Accelerating AES in JavaScript with WebGL

Yang Yang[1,2,3], Zhi Guan[1,2,3,⋆], Jiawei Zhu[1,2,3],
Qiuxiang Dong[1,2,3], and Zhong Chen[1,2,3]

[1] Institute of Software, School of EECS, Peking University, China
[2] MoE Key Lab of High Confidence Software Technologies (PKU)
[3] MoE Key Lab of Network and Software Security Assurance (PKU)
{yangyang,guanzhi,chen}@infosec.pku.edu.cn

Abstract. Cryptography is a fundamental building block for security sensitive Web applications. Because the architecture of JavaScript can not provide sufficient performance, the client-side web applications still lacks high performance cryptography primitives. In this paper we studied the feasibility of a new Web standard, i.e., the WebGL API for accelerating AES in JavaScript by exploiting the ability of GPU. We design and implemented AES using 128-bit key length. We compared the performance of our approach to the currently reported fastest pure JavaScript implementation and found our approach runs more than ten times faster in major browsers on all platform. Our work showed the potential optimization of using GPU via WebGL to accelerate JavaScript code.

Keywords: AES, WebGL, GPGPU, JavaScript.

1 Introduction

Recent years, the fast development of cloud computing makes it much easier for users to synchronize their personal data with the cloud to access the data anywhere for convenience. Since the service provider are untrusted, the unencrypted users' privacy may leak to curious employees or even the government, according to the recent report from Guardians[1]. It is necessary for many applications to encrypt the data before uploading to the cloud to preserve the privacy of users, especially sensitive photos, documents, musics, etc. As web browser is becoming a universal tool for interacting with remote servers, almost all popular applications provides a web interface, it is important to provide efficient cryptographic primitives for web applications to enhance their security, especially symmetric cryptography such as AES.

Although the performance of JavaScript has been experienced a continuous increasing recent years, there is still a remarkable gap between the performance of JavaScript code and native code because of the nature of a untyped scripting language dynamically interpreted running in a virtual machine. Unless a prominent improvement on the architecture of JavaScript occurs in the future, the gap

⋆ Corresponding author.
[1] http://www.guardian.co.uk/world/2013/jun/06/us-tech-giants-nsa-data

S. Qing et al. (Eds.): ICICS 2013, LNCS 8233, pp. 275–287, 2013.

may still exist for a long time. Another restriction for JavaScript is it doesn't support parallel computing. This means even the performance gap has been narrowed, pure JavaScript code can still not make full use of the processing power of the CPU. Since the performance of the single core has almost reached the limit, manufacturers tend to increase the performance mainly by increasing the number of cores in one CPU instead of increasing the performance of each core. This means unless there is a significant change in the architecture of JavaScript, the increasing of performance of JavaScript may be limited.

The poor performance of cryptographic primitives in JavaScript may deter potential users. In the experiment conducted by Chandra etc.[1], it takes more than 3 seconds to encrypt and transfer a file of 1MB in JavaScript, while about 90% of the time was consumed on encryption and decryption. Due to the existence of these restrictions in JavaScript, increasing the performance of cryptographic primitives in JavaScript is not simply a engineering problem, because the improvements is limited within the framework of pure JavaScript and the framework prevents the JavaScript code to make full use of the computation power of processors.

The emergence of WebGL(Web Graphics Library) provides us a choice to get rid of the restriction of JavaScript for more performance. WebGL is a web standard designed and maintained by the non-profit Khronos Group. It provides a JavaScript API based on OpenGL ES 2.0 for GPU accelerated rendering of 3D graphics within web browsers. The API is exposed through the HTML5 Canvas element as Document Object Model(DOM) interfaces. Developers can use WebGL to create shaders, textures, framebuffers in the graphics memory run shaders on GPUs directly. This indicates that we can run certain arithmetics on GPU directly by using WebGL APIs exposed in JavaScript. As GPU(Graphics Processing Unit) has been widely deployed as a de facto unit of personal computer and mobile devices, and provides highly parallel specialized processors the total throughput of which has surpassed CPU, the developers may gain great benefits on both performance and portability using WebGL for computation. This approach works similarly as the legacy GPGPU(General-Purpose Computation on Graphics Hardware)[2] technique which uses graphics APIs and shader language for general purpose programming and has been replaced by more dedicated GPGPU framework such as CUDA, OpenCL, etc. But in JavaScript, we found this technique showed us more advantages than it used to.

