Parallel Computing Scheme for the Encryption Process of DNSCrypt Protocol using CUDA

Fairuz Astra Pratama[1], Dr. Ir. Rinaldi Munir, MT. [2], Drs. Judhi Santoso, M.Sc. [3]
School of Electrical Engineering and Informatics,
Bandung Institute of Technology
Jl. Ganesha No.10, Lb. Siliwangi, Coblong, Bandung, Jawa Barat 40132, Indonesia
Email: pratamafairuz@gmail.com[1], rinaldi@informatika.org[2], judhi@informatika.org[3]

***Abstract* – DNSCrypt is a protocol that can be used to secure the communication in the DNS system. The usage of this protocol will add an additional processing in both client and server to encrypt and decrypt the message, negatively impacting system performance. CUDA can be used to reduce this negative effect by processing the encryption in the GPU, where each message block can be processed simultaneously in separate threads. This paper aims to explore the potential of using CUDA in DNSCrypt to improve performance by implementing several DNSCrypt system with different encryption algorithm using or not using CUDA, measuring the performance of each system, and analyzing it.**

***Keywords* — DNSCrypt, parallel processing, encryption, CUDA, performance**

I. Introduction

DNS is one of the component of the internet that provides a naming service. One of the most common usage of the DNS system is mapping a website URL to the IP address of the server that serve it. Every message that is sent in the DNS system are as a plaintext without any encryption. This makes the system vulnerable to eavesdropping and man-in-the-middle attack, where a third party can listen to every DNS message sent from a user to figure out the user's internet usage pattern.

One of the method that can be used to solve this problem is by using the DNSCrypt protocol. DNSCrypt secure the DNS communication protocol in a way similar to how TLS secure the HTTP protocol; by encrypting and validating every communication that occurs between a DNS client and a DNS resolver. This way, an attacker will not be able to find out what website a user is trying to access since every message sent between the user and the resolver will be encrypted.

Even though the size of each message is relatively small, the amount of DNS lookup in a network can be very high. A large DNS resolver may process up to 800000 DNS query every second. The usage of DNSCrypt protocol may result in a negative performance impact since every message that is sent and received need an additional processing in both end.

To help reduce the performance impact of using the DNSCrypt protocol, parallel processing can be used to accelerate the encryption and decryption process. This paper will propose a method of using CUDA to process the encryption process of the DNSCrypt protocol. Furthermore, this paper will also analyze the performance of a DNSCrypt system with various configuration (processing method and encryption algorithm) and compare it to a regular DNS system.

II. Literatures Study

A. DNSCrypt Protocol

DNSCrypt is a security extension to the commonly used DNS protocol. It works by encrypting and validating every DNS query and response sent between a client's DNS stub and the DNS resolver. Similar with TLS, this security protocol is designed to support many encryption scheme, and can use both TCP and UDP as a transport method. The way DNSCrypt works can be seen in Figure 1.

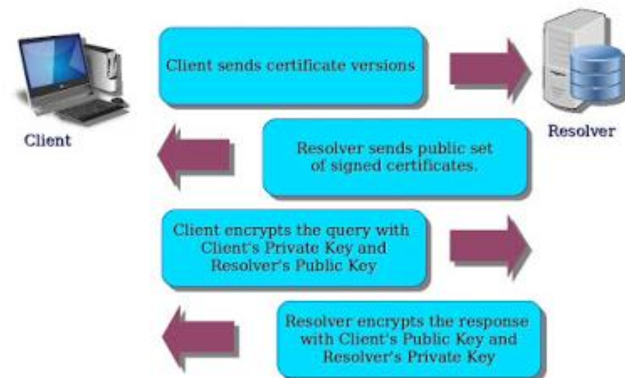

Figure 1 How DNSCrypt Protocol Works (Mitra, 2016)

The "certificate" that is referred to in Figure 1 is a digital document that contains information that is relevant for the client to be able to encrypt and validate DNS query and DNS response to and from the DNS resolver. At minimum the certificate will contain the protocol version and the resolver's public key that will be used to encrypt and decrypt messages for the current session. Since a resolver will be able to publish several certificates at once, every certificate will also have a "magic number" that the user will append to every message sent to the resolver. This way, the resolver know which certificate that each client is using to communicate.

