HiCrypt: C to CUDA translator for symmetric block ciphers

Keisuke Iwai, Naoki Nishikawa and Takakazu Kurokawa

Department of Computer Science
National Defense Academy of JAPAN
1-10-20 Hashirimizu, Yokosuka-shi 239-8686, JAPAN
Email: {iwai, ed11001, kuro}@nda.ac.jp

Abstract—Many-core computer systems with GPUs are coming into mainstream use from high-end computing, including supercomputers, to embedded processors. Consequently, the implementation of cryptographic methods on GPUs is also becoming popular because of such systems’ performance. However, many factors affect the performance of GPUs. To cope with this problem, we developed a new translator, HiCrypt, which can generate an optimized CUDA program from a cipher program written in a standard C language with directives. Users need only annotate variables and an encoding/decoding function, which are characteristics of cipher programs, with directives. To evaluate HiCrypt, three representative cipher programs are translated into CUDA programs by HiCrypt translator prototype. Generated programs perform high throughput almost identical to hand optimized CUDA programs for all three cipher programs. HiCrypt will contribute to development of new and various symmetric block ciphers using a GPU accelerator.

I. INTRODUCTION

Small and inexpensive portable devices such as smartphones and tablet PCs have become adopted explosively into widespread use. The necessity for high-speed cryptographic technology for use with such devices has also increased. However, demand for high-performance encryption at high-end fields such as data centers holding enormous data, so-called “Big-data”, continues growing too. These facts indicate that the field of security countermeasures is widely necessary. In fact, Intel and AMD have presented an AES-NI[1] instruction set that is implemented on a processor with an embedded cryptographic engine as a solution for high-speed encryption for a standard symmetric block cipher called Advanced Encryption Standard (AES)[2].

It drastically improves the speed of AES encryption by hardware acceleration. However, if AES is not used for a data encryption, then its high efficiency will not be valuable in the least, and the circuit area will have been wasted.

To date, heterogeneous many-core processors with a CPU–GPU (Graphics Processing Unit) integration architecture are becoming popular for their high-speed performance and low power consumption. A many-core architecture is scalable because the number of processor cores can be adjusted for use. For this reason, accelerating cryptographic processing while using a many-core processor with scalable architecture is expected to constitute one solution for high-speed and scalable cryptographic processing[3][4][5]. High-speed cryptographic processing accelerated by a many-core architecture such as that of GPGPUs will resolve problems that cannot be solved by AES-NI because those GPGPU accelerations based on a software implementation can accept any cryptographic algorithm. Moreover, such accelerators entail no extra hardware costs such as those associated with specialized circuits for cryptography because GPGPUs are embedded as a general purpose accelerator.

However another problem remains: difficulty of the program for GPGPU. Many factors are useful to exploit the performance of GPGPU for cipher applications. Particularly, problems such as optimizing the way of using various kinds of memory of CUDA and considering computation granularity trouble many programmers. To address these problems, we suggest a special-purpose translator for symmetric block ciphers: High-performance CRYPTOgraphic Program Translator (HiCrypt). HiCrypt generates optimized GPGPU programs using CUDA[6] from cipher programs written in a standard C language with directives. HiCrypt constitutes a simple method to produce optimized cipher programs with CUDA because the translator requires annotation only of elements that are characteristic for cipher programs.

The remainder of this paper is organized as follows. In the next two sections, we briefly introduce the CUDA architecture and optimization techniques for symmetric block ciphers on CUDA. In Section IV, details of HiCrypt directives are presented. In Section V, the translator design is shown. Implementation of the translator prototype and its evaluation using three representative 128-bit symmetric block cipher programs, AES, Camellia, and SC2000, are described in Section VI. Finally we explain our conclusions in Section VII.

II. CUDA

In CUDA, a GPGPU development environment released by Nvidia Corp., a GPU of the graphics processor is hidden as a parallel computing architecture. As shown in Figure II, The GPU has N × multiprocessors (MP) and global memory. Each MP has M × scalar processors (SP), shared memory, several 32-bit registers, and a shared instruction unit. The shared memory is a multi-port memory with multiple banks. In addition, the global memory has a special area called constant
memory, for which the access time is less than the global memory if each thread accesses the same address. As described in this paper, we chose an NVIDIA Tesla C2050 from the CUDA GPU family. Tesla C2050 incorporates the latest Fermi architecture, including 14 MPs; one MP incorporates 32 SPs.

