New Speed Records for Salsa20 Stream Cipher Using an Autotuning Framework on GPUs*

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

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

Abstract. Since the introduction of the CUDA programming model, GPUs are considered a viable platform for accelerating non-graphical applications. Many cryptographic algorithms have been reported to achieve remarkable performance speedups, especially block ciphers. For stream ciphers, however, the lack of reported GPU acceleration endeavors is due to their inherent iterative structures that prohibit parallelization. In this paper, we propose an efficient implementation methodology for data-parallel cryptographic functions in a batch processing fashion on modern GPUs in general and optimizations for Salsa20 in particular. We present an autotuning framework to reach the most optimized set of device and application parameters for Salsa20 kernel variants with throughput maximization as a figure of merit. The peak performance achieved by our implementation for Salsa20/12 is 2.7 GBps and 43.44 GBps with and without memory transfers respectively on NVIDIA GeForce GTX 590. These figures beat the fastest reported GPU implementation of any stream cipher in the eSTREAM portfolio including Salsa20/12, as well as the block cipher AES optimized by hand-tuning, and thus, to the best of our knowledge set a new speed record.

Keywords: CUDA, eSTREAM, GPU, Salsa20, Salsa20/r, stream cipher.

1 Introduction and Motivation

Performance enhancement on a GPU is a function of the extent of parallelism within the application. In case of symmetric block ciphers, for the encryption of long messages, the plaintext is first partitioned into chunks of the cipher's *blocksize* and then encrypted. For avoiding the weakness of generating identical ciphertexts for identical plaintext blocks, chaining dependencies between adjacent plaintext blocks are added, defined by *modes of operations*. The ciphertext C_i for the i^{th} plaintext block P_i under different modes of operations is

* This work was done in part while the second author was visiting RWTH Aachen, Germany as an Alexander von Humboldt Fellow.

A. Youssef, A. Nitaj, A.E. Hassanien (Eds.): AFRICACRYPT 2013, LNCS 7918, pp. 189–207, 2013.

given below. Here IV stands for the Initialization Vector and E_k stands for the encryption function parametrized by the secret key k.

Operation Mode	C_i
Electronic codebook (ECB)	$E_k(P_i)$
Counter (CTR)	$P_i \oplus E_k(nonce, counter)$
Cipher block chaining (CBC)	$E_k(P_i \oplus C_{i-1}), C_0 = IV$
Propagating CBC (PCBC)	$E_k(P_i \oplus P_{i-1} \oplus C_{i-1}), P_0 \oplus C_0 = IV$
Cipher feedback (CFB)	$E_k(C_{i-1}) \oplus P_i, C_0 = IV$

Observe that all the modes of operation of block ciphers are not parallelizable. The ECB and CTR modes of operation pose encryption as a massively parallel problem with all the plaintext blocks being available for simultaneous encryption without any dependency. However, in CBC, PCBC and CFB modes, due to the dependency or a "carry over" from the previous block, the encryption must progress sequentially on a block by block basis. Consequently, almost all the results of GPU based acceleration undertake block cipher encryption or decryption in Electronic Codebook (ECB) or Counter (CTR) mode for which the inter-dependency between data blocks does not exist and a parallel encryption of blocks of plaintext data is possible.

The use of stream ciphers is best suited for applications requiring high throughput and where the amount of data is either unknown, or continuous. A block cipher is generic in nature and can be used as a stream cipher in CTR mode. In comparison to block ciphers, stream ciphers are simpler and faster and typically execute at a higher speed than block ciphers [3]. Using a block cipher as a stream ciphers is therefore an *overkill* and consequently, results in a lower throughput of encryption in comparison. For example, on a Core 2 Intel processor, 20 rounds of Salsa20 stream cipher run at 3.93 cycles/byte, while 10 rounds of AES block cipher are reported to run more than twice as slow at 9.2 cycles/byte for long data streams (bitsliced AES-CTR) [2]. In this paper, we focus on the GPU implementation of Salsa20 series [4] of stream ciphers.

1.1 Why Salsa20?

The eSTREAM [1] competition was created to attract stream ciphers in two separate profiles, namely for software and hardware platforms. Out of 34 initial submissions, four software stream ciphers, namely, HC-128, Rabbit, Salsa20/12, SOSEMANUK and three hardware stream ciphers, namely, Grain v1, MICKEY 2.0 and Trivium made into the final portfolio [1]. Unlike the parallelizable modes of operations defined for block ciphers, most stream ciphers do not have the liberty of employing the "divide and rule" policy on chunks of plaintext and exhort parallelism on GPUs. Their highly iterative structures have inter-dependencies on subsequent keystream values generated.

For example, in case of HC-128 [14], the limitation on parallelization of the keystream generation routine from two S-boxes P and Q is severe. This is because there are inter-S-box as well as intra-S-box dependencies. The update of the values in S-boxes is a function of previous index values in the array (update of $Q[j]$ requires $Q[j \boxminus 3]$, $Q[j \boxminus 10]$ and $Q[j \boxminus 511]$, where $\boxminus$ is subtraction

modulo 512). This limitation renders no more than 3 parallel threads deployment to ensure correctness of results [8]. The other five eSTREAM finalists are also no different. The update of the internal states for generation of next block of keystream is dependent on its previous values.

Salsa20 [4] has an edge over the rest of the stream ciphers for mapping on GPUs, since it has no chaining or dependence between blocks of data during encryption / decryption. Hence a large number of parallel homogeneous threads can be subjected to plaintext data chunks enabling instruction execution in a Single Instruction Multiple Thread (SIMT) fashion exploiting well the parallelism offered by many-core architecture of GPUs. Each block takes a nonce, a secret key, constants and a counter and combines them to generate a block of keystream. For additive stream ciphers the keystream generation is independent of the plaintext. For generating ciphertext, keystream is simply XOR-ed with the plaintext. This property motivated us to take up and report an efficient implementation of Salsa20 stream ciphers on recent graphics hardware.