Version one of the DNSCrypt protocol is built using the X25519-XSalsa20Poly1305 encryption scheme. X25519 is an implementation of Elliptic-curve Diffie–Hellman, XSalsa20 is a stream cipher algorithm, and Poly1305 is MAC generation / validation algorithm. In this version, both client and server will have an X25519 key pair, where each participant will generate an identical secret key using their own secret key and the partner public key. After that, both parties can easily send and receive encrypted messages to and from each other using the XSalsa20 algorithm (every message sent will also be

accompanied by the nonce used to encrypt it). Additionally, Poly1305 will also be used to generate MAC that will be appended to the encrypted message, where upon receiving the message, the recipient will validate the MAC to make sure that the message originated from the correct host and has not been tampered by a third party.

B. CUDA Parallel Programming

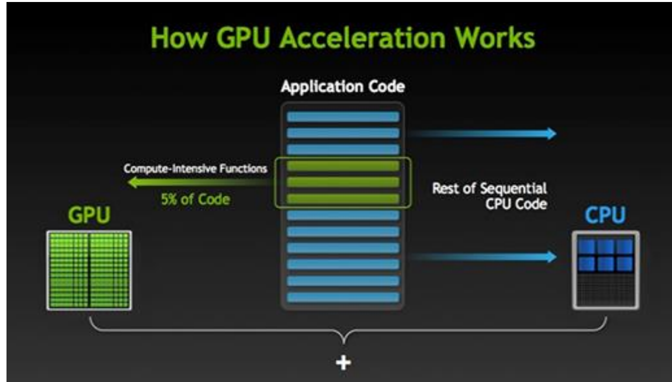

Figure 2 GPU Accelerated Computing (NVIDIA, 2007)

CUDA is a programming platform developed by NVIDIA that allows general-purpose program to be processed on NVIDIA's GPU. This concept is often called as "GPU-Accelerated Computing", where the majority of the code is still being processed in the CPU, but a small portion is processed in parallel using the GPU as can be seen in Figure 2. The main advantage of using the GPU instead of CPU, is that the GPU has a lot more processing core, making it more suitable for compute-intensive function or codes. The CUDA programming model is based on the C language and consist of several important concepts such as:

1. **Kernel Function**. A kernel function is a C function where the code within it will be executed N times in N separate CUDA thread. This function is declared using the __global__ declaration, and every parameter supplied must be a pointer to the GPU's global memory.
2. **Thread Hierarchies.** CUDA Threads is organized into a block of threads that is organized into a grid. Every thread will be able to access its thread id in its block, as well as its block id in its grid. When calling a kernel function, the code must also specify the number, structure and the dimension of both the thread blocks and the grid.
3. **Memory Hierarchies.** There are several different memory space in CUDA: Global and constant memory that can be accessed by the CPU and every CUDA threads, Shared memory that can accessed by every thread in the same block, as well as Local memory that can only be accessed by the owners thread.
4. **Heterogenic Programming.** In a CUDA program, the CPU will use the GPU using the following method:
   a. CPU will allocate the necessary GPUs global memory space for the parameter and output
   b. CPU will copy the parameter form the RAM to the GPUs global memory
   c. CPU call the kernel function and wait for it to finish
   d. Finally, the CPU will copy the processing result from the GPUs global memory to the RAM

III. PROPOSED SOLUTION

GPU processing had been proved to be effective at reducing the processing time of the encryption and decryption process of both AES (Li, Zhong, Zhao, Mei, & Chu, 2012) and Salsa20 (Khalid, Paul, & Chattopadhyay, 2013). This means that by altering the encryption and decryption processing scheme used at the DNSCrypt protocol implementation, the performance of the system may be improved. This section will propose a method of using CUDA to process the encryption and decryption in DNSCrypt protocol, as well as proposing the usage of a different encryption algorithm altogether.

