SSLShader: Cheap SSL Acceleration with Commodity Processors

Keon Jang[+], Sangjin Han[+], Seungyeop Han[*], Sue Moon[+], and KyoungSoo Park[+]

[+]*KAIST* [*]*University of Washington*

Abstract

Secure end-to-end communication is becoming increasingly important as more private and sensitive data is transferred on the Internet. Unfortunately, today's SSL deployment is largely limited to security or privacy-critical domains. The low adoption rate is mainly attributed to the heavy cryptographic computation overhead on the server side, and the cost of good privacy on the Internet is tightly bound to expensive hardware SSL accelerators in practice.

In this paper we present high-performance SSL acceleration using commodity processors. First, we show that modern graphics processing units (GPUs) can be easily converted to general-purpose SSL accelerators. By exploiting the massive computing parallelism of GPUs, we accelerate SSL cryptographic operations beyond what state-of-the-art CPUs provide. Second, we build a transparent SSL proxy, SSLShader, that carefully leverages the trade-offs of recent hardware features such as AES-NI and NUMA and achieves both high throughput and low latency. In our evaluation, the GPU implementation of RSA shows a factor of 22.6 to 31.7 improvement over the fastest CPU implementation. SSLShader achieves 29K transactions per second for small files while it transfers large files at 13 Gbps on a commodity server machine. These numbers are comparable to high-end commercial SSL appliances at a fraction of their price.

1 Introduction

Secure Sockets Layer (SSL) and Transport Layer Security (TLS) have served as de-facto standard protocols for secure transport layer communication for over 15 years. With endpoint authentication and content encryption, SSL delivers confidential data securely and prevents eavesdropping and tampering by random attackers. Online banking, e-commerce, and Web-based email sites typically employ SSL to protect sensitive user data such as passwords, credit card information, and private content. Operating atop the transport layer, SSL is used for various application protocols such as HTTP, SMTP, FTP, XMPP, and SIP, just to name a few.

Despite its great success, today's SSL deployment is largely limited to security-critical domains or enterprise applications. A recent survey shows that the total number of registered SSL certificates is slightly over one million [18], reflecting less than 0.5% of active Internet sites [19]. Even in the SSL-enabled sites, SSL is often enforced only for a fraction of activities (e.g., password submission or billing information). For example, Web-based email sites such as Windows Live Hotmail[1] and Yahoo! Mail[2] do not support SSL for the content, making the private data vulnerable for sniffing in untrusted wireless environments. Popular social networking sites such as Facebook[3] and Twitter[4] allow SSL only when users make explicit requests with a noticeable latency penalty. In fact, few sites listed in Alexa top 500 [2] enable SSL by default for the entire content.

The low SSL adoption is mainly attributed to its heavy computation overhead on the server side. The typical processing bottleneck lies in the key exchange phase involving public key cryptography [22, 29]. For instance, even the latest CPU core cannot handle more than 2K SSL transactions per second (TPS) with 1024-bit RSA while the same core can serve over 10K plaintext HTTP requests per second. As a workaround, high-performance SSL servers often distribute the load to a cluster of machines [52] or offload cryptographic operations to dedicated hardware proxies [3, 4, 6, 13] or accelerators [9, 10, 14, 15]. Consequently, user privacy in the Internet still remains an expensive option even with the modern processor innovation.

Our goal is to find a practical solution with commodity processors to bring the benefits of SSL to all private In-

[1] http://explore.live.com/windows-live-hotmail/
[2] http://mail.yahoo.com/
[3] http://www.facebook.com/
[4] http://www.twitter.com/

ternet communication. In this paper, we present our approach in two steps. First, we exploit commodity graphics processing units (GPUs) as high-performance cryptographic function accelerators. With hundreds of streaming processing cores, modern GPUs execute the code in the single-instruction-multiple-data (SIMD) fashion, providing ample computation cycles and high memory bandwidth to massively parallel applications. Through careful algorithm analysis and parallelization, we accelerate RSA, AES and SHA-1 cryptographic primitives with GPUs. Compared with previous GPU approaches that take hundreds of milliseconds to a few seconds to reach the peak RSA performance [37,56], our implementation produces the maximum throughput with one or two orders of magnitude smaller latency, which is well-suited for interactive Web environments.

Second, we build SSLShader, a GPU-accelerated SSL proxy that transparently handles SSL transactions for existing network servers. SSLShader selectively offloads cryptographic operations to GPUs to achieve high throughput and low latency depending on the load level. Moreover, SSLShader leverages the recent hardware features such as multi-core CPUs, the non-uniform memory access (NUMA) architecture, and the AES-NI instruction set.