1.2 Why Autotuning?

Another motivation of the work was the development of an autotuning framework for cryptographic kernels with optimization of throughput performance in mind. The recommended autotuning framework can tune optimally to other and newer devices of NVIDIA GPUs and can be extended to other cryptographic algorithms. *The need of such autotuners is emphasized by the fact that considering device occupancy as a figure of merit is not guaranteed to achieve maximized throughput.* Extensive experimentation is recommended with variation of factors like register usage, thread-block sizes, loop unroll factor etc. The implementation results after autotuning stand out in performance compared to hand-tuned codes for Salsa20.

Autotuning methodologies for multi-core devices are gaining popularity since hand-tuning a large number of parameters optimally for an algorithm on a machine is hard. Most of these autotuning efforts are limited to either a class of similar algorithms, a family of similar devices or an optimization strategy of one parameter for performance enhancement. Murthy *et al.* studied the effect of loop unrolling on various GPGPU programs and claimed 70 percent better throughput by optimally unrolling iterations [11]. A class of algorithms extensively undertaken for autotuning is General Matrix Multiply (GEMM), a part of Basic Linear Algebra Subprograms (BLAS) for matrix multiplication [18]. Kurzak *et al.* presented the optimized choice of tiling and thread arrangement for various versions of GEMM mapped on Fermi family of NVIDIA GPUs [9]. Our autotuning framework is similar in spirit to their work, however specialized for symmetric cryptographic schemes, for which no autotuning endeavors have been reported so far.

1.3 Our Contributions

The major contributions of our work are summarized as follows.

(1) We introduce a batch processing framework for processing any parallelizable cryptographic task in a hybrid CPU-GPU environment.
(2) We recommend a better memory hierarchy utilization for Salsa20 kernels, i.e., use of constant memory instead of shared memory for keeping the initial state vector for Salsa20 (boosting throughput considerably).
(3) We propose an autotuning framework for Salsa20 kernels with two goals in mind: fast device portability and selection of application-specific, device-specific and compiler-specific optimization parameters for throughput maximization. Performance tuning by various parameter search space generation and pruning is generic enough to be extended to other cryptographic schemes, for which no autotuning framework has been reported.
(4) Throughput curves for very long message streams encrypted by Salsa20/r, with and without memory transfers are presented. We hereby report so far the fastest implementation results for Salsa20 variants mapped on any GPU.

2 Parallelism Opportunities of Salsa20 in GPUs

We begin with a functional description of the Salsa20 stream cipher, followed by an overview of the CUDA programming model for NVIDIA GPUs. Then we connect these two by critically analyzing the parallelization opportunities of Salsa20 in GPU.

2.1 Description of Salsa20

Salsa20 accepts four types of inputs, each consisting of 32-bit words: an input key of either 256-bit $(k_0, k_1, ..., k_7)$ or 128-bit $(k_4 = k_0, ..., k_7 = k_3)$ size, a 64-bit nonce (n_0, n_1), a 64-bit counter (t_0, t_1) and four words of pre-defined constants ϕ_i, whose values are dependent upon the key size.

Initialization. These inputs are arranged in a predefined order into a 4x4 state vector X, as follows.

$$X = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix} = \begin{pmatrix} \phi_0 & k_0 & k_1 & k_2 \\ k_3 & \phi_1 & n_0 & n_1 \\ t_0 & t_1 & \phi_2 & k_4 \\ k_5 & k_6 & k_7 & \phi_3 \end{pmatrix}.$$

Keystream Generation. The state vector is subjected to a series of rounds composed of additions, cyclic rotations and XORs, to achieve a random permutation. Originally, the number of rounds was set to 20 (Salsa20/r, r=20); however, the version included in the eSTREAM portfolio [1], it was reduced to 12 rounds, for performance reasons.

Then Salsa20/r function for keystream generation can be represented mathematically as:

$Salsa20_k(X) = DoubleRound^{r/2}(X) + X$, with $DoubleRound(X) = RowRound(ColumnRound(X))$.

Each double round comprises of four *QuarterRounds* (in short, QR) performed first on the columns of the state vector X and then on the rows of the output.

$Y = (y_0, y_1, ..., y_{15}) = ColumnRound(X)$, and $Z = (z_0, z_1, ..., z_{15}) = RowRound(Y)$, where

$$(y_0, y_4, y_8, y_{12}) = QR(x_0, x_4, x_8, x_{12}), \quad (y_5, y_9, y_{13}, y_1) = QR(x_5, x_9, x_{13}, x_1),$$
$$(y_{10}, y_{14}, y_2, y_6) = QR(x_{10}, x_{14}, x_2, x_6), \quad (y_{15}, y_3, y_7, y_{11}) = QR(x_{15}, x_3, x_7, x_{11}),$$
$$(z_0, z_1, z_2, z_3) = QR(y_0, y_1, y_2, y_3), \quad (z_5, z_6, z_7, z_4) = QR(y_5, y_6, y_7, y_4),$$
$$(z_{10}, z_{11}, z_8, z_9) = QR(y_{10}, y_{11}, y_8, y_9), \quad (z_{15}, z_{12}, z_{13}, z_{14}) = QR(y_{15}, y_{12}, y_{13}, y_{14}).$$

Each $QuarterRound(a, b, c, d)$ consists of four $ARXrounds$, comprising of additions (A), cyclic rotations (R) and XOR (X) operations only, as below:

$$b = b \oplus ((a + d) \lll 7),\ c = c \oplus ((b + a) \lll 9),\ d = d \oplus ((c + b) \lll 13),\ a = a \oplus ((d + c) \lll 18).$$

Encryption and Decryption. A 16-word ciphertext block C is calculated simply by bitwise XOR-ing a 16-word plaintext block P with the 16-word keystream block S. On the receiver side, the same keystream, when bitwise XOR-ed with the ciphertext C, reproduces the plaintext P.

2.2 CUDA Programming Model Overview

CUDA defines a convenient programming model for heterogeneous computing environment for a CPU *host* and GPU *device*. This section briefly presents NVIDIA GPU architecture and its programming environment. The reader is kindly referred to CUDA C programming guide [16] and Fermi Architecture manual [15] for more information.

