Optimized GPU Implementation and Performance Analysis of HC Series of Stream Ciphers*

Ayesha Khalid[1], Deblin Bagchi[2], Goutam Paul[2], and Anupam Chattopadhyay[1]

[1] Institute for Communication Technologies and Embedded Systems,
RWTH Aachen University, Aachen 52074, Germany
ayesha.khalid@ice.rwth-aachen.de, anupam.chattopadhyay@ice.rwth-aachen.de
[2] Department of Computer Science and Engineering,
Jadavpur University, Kolkata 700 032, India
deblinbagchi@gmail.com, goutam.paul@ieee.org

Abstract. The ease of programming offered by the CUDA programming model attracted a lot of programmers to try the platform for acceleration of many non-graphics applications. Cryptography, being no exception, also found its share of exploration efforts, especially block ciphers. In this contribution we present a detailed walk-through of effective mapping of HC-128 and HC-256 stream ciphers on GPUs. Due to inherent inter-S-Box dependencies, intra-S-Box dependencies and a high number of memory accesses per keystream word generation, parallelization of HC series of stream ciphers remains challenging. For the first time, we present various optimization strategies for HC-128 and HC-256 speedup in tune with CUDA device architecture. The peak performance achieved with a single data-stream for HC-128 and HC-256 is 0.95 Gbps and 0.41 Gbps respectively. Although these throughput figures do not beat the CPU performance (10.9 Gbps for HC-128 and 7.5 Gbps for HC-256), our multiple parallel data-stream implementation is benchmarked to reach approximately 31 Gbps for HC-128 and 14 Gbps for HC-256 (with 32768 parallel data-streams). To the best of our knowledge, this is the first reported effort of mapping HC-Series of stream ciphers on GPUs.

Keywords: CUDA, eSTREAM, GPU, HC-128, HC-256, stream cipher.

1 Introduction

The eSTREAM [12] Portfolio (revision 1 in September 2008) contains the stream cipher HC-128 [21] in Profile 1 (SW) which is a lighter version of HC-256 [22] stream cipher born as an outcome of 128-bit key limitation imposed in the competition. Several research contributions exist on the cryptanalysis of HC-128 [14, 15, 13, 18, 20]. However, HC-256 has undergone fewer cryptanalytic attempts [16,

* This work was done in part while the second author was a summer intern and the third author was an Alexander von Humboldt Fellow at RWTH Aachen, Germany.

19]. For algorithmic details of HC-128 and HC-256, the reader may refer to Appendix A.

After NVIDIA introduced a general purpose parallel computing platform namely Compute Unified Device Architecture (CUDA) in November 2006 [24], many cryptographers harnessed GPUs for acceleration. The earliest successful effort of AES acceleration on GPUs, that outperformed CPU in throughput, was presented by Manavski [1] who reported a throughput of 8.28 Gbps for AES-128 encryption on NVIDIA GeForce 8800. His work was later criticized for having half of the throughput rates that it could achieve by using shared memory instead of constant memory for T-boxes [2]. A more recent work by Iwai *et al.* [3] reported 35 Gbps of throughput for AES encoding on NVIDIA GeForce GTX285 by exploiting memory granularity for independent threads.

Several endeavors undertook more than one cipher to present a suite of CUDA based crypto accelerator application. Liu *et al.* [4] studied the effect of number of parallel threads, size of shared memory for lookup tables and data coalescing in device memories for several block encryption algorithms (AES, TRI-DES, RC5, TWOFISH) processing on GPU using CUDA. Nishikawa *et al.* [5] targeted five 128-bit symmetric block ciphers from an e-government recommended ciphers list by CRYPTREC in Japan and achieved substantial speedup.

The block ciphers, when subjected to parallelism offered by CUDA, generally show high speedups compared to CPUs because of the absence of data dependency in the subsequent data blocks. Generally, the plaintext is broken into n-many blocks of same size and subjected to independent threads of GPUs. Higher sizes of plaintext give more data blocks and hence result in better throughput by achieving more data parallelism, till the device limit is reached.

Unlike block ciphers, stream ciphers in general cannot be subjected to this *'divide and rule'* strategy. The reason is the dependencies in the states/S-boxes that are used for keystream generation. The only endeavor of mapping eSTREAM (including HC-128) and SHA-3 cryptographic algorithms on GPUs was presented by D. Stefan in his masters thesis [7]. He reported a throughput of 2.26 Gbps (4.39 cycles/byte) for HC-128 implementation on NVIDIA GTX 295 GPU device[7]. This effort, however, lacks any optimization opportunity exploiting the structure of the algorithm and is, therefore, easily surpassed by our implementation in throughput.

This work presents a novel implementation of HC series of stream ciphers on recent graphics hardware. To the best of our knowledge, this is the first publication employing CUDA framework for GPU acceleration of any stream cipher.

2 Limitations in Parallelization of HC Ciphers

The *keystream generation* for HC series of stream ciphers has two steps, we name them as *self-update step* (SUS) of P/Q array and *keystream word generation step* (KWGS). In a serial implementation, each 32-bit word of P array SUS is followed by one KWGS. This goes on for 512 iterations in HC-128 and 1024 iterations for

HC-256. The same follows for Q array for exactly the same number of iterations. Ideally, a fast GPU-based implementation would be able to run all these steps in parallel by independent *threads* as long as the device capacity is not over-budgeted. However, ciphers like HC have highly iterative structures, prohibiting parallelization beyond a limit.

