Test] - Starting...
 Running on...

  Device 0: GeForce GT 620
  Quick Mode

  Device to Device Bandwidth, 1 Device(s)
  PINNED Memory Transfers
    Transfer Size (Bytes)         Bandwidth(MB/s)
    33554432                      5850.6

 Result = PASS
```

Appendix B: Tegra K1

```
CUDA Device Query (Runtime API) version (CUDART static linking)
 Detected 1 CUDA Capable device(s)
 Device 0: "GK20A"
   CUDA Driver Version / Runtime Version          6.0 / 6.0
   CUDA Capability Major/Minor version number:    3.2
   Total amount of global memory:                 1746 MBytes
          (1831051264 bytes)
   ( 1) Multiprocessors, (192) CUDA Cores/MP:     192 CUDA Cores
   GPU Clock rate:                                852 MHz (0.85 GHz)
   Memory Clock rate:                             924 Mhz
   Memory Bus Width:                              64-bit
   L2 Cache Size:                                 131072 bytes
   Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D
          =(65536, 65536), 3D=(4096, 4096, 4096)
   Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048
          layers

   Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384),
          2048 layers
   Total amount of constant memory:               65536 bytes
   Total amount of shared memory per block:       49152 bytes
   Total number of registers available per block: 32768
   Warp size:                                     32
   Maximum number of threads per multiprocessor:  2048
   Maximum number of threads per block:           1024
   Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
   Max dimension size of a grid size    (x,y,z): (2147483647, 65535,
          65535)
   Maximum memory pitch:                          2147483647 bytes
   Texture alignment:                             512 bytes
   Concurrent copy and kernel execution:          Yes with 1 copy
          engine(s)
   Run time limit on kernels:                     No
   Integrated GPU sharing Host Memory:            Yes
   Support host page-locked memory mapping:       Yes
   Alignment requirement for Surfaces:            Yes
   Device has ECC support:                        Disabled
   Device supports Unified Addressing (UVA):      Yes
   Device PCI Bus ID / PCI location ID:           0 / 0
   Compute Mode:
      < Default (multiple host threads can use ::cudaSetDevice() with
          device simultaneously) >

 deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 6.0, CUDA
       Runtime Version = 6.0, NumDevs = 1, Device0 = GK20A
 Result = PASS


 [CUDA Bandwidth Test] - Starting...
 Running on...

  Device 0: GK20A
  Quick Mode

  Device to Device Bandwidth, 1 Device(s)
  PINNED Memory Transfers
    Transfer Size (Bytes)         Bandwidth(MB/s)
    33554432                      11897.3

 Result = PASS
```