The aim of this work is to investigate how the WebGL can be used to accelerate browser side JavaScript cryptography computation. We selected the AES[2] as the focal algorithm to demonstrate the possibility and efficiency of WebGL acceleration. We made following contributions in this work:

- We analyzed the features provided by WebGL and discussed some issues that may lead to mistakes when using WebGL for GPGPU.
- We designed and implemented a WebGL version of AES that use the power of GPU for accelerating the AES encryption in JavaScript through WebGL API.

[2] `http://www.gpgpu.org/`

– We evaluated the performance of our implementation and compare the result with the leading AES implementation in pure JavaScript.

We found that our implementation runs several times faster than AES implementation in pure JavaScript on the platform with a powerful graphics card. Even on a machine with a low end integrated graphics card our implementation also runs well and performs almost as fast as the pure JavaScript Implementation. Our research demonstrates that it is possible and efficient to use WebGL to accelerate the general purpose computing in browsers, and provides a way to make the encryption and decryption practical in browsers.

The remainder of this paper is organized as follows. Section 2 gives a overview of related work. Section 3 gives a brief introduction to WebGL and detailed several important issues in general purpose computing using WebGL. Section 4 introduced the standard approach and the fast approach of AES algorithm. Section 5 describes how we design and implemented the WebGL version of AES. Section 6 shows how we conducted the experiment and the result of experiment. At last we conclude our paper and talk about our future work.

2 Related Work

There is no cryptographic primitives and high performance general computation APIs currently available in major browsers. Web cryptography API is trying to provide common cryptographic services in JavaScript through the object window.crypto, but there is only a draft at the moment and no browser has announced a time table to support this standard.

Some modern browsers also provide other ways to implement the logic of web applications besides JavaScript, such as programmable plug-ins and Java Applets, while all these techniques are not portable for browsers on all platform, such as smart phones, and the extra installations they require also bothers users.

Asm.js[3] provides a framework to compile the JavaScript code to a well defined subset of JavaScript instructions which are easier to optimize. It reduce the performance gap between JavaScript code and native code greatly, but it doesn't support parallel computing, either. WebCL[4] is a JavaScript binding to OpenCL for heterogeneous parallel computing, but there are only several prototypes at the moment. Native Client[3] gives browser-based applications the computational performance of native applications without compromising safety, but currently only Chrome on X86 platform is supported.

There has been several cryptography libraries implemented in pure JavaScript[5,6]. The most effective research previous published on accelerating symmetric cryptography in JavaScript was done by Stark et al.[4]. They studied a few optimizations and trade-offs for implementing AES effectively in JavaScript

[3] http://asmjs.org/
[4] http://www.khronos.org/webcl/
[5] http://people.eku.edu/styere/Encrypt/JS-AES.html
[6] https://code.google.com/p/crypto-js/

and built a highly optimized AES implementation in JavaScript. Their AES implementation is both faster and smaller than any other AES implementation in JavaScript before their work, but still dozens of times slower than native implementation.

D. Cook et al. [5] firstly implemented AES-128 on GPU by mapping the AES cipher to the standard fixed graphics pipeline using OpenGL, but their performance was only 184Kbps–1.53Mbps on Geforce3 Ti200, which was 40 to 100 times slower compared with CPU. Harrison et al.[6] used the shader-based programmable pipeline to implement AES and got a much better performance than Cook's research, but still under performed compared to some optimized implementations on standards CPU. Fleissner[7] accelerated the Montgomery exponentiation with OpenGL to more than 100 times faster than the standard algorithm. Moss[8] implemented RSA using an RNS based approach using OpenGL and gave results comparable to the fastest CPU implementation.