2.1 Intra-S-Box Dependency in Self Update Step of S-Boxes

The gain of parallelization offered by CUDA programming model can be exploited easily if each iteration of a given iterative code block is independent of its past execution. Such loops can be converted to parallel kernels by complete unrolling where each loop iteration is executed by an independent *thread*. If an array value being computed by a loop iteration has an intra-array-dependency, such parallelism cannot be harnessed.

The SUS of HC-128 has a data dependency, the update of element $P[j]$ depends on its current and past values, i.e., $P[j]$, $P[j \boxminus 3]$, $P[j \boxminus 10]$ and $P[j \boxminus 511]$. Since the nearest dependency in the SUS of $P[j]$ is on $P[j \boxminus 3]$, one cannot unroll the loop more than 3 times.

```
//Three times unrolled version of P array SUS
for(j = 0; j < 512; j = j + 3)
{
    P[j] = P[j] + g1(P[j ⊟ 3], P[j ⊟ 10], P[j ⊟ 511]);
    P[j + 1] = P[j + 1] + g1(P[j ⊟ 2], P[j ⊟ 9], P[j ⊟ 510]);
    P[j + 2] = P[j + 2] + g1(P[j ⊟ 1], P[j ⊟ 8], P[j ⊟ 509]);
}
```

Fig. 1 describes the data dependencies for calculating the values at the i^{th}, $(i+1)^{th}$ and $(i+2)^{th}$ indices of P array pictorially. Calculation of $(i+3)^{th}$ index value requires the value at i^{th} index of the array, making a simultaneous update of values at indices i and $(i+3)$ impossible. This dependency limits the number of *threads* carrying out the SUS of P/Q array to no more than 3. The same arguments can be extended for HC-256 SUS. Moreover, due to similar limitations, we cannot harness more than 2 and 3 simultaneous *threads* for Step 1 and 3 respectively of *initialization* phase in HC series of stream ciphers.

2.2 Inter-S-Box Dependency in Keystream Generation

For exploiting parallelism we try to investigate if it is possible to carry out SUS P and Q arrays simultaneously (no spatial data dependency) or their current and future copies simultaneously (no temporal data dependency).

Inter-S-Box Spatial Data Dependency. Consider the *keystream generation* phase of HC-128 as given in Appendix A. The SUS of P and Q arrays does not require values from each other. However, KWGS after SUS of P array has a

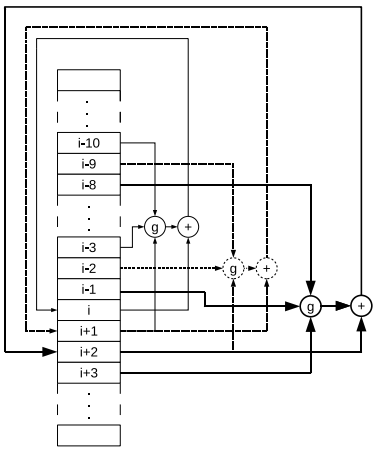

Fig. 1. Dependency in SUS at indices i, $i+1$ and $i+2$ in S-Boxes

dependency on Q array and vice versa. Hence a naive implementation with simultaneous update of P and Q arrays will not bear correct results for KWGS. In HC-256, even the SUS of the two S-Boxes is dependent on each other. Moreover, the KWGS dependency after SUS in HC-256 is the same as in HC-128.

Inter-S-Box Temporal Data Dependency. Temporal data dependency between the current instance of S-Boxes and their future instance is investigated to exploit the possibility of simultaneous *keystream generation* from these arrays for multiple data blocks. Consider two temporal instances of P array. Let $P_{current}$ contain the expanded values after *initialization* phase and P_{future} be the one that will have the future values of P array after SUS. Note that SUS of P_{future} has a dependency on $P_{current}$, hence making it impossible to simultaneously update multiple temporal instances of P/Q arrays. Arguing along the same lines, its evident to see data dependency of P/Q arrays on their past instances in HC-256 too.

2.3 Data-intensiveness

When comparing the computational nature of stream ciphers with block ciphers, a striking trend can be seen. Stream ciphers are predominantly *data intensive* while block ciphers are *computation intensive*. HC series of stream ciphers are no exception. Appendix B gives the list and frequency of various 32-bit binary operations required by the SUS and KWGS of HC-128 and HC-256. The high ratio of memory accesses to the arithmetic operations can be seen to be quite high.

3 Optimization Strategies for GPU Implementation of HC Series of Stream Ciphers

Kernels in CUDA compatible devices are assigned a small budget of thread-local registers. Shared memory is local to a block of threads and is comparatively bigger. The biggest memory in size is the grid-local global memory whose access incurs a 100x penalty as compared to register access [9]. Our device NVIDIA GeForce GTX 590 has 3 GB of global memory, 48 KB of shared memory per MP and a maximum of 64 registers per thread. Considering the memory hierarchy, the fastest single data-stream implementation of the algorithm should use the fastest memory, i.e., the registers. However, the S-boxes of HC-128 (4 KB) and HC-256 (8 KB) are far too big to fit in them. The next best possibility is to put the P and Q arrays in the shared memory and let the registers hold their smaller 16-element snapshot as suggested for the optimized implementation in [21, 22]. However, this single thread implementation of *keystream generation* does not lead to significant throughput. For example, HC-128 on our device yielded a throughput of only *0.24* Gbps.