Parallel processing on CUDA is attributed to many-thread parallelism. The resources in MPs are assigned evenly to numerous active threads. Threads can be processed effectively, provided that the number of threads is in multiples of 32. Especially, the SP pipeline can be kept filled if more than 192 threads are activated, which engenders a considerable performance increase[7].

III. TYPICAL IMPLEMENTATION OF HIGH-SPEED SYMMETRIC BLOCK CIPHERS

AES is adopted as an example of symmetric block cipher in this paper, although HiCrypt tries to translate many symmetric block ciphers. Moreover, some conditions are fixed to simplify the description herein. For example, the secret key size is fixed to 128-bit. The calculation time of an extended key, hereinafter referred to as “key”, which is calculated from the secret key is not discussed because it is much smaller than the encryption time. Moreover, an encoding algorithm of AES is only discussed because decoding will apply the same technique as that shown in the encoding.

A. AES

AES is a symmetric block cipher introduced in 2001 by NIST[2]. Although 128-bit, 192-bit, and 256-bit key size can be selected, we discuss only 128-bit in this paper. Its algorithm of the 128-bit key defines 10-round processes. Each round includes four transformations: SubBytes, ShiftRows, MixColumns, and AddRoundKey. The final round differs slightly from the other rounds; it does not include MixColumns.

In AES, a round process can be combined into a transformation simply using a look-up table called T-box and XOR operation[5]. Letting $a$ be the round input, which is divided into four inputs $a_0$, $a_1$, $a_2$, $a_3$, each of which consists of 32 bits, the round output $e$ is represented as

$$e_j = T_0[a_0,j] \oplus T_1[a_1,j+1] \oplus T_2[a_2,j+2] \oplus T_3[a_3,j+3] \oplus k_j$$

where $T_0$, $T_1$, $T_2$, and $T_3$ are look-up tables and $k_j$ is the $j$-th column of a round key. This algorithm requires only four look-up table transformations and four XOR operations. Using a pre-computed look-up table such as T-box is valid for other symmetric block ciphers, but the look-up table requires more memory space. In fact, almost all software implementations of the block cipher use the look-up table.

Block ciphers have some modes - Electric Code Book (ECB), Cipher Block Chaining (CBC), CountTeR (CTR) and so on. Electronic Code Book (ECB), CountTeR (CTR), or Xor-encrypt-xor Tweakeable code book mode with ciphertext Stealing (XTS)[9] are known as parallelizable modes in block cipher. ECB uses a single key applied to all plaintexts. CTR uses a key stream generated from a secret key and combined with plaintexts. In the CTR mode, the generation of the key stream is conducted in the same manner as ECB. In the XTS mode, plaintexts are encrypted using two ECB modes. Therefore, we discussed only ECB mode to evaluate the parallel processing of symmetric block ciphers.

B. Optimization for AES on CUDA

Some previous works had clarified which techniques are valid for the AES program with CUDA. Biagio et al. [10] described a study of the relations between the parallelizing granularity and its performance. The effect of the number of threads and file size are also shown. Iwai et al. [11] further discussed granularity and the variety of memory allocations for variables of cipher programs. Efficient CUDA programs make a difference of more than 50 times in performance. This section explains the optimization technique introduced in a previous work.

1) Granularity: In 128-bit symmetric block ciphers, the input data are separated specifically to 128-bit block size and are then encrypted. Each 128-bit block can be processed by each thread in parallel. There is greater computation granularity such as 64-bit, 32-bit and 16-bit per thread. Even greater 128-bit per thread granularity has the benefit of no synchronization instruction during parallel processing by multiple threads compared with the other granularity. For this reason, a HiCrypt translator set the computation granularity to 128-bit per thread in parallel encryption processing on GPUs.