Our contributions are summarized as follows:

(i) We provide detailed algorithm analysis and parallelization techniques to scale the performance of RSA, AES and SHA-1 in GPUs. To the best of our knowledge, our GPU implementation of RSA shows the highest throughput reported so far. On a single NVIDIA GTX580 card, our implementation shows 92K RSA operations/s for 1024-bit keys, a factor of 27 better performance over the fastest CPU implementation with a single 2.66 GHz Intel Xeon core.

(ii) We introduce opportunistic workload offloading between CPU and GPU to achieve both low latency and high throughput. When lightly loaded, SSLShader utilizes low-latency cryptographic code execution by CPUs, but at high load it batches and offloads multiple cryptographic operations to GPUs.

(iii) We build and evaluate a complete SSL proxy system that exploits GPUs as SSL accelerators. Unlike prior GPU work that focuses on microbenchmarks of cryptographic operations, we focus on systems interaction in handling the SSL protocol. SSLShader achieves 13 Gbps SSL large-file throughput handling 29K SSL TPS on a single machine with two hexa-core Intel Xeon 5650's.

The rest of the paper is organized as follows. In Section 2, we provide a brief background on SSL, popular cryptographic operations, and the modern GPU. In Sections 3 and 4 we explain our optimization techniques for RSA, AES and SHA-1 implementations in a GPU. In Sections 5 and 6, we show the design and evaluation of SSLShader. In Sections 7 and 8 we discuss related work and conclude.

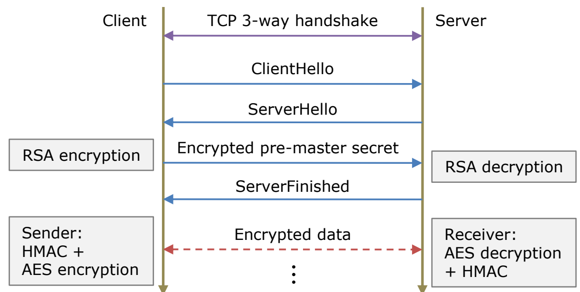


Figure 1: SSL handshake and data

2 Background

In this section, we provide a brief introduction to SSL and describe the cryptographic algorithms used in TLS_RSA_WITH_AES_128_CBC_SHA, one of the most popular SSL cipher suites. We also describe the basic architecture of modern GPUs and strategies to exploit them for cryptographic operations. In this paper we use TLS_RSA_WITH_AES_128_CBC_SHA as a reference cipher suite, but we believe our techniques to be easily applicable to other similar algorithms.

2.1 Secure Sockets Layer

SSL was developed by Netscape in 1994 and has been widely used for secure transport layer communication. SSL provides three important security properties in private communication: data confidentiality, data integrity, and end-point authentication. From SSL version 3.0, the official name has changed to TLS and the protocol has been standardized by IETF. SSL and TLS share the same protocol structure, but they are incompatible, since they use different key derivation functions to generate session and message authentication code (MAC) keys.

Figure 1 describes how the SSL protocol works. A client sends a ClientHello message to the target server with a list of supported cipher suites and a nonce. The server picks one (asymmetric cipher, symmetric cipher, MAC algorithm) tuple in the supported cipher suites, and responds with a ServerHello message with the chosen cipher suite, its own certificate and a server-side nonce. Upon receiving the ServerHello message, the client verifies the server's certificate, generates a pre-master secret and encrypts it with the server's public key. The encrypted pre-master secret is delivered to the server, and both parties independently generate two symmetric cipher session keys and two MAC keys using a predefined key derivation function with the pre-master key and the two nonces as input. Each (session, MAC) key pair is

used for encryption and MAC generation for one direction (e.g., client to server or server to client).

In the Web environment where most objects are small, the typical SSL bottleneck lies in decrypting the pre-master secret with the server-side private key. The client-side latency could increase significantly if the server is overloaded with many SSL connections. When the size of an object is large, the major computation overhead shifts to symmetric cipher execution and MAC calculation.

2.2 Cryptographic Operations

TLS_RSA_WITH_AES_128_CBC_SHA uses RSA, AES, and a Secure Hash Algorithm (SHA) based HMAC. Below we sketch out each cryptographic operation.