For exploiting parallelism, we strive to launch multiple threads simultaneously. As registers are local to one kernel, we use shared memory instead and discuss various optimization strategies for single data-stream implementation in Section 3.1. For multiple data-streams implementation, the use of on-chip block-local shared memory instead of off-chip grid-local global memory can boost the speedup significantly. However, each data-stream requires a memory budget m for P and Q arrays, where $m = 4$ KB for HC-128 and $m = 8$ KB for HC-256 and hence the number of parallel data-streams per MP is restricted to s/m, where $s = 48$ KB is the shared memory size. Therefore, we perform the multiple data-streams implementation using global memory, as discussed in detail in Section 3.2. A brief overview of the CUDA programming model for GPUs is given in Appendix C.

3.1 Single Data-Stream Optimizations

Program listing of a simple implementation of *keystream generation* code for HC-128 with the degree of parallelism that is straightforward to manipulate is given in Table 1. Since the *initialization* phase is similar and simpler, its explanation is skipped. The intra-dependency of S-Box arrays does not allow more than 3 parallel threads to update P/Q arrays as described in Section 2.1. The CUDA kernel is called with 1 block of 512 threads. The code is divided into *four* parts. The first and third parts give SUS for P and Q arrays respectively while part two and four perform KWGS. Only 3 out of 512 threads update P array in part one, requiring 171 (512/3) times execution for completely updating P array. In part 2, the S-Boxes are employed to generate 512 words of keystream using 512 threads simultaneously. Part 3 updates the Q array followed by 512 words of KWGS in part 4. This implementation yields a throughput of *0.37* Gbps for *keystream generation* in HC-128.

$if(threadIdx.x <= 2)$ $\quad for(i = threadIdx.x; i < 512; i = i + 3)$ $\quad\quad P\_s[i] = P\_s[i] + g1(P\_s[i \boxminus 3], P\_s[i \boxminus 10], P\_s[i \boxminus 511]);$
$i = threadIdx.x;$ $s[i] = h1(Q\_s, P\_s[i \boxminus 12]) \oplus P\_s[i];$
$if(threadIdx.x <= 2)$ $\quad for(i = threadIdx.x; i < 512; i = i + 3)$ $\quad\quad Q\_s[i] = Q\_s[i] + g2(Q\_s[i \boxminus 3], Q\_s[(i \boxminus 10)], Q\_s[i \boxminus 511]);$
$i = threadIdx.x;$ $s[i + 512] = h2(P\_s, Q\_s[i \boxminus 12]) \oplus Q\_s[i];$

Table 1. *Keystream generation* implementation of HC-128 using three threads

Next we discuss the optimization strategies undertaken to improve the parallelism and consequently the throughput of this simple parallel CUDA based implementation of HC-128. In case the strategies are applicable only to one of the ciphers in HC series of stream ciphers, it has been explicitly mentioned.

Parallelization of P/Q Array SUS with Key Generation(512 words). One way of increasing the degree of parallelism in HC-128 algorithm was suggested by Chattopadhyay *et al.* [23]. The authors proposed carrying out SUS of either of the S-Boxes along with a simultaneous KWGS from the other S-Box. The parallelism can be employed ensuring correct results by keeping multiple temporal copies of S-Boxes (say $P0$, $Q0$, $P1$, $Q1$). If the shared memory of the GPU device used for S-Box instances is not over-budgeted, this strategy can be employed for achieving parallelism. As seen from Appendix A, each round of HC-128 *keystream generation* for 1024 words has a P-SUS and P-KWGS for 512 words, followed by a similar Q-SUS and Q-KWGS for 512 words. With two copies of S-Boxes, we can parallelize the P-SUS with Q-KWGS and vice versa. The series of steps as proposed in [23] are summarized in Table 2. After *initialization* routine, arrays $P0$, $Q0$ contain the expanded key and IV. SUS of P array starts by reading values from $P0$ (past values) and updating $P1$ (current values). No more than 3 parallel threads (due to intra-data-dependency) execute iteratively updating the entire 512 words array. In step 1 the Q array is updated reading values from $Q0$ (past values) and updating $Q1$ (current values). KWGS using $P1$ and $Q0$ is done by 512 parallel threads simultaneously - we denote this by Keygen($Q0$,$P1$). Similar notations describe the other steps.

Step #	KWGS	SUS	Comments
Step 0	-	P1	3 threads for SUS
Step 1	Keygen($Q0$,$P1$)	Q1	3 active threads (out of a warp) for SUS + 512 threads for KWGS
Step 2	Keygen($Q1$,$P1$)	$P0$	
Step 3	Keygen($Q1$,$P0$)	$Q0$	
Step 4	Keygen($Q0$,$P0$)	$P1$	

Table 2. Parallelizing one SUS warp with one KWGS block

After the initial step, $Q1$, $P0$, $Q0$, $P1$ are updated in successive steps, each time simultaneously generating keystream words from the S-Box updated in the

previous step. This goes on by repetition of step 1 till 4 for as many keystream values as required. CUDA framework for HC-128 parallel implementation employs 544 threads for *keystream generation* in total. Out of these, 512 threads carry out KWGS from an entire array of S-Box words simultaneously. One thread warp with three active threads carry out the SUS of the S-Box. Here parallelism is achieved at the cost of extra resources, since only multiple copies of the S-Boxes guarantee correct results for parallel implementation. This strategy is applied to HC-256 as well. Similarly, one warp with 3 active threads remains under-utilized; however KWGS is carried out by 1024 parallel threads for larger S-Boxes in HC-256.