2) Memory allocation: Generally in optimized software implementation of symmetric block ciphers, the same look-up tables are referred multiple times. Their contents are not modified. Therefore, tables stored in shared memory with low access latency engender higher performance than those stored in global memory. If the cache mechanism works well, then constant memory will provide the same low latency as shared memory. However, the look-up table operation of block cipher requires random access[11]. Therefore, constant memory should be a wrong selection to store for such look-up tables.
By contrast, the key can be stored in the shared memory or the constant memory because it is referred as common data by all threads and accessed in a regular pattern.

3) Thread management: The following two solutions should exist to parallelize the encryption process of 128-bit block ciphers. One is to generate the same number of threads as the number of 128-bit blocks that construct a plaintext. Another one is to process several 128-bit blocks in a repetitive fashion by each thread. This technique shows 3–5% improvement over the first because look-up tables on a shared memory can be reused.

IV. DESIGN OF THE DIRECTIVES

A. Directives

Directives of HiCrypt are specified with the #pragma mechanism. The syntax of directive is displayed as described below.

#pragma cryptcuda directive-name [clause] new-line

Each directive starts with #pragma cryptcuda. The remainder of the directive follows the C conventions for program. The format of this directive is similar to OpenACC[12] and so on. All directives defined by HiCrypt are shown in Table I with their brief descriptions.

Directives of HiCrypt consist of directive groups of two kinds. The first one includes must directives, which annotate elements for characteristics of cipher programs. Another group consists of parameters for optimization and hints for a translator such as the size of the plaintext. Must directives, plaintext, ciphertext, key, table, kernelCall, and kernelFunc, are introduced here. Directives plaintext, ciphertext, key, and table annotate variables. They apply to an immediately following statement that is a variable declaration.

The remaining two must directives, kernelCall and kernelFunc, annotate an encoding function call and its definition. kernelCall applies to an immediately following single statement that encodes function call so that the function can be offloaded to GPU. kernelFunc applies to an immediately following structured function definition block that is definition of function annotated kernelCall. The structured function definition block will be translated by a CUDA C program for a CUDA compiler to execute by the GPU. The optional directive device provides a solution to call functions from an inner function, which is annotated as kernelFunc.

Other directives for optimization are arch, threadNum and blockNum. arch defines an architecture of CUDA GPU such as sm_20 and sm_13. It provides a hint for optimization to the translator. Furthermore, threadNum and blockNum provide a solution when users would like to optimize the number of threads and thread blocks manually, although HiCrypt has default numbers of them.

Directives filesize and blocksizeOfCipher provide a hint of a targeted cipher algorithm to the translator. If filesize and blocksizeOfCipher are defined, then the translator will work reliably, but the translator will seek the file size of a plaintext and the block size of a cipher algorithm.

B. Restrictions

To use HiCrypt directives, some restrictions must be made on a program format. This section shows all restrictions required by the translator. Variables, which represent plaintext, ciphertext, and key, must be declared as a single-dimension array. Look-up tables annotated by table must be declared as single-dimension arrays too, although look-up tables for symmetric block cipher are sometimes declared as multi-dimension arrays. In any case, most optimized programs for a symmetric block cipher use look-up tables that are declared as a single-dimension array. These restrictions are presented in Table II. These pointer variables shown in Table II must have no aliases because the translator uses their names directly to generate a CUDA program.

There are two additional restrictions. The first is a restriction on order and members of argument of the function definition, shown as below:

```c
void function-name(integertype *plaintext, integertype *ciphertext, integertype *key, integertype numberofblocks);
```

Types of the dummy arguments except numberofblocks accept any integer pointer type, such as unsigned char*, int* and long*. The type of the plaintextpointer and ciphertextpointer must be the same. Moreover, dummy arguments plaintextpointer, ciphertextpointer and key must have no aliases.

Another restriction is format of the “for” loop, which encodes a cipher block repetitively in an encoding function annotated by the directive. The loop must be written according to the format shown below:

```c
for (counter=0; counter < numberofblock; counter++, plaintextpointer += blocksize, ciphertextpointer += blocksize);
```

The parameter numberofblock represents the number of plaintext blocks and blocksize represents the block size of the block cipher. For example, the block size of AES is described as “16” in byte size or “4” in word (32-bit) size.