Execution Model. CUDA device execution model is depicted in Fig. 1. Parallel portions of an application, executed on the device are called *kernels*. A Kernel call launches a number of *threads*, each executing the same code but having a unique *threadID*. Threads are forwarded to the CUDA device in groups called *warps* for execution. A *threadblock* is a batch of threads that may or may not cooperate with each other by sharing data or by synchronizing their execution. Threads from different threadblocks cannot cooperate.

Kernels are launched in *grids* for execution, comprising of one or more threadblocks. The grid dimensions are specified by *blocksPerGrid* and *threadsPerBlock*. A CUDA device consists of several *Streaming Multiprocessors* (*SM*), each responsible for handling one or more blocks in a grid. Threads in a block are not divided across multiple SMs.

Memory Model. CUDA provides explicit methods to organize memory architecture. Local variables within a kernel reside in registers (*regs*) or in the off-chip local memory (*lmem*). Shared memory (*shmem*) is shared by each threadblock. Global memory (*gmem*) is accessible by all threads as well as host. The lifetime of global memory is from allocation to de-allocation by the host. However, for the other memories mentioned, the lifetime is only during the kernel execution. Other than these memories, each thread within a grid can access *read-only*, *constant* and *texture* memories. These memories can be modified from the host only,

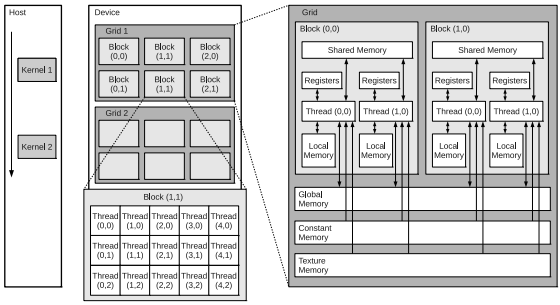

Fig. 1. CUDA GPU execution model [16]

and are useful for storing immutable data structures like lookup tables. The performance of any algorithmic implementation on the GPUs depends heavily on the proper utilization of this memory hierarchy.

2.3 Analyzing Parallelism Opportunities of Salsa20

The immense parallelism offered by the GPUs for acceleration can be better harnessed by a careful study of the parallelism opportunities offered by the application intended for mapping on it. The degree of parallelism also effects the potential throughput performance achievable after mapping. For Salsa20, we observe two categories of parallelism.

Functional Parallelism. As evident from Section 2.1, each block of 64 bytes of Salsa20 keystream can be independently generated and mixed with data to get the ciphertext. Salsa20/r has $r/2$ *DoubleRounds*, each comprising of a *ColumnRound* and a *RowRound*. These *Column* or *Row-Rounds* undergo 4 *ARXrounds* for each row/column. Hence a total of $16 \times r$ invertible *ARXrounds* complete the keystream generation for one block of Salsa20/r. A CUDA compatible device, capable of launching t parallel threads, each undertaking one data block of plaintext, will give a throughput of $(t \times 64)/(16 \times r \times \alpha)$ Bytes/sec if α is the time taken for one *ARXround* as depicted in Fig. 2. We ignore the final addition of *DoubleRound* output with the state vector for keystream generation since its overhead is negligible in comparison.

Data Parallelism. In Salsa20, each *QuarterRound* operates on either a row or a column of the 4x4 array. Each of the four *ARXrounds* constitutes of a *QuarterRound*, modifying exactly one value of that row or column. Hence 4 parallel *QuarterRounds* can be executed due to absence of inter-column/row dependence. Consequently, $16 \times r$ transformations of one Salsa20 block can be broken down as $4 \times r$ transformations mapped on 4 parallel threads giving a throughput of $(t \times 64)/(4 \times r \times \alpha)$ Bytes/sec, or 4 times higher than a single thread per block mapping. Exploiting further parallelism within one *ARXround* is not possible due to dependence of XOR (X) operation on the output of rotation (R) and addition (A) as shown in Fig. 2.

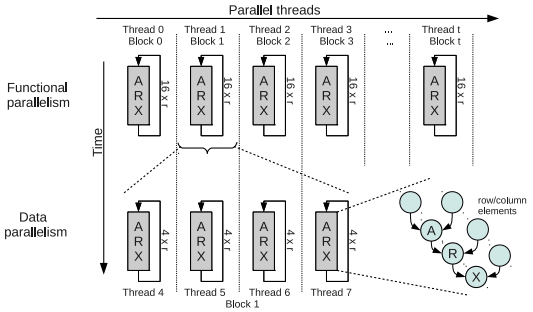

Fig. 2. Parallelism in Salsa20/r

For coding Salsa20 kernels employing functional parallelism (one thread per data block), internal registers and shared memory were used for storing results of *DoubleRounds* and X respectively. For manipulating data parallelism (four threads per data block) inter-thread communication is required within the threads of a threadblock. Therefore the results of *DoubleRounds* are also held in shared memory. For this implementation the need of thread-synchronization makes it lag behind in performance compared to single thread per block implementation. Experimentation of mapping AES on GPUs with different granularities also conform to our findings, as the best throughput performance is achieved when no synchronization is required between different threads [10,7]. Another reason for avoiding intra-block synchronization is that for most GPU devices is as follows. *The limited number of shared memory limits instruction-level parallelism by restricting the number of threads launched, lowering occupancy. Hence, for the rest of the discussion, we consider only the single thread per data block implementation due to its superior throughput performance.*

3 Batch Processing Framework

Salsa20 algorithm is a classic case of a parallelizable application, for which performance is dependent on the amount of parallel work received. For all such applications, a batch processing framework of operation is recommended. It is termed *batch* processing, since a batch of threads work simultaneously to encrypt one data block each and iterating in a loop for encryption of more plaintext. The batch of active threads die when all the data to be encrypted is exhausted.