The emergence of dedicated GPGPU frameworks such as CUDA[9], Brooks[10] and OpenCL[11] inspired the research on accelerating crypto primitives with GPU. Manavski[12] implemented AES using CUDA and showed GPU can perform as an efficient cryptographic accelerator for the first time, their solution was 20 times faster than the native implementation. Szerwinski et al.[13] used CUDA to accelerated DSA, RSA and ECC. Zhang et al.[14] accelerated composite order bilinear pairing with CUDA.

3 WebGL Background

WebGL is a standard of graphics APIs based on OpenGL ES 2.0 developed by Khronos group, the version 1.0 of the WebGLspecification was released March 2011[15]. Until the time of this paper writing, most major desktop browsers and mobile browsers have supported WebGL officially or internally. This indicates our approach can be used in most browsers across platforms without modification.

WebGL is a shader-based API using OpenGL Shading Language(GLSL), which makes full use of programmable pipelines in GPUs and provides a great convenience for general purpose computing. As illustrated in Figure 1, using WebGL for general purpose computing works in similar as rending a frame in graphics computing: developer passes bunch of input data into graphics memory as textures, implements the computing logic in shaders, renders the computing result into framebuffers, then read the result back to the main memory or use the result as the input of the next iteration. These steps involves much glue code, how to launch these steps and what these steps mean can be found at WebGL tutorials and references books.

The key to general purpose computing in GPU via WebGL is how to map the computing procedure to graphics rendering precisely and effectively. Since WebGL is designed for graphics computing, it supports only limited data structures, especially for input and output, and it does not have full support of integer arithmetics until now. This indicates that developers sometimes have to map the unsupported arithmetics and data structures to supported ones in order

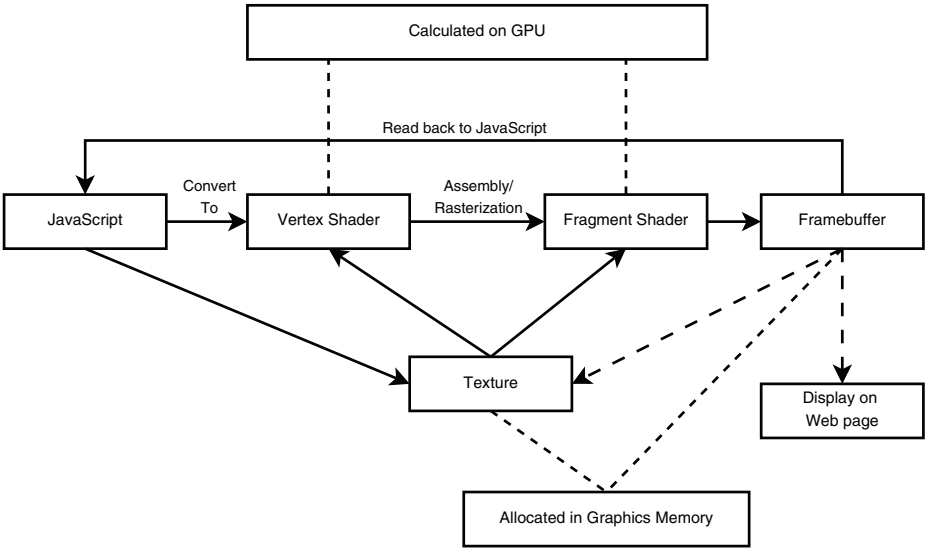


Fig. 1. A simplified view of WebGL pipeline model of computation

to leverage WebGL and GPU for computing. And any computing that is used in shaders must take the precision that shaders supported into consideration in order to prevent the unexpected truncation affect the result. As GPU is not effective at executing the serial control logic, the algorithm of the computing must be optimized for parallel. From our observation and experiments, the following concrete principles are helpful when using WebGL and GPU for general purpose computing.