These restrictions must be accepted to use HiCrypt. However, these restrictions are reasonable for users because many optimized programs for symmetric ciphers are written similarly. To explain details of these directives and the behavior of the translator, a base code for AES with HiCrypt’s directives is shown in Figure 2 as an example. This code shows only important statements and structures of the program.
TABLE I
BRIEF DESCRIPTIONS OF THE DIRECTIVES.

<table>
<thead>
<tr>
<th>Annotate variables</th>
<th>For optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>plaintext</td>
<td>arch†</td>
</tr>
<tr>
<td>ciphertext</td>
<td>ciphertext, pinter</td>
</tr>
<tr>
<td>key</td>
<td>threadNum†</td>
</tr>
<tr>
<td>table</td>
<td>blockNum†</td>
</tr>
<tr>
<td></td>
<td>CUDA architecture</td>
</tr>
</tbody>
</table>

For functions

- kernelCall: Earmark of kernel call
- kernelFunc: Kernel function definition
- device: Definition of function called from kernel

<table>
<thead>
<tr>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>filesize†</td>
</tr>
<tr>
<td>key, pointer</td>
</tr>
<tr>
<td>threadNum†</td>
</tr>
<tr>
<td>Number of threads</td>
</tr>
<tr>
<td>Number of thread blocks</td>
</tr>
</tbody>
</table>

†Directives operated by themselves with a clause.

```
// <abbreviate program header>
// Compute capability
// for CUDA (optional)
#pragma cryptcuda arch sm_20

// look-up tables
#pragma cryptcuda table
const char Sbox1[256]={...};
#pragma cryptcuda table
const char Sbox2[256]={...};
// <and more tables...>

//prototype
void AESEncode(char*, char*, char*, int);
.. main()
#pragma cryptcuda plaintext
char *pt;
#pragma cryptcuda ciphertext
char *ct;
#pragma cryptcuda key
char key[64];
.. // pt, ct and key initialize
pt = (char *)malloc(pt, sizeof(char)*FILESIZE);
```

V. DESIGN OF THE TRANSLATOR

A. Behavior overview of the translator

An implemented translator prototype described in Section VI scans a source program twice. On the first pass, it finds all directives and annotated elements in the source program. Then it stores them in its working memory, called a directive table. Contents of the directive table for AES are shown in Table III for example. On the second pass, it translates statements of the source program annotated by the directives to an optimized CUDA program. This transformation technique consists of four primary parts, creating new variables, generating allocate and transfer codes, generating a kernel function call, and translating a kernel function definition.

B. Creating new variables

1) Plaintext and Ciphertext: Two variables pt annotated by plaintext and ct annotated by ciphertext in Figure

2 must be allocated on the global memory of a CUDA GPU. The translator creates new variables to which are added prefix “d_” to the basename to allocate them on the global memory. Their declarations are also created at the beginning of the inner main() function.

TABLE III
DIRECTIVE TABLE FOR AES BASE CODE.