Parallelization of P and Q SUS with Key Generation (1024 words). Further parallelization of HC-128 is possible by simultaneous P-SUS and P-KWGS of 512 words as well as the Q-SUS and Q-KWGS of 512 words in *keystream generation* phase as described in Appendix A. Thus both the S-Boxes can be updated in parallel along with simultaneous generation of 1024 words of keystream. However, step 1 and 3 of *keystream generation* in Table 2, reveal a data dependency. $Q0$ is needed for generating key from $P1$, and $Q1$ for generating key from $P0$. Hence, update of $P0$, $Q0$ and generating 1024 keystream words using Keygen($Q0$, $P1$) and Keygen($P1$, $Q1$) gives rise to a *race condition*, commonly called a *Read After Write (RAW) hazard* where the keystream values would depend upon which statement gets executed first. This can be successfully avoided by using 2 more copies of Q arrays, namely Q_{Buff0} and Q_{Buff1} for keeping backups of $Q0$ and $Q1$ respectively. For preserving correctness, these buffers need to be updated at every alternate step. All arrays are stored in the shared memory for fast access.

Table 3 describes a step by step execution. After *initialization*, the expanded key and IV reside in $P0$, $Q0$. All other temporal S-Box copies i.e., $P1$, $Q1$, Q_{Buff0} and Q_{Buff1} are left un-initialized. Simultaneous SUS of P and Q arrays is carried out by reading values from $P0$, $Q0$ (past values) and updating $P1$, $Q1$ (current values) respectively. A copy of $Q0$ is backed up in Q_{Buff0} simultaneously. In this step, 6 threads of 2 warps carry out the SUS for $P1$ and $Q1$. For $Q0$ backup, 512 parallel threads make a copy.

Q_{Buff} copy	KWGS		SUS		Comments
Q_{Buff0} copy	-	-	$P1$	$Q1$	3 + 3 threads for SUS, 512 threads for copying $Q0$ to Q_{Buff0}
Q_{Buff1} copy	Keygen ($Q1$,$P1$)	Keygen (Q_{Buff0},$P1$)	$P0$	$Q0$	3 + 3 threads for SUS, 512 threads for Keygen($Q1$,$P1$), 512 threads for copying $Q1$ to Q_{Buff1} and Keygen(Q_{Buff0},$P1$)
Q_{Buff0} copy	Keygen ($Q0$,$P0$)	Keygen (Q_{Buff1},$P0$)	$P1$	$Q1$	3 + 3 threads for SUS, 512 threads for Keygen($Q0$,$P0$), 512 threads for copying $Q0$ to Q_{Buff0} and Keygen(Q_{Buff1},$P0$)

Table 3. Parallelizing 2 S-Box SUS warps with 2 KWGS blocks

In step 1, we employ a block of 1024 threads for generating 1024 words of keystream, each thread generates one word of keystream. Out of these, 512 threads are used to execute the extra step of copying values to the buffers. Alternate updates of $P0$, $Q0$ and $P1$, $Q1$ follows, simultaneously generating 1024 words of keystream. Hence Step 1 and 2 are repeated as long as the *keystream generation* is required.

A single kernel cannot be invoked with more than 1024 threads in a block. We break the thread budget in two blocks, each having 544 threads. The two blocks run concurrently, one warp in each carrying out SUS and 512 threads generating keystream. GPUs with compute capability 2.0 or more have the capability of calling concurrent kernels at the same time as well.

This strategy of achieving parallelism cannot be extended for HC-256 since its SUS of the S-Boxes is dependent on each other.

3.2 Multiple Data-Streams Optimization

The GPU clock is slower than the CPU clock speed. Thus speedup in GPU devices can be achieved in two ways. One way is by employing parallel threads respecting data dependencies in a single stream of data as investigated in Section 3.1. A better alternative in terms of resource utilization and throughput is to employ all the SPs (stream processors) of the CUDA device by employing ciphers of multiple data-streams in parallel. Due to the limited size of shared memory, we employ the larger albeit slower global memory for ciphering multiple parallel streams of data.

Performance tuning on the GPU requires understanding device specifications and accordingly finding and exposing enough parallelism to populate all the multiprocessors (MPs). NVIDIA GeForce GTX 590 can accommodate up to 8 blocks (or 48 warps) per MP. Since each warp can have 32 homogeneous threads, an MP can process up to 1536 threads (48×32). To fully utilize each MP, the number of threads it should get assigned should be no more than 192 per block (1536/8). This limit is kept in mind when assigning the thread budget to each MP for HC series of stream ciphers.

For HC-128, the 3 threads for SUS of each of the S-Boxes constitute one warp. Since these threads execute a total of 171 times (512/3) for complete update of either of the S-Boxes, the number of parallel threads employed for KWGS can be adjusted so that the budget of total number of 192 threads per block is never exceeded. We employ 128 threads for KWGS and 2 warps for S-Box update in case of HC-128. Hence 2 warps of S-Box SUS and 4 warps of KWGS are kept in the same block of 192 threads. For HC-256, however, only one warp is used for SUS and 4 for KWGS, making the total thread budget equal to 160 per block. This strategy ensures maximum number of parallel data-streams the device can encrypt simultaneously, showing noticeable increase in the throughput of both HC-128 and HC-256.