3.1 CPU-GPU Interaction

Algorithm 1 explains the batch processing framework for encryption or decryption in a hybrid environment (CPU-GPU). We consider plaintext (P), given as 1-D data to be the input to the application. Inputs to the framework for Salsa20 encryption have already been explained in the functional description of the algorithm in Section 2.1. The byte-length of a data block for encryption or

decryption is called the *blocksize*. The initial state vector (X) is set up at the host machine using algorithm specific *Initialization* routine and transferred to the global memory (*gmem*). Assuming encryption of P having size larger than global memory (*gmem*) size, P is divided into chunks equal to size of *gmem*, termed as P_k. Every k^{th} iteration encrypts a portion of plaintext P_k into an equal sized ciphertext C_k (line no. 3). For simplicity, we assume the total size of plaintext to be a multiple of the size of *gmem*, in case of non-conformity, the number of data blocks forwarded to kernel for encryption is changed to the residue after division with $size(gmem)$ in the last iteration. For Salsa20, X is a 16-word array and its subscript represents its existence location, i.e., h, g, s, r representing host, global memory, shared memory and registers respectively. After the transfer of P_k to device's *gmem*, launch of kernel is kick-started in an iterative fashion. One batch of threads or *threadsPerGrid*, executed in parallel on device, is $blocksPerGrid \times threadsPerBlock$. In every iteration, when the kernel call is terminated, *gmem* contains the cipher text, that must be read out by the host (line no. 7) before writing the next plaintext chunk into the device memory (line no. 4).

Input: $key(k)$, $nonce(n)$, $counter(t)$, $constants(\phi_i)$, $rounds$, $blocksize$, $plaintext(P)$
Output: $ciphertext$ (C)
1 $X_h = Initialization(key, constants, counter, nonce)$;
2 $X_h : host \Rightarrow gmem$;
3 **for** k=*1* **to** $\lceil \frac{size(P)}{size(gmem)} \rceil$ **step** *1* **do**
4 $P_k : host \Rightarrow gmem$;
5 $Salsa20\_kernel \lll blocksPerGrid, threadsPerBlock \ggg (rounds, size(gmem)/blocksize)$;
6 $X_g : gmem \Rightarrow host$;
7 $C_k : gmem \Rightarrow host$;
end

Algorithm 1. Batch processing for a cryptographic kernel

3.2 The CUDA Kernel

Algorithm 2 is the CUDA kernel call and is executed on the GPU device. Although CUDA kernel functions do not have any output, the algorithm represents a pseudo-code and the output specified is not the output of the kernel function. Two local variables called *counter* and *batch* are declared and initialized, containing the unique threadID and the total number of threads in a batch respectively. Variable *counter* is used to update the counter in the state vector of Salsa20, incremented by the variable *batch* after every iteration. When a thread finishes encrypting a block, it encrypts again the block corresponding to that thread index plus the total number of active threads running (*batch*), which is constant and device dependent.

The state vector, residing in global memory, is first copied to faster *shmem*. As the size of global memory of newer NVIDIA GPUs is in GBs, a single batch of parallel threads each encrypting one data block, will not finish up the P_k,

```
   Input: rounds, dataBlocks, plaintext(P_k)
   Output: ciphertext (C_k)
 1 counter = blockDim.x * blockIdx.x + threadIdx.x;
 2 batch = gridDim.x × blockDim.x;
 3  X_g : gmem ⇒ shmem;
 4  for i=1 to dataBlocks step dataBlocks/batch do
 5  |   X_s = X_s + counter;
 6  |   X_s : shmem ⇒ regs;
 7  |   for j=1 to rounds step 2 do
 8  |   |   state_r = DoubleRound(state_r);
    |   end
 9  |   S_i = X_s ⊕ state_r;
10  |   P_ki : gmem ⇒ regs;
11  |   C_ki = P_ki ⊕ S_i;
12  |   C_ki : regs ⇒ gmem;
13  |  counter+ = batch
    end
14  X_s : shmem ⇒ gmem;
```

Algorithm 2. Salsa20 kernel

requiring iterations over variable i, as given in line no. 4. Here too, for the sake of simplicity, we consider the number of *dataBlocks* forwarded to the kernel for encryption or decryption to be a multiple of *batch* of threads. In case of non-conformity, the pseudo code can be modified to launch lesser number of threads in the batch in the last iteration. The state vector is updated with the counter value as given in line no. 5. Since threadID is different for each thread in a batch, all threads get a different state vector. The variable $state_r$ refers to the register copy of the state vector (it is copied from *shmem* to *regs* in line no. 6).

The value of *rounds* is either 8, 12 or 20 for various flavors of Salsa20/r. A copy of $state_r$ in thread-local registers apply *DoubleRound* transformations for $\frac{rounds}{2}$ times. One *block* of keystream, generated by XOR-ing the state vector with its transformed copy in local registers (line no. 9), is held in S_i.

The last step is the encryption of the plaintext with the generated keystream. Plaintext is read from *gmem*, one *block* at a time (P_{ki}), XOR-ed with the generated keystream to produce a block of ciphertext (C_{ki}) and then is written back to *gmem*. Saving of state vector into *gmem* is required before exiting the kernel, since its lifetime in shared memory lasts only as long as threadblock's lifetime.

3.3 Programming Recommendations

CUDA programmers are recommended to follow the guides [16,17] to achieve the best performance. We summarize some more relevant recommendations for good throughput performance when Algorithm 1 and 2 are mapped onto a GPU.

Avoiding threadBlock Switching Overhead. Each kernel launch on the device bears overheads of a kernel call, memory allocation and argument copying into the device. If the amount of work per kernel is small in comparison to the total workload, the run-time of the application is dominated by these overheads

instead of the actual computation time. In order to decrease these overheads, the amount of work per kernel call should be increased. Hence we resort to iterations computed inside a kernel call (loop indexed with i in Algorithm 2) to continue as long as the entire workload is finished instead of launching a new batch of threads. This strategy amortizes the overhead of multiple kernel calls across more computation and boosts throughput.

Reuse Memory. For cryptographic applications, the plaintext P_k once handed over to the device is not needed back by the host device. A prudent decision is to *overwrite* the plaintext with ciphertext in the *gmem*. It saves the iterations of loop indexed by k by half in Algorithm 1.

Data Coalescing: Global memory accesses incur a 100x access penalty compared to kernel local registers [16]. If these accesses are close to each other and dispatched in a group, they are *coalesced* as a single access. The device can read 4-byte, 8-byte, or 16-byte words from global memory into registers in a single instruction. Mixing of plaintext for generating ciphertext requires reading, XORing with keystream and writing back into the global memory as given by line no.s 10, 11 and 12 respectively (in Algorithm 2). Maximum memory coalescing that the device supports gives good saving in access time.