Simplify the logic of shaders. As we have mentioned, GPU is not good at executing the control logics, such as the conditional branch. Another reason is complex logic may consume much longer time to compile. A possible mitigation is to complete the computing with multiple renders, and use CPU to execute the control flow between renders.

Reduce the number of data transfers on bus. As the bandwidth of the bus is limited, transfer data between the main memory and the graphics memory is time consuming.

Batch processing the data. Because of the executing model of GPU, computing with a block of input may take the same time as computing with 1,000 blocks.

Memory access pattern. As most circuits of GPU are used to implement the arithmetic units, the space of cache and registers is relatively small, there will be a great penalty on the performance if the cache miss occurs too much. It is important for the developer to optimize the memory access pattern, for example, try not to access a large part of data randomly.

Number conversion. In WebGL, the data supplied to the texture in JavaScript should be 8-bit integers, but the data got in GLSL are floating

numbers in $[0, 1]$. In brief, the integers are linearly mapped to the floating number when they are transferred from main memory to graphics memory, and mapped in the reverse way when floating numbers are transferred back from graphics memory to main memory. So any linear transformations on the floating form in GLSL would be equally applied to the integer form in JavaScript. This is a feature or restriction in another word of GLSL, the developers must be conscious of this.

Texture coordinate. In shaders, the texel is accessed by function `texture2D()` using the coordinate (x, y). Each dimension of the coordinate is a floating number within the range of $[0, 1]$, which means there is not only one single value can be used to look up any texel, but a range of coordinates. Generally speaking, any coordinate within the scope of the texel can be used to look up the texel, but there is no guarantee for different hardware to behave totally the same for boundary values. For the texel at i-th column and j-th row in a texture whose width is w and height is h, using $(\frac{i}{w-1}, \frac{j}{h-1})$ as the coordinate is acceptable in most cases, but since accuracy is critical for general purpose computing, it is necessary to calculate the coordinate by $(\frac{2\times i+1}{2\times w}, \frac{2\times j+1}{2\times h})$ sometimes.

4 AES Background

AES(Advanced Encryption Standard) [2] is a symmetric block cipher that encrypt plain text blocks of 128 bits with various key length of 128 bits, 192 bits or 256 bits. It is a restricted version of Rijndael symmetric block cipher that can encrypt and decrypt blocks of 128 bits using a key size of 128-bit, 192-bit, and 256-bit length. The cipher is basically a series of round transformations on blocks with round keys expanded from the original key using a key schedule algorithm [16,2], the output of each round is the input of the next round. The number of the rounds is determined by the key length: 10 rounds for 128-bit, 12 rounds for 192-bit, and 14 rounds for 256-bit. The block is depicted as a 4×4 column-major order matrix of bytes, termed **state**.

The standard implementation of AES encryption starts with an AddRoundKey operation on the state, followed by 10/12/14 round transformations depending on the length of the key. Each round transformation includes 4 successive steps except the final round: SubBytes, ShiftRows, MixColumns and AddRoundKey. The final round is similar except the lack of the MixColumns step. Decryption is done by reversing each step of encryption using the same key.

SubBytes. Each byte of the state is substituted independently using a predefined substitution box(S-box) computed over the Galois Field $GF(2^8)$ [2].

ShiftRows. Rows are rotated by 0, 1, 2 and 3 bytes, respectively, to the left.

MixColumns. A substitution that makes use of arithmetic over the Galois Field $GF(2^8)$.

AddRoundKey. A simple bitwise XOR of the state with a piece of the expanded round key.