A. Parallelizing the XSalsa20 Algorithm

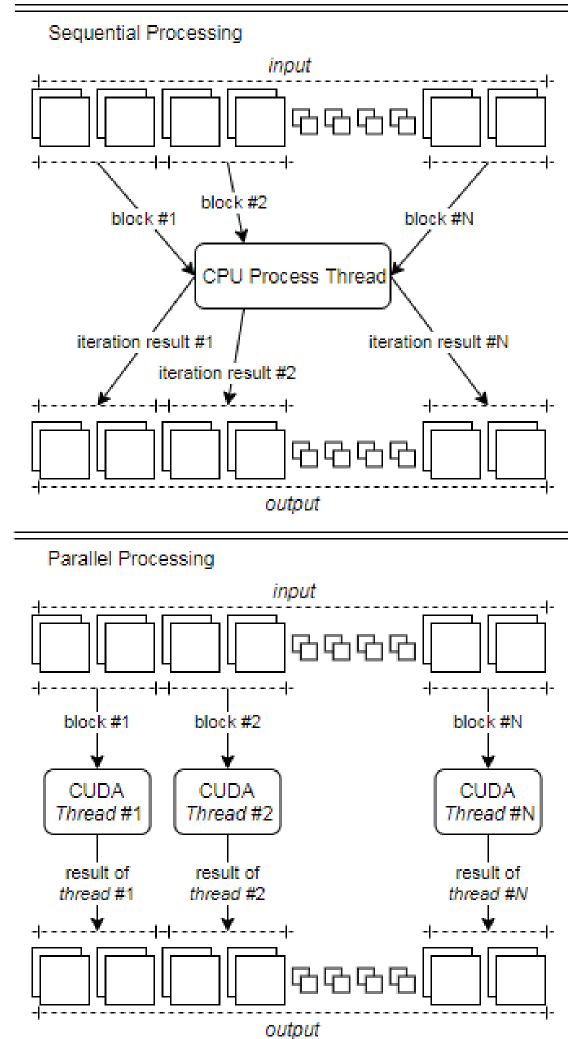

Figure 3 Mapping of a Sequential Encryption Algorithm to its Parallel Version

XSalsa20 that is used in the DNSCrypt protocol version one is an extension of the Salsa20 algorithm, and it function by combining the 256 bit XSalsa20 encryption key with the first 128 bit out of the 192 bit nonce into a 256 bit Salsa20

encryption key using a hash function. This key, as well as the 64 bit nonce that is not used is then used by the Salsa20 algorithm to perform the actual encryption/decryption process.

Salsa20 is a symmetric stream cipher that will first generate a keystream from the secret key and nonce. After that, both the encryption and decryption is done by XOR-ing the plaintext / ciphertext with the keystream. The keystream itself is generated in a 64 byte chunk using some sort of hash function using the encryption key, nonce, as well as the chunk id, this means that every chunk can be generated independently. This also means that the data being processed (both encryption and decryption) can also be separated into 64 byte chunk/block, where each ciphertext / plaintext block can be mapped to its plaintext / ciphertext block independently.

In the sequential algorithm the mapping of each 64 byte input block is done sequentially. Using CUDA, the mapping of each block can be done in parallel, as can be seen in Figure 3. This way, the encryption will be processed in several CUDA processing cores, instead of one CPU core. Using this base design, there are still several issues that needs addressing, namely the memory allocation and the kernel function itself.

1. Memory Allocation and Parameter Copying

To make sure that every CUDA thread will be able to access the function parameter, CPU needs to allocate and copy the parameters before every kernel function invocation. The allocated memory will also need to be freed after the kernel function is done, and this whole process can take even longer than the kernel function itself. To remedy this, the program can allocate the input and output memory space in the GPUs global memory when the program first start. This way, every function invocation will only need to copy the parameters and result from the allocated memory space, drastically reducing the processing time.

2. Kernel Function Design

The kernel function design can be seen in Figure 4. In general, once the kernel function invoked, the first few thread in each thread block will copy the key and nonce parameter from the global memory to the shared memory. After that, each thread will calculate its absolute id in the whole grid by calculating “thread id in grid + (number of thread in a block * block id in grid)” After knowing which block id it needs to process, the thread will then check if its block id is bigger than the number of block that needs to be processed, and stop if it is. Otherwise, the thread will read the input block from the global memory, calculate the keystream block using the key and nonce from the shared memory, XOR both of those value, and store the result in the global memory.