Autotune. The choice of grid dimensions, $blocksPerGrid$ and $threadsPerBlock$ is critical since it affects the throughput. It is discussed in detail in Section 4.

3.4 Optimization for Salsa20

For a given key, the initial state vector for multiple blocks of Salsa20 encryption remains the same except for a counter value, that is incremented for each block. Hence it can be treated as a constant array, while the counter is taken care of by each thread kernel individually by its $threadID$. Keeping the initial state vector in fast read-only constant memory, instead of shared memory, is therefore useful as constant memory is optimized for broadcast due to data coalescing. Since each block of Salsa20 requires reading of initial state vector twice, once before the *DoubleRound* iterations and once after it (line no. 6 and 9 respectively in Algorithm 2), the use of constant memory is highly suited. CUDA specific function *cudaMemcpyToSymbol* writes the initial state vector in the constant memory. This strategy cannot be generalized to all ciphers. However, a prudent use of a faster memory, whenever applicable, always enhances performance for CUDA applications. This factor alone boosts the peak throughput for Salsa20/12 (for 1 GB of plaintext) by 4 GBps.

4 Autotuning Framework for Performance Optimizations

In context of a CUDA back-end application, our autotuning framework automatically chooses tunable parameters of application mapping with the aim of

improving a designated Figure of Merit (FoM). The tunable parameters may be application specific, device dependent or compiler optimizations. Finding the optimal values of these parameters may require extensive experimentation on a case-by-case basis. Apart from promising a performance boost, another reason for developing an autotuning framework is the provision of portability across different devices belonging to the same architecture family. Some common figures of merit are listed below.

Occupancy. Device occupancy (or concurrency) is the ratio of active resident threads to the maximum number of resident threads whose resources can be stored on-chip simultaneously. Occupancy serves as a guideline for performance, but does not guarantee optimized throughput.

GFLOPs. Comparison of application's GFLOPs (Giga FLoating point Operations Per Second) against peak GFLOPs specified for a device. However, peak GFLOPs is quoted strictly for Floating point instructions.

Throughput. It is the measured output rate ($Bytes/sec$) of an application using timing functions on the device. Given the application in hand, we chose it to be our FoM.

The aim of an autotuning framework is to admit a large range of tunable parameters to CUDA application and select the one that makes the kernel run most efficiently. The range of these tunable parameters may be dependent on the constraints imposed by the device, application or both. The task of identifying *what* parameters should be subjected to tuning is critical, since they vary widely between algorithms. Keeping in mind the operation flow of the framework, we classify these parameters optimizations as *Compiler-specific* and *Device-specific*.

4.1 Device-Specific Optimizations

Device-specific optimizations are the ones that are tweakable at the runtime of the application, e.g., device grid dimensions. The given CUDA application is enhanced by the addition of the provision of the kernel variants being subjected to all possible combinations of these parameters after pruning by certain checks. Benchmarking for throughput is also added for later use. As shown in Fig. 3, enhancement of the application with the addition of device specific optimizations and benchmarking provision is the first step of the autotuning framework. However, execution of these enhancements does not manifest before various compiler optimizations have been done and multiple copies of the code executables are ready. Final runs of these programs result in sifting the fastest implementation with recommended parametrization choices.

Algorithm 3 gives the pseudo-code of the device specific optimizations setup. Out of the 4 different inputs, device properties (obtained by *cudaDeviceProp* function) and compute capability properties (obtained from a lookup table corresponding to major and minor compute capability) are device dependent. Kernel constraints are application dependent and are obtained after compilation of the

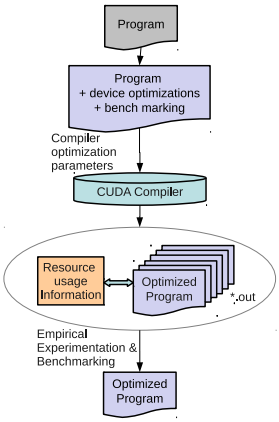

Fig. 3. Autotuning Framework Flowchart

program. *minOccupancy* is specified by user to filter out kernels with too low occupancy from experimentation. A higher value will prune the search space more but might miss the fastest kernels too; a lower value, on the other hand, will compromise on speed due to large search space for the fastest kernel.

All possible values of the two critical device parameters, $threadsPerBlock$ and $blocksPerSM$, are considered for experimentation within their permitted range in nested loops as specified by line no. 1 and 2. Threadblock size should always be a multiple of $warpSize$, because kernels issue instructions in warps. The next four lines of code calculate the resource budget for the current configuration of the device parameters. Total resident $threadsPerSM$ is a product of $blocksPerSM$ and $threadsPerBlock$. The next two lines calculate the resource usage of register and shared memory per kernel from the application specific parameters.

A kernel is subjected to experimentation with a set of possible device parameters configuration after pruning by 4 checks as specified from line no. 7 to 10. Check 1 ensures that the maximum number of possible threads executable on an SM is not exceeded. Check 2 and 3 ensure that the register budget and the shared memory budget specific to one kernel is not exceeded. Check 4 makes sure that the current device configuration has an occupancy higher than the minimum specified by the user. Functions to calculate the time elapsed before and after the kernel call are used to carryout the time duration benchmarking.