For processors supporting 32-bit or greater word length, Daemen and Rijmen detailed a fast implementation approach that combines the SubBytes, ShiftRows and MixColumns transformations into four 256-entry(each entry is 4 bytes) lookup tables("T-Table")[16,2]. The T-Table approach reduces the SubBytes, ShiftRows, MixColumns operations in round transformation to simply updating the j-th column of the state according to the Equation 1.

$$[s'^{j}_{0}, s'^{j}_{1}, s'^{j}_{2}, s'^{j}_{3}]^T = \bigoplus_{i=0}^{3} T_i[s_{i,j} + C_i], 0 \leq j \leq 3 \quad (1)$$

where $s_{j,k}$ is the byte in the j-th row and k-th column of the state, and C_i is a constant equivalently doing the ShiftRows in place. Each T_i is a rotation of the other. After the state is updated, the step AddRoundKey is performed to complete the round operation. As we can see from the equation, there are only XORs and table lookups needed in this technique.

The block cipher itself is not sufficient for the security of multiple blocks, a mode of operation is needed. The cipher mode is important for both performance and security for block ciphers. The CBC mode, OFB mode and other chained modes are secure but not efficient for parallel computing since the computation of the next block depends on the result of the previous one. The ECB mode is efficient for parallel computing but insecure. CTR mode is both secure and efficient for parallel computing as the counter can be precomputed efficiently and simple to implement. Another advantage for CTR mode is that only the encryption procedure of the cipher is needed for both encryption and decryption in CTR mode.

5 WebGL Version of AES

5.1 Overview

We implemented AES encryption using 128-bit key length using WebGL. The decryption and other key lengths can be implemented with minor modifications. We implemented the ECB mode for plain text of various length, and other modes such as CTR mode can be implemented simply with a extra shader. As our purpose here is to demonstrate the feasibility and efficiency of WebGL for cryptography, we just keep our implementation simple but convincing.

Specifically, the algorithm we implemented is the fast approach for 32-bit processors as mentioned in section 4, because this approach mainly involves two types of operation: table lookup and XOR. Since the arithmetic operation supported by GLSL is limited, this feature will facilitate our work. We did not implement the key scheduling in WebGL, because this procedure is a serial of limited operations that can be done in no time in JavaScript in CPU. And once the key schedule is expanded, it can be used repeatedly to encrypt message of any length. The key schedule, the plain text and other parameters would be packed into textures as input to shaders, the cipher text would be written to the framebuffer by shaders and read back into JavaScript. We can encrypt multiple blocks in the same time with GPU and WebGL in order to exert the power of

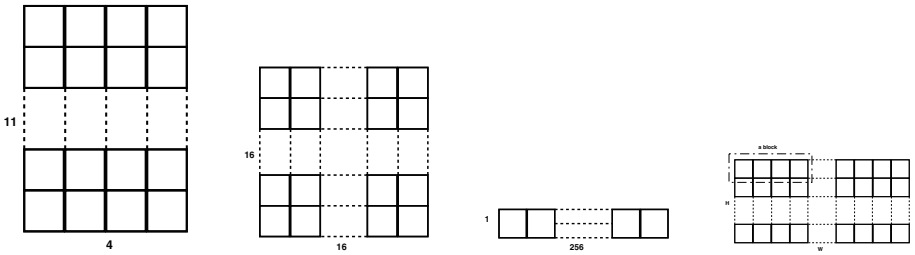
high throughput of GPU. We denote the blocks we encrypt at the same time as a packet.

The multiple-target technique is not supported in WebGL, so the output of a shader could be only a texel which is no more than four bytes. As the size of one block is 16 bytes, it takes four shader instances to produce the result of one block. As each round operation requires the whole state output by the previous round operation, there should be a synchronization after each round, or each shader instance has to execute all the rounds completely before the final one. The too complex logic takes much more time to compile and run, therefore we decided to split the encryption logic into 3 independent shader programs: the first implemented the initial round which just combines the key with the state by a XOR operation, the second implemented the round operation between the state and a piece of round key that can be specific by a parameter, the third implemented the final round operation. In this case, we have to render for 11 times in total to encrypt a packet regardless of the size of the packet with a 128-bit key: 1 for the initial round, 9 for round operations, and 1 for the final round. As we don't want to read the intermediate result back into main memory, we used two framebuffers alternatively: the one holding the result of the previous round will be used as the input to the next round, since the framebuffer of one render can also be used as the texture to another render.