4 Experimental Results

Throughput performances of HC ciphers for single and multiple parallel data-streams were benchmarked on NVIDIA GeForce GTX 590. We used an AMD Phenom^{TM}II X6 1100T Processor with 8 GBs of RAM as host CPU. Each test was conducted 1000 times and the results were averaged. Appendix D summarizes the hardware specifications of the two computation platforms.

4.1 Encryption of Single Data-Stream

Initialization phase of HC ciphers has been implemented using shared memory and global memory in two separate experiments. The last step of *initialization* phase is similar to SUS phase, consequently 3 parallel *threads* are employed for it. In the second step of *initialization* phase, intra-dependency for W is even more severe, limiting the number of simultaneous *threads* to 2. Using faster shared memory instead of global memory accelerates *initialization* phase as shown in Table 4. It however, incorporates the overhead of copying P, Q and W arrays on shared memory that can be done simultaneously using 512 and 1024 parallel threads in case of HC-128 and HC-256 respectively.

	NVIDIA GeForce GTX 590		AMD Phenom^{TM}II
	Global memory	Shared memory	X6 1100T
HC-128	1.386 ms 22.53 Mbps	1.078 ms 28.98 Mbps	27 μ s 1.15 Gbps
HC-256	1.930 ms 32.35 Mbps	1.666 ms 53.56 Mbps	60 μ s 1.04 Gbps

Table 4. Duration and throughput of *initialization* phase of HC series of stream ciphers

The performance results of *keystream generation* phase are presented in Fig. 2 and Fig. 3 for HC-128 and HC-256 respectively. The throughput shows an increasing trend, till it saturates for higher data sizes considered. The maximum throughput when using the global memory for storing S-Boxes of HC-128 is *0.41* Gbps. Using shared memory gives a boost to performance because of its smaller access time. A similar trend is observed for HC-256. The size of the S-Boxes is

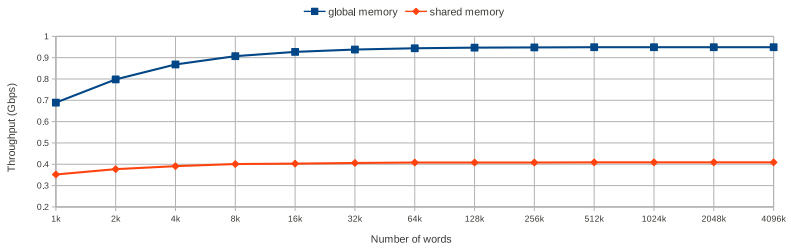

Fig. 2. HC-128 *keystream generation* throughput using shared and global memory

double compared to that of HC-128, the amount of shared memory used by the optimized version of our algorithm is 16 KB (two copies of each S-Box). A GPU device with lower compute capability has no more than 16 KB of shared memory per MP. Hence, this optimized implementation of HC-256 on one thread block of such devices is not possible. The maximum throughput from the global memory implementation of HC-256 is 0.15 Gbps and for shared memory implementation is 0.41 Gbps.

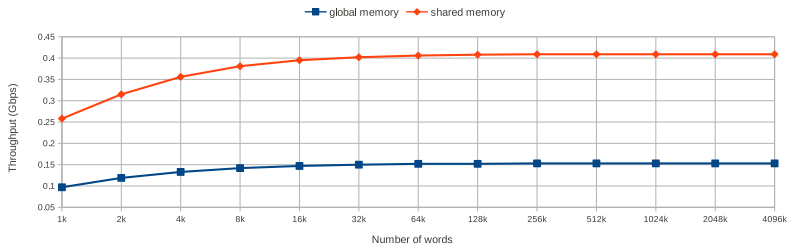

Fig. 3. HC-256 *keystream generation* throughput using shared and global memory

4.2 Encryption of Multiple Data-Streams in Parallel

The parallelism offered by the CUDA device can be well exploited using multiple parallel streams of data. For simulation purposes we start with a single stream of data and double them up to 32K parallel streams. Fig. 4 gives the throughput of HC-128 and HC-256 for increasing number of parallel data-streams on our CUDA device. The trend of throughput rise shown by the two ciphers is similar, having an apparent peak for 64 parallel streams. The CUDA device used has a total of 16 MPs and each MP can accommodate 8 blocks at most. Maximum utilization of MPs is achieved for 128 parallel streams of data (16×8). Further increase in the number of parallel data-streams shows a slight improvement in the throughput. The reason is that the parallel streams in excess of 128 are waiting in instruction queue and are launched with negligible context switch time. The maximum throughput achieved is 31 Gbps for HC-128 and 14 Gbps for HC-256 employing 32768 parallel streams.