The reason that each block thread will copy the nonce and key from the global memory to the shared memory is that reading value from the shared memory is a lot faster than reading from the global memory and every thread in each block will need to access both those value regardless of which input block it will process. By making a couple of threads collaboratively copy part of the nonce and key from the global memory, the number of global memory access will be reduced significantly, improving overall performance.

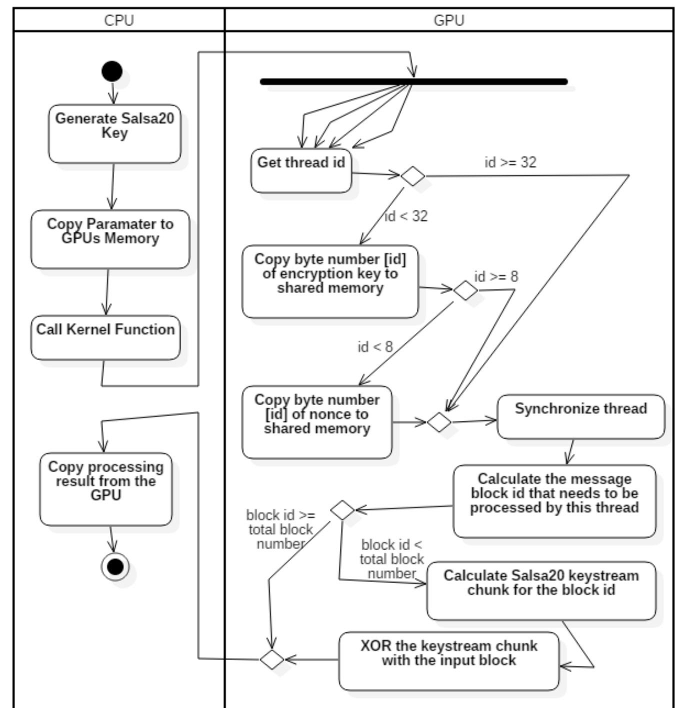

Figure 4 Activity Diagram of XSalsa20 Parallel Computing Scheme using CUDA

B. Using AES as an Alternative to XSalsa20

AES is a standard symmetric block cipher that had been used in several security protocol in the internet, including the TLS. Since the Poly1305 validation algorithm does not depend on the encryption algorithm used, and since AES can still use the 256 bit key that is generated by the X25519 algorithm, AES can easily be used as an alternative to XSalsa20 in the DNSCrypt protocol. Furthermore, since DNSCrypt can support multiple encryption scheme at the same time using the certificate method, a new encryption scheme of X25519-AESPoly1305 can be easily added to the system without changing much of the underlying system.

AES is a block cipher that will encrypt a 16 byte block of data using a 256 bit key. To be able to process data larger than 16 byte, an appropriate operation mode must be chosen. In this paper, the CTR mode will be used, since it function in a similar way to XSalsa20 and also can be easily processed in parallel. AES-CTR will encrypt and decrypt data by producing a keystream from a 256 bit secret key and a 64 bit nonce, and XOR-ing said keystream with the data to encrypt and/or decrypt it. Similar to XSalsa20, the keystream is generated in a 16 byte chunk, and each part can be calculated independently.

This means that AES-CTR parallel computing scheme is similar to XSalsa20s that can be seen on Figure 3. The memory allocation, parameter copying, and the kernel function itself will be similar in general. The only differences between the two—besides the different keystream generator method—is that AES has a block size of 16 bytes, and a lookup-table that needs to be accessible to all threads during processing.

To make sure that each thread can read the AES look-table properly, a method similar to the memory allocation method for

the parameter can be used. When the program start and allocate the global memory for the AES parameter, it will also allocate and copy the AES lookup-table to the GPUs constant memory. Furthermore, when the kernel function is invoked, each thread in a block will collaborate to copy this lookup-table to the block shared memory alongside the encryption key and the nonce. This is done since the AES algorithm will frequently access this table in a random manner, and the shared memory is more suitable for this than the constant memory (Iwai, Nishikawa, & Kurokawa, 2012).