5.2 XOR Operation

The AES fast approach requires to calculate the XOR of two 32-bit unsigned integers, while both 32-bit integer and XOR operation are not supported in GLSL. Although each texel can hold up to 32-bit data, it actually consists of four 8-bit floating numbers in GLSL or four 8-bit integer in JavaScript. As XOR of two integers is just the combination or XOR of each bits at corresponding position, we can just hold the 32-bit integers in texels and calculate the XOR of two texels by just calculating each 8-bit component of them. Therefore we can construct a table whose element at i-th column and j-th row is $i \oplus j$, then the calculation of $i \oplus j$ can be transformed to looking up the element at i-th column and j-th row in the table. In GLSL, the situation is a little sophisticated: there is slightly difference between the mapped floating number and the texture coordinate as mentioned the Section 3, an integer i will be transformed to $i/255.0$ in GLSL, as the reliable coordinate to access the i-th column or row is $\frac{i \times 2+1}{256 \times 2}$, it is better to convert the floating form f_i of the integer i to the texture coordinate by $\frac{f_i \times 255.0 \times 2+1}{256.0 \times 2}$ for accuracy, especially on GPUs with lower precision.

For XOR of two 8-bit integers, there would be $256 \times 256 = 65536$ entries in the table. The random access to a table of this size in GPU may cause plenty of cache misses. An optimization has been proposed in the paper[17]: We can construct a table holding the XOR of any two 4-bit integers instead of for any two 8-bit integers. For any two 8-bit integers first we divide it equally into two 4-bit parts and calculate the XOR of each part, then combine the result to get the XOR of original 8-bit integer. This optimization reduces the table from 256×256 to 16×16 and has been proved to be much faster in the experiments[17].



(a) Key Schedule (b) XOR Table (c) T-Table (d) States

Fig. 2. The memory layout of the key schedule, the XOR table, the T-Table, and the state, each square in the figure stands for a texel

5.3 Memory Layout

We will discuss the memory layout of the input, output and parameters in graphics memory and how to access them in this part. As the texture and the framebuffer has two dimension, there could be many possibilities for memory layout and access pattern of the same piece of data, our purpose is to choose the most efficient and easy-to-use one.

The key schedule is common for all plain text blocks encrypted using the same key. For the key of the same length, there are 11 round keys including the original one, each round key is 16 bytes. Each texel holds 4 bytes at most, we use 4 continuous texels for one piece of round key, and each round key is held in a row in sequence in a texture as shown in Figure 2(a). In this layout, the coordinate used to access the i-th element of the j-th round key is easy to calculate as $(\frac{i\times 2+1}{2\times 4}, \frac{j\times 2+1}{2\times 11})$ as described in Section 3.

The original T-Table takes up 4KB memory in total. As the T-Table has to be accessed randomly, it is better if we can reduce the memory usage to reduce the cache miss. Since each T-Table is a rotation of the other, the Equation 1 can be optimized to the Equation 2:

$$\begin{aligned}[s'^{j}_{0}, s'^{j}_{1}, s'^{j}_{2}, s'^{j}_{3}]^T = \quad & T_0[s_{0,j} + C_0] \oplus Rot(T_0[s_{1,j} + C_1] \oplus \\ & Rot(T_0[s_{2,j} + C_2] \oplus Rot(T_0[s_{3,j} + C_3])))\end{aligned} \tag{2}$$

In this case, there is only one T-Table needed and takes up 1KB memory space only. Each entry of the T-Table is 32-bit long and can be packed into one texel, so we put the whole table in a texture whose size is 256×1. The layout of T-Table is shown in Figure 2(c) The i-th element of the table can be looked up with the coordinate $(\frac{i\times 2+1}{256\times 2}, 0.5)$.

The XOR table is a 16×16 matrix as shown in Figure 2(b) and mentioned above, we use the alpha component of the texel to hold each XOR result.