4.2 Compiler-Specific Optimizations

Compiler-specific optimizations are the ones that are subjected to the *nvcc* compiler at the compile time, e.g., preprocessor directives. As shown in Fig. 3, this step generates a number of optimized programs, each pertaining to a possible permutation out of the range of all the compiler-specific optimization parameters. Other than getting these executables, compiler generates information regarding the resource usage of the application in question, i.e., global, constant memory usage per grid, register count, local memory and shared memory usage

Input: 4 types of inputs:
1. **Device**: $warpSize$, $maxRegsPerBlock$, $maxShMemPerBlock$, $maxThreadsPerBlock$, $maxSM$;
2. **Compute Capability**: $maxBlocksPerSM$, $maxWarpsPerSM$;
3. **Kernel constraints**: $regsPerThread$, $shMemPerThread$;
4. **User constraints**: $minOccupancy$.
Output: Valid parameter variants for benchmarking

1 **for** $threadsPerBlock = warpSize$ **to** $maxThreadsPerBlock$ **step** $warpSize$ **do**
2 **for** $blocksPerSM = 1$ **to** $maxBlocksPerSM$ **step** 1 **do**
3 $threadsPerSM = blocksPerSM \times threadsPerBlock$;
4 $regsPerSM = threadsPerSM \times regsPerThread$;
5 $ShMemPerSM = threadsPerSM \times shMemPerThread$;
6 $occupancy = \frac{threadsPerSM}{(maxWarpsPerSM \times warpSize)}$;
7 ***Check1:*** $threadsPerSM \leq (maxWarpsPerSM \times warpSize)$;
8 ***Check2:*** $regsPerSM \leq maxRegsPerBlock$;
9 ***Check3:*** $ShMemPerSM \leq maxShMemPerBlock$;
10 ***Check4:*** $occupancy \geq minOccupancy$;
11 $blocksPerGrid = maxSM \times blocksPerSM$;
12 $success = kernel\_launch \lll blocksPerGrid, threadsPerBlock \ggg$
 end
end

Algorithm 3. Device-specific optimizations: Search space generation and pruning

per kernel. These resources are used as constraints during the empirical experimentation before reaching the performance-optimized kernel. The two compiler-specific optimizations applicable for the current application are loop unrolling and restricting per kernel register budget. Both of these manifest as a compromise between parallelism and register pressure.

Unroll Factor. Loop Unrolling replaces the main body of a loop with multiple copies of itself, adjusting the control logic accordingly. *#pragma unroll* n is a preprocessing directive where n defines the unroll factor ($n = 1$ means no unrolling, $n = k$ means full unrolling, where the trip count of the loop is k). On the positive side, loop unrolling results in reduced dynamic instructions (compare and jump) count, boosting speedup. On the negative side, however, unrolling increases the total instruction count of the loop body and leads to an increased register pressure. Since registers are partitioned among threadblocks, an increased use of registers per threadblock reduces the device occupancy. This may or may not affect the throughput and requires experimentation for assurance. For Salsa20, the three flavors of the algorithm iterate for 4, 6 and 10 times for Salsa20/8, Salsa20/12 and Salsa20/20 respectively. For each of these, unroll factor from no unrolling to maximum unrolling is considered for experimentation.

Register Budget. A CUDA programmer can force a restricted number of registers by specifying cuda-nvcc-opts=-maxrregcount R, limiting the register use to R per kernel. Lowering register count allows increased occupancy which may

result in increased throughput. On a negative note, it may cause spilling into the local memory when the register limit is exceeded. The local memory is as slow as the global memory and spilling into it can consequently cause severe performance degradation despite the higher occupancy. For all Salsa20 kernel variants, the register budget varies from 26 to 43 for no unrolling to maximum unrolling. For parametrization of register budget, all the multiples of 5 within this minimum and maximum register use are considered. Lowering the register budget any further than the minimum limit causes spilling and hence these cases are omitted from benchmarking.

5 Results and Discussion

In this section, we present detailed experimental results and compare them with the available state-of-the art benchmarks.

5.1 Experimental Setup

Throughput performances of Salsa20 stream cipher is reported for NVIDIA GeForce GTX 590, although the autotuning framework is generic enough to cater for any Fermi NVIDIA GPU device. To quantify the speedup against a general purpose computer, a single threaded application program written in C was run on an AMD Phenom 1055T Processor (clockspeed 2.8 GHz) with 8 GB of RAM and Linux operating system. For a good approximation, each experiment was run 100 times and the timing results were averaged.

5.2 Search Space Generation and Pruning

Table 1 gives the possible range of parameters for the Salsa20 application kernel for NVIDIA GeForce GTX 590. The register budget range was chosen within the minimum and the maximum register requirements with no unroll and full unroll respectively, in steps of 5. All possible values of the unroll factor are taken into consideration. Grid dimension's permitted range is dependent on the device. Minimum occupancy was chosen to be 0.16, i.e., 256 threads per SM (256/1536), since tests with selective lower occupancies gave inferior throughputs.

In order to give an idea of the magnitude of the possible kernel configurations on which the autotuning framework carried out experimentation, some numbers are presented. For Salsa20/20, 10 possible unroll factors generate 10

Table 1. Range of parameters for autotuning Salsa20 kernel on a GTX 590

	Parameter	Range
Compiler-specific optimizations	Register budget	$26, 30, 35, 40, 43$
	Unroll factor	$1, 2, ..., r/2$
Device-specific optimizations	Threads per block	$32, 64, ..., 1024$
	Blocks per SM	$1, 2, ...8$
	Minimum occupancy	0.16

optimized versions of the program, each a candidate for experimentation. Further, each of them is subjected to restricted register budget to generate multiple versions. Considering only the case of full unroll and unrestricted use of registers for Salsa20/20 kernel subjected to device specific constraints, the number of allowed grid size combinations comes out to be 55. Extensive experimentation of all possible combinations of parameters after pruning was carried out for benchmarking.

5.3 Compile Time Optimization of Register Pressure

To find the optimal trade-off between active concurrent threads and registers availability per thread, two parameters have been tweaked. These are the restricted use of register budget and register unrolling. Restricting the use of registers per thread was benchmarked to always have a deteriorating effect on the throughput, in spite of increased occupancy. Changing the unroll factor, however, gives improved performance results. Fig. 4(a) gives the effect of unrolling factor on the registers used per thread for Salsa20/8, Salsa20/12 and Salsa20/20. Since Salsa20/20 requires 10 loop iterations of *DoubleRound* function, unrolling factors range from 1 to 10. Unrolling an n-iteration loop more than n times makes no difference and is considered by the CUDA compiler as a full unroll. Consequently, the unrolling of Salsa20/8 and Salsa20/12 kernels show no change after unroll factor of 4 and 6 respectively.