IV. IMPLEMENTATION

Implementation of the designed solution in the previous section are done using an existing open source implementation of the DNSCrypt protocol as a base project. For the client, the dnscrypt-proxy project (https://github.com/opendns/DNSCrypt-proxy) will be used while dnscrypt-wrapper (https://github.com/cofyc/DNSCrypt-wrapper) will be used for the server implementation. Both of the initial implementation used are a proxy application, where the client proxy application is connected to regular DNS client and the server proxy application that is also connected to a regular DNS resolver.

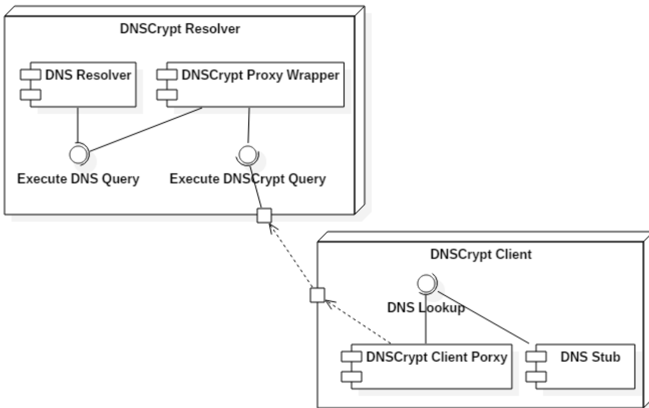

Figure 5 Full DNSCrypt System Architecture

The architecture for the full system using both of these proxy application can be seen in Figure 5 below. The libsodium (https://github.com/jedisct1/libsodium) and libssl (https://github.com/openssl/openssl) cryptography library will also be used as reference for adding the XSalsa20-CUDA and AES encryption scheme to the initial implementation. Furthermore, the GPU used in this implementation and the following experiment is the GeForce GTX 950M, which is an NVIDIA GPU that has a total of 640 processing core.

V. EXPERIMENTS RESULTS AND ANALYSIS

The experiment discussed in this section is done to measure, compare, and analyze the performance of the DNSCrypt system seen in Figure 5 that uses various encryption scheme. To achieve this, testing will be done in two separate scenario. The first step is to compare the performance of all the encryption function made, which is both the encrypt and decrypt function of the X25519-XSalsa20Poly1305 and X25519-AESPoly1305; each of which has a version where the encryption / decryption is processed in CPU and GPU (a total of eight different function). The second step is to compare the performance of the several DNSCrypt system implementation variants that uses a different encryption (XSalsa20 or AES) and processing (CPU or GPU) scheme.