2.2.1 RSA

RSA [53] is an asymmetric cipher algorithm widely used for signing and encryption. To encrypt, a plaintext message is first transformed into an integer M, then turned into a ciphertext C with:

$$C := M^e \bmod n \quad (1)$$

with a public key (n, e). Decryption with a private key (n, d) can be done with

$$M := C^d \bmod n \quad (2)$$

C, M, d, and n are k-bit large integers, typically $1,024$, $2,048$, or even $4,096$ bits (or roughly 300, 600, or $1,200$ decimal digits). Since e is chosen to be a small number (common choices are 3, 17, and $65,537$), public key encryption is 20 to 60 times faster than private key decryption. RSA operations are compute-intensive, especially for SSL servers. Because servers perform expensive private key decryption for each SSL connection, handling many concurrent connections from clients is a challenge. In this paper we focus on private key RSA decryption, the main computation bottleneck on the server side.

2.2.2 AES

Advanced Encryption Standard (AES) [32] is a popular symmetric block cipher algorithm in SSL. AES divides plaintext message into 128-bit fixed blocks and encrypts each block into ciphertext with a 128, 192, or 256-bit key. The encryption algorithm consists of 10, 12, or 14 rounds of transformations depending on the key size. Each round uses a different round key generated from the original key using Rijndael's key schedule.

We implement AES encryption and decryption in cipher-block chaining (CBC) mode. In CBC mode, each plaintext block is XORed with a random block of the same size before encryption. The i-th block's random block is simply the $(i-1)$-th ciphertext block, and the initial random block, called the Initialization Vector (IV), is randomly generated and is sent in plaintext along with the encrypted message for decryption.

2.2.3 HMAC

Hash-based Message Authentication Code (HMAC) is used for message integrity and authentication. HMAC is defined as

$$HMAC(k,m) = H((k \oplus opad) \| H((k \oplus ipad) \| m)) \quad (3)$$

H is a hash function, k is a key, m is a message, and $ipad$ and $opad$ are predefined constants. Any hash function can be combined with HMAC and we use SHA-1 as it is the most popular.

2.3 GPU

Modern GPUs have hundreds of processing cores that can be used for general-purpose computing beyond graphics rendering. Both NVIDIA and AMD provide convenient programming libraries to use their GPUs for computation or memory-intensive applications. We use NVIDIA GPUs here, but our techniques are applicable to AMD GPUs as well.

A GPU executes code in the SIMD fashion that shares the same code path working on multiple data at the same time. For this reason, a GPU is ideal for parallel applications requiring high memory bandwidth to access different sets of data. The code that the GPU executes is called a *kernel*. To make full use of massive cores in a GPU, many threads are launched and run concurrently to execute the kernel code. This means more parallelism generally produces better utilization of GPU resources.

GPU kernel execution takes the following four steps: (*i*) the DMA controller transfers input data from host memory to GPU (device) memory; (*ii*) a host program instructs the GPU to launch the kernel; (*iii*) the GPU executes threads in parallel; and (*iv*) the DMA controller transfers the result data back to host memory from device memory.

The latest NVIDIA GPU is the GTX580, codenamed Fermi [20]. It has 512 cores consisting of 16 Streaming Multiprocessors (SMs), each of which has 32 Stream Processors (SPs or CUDA cores). In each SM, 48 KB shared memory (scratchpad RAM), 16 KB L1 cache, and $32,768$ 4-byte registers allow high-performance processing. To hide the hardware details, NVIDIA provides Compute Unified Device Architecture (CUDA) libraries to software programmers. CUDA libraries allow easy programming for general-purpose applications. More details about the architecture can be found in [47,48].

The fundamental difference between CPUs and GPUs comes from how transistors are composed in the processor. A GPU devotes most of its die area to a large array of Arithmetic Logic Units (ALUs). In contrast, most CPU resources serve a large cache hierarchy and

a control plane for sophisticated acceleration of a single thread (e.g., out-of-order execution, speculative loads, and branch prediction), which are not much effective in cryptography. Our key insight of this work is that compute-intensive cryptographic operations can benefit from the abundant ALUs in a GPU, given enough parallelism (intra- and inter-flow).

3 Optimizing RSA for GPU

For RSA implementation on GPUs, the main challenge is to achieve high throughput while keeping the latency low. Naïve porting of CPU algorithms to a GPU would cause severe performance degradation, wasting most GPU computational resources. Since a single GPU thread runs at 10x to 100x slower speed than a CPU thread, the naïve approach would yield unacceptable latency.

In this section, we describe our approach and design choices to maximize performance of RSA decryption on GPUs. The key point in maximizing RSA performance lies in high parallelism. We exploit parallelism in the message level, in modular exponentiation, and finally in the word-size modular multiplication. We show that our parallel Multi-Precision (MP) algorithm obtains a significant gain in throughput and curbs latency increase to a reasonable level.

3.1 How to Parallelize RSA Operations?

Our main parallelization idea is to batch multiple RSA ciphertext messages and to split those messages into thousands of threads so that we can keep all GPU cores busy. Below we give a brief description of each level.