5.4 Register Unroll vs. Throughput

The register unrolling positively effects the throughput in general as shown in Fig. 4(b). These results are obtained after benchmarking more than 2500 kernel variants considering the full range of unroll factors and all grid dimensions supported by the device. Constraint of *minOccupancy* is applied, but register use restriction at compile time is skipped since it does not boost the throughput. The size of plaintext is kept 32 KB for encryption by the kernel.

Interestingly, the highest throughput for a Salsa20 kernel variant is obtained when the inner loop is unrolled by a factor one *less* than the full unrolling. Considering the case of Salsa20/20, the registers used per kernel remain unchanged till the unroll factor is raised from 1 till half of the full unroll factor, i.e., 5 as given in Fig. 4(a). For an unroll factor of 6 to 9, the no. of registers per kernel increases from 39 and saturates to a maximum of 43 for the full unroll. By varying the grid dimension, we find that the best throughput figures are obtained when the unroll factor is 9. Although partially unrolled loops may require some cleanup code to account for loop iterations that are not an exact multiple of the unrolling factor, it may or may not decrease the performance in practice. Hence considering a range of unroll factors for experimentation proves beneficial in reaching the optimized performance.

Similarly, for the other flavors of Salsa20, i.e., for Salsa20/8 and Salsa20/12, the highest throughput is achieved when unroll factor is 3 and 5 respectively, as shown in Table 2. For these unroll factors, the register usage in the three kernels

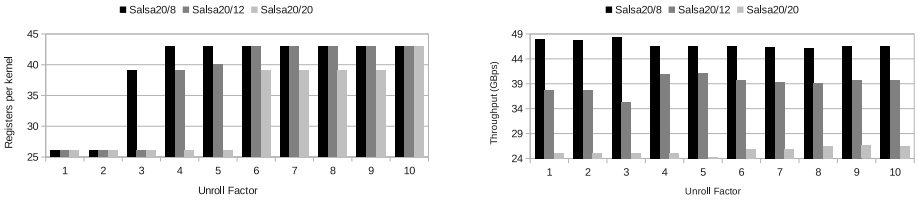

Fig. 4. (a) Register pressure and (b) kernel throughput against unroll factor

Table 2. Salsa20 optimized parameters for GTX 590 (32 KB plaintext)

	Autotuned					Hand-tuned	
Kernel variant	Unroll factor	Threads per block	Blocks per SM	Device occupancy	Throughput (GBps)	Throughput (GBps)	Improvement (GBps)
Salsa20/8	3	448	1	0.29	48.29	45.77	2.52
Salsa20/12	5	320	2	0.41	41.14	39.91	1.23
Salsa20/20	9	512	1	0.33	26.60	24.42	2.18

restricts the occupancy of the device. With 40 registers, no more than 25 warps can be launched on each SM for GTX 590 (register limit on the device being 32K) restricting the device occupancy to 0.52. Table 2 gives the throughput performance with hand-tuned parametrization for maximum device occupancy. The improvement in throughput obtained emphasizes the need of autotuning as a necessary requirement for performance enhancement of a CUDA application.

5.5 Workload vs. Performance

Fig. 5 shows the performance of Salsa20 variants on a GTX 590 for varying plaintext sizes. For throughput estimation, the plaintext blocksize is increased from 1 Byte till 1 GB. For GTX 590, we cannot go beyond 1.5 GB in one batch of plaintext encryption due to the size of the global memory (obtained from *cudaDeviceProp* function). It is easy to see that the performance of Salsa20 is highly dependent on the amount of parallel work it receives. We find the

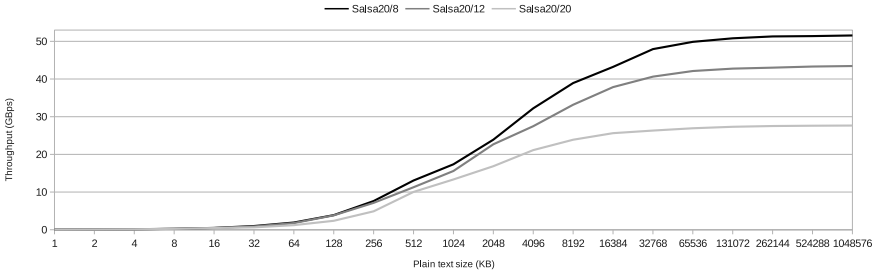

Fig. 5. Salsa20 throughput on GTX 590 for varying plaintext sizes (w/o mem trans.)

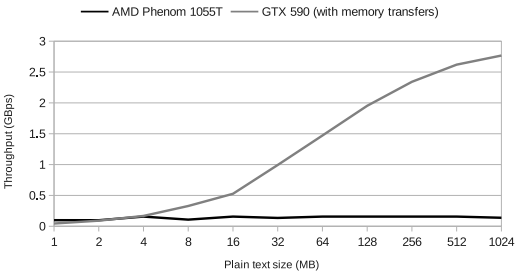

Fig. 6. Salsa20 throughput comparison on a CPU and GPU

peak throughput performance of Salsa20/8, Salsa20/12 and Salsa20/20 to reach 51.55, 43.44 and 27.65 GBps, respectively, outperforming the best reported GPU implementations so far.

We also took into account the overhead attributable to the plaintext data transfer from CPU to GPU and ciphertext data transfer from GPU to CPU to get the effective throughput, as given in Fig. 6. The peak performance for the GPU under consideration reaches around 2.8 GBps with memory transfer overheads. The severe drop in the throughput clearly indicates that the bottleneck in the system is the data transfer bandwidth: PCIe bandwidth. For the host CPU, i.e., for AMD Phenom 1055T, the peak performance reaches 157 MBps.