A. *Encryption Function Performance*

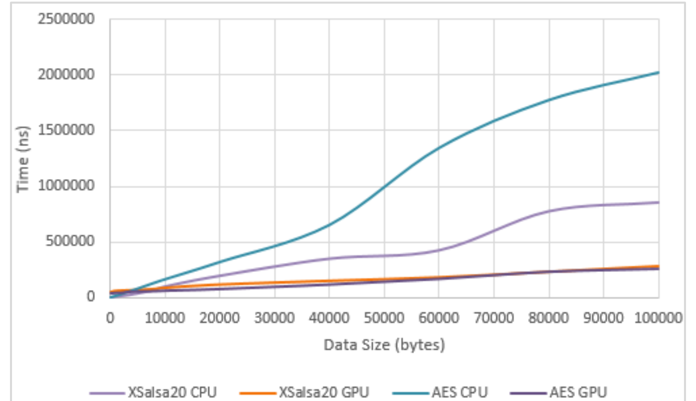

Figure 6 Processing Time of Different Encryption Function

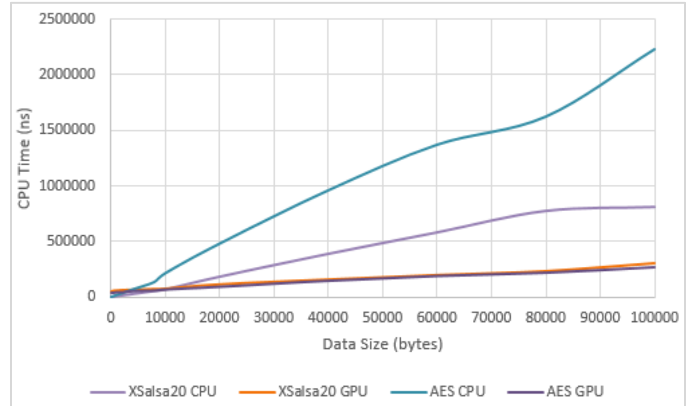

Figure 7 CPU Time of Different Encryption Function

The performance measured in this experiment is the average processing and CPU time that is needed by the encryption function to process data with varying size. The processing and CPU time measuring in this experiment is done using the clock_gettime function that is available in the C standard library. After carrying out this experiment, the processing time comparison of the different encryption function can be seen in Figure 6, while the CPU time difference can be seen in Figure 7 (both shows the average of the encrypt and decrypt version for each scheme). There are several pattern that stands out and needs further analyzing:

1. CPU Usage Analysis

From the experiment result the CPU processing time for all possible encryption function is more or less similar to the total processing time. While this is expected for the CPU version, the usage of GPU is expected to "share" the processing load with the CPU, making the CPU usage time lower than the total processing time. After further research it is known that this happened due to the method of how the CPU wait for the kernel function to complete. There are two main ways to do this:

"Polling" where the CPU is constantly checking the GPU status similar to the busy-waiting method, and "Blocking" where the CPU process thread is turned off until the kernel function is complete. Since the default mode used is "Polling", this resulted in the CPU remaining busy while the GPU process the kernel function, making the CPU usage time similar to the total processing time. To find out the effect of using the "Blocking" method, the experiment will be re-run after changing the GPU configuration. The result of this experiment can be seen in the Figure 8 below.

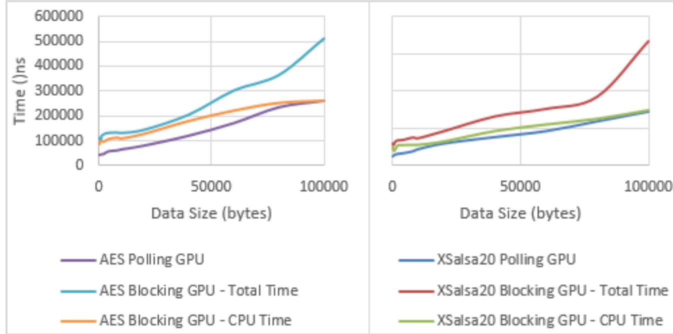

Figure 8 Comparison of "Polling" and "Blocking" Method

From the comparison result, it is seen that while the CPU usage time of the "Blocking" method is less than its total processing time, it is still more than the total processing time (and the CPU usage time) of the "Polling" method. This means that the "Blocking" method has a higher overhead cost than "Polling", and are unsuitable if the kernel function only take a short amount of processing time. Since the data that needs to be processed in a DNSCrypt system is usually relatively small, it can be concluded that the "Polling" method is more suitable when using CUDA in the DNSCrypt protocol.

2. Processing Time and GPU Core Usage Analysis

From the experiment result the CUDA version of each algorithm can be seen as having a better performance than the CPU version when processing a bigger data. This can be better seen after computing the speed-up of the algorithm as can be seen in Table **1**. For small data, the GPU version is slower than the CPU version. This happens because the CUDA usage overhead (such as parameter copying) is larger than the time it took to process the encryption / decryption in the CPU. On the other hand, as the data size grow, the processing time for all function version also rise. But, the rate of which the GPU version increase is generally lower than the CPU version. This happens due to the increase in GPU core usage.

In the parallel computing scheme used in this experiment, each message block will be processed in a CUDA thread which is processed in a CUDA processing core. This means that for a small data, the amount of thread made and the number of CUDA core used is relatively low. On the other hand, the bigger the data being processed, the more core is able to be used to process it, increasing the efficiency. This, combined with the GPU usage overhead, explains why the algorithm speedup is lower than one for small data, and gradually gets better the bigger the data is. This also explain why AES-GPU has a better performance than XSalsa20 even if the algorithm itself is inherently slower. The smaller block size makes AES able to be processed in a bigger number of core than XSalsa20, increasing its performance.