4.3 Throughput comparison of HC Series of Stream Ciphers on Various Platforms

We compare our acceleration results with the only available figures for HC-128 acceleration on GPUs by D. Stefan in his masters thesis [7]. Without employing parallelism within a single data-stream for HC-128, he assigned one thread to one data-stream. For supporting multiple data-streams, he employed global memory for S-boxes. The highest throughput achieved is reported and compared with our implementation in Table 5. For the same number of blocks, our throughput

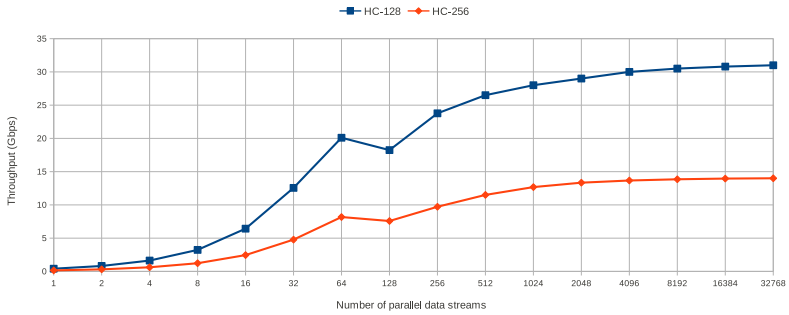

Fig. 4. *keystream generation* throughput for varying number of multiple data-streams

is approximately 14 times higher. Comparing the cycles/byte performance also shows a significant decrease. Results for *initialization* phase are not available for comparison.

	Implementation by D. Stefan[7]	Our Implementation
NVIDIA device	GeForce GTX 295	GeForce GTX 590
Release date	January 8, 2009	March 24, 2011
Compute Capability	1.3	2.0
Memory Used	Global Memory	Global Memory
Threads / data-stream	1	192
data-stream / Block	256	1
Total blocks used	680	680
Total data-streams	680×256	680
Total threads used	680×256	192×680
Performance(Cycles/byte)	4.39	0.279
Throughput(Gbps)	2.26	31

Table 5. Comparison of our HC-128 acceleration with D. Stefan [7]

The HC-128 performance evaluation on CPU was done using the *eSTREAM testing framework* [6]. The C implementation of the testing framework was installed in the CPU machine (specs given in Appendix D) on CentOs 5.8 (Linux version 2.6.18-308.11.1.el5xen). For the benchmark implementation of HC-128 and HC-256 the highest *keystream generation* speeds were found to be 2.36 cycles/byte and 3.63 cycles/byte respectively. Table 6 gives a comparison of throughput of HC series of stream ciphers on various platform. The throughput obtained on an AMD Phenom^{TM} II X6 1100T Processor is 10.94 Gbps and 7.5 Gbps for *keystream generation* phase of HC-128 and HC-256 respectively. The high speed rendered by CPU is primarily because it has to incur no memory overhead for RAM located contents unlike the GPU memory accesses. Moreover, the limitation of SIMD architecture of GPUs requires homogeneity of warp threads which is not a limitation in CPUs. Consequently the CUDA mapping of the HC family of ciphers is 11-18 times slower. The ASIC based implementation proposed by Chattopadhyay *et al.* is so far the fastest reported implementation

of HC-128 claiming a throughput of 22.88 Gbps [23]. The throughput results of HC-256 are however not reported.

	AMD Phenom™ II X6 1100T		NVIDIA GeForce GTX 590		ASIC [23] (65nm Technology)	
HC-128	10.9 Gbps	2.36 C/B	0.95 Gbps	9.27 C/B	22.88 Gbps	0.5 C/B
HC-256	7.5 Gbps	3.63 C/B	0.41 Gbps	21.82 C/B	Not reported	Not reported

Table 6. Throughput (Gbps), Cycles/Byte (C/B) of a single data-stream HC ciphers

For multiple data-streams we get promising results which for CPUs is not straightforward to implement. For 32768 parallel data-streams, our GPU gives a throughput of 31 Gbps for HC-128 and 14 Gbps for HC-256. Hence we conclude that HC-series of stream ciphers is unfit to be off-loaded to GPUs in case of a single data-stream application. In contrast, an application exploiting multiple parallel data-streams can achieve GPU acceleration up to 2.8 times faster in case of HC-128 and 1.87 times faster for HC-256 (with 32768 parallel data-streams).

5 Conclusion and Future Work

This work presents the first detailed study of algorithmic acceleration limitations in HC series of stream ciphers for mapping on a GPU device. The high degree of data dependency in their S-box update procedures puts strict limitations on exploiting the inherent parallelism that a graphics device offers. Moreover these ciphers are primarily data intensive in nature. These limitations explain the absence of relevant scientific publications in this arena. We present various strategies to improve the throughput of the HC-128 and HC-256 ciphers at the cost of replicated copies of S-Boxes. However, for a single data-stream acceleration, our throughput does not go beyond 0.95 Gbps and 0.41 Gbps for HC-128 and HC-256 respectively on a GeForce GTX 590 (leaving it 11-18 times slower than a standard CPU in throughput).

For multiple data-streams, however, we beat the CPU performance. We did a thorough tuning on the GPU for optimizing all the architectural features that the device could offer. Thread and warp grouping is done so as to expose enough parallelism to the device to keep all the MP cores busy all the time. Our GPU based acceleration resulted in being 2.8 times faster than CPU in case of HC-128 and 1.87 times faster for HC-256 (with 32,768 parallel data-streams). Hence we conclude that GPUs can successfully be employed as a co-processor with a CPU host to accelerate HC series of stream ciphers using multiple parallel streams of data. As future work, we plan to investigate the parallelism opportunities offered by the entire eSTREAM portfolio [12] of software stream ciphers and compare the performance against today's CPUs.