<table>
<thead>
<tr>
<th>Directive ID</th>
<th>Type</th>
<th>Size</th>
<th>Line</th>
<th>Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>plaintext</td>
<td>pt</td>
<td>char*</td>
<td>FILESIZE</td>
<td>593</td>
</tr>
<tr>
<td>ciphertext</td>
<td>ct</td>
<td>char*</td>
<td>FILESIZE</td>
<td>595</td>
</tr>
<tr>
<td>key</td>
<td>e_key</td>
<td>char*</td>
<td>64</td>
<td>591</td>
</tr>
<tr>
<td>table</td>
<td>Sbox1</td>
<td>uint32[]</td>
<td>256</td>
<td>52</td>
</tr>
<tr>
<td>table</td>
<td>Sbox2</td>
<td>uint32[]</td>
<td>256</td>
<td>187</td>
</tr>
<tr>
<td>table</td>
<td>Sbox3</td>
<td>uint32[]</td>
<td>256</td>
<td>321</td>
</tr>
<tr>
<td>table</td>
<td>Sbox4</td>
<td>uint32[]</td>
<td>256</td>
<td>484</td>
</tr>
<tr>
<td>kernelCall</td>
<td>AESEncode</td>
<td>NA</td>
<td>NA</td>
<td>631</td>
</tr>
<tr>
<td>kernelFunc</td>
<td>AESEncode</td>
<td>NA</td>
<td>NA</td>
<td>659</td>
</tr>
</tbody>
</table>
2) **Key and Table:** Variables annotated by `key` and `table` are allocated on the constant memory. The translator creates new variables to which are added prefix “c_” to the basename to allocate these variables in the constant memory. Their declaration is also created in front of a `main()` function. Tables must be allocated on the shared memory of the GPU. However, data cannot be sent from CPU directly to the shared memory. Tables need not always be allocated on constant memory. However, the current version of the translator allocates them on the constant memory first. Furthermore, allocating a variable for a key on the shared memory is valid. The key can be allocated on the shared memory or constant memory because each allocation does not make a difference for their performance[11]. Figure 3 portrays a generated program by the translator. Variables created by the translator are placed at lines 11–18 in Figure 3.

**C. Generating code for an allocation and a transfer**

When the translator finds a `kernelCall` directive in `main()` function, first of all, `CUDA_INIT()` is created as a string constant in front of `kernelCall`. Allocation and transfer codes follow `CUDA_INIT()`. Code for device memory allocation is provided by the API `cudaMalloc()`. To allocate a ciphertext and a plaintext, two `cudaMalloc()` codes are generated. Their parameters, variable name, its type, and size are generated by information stored in the directive table. Then data transfer codes for all created variables are generated. A key and tables are sent to the constant memory using API `cudaMemcpyToSymbol()`. A transfer code for a plaintext is generated with `cudaMemcpy()`. Information of parameters of these functions is stored in the directive table too. To set directions of the transfer between CPU and GPU, `cudaMemcpyHostToDevice` is attached on this API as one parameter. Then, to receive ciphertext processed on GPU, `cudaMemcpy()` is generated after the encode function call with the parameter `cudaMemcpyDeviceToHost`. They are shown in Figure 3 at lines 21–38.

**D. Generating kernel function call**

The directive `kernelCall` annotates a function call for encoding, which is represented as `AESEncode` in base code for example.

To make a kernel call of CUDA, new variables `grid` and `thread` are declared to assign the number of blocks and threads. They are declared the type of `dim3` variables according to the basic CUDA. The number of threads and blocks are given by directives `threadNum` and `blockNum`. If they are not specified, then the translator will attempt to calculate the best numbers according to an architecture of GPU and a characteristic of CUDA. Default value of the `blockNum` is 28 or its multiples for Tesla C2050 currently. Default value of the `threadNum` is 256 or its multiples. To minimize a loss of performance attributable to padding in case of odd size of plaintext, it is better to set the number of thread and thread blocks as not too numerous. In contrast, to obtain higher performance, a certain number of thread blocks and threads are necessary. The kernel call is shown in Figure 3 at lines 34–35.

**E. Definition of a kernel function**

Translation of the kernel function definition is another important task for the translator. This work consists of two parts: shared memory management and loop distribution.

1) **Shared memory management:** When the translator meets up with the directive `kernelFunc`, the translator adds keyword “__global__” to before the kernel function definition. Then, new shared variables with a prefix “s_” added to variables annotated with `table` are created. Declarations of new variables are placed at the beginning of a body of the kernel function definition with keyword “__shared__”. A variable annotated with `table` is called “table” hereinafter and a new variable allocated on the shared memory is hereinafter called an “s_table”. A table variable allocated on the constant memory is called a “c_table”.

To fill up each `s_table` with the content of each table that is declared on the base code, initialization codes for each `s_table` are created. Three kinds of code are used to initialize `s_tables`. An initialization code is generated according to Algorithm.
1. The algorithm shows three cases, the case of the number of threads is equal to, greater than, or less than the table size. After generating an initialization code for all tables, API syncthreads() will be inserted to synchronize the threads. Finally, all table references must be replaced to the s_table. Figure 4 at lines 4–11 shows this translation.