Table 1 Processing Parallelization Speed-Up

Data Size (Bytes)	XSalsa20 GPU vs XSalsa20 CPU	AES GPU vs AES CPU	AES GPU vs XSalsa20 CPU
100	0,029	0,064	0,036
200	0,040	0,098	0,050
400	0,061	0,169	0,080
800	0,103	0,299	0,142
1000	0,127	0,372	0,175
2000	0,222	0,704	0,316
4000	0,408	1,134	0,496
8000	0,844	**2,209**	1,111
10000	1,106	**2,570**	**1,541**
20000	**1,640**	**4,041**	**2,470**
40000	**2,283**	**5,474**	**2,935**
60000	**2,304**	**7,919**	**2,504**
80000	**3,291**	**7,601**	**3,330**
100000	**2,980**	**7,793**	**3,303**

From this experiment, it can be concluded that the parallel version of the encryption algorithm only suitable for large enough data. Since the DNS query is consistently small (only contains the requested record name and type), it is unsuitable to process its encryption in parallel. On the other hand, a DNS response can reach up to 64KB size, and can be processed in parallel effectively.

B. *DNSCrypt System Performance*

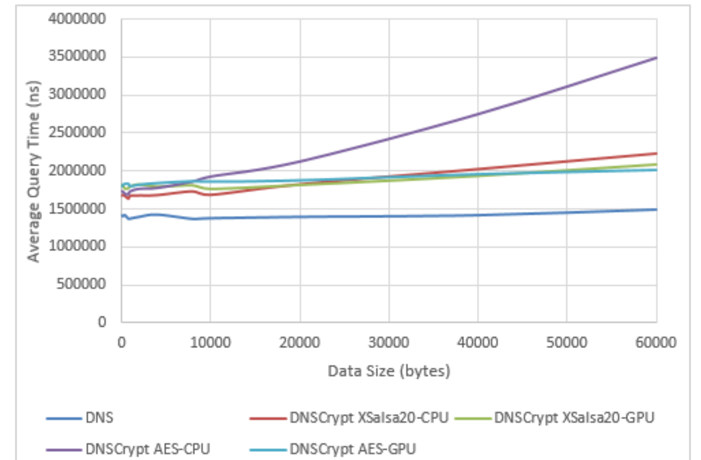

Figure 9 DNS Query Processing Time in Varying System

The performance measured in this experiment is the average processing time of a DNS query with varying response size. To measure this, after the DNSCrypt system had been set up, a bash script will be executed to fetch a specific DNS record with a known response size using the 'dig' command. The script will send several DNS request at the same time, and measure the time required by the system to answer all the queries using the 'date' command. To get a baseline this method will also be used to measure a regular DNS system performance. After carrying out this experiment, the query processing time of the each systems can be seen in Figure 9.

In a regular DNS system, the query processing time tends to remain constant even with a bigger response size. The usage of DNSCrypt will require each response to be encrypted and decrypted, and this process will take longer the bigger the data being processed. By comparing each DNSCrypt system to the regular DNS system, performance impact due to the usage of each DNSCrypt encryption and processing scheme can be calculated, and can be seen in Table 2 below.

Table 2 DNSCrypt vs DNS Performance Impact

Response Size (Bytes)	XSalsa20 CPU	XSalsa20 GPU	AES CPU	AES GPU
100	15,4 %	22,1 %	17,9 %	21,4 %
200	16,4 %	22,3 %	18,1 %	22,7 %
400	15,0 %	19,8 %	16,1 %	22,4 %
800	15,8 %	22,6 %	18,5 %	25,0 %
1000	17,6 %	23,5 %	20,1 %	23,6 %
2000	16,5 %	23,4 %	20,4 %	23,2 %
4000	15,0 %	20,5 %	19,3 %	22,5 %
8000	20,6 %	24,2 %	25,9 %	26,3 %
10000	17,9 %	21,8 %	**28,0 %**	**25,7 %**
20000	**23,4 %**	**23,1 %**	**34,0 %**	**25,5 %**
40000	**30,0 %**	**26,7 %**	**48,3 %**	**27,4 %**
60000	**33,3 %**	**28,5 %**	**57,3 %**	**25,9 %**