References

1. S. A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography In International Signal Processing and Communications (ICSPC) 2007, pages 65-68, IEEE.
2. A. Biagio, A. Barenghi, G. Agosta and G. Pelosi. Design of a parallel AES for graphics hardware using the CUDA framework. In International Symposium on Parallel & Distributed Processing (IPDPS) 2009, pages 1-8, IEEE.
3. K. Iwai, N. Nishikawa, and T. Kurokawa. Acceleration of AES encryption on CUDA GPU. In International Journal of Networking and Computing 2012, pages 131-145, vol. 2, no. 1.
4. G. Liu, H. An, W. Han, G. Xu, P. Yao, M. Xu, X. Hao and Y. Wang. A Program Behavior Study of Block Cryptography Algorithms on GPGPU. In Fourth International Conference on Frontier of Computer Science and Technology 2009, FCST'09, pages 33-39, IEEE.
5. N. Nishikawa, K. Iwai, and T. Kurokawa. High-Performance Symmetric Block Ciphers on Multicore CPU and GPUs In International Journal of Networking and Computing, 2012, pages 251-268, vol. 2, no. 2.
6. C. D. Cannire. eSTREAM testing framework. Available at `http://www.ecrypt.eu.org/stream/perf`.
7. D. Stefan. Analysis and Implementation of eSTREAM and SHA-3 Cryptographic Algorithms. 2011. Available at `http://hgpu.org/?p=5972`.
8. M. Bauer, H. Cook and B. Khailany. CudaDMA: Optimizing GPU memory bandwidth via warp specialization. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011, ACM, New York, NY, USA, Article 12.
9. Available at `http://stanford-cs193g-sp2010.googlecode.com/svn/trunk/lectures/lecture_4/cuda_memories.pdf`
10. D. Bernstein. Cache-timing attacks on AES. 2005. Available at `http://cr.yp.to/papers.html#cachetiming`.
11. D. Boneh, R. A. Demillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. In EUROCRYPT 1997, pages 37-51, vol. 1233, Lecture Notes in Computer Science, Springer.
12. eSTREAM: the ECRYPT Stream Cipher Project. Available at `http://www.ecrypt.eu.org/stream`.
13. A. Kircanski and A. M. Youssef. Differential Fault Analysis of HC-128. In Africacrypt 2010, pages 360-377, vol. 6055, Lecture Notes in Computer Science, Springer.
14. Y. Liu and T. Qin. The key and IV setup of the stream ciphers HC-256 and HC-128. In International Conference on Networks Security, Wireless Communications and Trusted Computing 2009, pages 430-433, IEEE.
15. S. Maitra, G. Paul, S. Raizada, S. Sen and R. Sengupta. Some observations on HC-128. In *Designs, Codes and Cryptography*, 2011, pages 231-245, vol. 59, no. 1-3.
16. E. Zenner. A Cache Timing Analysis of HC-256. In SAC 2008, pages 199-213, vol. 5381, Lecture Notes in Computer Science, Springer.
17. D. A. Osvik, A. Shamir and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In CT-RSA 2006, pages 1-20, vol. 3860, Lecture Notes in Computer Science, Springer.

18. G. Paul, S. Maitra and S. Raizada. A Theoretical Analysis of the Structure of HC-128. In IWSEC 2011, pages 161-177, vol. 7038, Lecture Notes in Computer Science, Springer.
19. G. Sekar and B. Preneel. Improved Distinguishing Attacks on HC-256. In IWSEC 2009, pages 38-52, vol. 5824, Lecture Notes in Computer Science, Springer.
20. P. Stankovski, S. Ruj, M. Hell and T. Johansson. Improved distinguishers for HC-128. In *Designs, Codes and Cryptography*, 2012, pages 225-240, vol. 63, no. 2.
21. H. Wu. The Stream Cipher HC-128. Available at `http://www.ecrypt.eu.org/stream/hcp3.html`.
22. H. Wu. A New Stream Cipher HC-256. In FSE 2004, pages 226-244, vol. 3017, Lecture Notes in Computer Science, Springer. The full version is available at `http://eprint.iacr.org/2004/092.pdf`.
23. A Chattopadhyay, A. Khalid, S. Maitra and S. Raizada. Designing High-Throughput Hardware Accelerator for Stream Cipher HC-128. In International Symposium on Circuits and systems (ISCAS) 2012, pages 1448-1451, IEEE.
24. NVIDIA CUDA. Available at `http://developer.NVidia.com/object/CUDA.html`

Appendix A: Description of HC-128 and HC-256 Keystream Generation

HC-t uses t-bit secret key and IV, and 32-bit element internal arrays P and Q each of length $4t$, where t is either 128 or 256. We briefly sketch the *keystream generation* phase of the algorithms here. For details of key and IV setup, one may refer to [21, 22]. The operators used are $+$ (addition modulo 2^{32}), $\boxminus$ (subtraction modulo 512), $\oplus$ (bit-wise exclusive OR), $\gg, \ll$ (32-bit shifts) and $\ggg, \lll$ (32-bit rotations). Let s_r denote the keystream word generated at the r-th step, $r = 0, 1, 2, \ldots$. The functions g_1 and g_2 (3 inputs for HC-128 and 2 inputs for HC-256) are used for self-update of P and Q and functions h_1 and h_2 are used in the *keystream generation*, as follows.