2) Loop distribution: The translator finds a loop that encodes the plaintext blocks by searching a keyword for and analyzes its parameters. The translator replaces an original for structure to a new loop divided by the fixed number of threads as shown in Section III-B3. Therefore, each thread has a loop for encryption cipher blocks, so that the number of iterations is represented as $(\text{number of cipher blocks})/(\text{total number of threads})$. When the number of cipher blocks is broken, the translator uses a padding technique. Moreover, to exploit further performance, each thread must access the global memory in sequence. It provides a coalesced access to the global memory. A translated loop header is shown in Figure 4 at lines 18–19, 24–25.

VI. IMPLEMENTATION AND EVALUATION

A. Prototype of the translator

A prototype of HiCrypt translator was implemented to confirm its effectiveness through performance evaluation of generated CUDA codes. According to the design of the translator shown before, it requires no syntax analysis. For this reason, this prototype has been implemented using the Ruby programming language. It has achieved almost all functions of HiCrypt by pattern matching and replacing. A description for PC, OS, CUDA, and GPU used for this evaluation is shown in Table IV.

B. Cipher algorithms

For this evaluation, we have selected two more cipher algorithms, which are Camellia and SC2000, aside from the AES. The reasons we have selected these cipher algorithms are the data randomization structure of AES, Camellia, and SC2000 differ, SPN, Feistel, and the hybrid of SPN and Feistel. In addition, these three ciphers are listed as e-government recommended ciphers by the Cryptography Research and Evaluation Committees (CRYPTREC). AES and Camellia are also recommended by the New European Schemes for Signatures, Integrity and Encryption (NESSIE)[13]. Moreover, SC2000

<table>
<thead>
<tr>
<th>Table IV</th>
<th>SPECIFICATION OF ENVIRONMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Core i7-2600K 3.40 GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>32 GB</td>
</tr>
<tr>
<td>Motherboard</td>
<td>ASUS P8Z68-V</td>
</tr>
<tr>
<td>OS</td>
<td>CentOS 6.0 (Kernel ver. 2.6.32-71)</td>
</tr>
<tr>
<td>Compiler</td>
<td>gcc ver 4.4.4 (option -O3)</td>
</tr>
<tr>
<td>CUDA platform</td>
<td>Nvidia CUDA toolkit 4.0</td>
</tr>
</tbody>
</table>

HiCrypt by pattern matching and replacing. A description for PC, OS, CUDA, and GPU used for this evaluation is shown in Table IV.

<table>
<thead>
<tr>
<th>Table V</th>
<th>SUMMARY OF THE CIPHER SPECIFICATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Structure</td>
</tr>
<tr>
<td>AES</td>
<td>SPN</td>
</tr>
<tr>
<td>Camellia</td>
<td>Feistel</td>
</tr>
<tr>
<td>SC2000</td>
<td>Hybrid</td>
</tr>
</tbody>
</table>

*in case of 128-bit key length.

**Optimized as speed. Num. of tables and size are selectable in SC2000.
have achieved the best throughput in the symmetric block ciphers currently.[14].

1) Camellia: For Camellia, an optimization method for software implementation is presented in its specification document[15]. Its 128-bit key algorithm includes 18 round processes. Each round includes an F-function. In the F-function, a 64-bit input is transformed simply using eight S-table substitutions with 8-bit input and 32-bit output and XOR operations. The FL and FL<sup>-1</sup> functions, consisting of 32-bit AND, OR, XOR, and cyclic left shift, are inserted per six rounds.

2) SC2000: As for SC2000, an optimization for a software implementation is shown in its specification document[16]. Each round executes five transformations in sequence: I, B, I, R, and R-function. In the R-function, the 32-bit input is separated into several fragments of arbitrary size, for example (6-bit, 5-bit, 5-bit, 5-bit, 6-bit) or (11-bit, 10-bit, 11-bit). Then they are replaced by S<sub>6</sub> and S<sub>7</sub>-table or S<sub>11</sub>, and S<sub>10</sub>-table, etc.

Table V presents a summary of the specification of three ciphers.