The result received from this experiment is consistent with the previous one. If the average response size in a system is relatively low, the CPU version will perform better than the GPU version. As the average response size grow, the GPU version will start outperforming the CPU ones. Besides that, the table also show that DNSCrypt AES-GPU scheme has a worse performance than the XSalsa20-GPU, even if the previous experiment prove otherwise. This happen because even though AES-GPU is more efficient than XSalsa20-GPU, AES-CPU is a lot slower than XSalsa20-CPU. Since DNS query encrypting and decrypting is processed in the CPU for all of configuration, the overall performance of DNSCrypt AES-GPU will be lower than XSalsa20-GPU in most cases.

VI. CONCLUSIONS

CUDA can be used in the DNSCrypt protocol to process the encryption and decryption process in parallel. It does this by processing each message block in a separate CUDA thread. The usage of “Polling” method in the GPU will also yield a better performance than the “Blocking” alternative since the size of message processed is relatively small and quick to process. The effect of the usage of DNSCrypt in a DNS system, as well as the effect of the usage of CUDA in said DNSCrypt protocol can be seen in Table 2. In general, the usage of CUDA can increase the DNSCrypt performance, but only if the message size that is processed is big enough to be able to take advantage of the big number of the CUDA processing core. Additionally, AES can also be used as an alternative to XSalsa20 for the encryption algorithm used in DNSCrypt and still use the X25519 as the key exchange algorithm and Poly1305 for the message validation.

For further research, a batch-processing method can be developed for the DNSCrypt protocol. Since most queries and responses in a DNS system is relatively small, batch-processing will allow the use of CUDA more effectively, since the size of the data to be processed at the same time will increase. Additionally, the cryptanalysis or security analysis of the usage of AES as the alternative of XSalsa20, as well as the usage of GPU in processing message encryption / decryption is also an interesting research topic.

REFERENCES

[1] K. Iwai, N. Nishikawa and T. Kurokawa, "Acceleration of AES encryption on CUDA GPU," *International Journal of Networking and Computing,* 2012.

[2] Q. Li, C. Zhong, K. Zhao, X. Mei dan X. Chu, “Implementation and Analysis of AES Encryption on GPU,” 2012.

[3] D. J. Berstein, "Salsa20 Spesification," The University of Illinois, Chicago, 2005.

[4] P. Mockapetris, “DOMAIN NAMES - CONCEPTS AND FACILITIES,” November 1987. [Online]. Available: https://tools.ietf.org/html/rfc1034. [Accessed 17 April 2018].

[5] F. Denis dan Y. Fu, “DNSCRYPT-V2-PROTOCOL,” 17 Februari 2018. [Online]. Available: https://github.com/DNSCrypt/dnscrypt-protocol/blob/master/DNSCRYPT-V2-PROTOCOL.txt. [Accessed 17 April 2018].

[6] A. Mitra, "What is DNSCrypt?," 31 Maret 2016. [Online]. Available: https://computersecuritypgp.blogspot.co.id/2016/03/what-is-dnscrypt.html. [Accessed 18 April 2018].

[7] “X25519 key exchange,” [Online]. Available: https://cryptography.io/en/latest/hazmat/primitives/asymmetric/x25519/. [Accessed 17 April 2018].

[8] NVIDIA, “What’s the Difference Between a CPU and a GPU?,” 16 Desember 2009. [Online]. Available: https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/. [Accessed 17 April 2018].

[9] NVIDIA, “WHAT IS GPU-ACCELERATED COMPUTING?,” 2007. [Online]. Available: http://www.nvidia.com/object/what-is-gpu-computing.html. [Accessed 18 Agustus 2017].

[10] NVIDIA, “NVDIA CUDA Developer Zone,” 5 Maret 2018. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [Accessed 17 April 2018].

[11] S. Bortzmeyer, “DNS Privacy Considerations,” August 2015. [Online]. Available: https://tools.ietf.org/html/rfc7626. [Accessed 17 April 2018].

[12] A. Khalid, G. Paul dan A. Chattopadhyay, “New Speed Records for Salsa20 Stream Cipher Using an Autotuning Framework on GPUs,” RWTH Aachen University, Aachen, 2013.