	HC-128	HC-256
t	128	256
g_1	$((x \ggg 10) \oplus (z \ggg 23)) + (y \ggg 8)$	$((x \ggg 10) \oplus (y \ggg 23)) + Q[(x \oplus y) \bmod 4t]$
g_2	$((x \lll 10) \oplus (z \lll 23)) + (y \lll 8)$	$((x \ggg 10) \oplus (y \ggg 23)) + P[(x \oplus y) \bmod 4t]$
h_1	$Q[x_{(0)}] + Q[2t + x_{(2)}]$	$Q[x_{(0)}] + Q[t + x_{(1)}] + Q[2t + x_{(2)}] + Q[3t + x_{(3)}]$
h_2	$P[x_{(0)}] + P[2t + x_{(2)}]$	$P[x_{(0)}] + P[t + x_{(1)}] + P[2t + x_{(2)}] + P[3t + x_{(3)}]$
$P[j] +=$	$g_1(P[j \boxminus 3], P[j \boxminus 10], P[j \boxminus 511])$	$P[j \boxminus 10] + g_1(P[j \boxminus 3], P[j \boxminus 1023])$
$Q[j] +=$	$g_2(Q[j \boxminus 3], Q[j \boxminus 10], Q[j \boxminus 511])$	$Q[j \boxminus 10] + g_2(Q[j \boxminus 3], Q[j \boxminus 1023])$

The last two rows of the above table show the self-update steps (SUS) for the arrays P and Q. Here $x = x_{(3)} \| x_{(2)} \| x_{(1)} \| x_{(0)}$ is a 32-bit word, with $x_{(0)}, x_{(1)}, x_{(2)}$ and $x_{(3)}$ being the four bytes from right to left. The *keystream generation* phase happens in cycles of $8t$ rounds, in the first $4t$ of which the array P is updated followed by a keystream word generation step (KWGS) $s_i = h_1(P[j \boxminus 12]) \oplus P[j]$. In the next $4t$ rounds, the array Q is updated and the corresponding KWGS is given by $s_i = h_2(Q[j \boxminus 12]) \oplus Q[j]$.

Appendix B: List of Operations for Keystream Generation in HC-128 and HC-256

	HC-128 SUS	HC-128 KWGS	HC-256 SUS	HC-256 KWGS
Modulo Additions	2	2	3	7
Xor	1	1	2	1
Modulo Subtractions	3	1	3	1
Rotations	3	0	2	0
Shifts	0	1	0	3
Total operations	**9**	**5**	**10**	**12**
Memory Reads	4	4	5	6
Memory Writes	1	1	1	1
Total memory accesses	**5**	**5**	**6**	**7**

Appendix C: Overview of CUDA Programming Model

CUDA exposes the device as a repository of thousands of parallely executable threads as shown in Fig. 5. The GPU chip is organized as a collection of multiprocessors (MPs). Each MP has a number of Stream Processors (SPs), each handling one thread. Each MP is responsible for handling one or more thread blocks. Since thread blocks have no dependencies among themselves, their assignment is independent of MPs allowing transparent scaling of programs across different GPUs. Here are some technical terms relevant to the CUDA execution model.

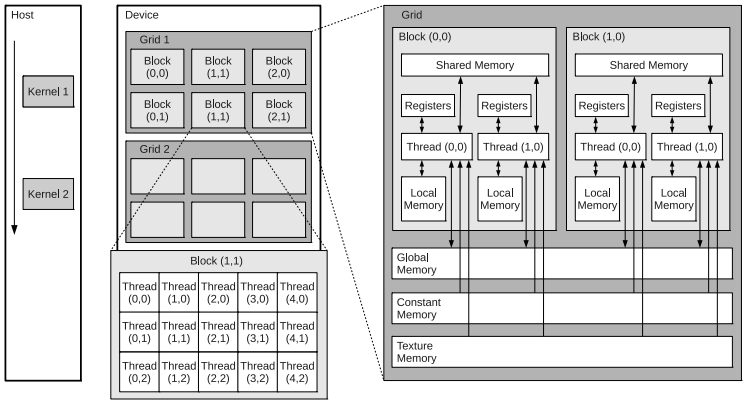

Fig. 5. CUDA GPU execution model

1. **Thread**: the smallest unit of execution in CUDA.
2. **Warp**: the threads are forwarded to the CUDA MPs in groups (*warps*) of 32 for execution. If all thread kernels in a warp are homogeneous, all the SPs in an MP execute the same instruction in a true SIMD fashion.
3. **Block**: a group of threads each with exclusive local memories and registers and a single shared memory as shown in Fig. 5.

4. **Grid**: one or more thread blocks being executed by a kernel in memory form a grid. Each MP handles one or more blocks in a grid. Threads in a block are not divided across multiple MPs.
5. **Kernel**: a block of code called from the host CPU, and then sent to the device with a grid of thread blocks. CUDA gives the freedom of choosing the threads and block structure and dimension to the coder.

Appendix D: Hardware Specifications of CPU and GPU used for Throughput Comparison

	AMD Phenom^{TM}II X6 1100T	NVIDIA GeForce TX 590
Transistors	904 million	6 billion
Processor Frequency (GHz)	3.31	1.2
Cores/SPs	6	1024
Cache/shared Memory	L2-512 KB, L3-6 MB×6	48 KB×32
Threads executed per cycle	6	1024
Active Hardware threads	6	49152 (maximum)