3) Creating a source program with directives: Source codes of AES and Camellia are made from OpenSSL/TLS source code. Particularly, primary encoding codes are almost a direct quote from it. By contrast, SC2000 is implemented based on the specification[16] by one author of this paper. All programs are implemented with a restricted format shown as before. The directives, plaintext, ciphertext, key, table, kernelCall and kernelFunc are used. Therefore, the format of these programs is the same as a base code of AES shown in Figure 2. The directive device is annotated to call the function in the kernel function in case of SC2000. However all programs have different data randomize the source code, and directives are easily inserted to all three ciphers. Because annotated variables are distinguished for each cipher program, making a program with HiCrypt directives is easy for programmers.

C. Automatic translation results

Results show that the translator prototype generated CUDA programs for all three ciphers. Performances of generated programs and hand-optimized programs are presented in Table VI. To evaluate their performance in cases near peak performance, each parameter is determined: the number of threads in a thread block is 512, the number of thread blocks is 56 and the size of plaintext is 10 MByte. Each throughput is similar to each recent highest throughput shown in [14]. Furthermore, all three programs have been compiled using a standard C compiler that ignored directives of HiCrypt. Throughputs which have been processed using a CPU with a single thread are also shown. As shown there, the programs generated by HiCrypt achieved almost identical performance as those of hand-optimized programs. Therefore, the generated programs seem almost identical to the hand-optimized program, except for the order of the API call and the places of declarations of variables.

HiCrypt has two more features, allowing nested loops and device function that calls an inner kernel function. Symmetric cipher algorithms consist of rounds which are repeated in the encoding process. To repeat rounds, a loop structure such as a “for” or “while” is used in an encoding function. In other words, loops for encoding to cipher blocks have a nested loop structure. Moreover, some data-randomize operations in rounds are defined as function calls to make a readable program. HiCrypt provides the directive device to support such a function. Results show that HiCrypt can translate such nested loops and device function calls.

In contrast, general purpose automatic parallelization compilers for accelerator including GPGPU have been appearing recently, such as OpenACC[12], PGI Accelerator[17], and CAPS HMPP[18]. Moreover, academic suggestions to assist to make up OpenCL and CUDA programs have also been presented[19][20][21]. HiCrypt differs from them in their purpose, which is specialized for symmetric block ciphers. To make a small target, HiCrypt generated highly optimized CUDA programs from symmetric block cipher programs.

The technique described in this paper is also valid for AMD GPUs or OpenCL programs. To generate OpenCL to support other heterogeneous multi-core architectures such as AMD GPUs remains as a subject for future work. Moreover, we would like to develop other features of HiCrypt such as automatic translation for public key cryptography. Furthermore, syntax analysis will be implemented to HiCrypt to relax its restriction.

VII. CONCLUSION

This paper described a new translator, HiCrypt, for symmetric block ciphers. HiCrypt can generate optimized CUDA programs from symmetric block cipher programs written in a standard C language with HiCrypt directives. Usage of directives is simple for cipher programmers because directives require annotation of some elements that are characteristics of the cipher programs. To evaluate HiCrypt, we implemented a translator prototype and evaluated its performance through implementation of three symmetric block cipher programs, AES, Camellia, and SC2000, with directives. Results show that the translator prototype generated optimized CUDA programs that exhibit almost identical performance as hand-optimized

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Translator</td>
<td>49.5</td>
<td>31.1</td>
<td>76.2</td>
<td>512</td>
<td>56</td>
</tr>
<tr>
<td>Hand optimized</td>
<td>49.6</td>
<td>51.1</td>
<td>76.2</td>
<td>512</td>
<td>56</td>
</tr>
<tr>
<td>Base code (CPU, single thread)</td>
<td>1.2</td>
<td>0.9</td>
<td>1.0</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
programs for all three cipher programs. Moreover, the translator accepts a nested loop as a target loop and a calling function in the target loop.

Although the heterogeneous multiprocessors that embed a GPU have become widely used, they will be able to accelerate ciphers. We expect that HiCrypt will contribute to the development of new symmetric block ciphers.

REFERENCES