The size of the state is 16 bytes, so we can keep it in sequence in a row in the texture. Since it is not effective to encrypt just one block at one time, we have to supply multiple blocks to the input texture. The difficulty is how to locate the right block of state for each shader instance when there are multiple

blocks supplied. As the size of input and output of the encryption are same, we can construct the framebuffers holding the result and the texture holding the input of the same size, so there would be a one to one map between the input and the output. This corresponding relation can be constructed by rendering a rectangle to fill the viewport with a designed vertex shader program easily. A simple approach is to place only one block in a row in the texture or framebuffer, then for the shader instance outputting to (s, t) in the framebuffer, it can locate the input to it at the t-th row of the input texture easily. Since the side length of the texture and framebuffer that supported by GPU is limited to the order of thousands, this approach can encrypt only thousands of blocks at the same time, so it is necessary to put multiple blocks in each row as shown in Figure 2(d). In this case, for the shader instance outputting to (s, t) in the framebuffer whose size is $w \times h$, the coordinate of the i-th column($0 \leq i < 4$) can be calculated by Equation 3. In this approach, we can encrypt millions of blocks at the same time.

$$\begin{aligned} s_i &= \frac{(floor(s \times w \times 2.0/8.0) \times 8.0 + 2.0 \times i + 1.0)}{(w \times 2.0)} \\ t_i &= t \end{aligned} \tag{3}$$

6 Experiment

Table 1. The configuration of test machines

Machine	A	B	C
Platform	Desktop	Laptop	Pad
Model	Dell OptiPlex 990MT	Lenovo Y400N	Nexus 4
OS	Ubuntu 12.04	Windows 8 Pro	Android 4.2.2
CPU	i7 2660	i5 3230M	A5
GPU	NVIDIA 560TI	NVIDIA G750M	PowerVR SGX 543

We expected to find out how fast the WebGL AES can be, and the impaction of packet size on performance through the experiment. So we designed the experiment as follows: We launched the the implementation with different packet sizes, start from one block which is 16 bytes, up to 1M bytes, and the next packet size is always four times of the previous one. We used the implementation to encrypt the randomly generated plain text whose length is 3 times of the packet size, so the algorithm would run 3 times to finish the encryption. Then we get the time consumed during the encryption and calculate the throughput of the implementation. We also launched the experiment on SJCL(Stanford JavaScript Crypto Library)[4] which is the fastest pure JavaScript AES implementation currently reported in the same way for comparison.

We launched the experiment in different browsers on different platforms in order to have a comprehensive view of the implementation. We used two most popular browsers of their latest version: Chrome(27.0) and Firefox(23.0). The machines used belongs to different platforms: desktop, laptop and pad. The major configuration are summarized in Table 1 and indexed by characters.

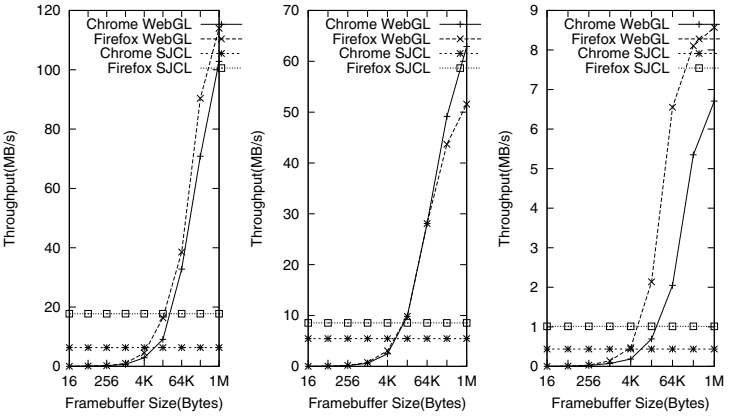


Fig. 3. The throughput of different approaches with different packet size

The result of experiment is shown in Figure 3. We found that the WebGL approach ran well on various operating systems and hardware platforms. The speed of pure JavaScript AES implementation is almost stable regardless of the packet size, while the speed of WebGL AES implementation continuously increased as the packet size increased. From the figure we can see that when the packet size is small, the WebGL version AES ran slowly, but when the packet size became larger (more than 16K), the WebGL version AES ran much faster than pure JavaScript version AES on all platforms. And when the packet size is large enough(greater or equal than 1MB), the speed of WebGL version AES could be more than 10 times of pure JavaScript version AES. Considering the size of pictures, songs or other multimedia files users daily use could all be several MBs or even dozens of MBs, the WebGL version AES could surely exert its full power in modern Web applications.