5.6 Comparison with Other Works

Table 3 gives a comparison of our work on Salsa20 acceleration on GPUs with the results presented by D. Stefan [13] and S. Neves [12]. We also compare the performance with the fastest reported AES implementation on GPUs [7]. For a fair comparison, we scale up the throughput figures of other devices (without memory transfers) in accordance with our newer GPU device by considering the number of processor cores per device. Although the processing frequency of our device is slower in comparison, we ignore this factor for scaling the throughput calculation. The throughputs (GBps) per core from [13,12] is (5.3/480 and 9/192), which is multiplied with the number of cores of our device (512) to get 5.7 GBps and 24 Gbps respectively. These scaled throughputs are surpassed by our peak performance of 43.44 GBps. In [13], the maximum throughput of 5.3 GBps (without memory transfers) was achieved for Trivium and that is also far behind (even after scaling) the throughput of our implementation. Scaling on similar lines, the AES implementation by Iwai *et al.* [7] results in a throughput of 9.3 GBps which is about 4.6 times slower than our reported peak performance for Salsa20/12. This re-scaling formula would be invalid for throughput calculation with memory transfers, since, like most of the cryptographic algorithms, Salsa20 and AES are data intensive in nature and show performance dependence on external memory access speed. The main factor contributing to our performance gain is the use of constant memory instead of shared memory for keeping the copy

Table 3. Comparison of peak performance (Tp stands for Throughput)

	D. Stefan[13]	S. Neves[12]	This work	Iwai *et al.*[7]
Algorithm	Salsa20/12	Salsa20/12	Salsa20/12	AES
NVIDIA device	GTX 295	GTX 260	GTX 590 (one GF110)	GTX 285
Release (DD/MM/YYYY)	08/01/2009	16/06/2008	24/03/2011	15/01/2009
Compute Capability	1.3	1.2	2.0	1.3
Processor cores	480	192	512	240
Shader Frequency (MHz)	1242	1350	1215	1470
Threads / Block	256	256	320	512
Tp (GBps)(w/ m)	-	1.3	2.8	2.8
Tp (GBps)(w/o m)	5.3	9	43.44	4.4
Scaled Tp (GBps)(w/o m)	5.7	24	43.44	9.3

of the initial state vector. Moreover, our autotuning framework to sift out the choice of parameters maximizing throughput also helps in reaching the claimed performance. According to Table 3, our throughput with memory transfer is the same as the best result known for AES. However, the claimed speed of 2.8 GBps for AES with memory transfers is reported after being improved by 68% by optimization of overlapping GPU processing and data transfers [7]. Our current framework does not support this optimization. However, the search for an optimal transfer blocksize to hide the transfer latency is on our roadmap.

6 Conclusion

We present an autotuning framework for Salsa20 series of stream cipher. It not only guarantees fast portability for Fermi GPUs and optimized throughput performance, but it can be generalized and extended to other massive parallel cryptographic operations also. Moreover, our peak throughput figure of 43.44 GBps surpasses the fastest GPU based performance reported so far for all stream ciphers (both hardware and software) in the eSTREAM portfolio [12,13,8], as well as AES in CTR mode [7].

Regarding the future work, we plan to extend our efforts in different directions. Firstly, we intend to benchmark GPU implementation of other parallelizable stream ciphers (e.g., ChaCha [5], a variant of Salsa20). Secondly, we plan to extend our autotuning framework to handle plaintext data ordered as a 2-D array for multimedia applications. Generalization of our autotuning framework for optimizing any symmetric key cryptographic kernel is also intended.

References

1. eSTREAM: the ECRYPT Stream Cipher Project, `http://www.ecrypt.eu.org/stream`
2. Bernstein, D.J.: Hash functions and ciphers. In Notes on the ECRYPT Stream Cipher Project (eSTREAM), `http://cr.yp.to/streamciphers/why.html`
3. Bernstein, D.J.: eBACS: ECRYPT Benchmarking of Cryptographic Systems, `http://bench.cr.yp.to/results-stream.html`

4. Bernstein, D.J.: The salsa20 family of stream ciphers. In: Robshaw, M., Billet, O. (eds.) New Stream Cipher Designs. LNCS, vol. 4986, pp. 84–97. Springer, Heidelberg (2008)
5. Bernstein, D.J.: ChaCha, a variant of Salsa20. Workshop Record of SASC 2008: The State of the Art of Stream Ciphers, http://cr.yp.to/papers.html#chacha
6. Biagio, A., Barenghi, A., Agosta, G., Pelosi, G.: Design of a parallel AES for graphics hardware using the CUDA framework. In: International Symposium on Parallel & Distributed Processing (IPDPS), pp. 1–8. IEEE (2009)
7. Iwai, K., Nishikawa, N., Kurokawa, T.: Acceleration of AES encryption on CUDA GPU. International Journal of Networking and Computing 2(1), 131–145 (2012)
8. Khalid, A., Bagchi, D., Paul, G., Chattopadhyay, A.: Optimized GPU implementation and performance analysis of HC series of stream ciphers. In: Kwon, T., Lee, M.-K., Kwon, D. (eds.) ICISC 2012. LNCS, vol. 7839, pp. 293–308. Springer, Heidelberg (2013), http://eprint.iacr.org/2013/059
9. Kurzak, J., Tomov, S., Dongarra, J.: Autotuning GEMM kernels for the Fermi GPU. In: Transactions on Parallel and Distributed Systems, pp. 2045–2057. IEEE (2012)
10. Manavski, S.A.: CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In: International Signal Processing and Communications (ICSPC), pp. 65–68. IEEE (2007)
11. Murthy, G.S., Ravishankar, M., Baskaran, M.M., Sadayappan, P.: Optimal loop unrolling for GPGPU programs. In: International Symposium on Parallel & Distributed Processing (IPDPS), pp. 1–11. IEEE (2010)
12. Neves, S.: Cryptography in GPUs. Master's thesis (2009), http://eden.dei.uc.pt/~sneves/pubs
13. Stefan, D.: Analysis and Implementation of eSTREAM and SHA-3 Cryptographic Algorithms. Master's thesis (2011), https://github.com/deian/gSTREAM.
14. Wu, H.: The Stream Cipher HC-128, http://www.ecrypt.eu.org/stream/hcp3.html
15. NVIDIA's Next Generation CUDA Compute Architecture: Fermi, http://stanford-cs193g-sp2010.googlecode.com/svn/trunk/lectures/lecture_4/cuda_memories.pdf
16. CUDA C Programming Guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#ptx-compatibility
17. CUDA C Best Practices Guide, http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html
18. Basic Linear Algebra Subprograms Technical Forum Standard (August 2001), http://www.netlib.org/blas/blast-forum/blas-report.ps