7 Conclusion

In this paper, we proposed a new approach to accelerate AES in JavaScript via WebGL. Our primary contribution is demonstrating the feasibility of using GPU via WebGL to provide much better performance than pure JavaScript since JavaScript is an untyped language and it doesn't support parallel computing.

Our approach was more than ten times faster than pure JavaScript implementation of AES. The performance is sufficient for most cryptographic operations in web applications to provide a smooth user interface. This also demonstrated that legacy techniques such as using graphics API for GPGPU can be very powerful in certain runtime environment.

Acknowledgement. I would like to thank my supervisor, Researcher Zhi Guan and Professor Zhong Chen for their excellent guidance throughout the writing of the paper.

References

1. Chandra, R., Gupta, P., Zeldovich, N.: Separating web applications from user data storage with BSTORE. MIT web domain (June 2010)
2. NIST. Specification for the Advanced Encryption Standard (AES). Technical Report Federal Information Processing Standards (FIPS) 197, National Institute of Standards and Technology (November 2001)
3. Yee, B., Sehr, D., Dardyk, G., Chen, J.B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., Fullagar, N.: Native client: A sandbox for portable, untrusted x86 native code. In: 2009 30th IEEE Symposium on Security and Privacy, pp. 79–93 (2009)
4. Stark, E., Hamburg, M., Boneh, D.: Symmetric Cryptography in Javascript. In: ACSAC, pp. 373–381. IEEE Computer Society (2009)
5. Cook, D.L., Ioannidis, J., Keromytis, A.D., Luck, J.: cryptographics: Secret key cryptography using graphics cards. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 334–350. Springer, Heidelberg (2005)
6. Harrison, O., Waldron, J.: AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 209–226. Springer, Heidelberg (2007)
7. Fleissner, S.: GPU-accelerated montgomery exponentiation. In: Shi, Y., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2007, Part I. LNCS, vol. 4487, pp. 213–220. Springer, Heidelberg (2007)
8. Moss, A., Page, D., Smart, N.P.: Toward acceleration of RSA using 3D graphics hardware. In: Galbraith, S.D. (ed.) Cryptography and Coding 2007. LNCS, vol. 4887, pp. 364–383. Springer, Heidelberg (2007)
9. C. CUDA. Programming guide. NVIDIA Corporation (July 2012)
10. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: stream computing on graphics hardware. ACM Trans. Graph. 23(3), 777–786 (2004)
11. Munshi, A. (ed.): Khronos OpenCL Working Group. The opencl specification (2008)
12. Manavski, S.A.: CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In: ICSPC 2007, pp. 65–68 (November 2007)
13. Szerwinski, R., Güneysu, T.: Exploiting the Power of GPUs for Asymmetric Cryptography. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 79–99. Springer, Heidelberg (2008)

14. Zhang, Y., Xue, C.J., Wong, D.S., Mamoulis, N., Yiu, S.M.: Acceleration of composite order bilinear pairing on graphics hardware. In: Chim, T.W., Yuen, T.H. (eds.) ICICS 2012. LNCS, vol. 7618, pp. 341–348. Springer, Heidelberg (2012)
15. Marrin, C.: Webgl specification. Khronos WebGL Working Group (2011)
16. Daemen, J., Rijmen, V.: The design of Rijndael: AES–the Advanced Encryption Standard. Springer, Berlin (2002)
17. Harrison, O., Waldron, J.: AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 209–226. Springer, Heidelberg (2007)