Independent Messages: At the coarsest level, we process multiple messages in parallel. Each message is inherently independent of other messages; no coordination between threads belonging to different messages is required.

Chinese Remainder Theorem (CRT): For each message, (2) can be broken into two independent modular exponentiations with CRT [51].

$$M_1 = C^{d \bmod (p-1)} \bmod p \tag{4a}$$

$$M_2 = C^{d \bmod (q-1)} \bmod q \tag{4b}$$

where p and q are $k/2$-bit prime numbers chosen in private key generation ($n = p \times q$). All four parameters, p, q, $d \bmod (p-1)$, and $d \bmod (q-1)$, are part of the RSA private key [38].

With CRT, we perform the two $k/2$-bit modular exponentiations in parallel. Each of which requires roughly 8 times less computation than k-bit modular exponentiation. Obtaining M from M_1 and M_2 adds only small overheads, compared to the gain from two $k/2$-bit modular exponentiations.

Large Integer Arithmetic: Since the word size of a computer is usually 32 or 64-bit, large integers must be broken into small multiple words. We can run multiple threads, each of which processes a word. However, we need carry-borrow processing or base extension in order to coordinate the outcome of per-word operations between threads. We consider two algorithms, standard Multi-Precision (MP) and Residue Number System (RNS), to represent and compute large integers. These algorithms are commonly used in software and hardware implementations of RSA.

3.2 Optimization Strategies

In our MP implementation we exploit the following two optimization strategies: (*i*) reducing the number of modular multiplications with the Constant Length Nonzero Windows (CLNW) partitioning algorithm; (*ii*) adopting Montgomery's reduction algorithm to improve the efficiency of each modular multiplication routine performed at each step of the exponentiation. These optimization techniques are also helpful for both serial software and hardware implementations, as well as for our GPU parallel implementations.

CLNW: With the binary square-and-multiply method, the expected number of modular multiplications is $3k/2$ for k-bit modular exponentiation [41]. For example, the expected number of operations for 512-bit modular exponentiation (used for 1024-bit RSA with CRT) is 768. The number can be reduced with sliding window techniques that scan multiple bits, instead of individual bits of the exponent.

We have implemented CLNW and reduced the number of modular multiplications from 768 to 607, achieving a 21% improvement [28]. One may instead use the Variable Length Nonzero Window (VLNW) algorithm [26], but it is known that VLNW does not give any performance advantage over CLNW on average [50].

Montgomery Reduction: In a modular multiplication $c = a \cdot b \bmod n$, an explicit k-bit modulo operation following a naïve multiplication should be avoided. Modulo operation requires a trial division by modulus n for the quotient, in order to compute the remainder. Division by a large divisor is very expensive in both software and hardware implementations and is not easily parallelizable, and thus inappropriate especially for GPUs.

Montgomery's algorithm allows a modular multiplication without a trial division [45]. Let

$$\overline{a} = a \cdot R \bmod n \tag{5}$$

be the *montgomeritized* form of a modulo n, where R and n are coprime and $n < R$. Montgomery multiplication

Algorithm 1 MMUL: Montgomery multiplication

Input: $\bar{a}, \bar{b}$
Output: $\bar{a} \cdot \bar{b} \cdot R^{-1} \bmod n$
Precomputation: R^{-1} such that $R \cdot R^{-1} \equiv 1 \pmod{n}$
n' such that $R \cdot R^{-1} - n \cdot n' = 1$

1: $T \leftarrow \bar{a} \cdot \bar{b}$
2: $M \leftarrow T \cdot n' \bmod R$
3: $U \leftarrow (T + M \cdot n)/R$
4: **if** $U \geq n$ **then**
5: **return** $U - n$
6: **else**
7: **return** U
8: **end if**

is defined as in Algorithm 1. If we set R to be 2^k, the division and modulo operations with R can be done very efficiently with bit shifting and masking.

Note that the result of Montgomery multiplication of $\overline{a}$ and $\overline{b}$ is still $\overline{a} \cdot \overline{b} \cdot R^{-1} \bmod n = \overline{a \cdot b} \bmod n$, the montgomeritized form of $a \cdot b$. For a modular exponentiation, we convert a ciphertext C into $\overline{C}$, get $\overline{C^d}$ with successive Montgomery multiplication operations, and invert it into $C^d \bmod n$. In this process, expensive divisions or modulo operations with n are eliminated.

The implementation of Montgomery multiplication depends on data structures used to represent large integers. Below we introduce our MP implementation.

3.3 MP implementation

The standard Multi-Precision algorithm is the most convenient way to represent large integers in a computer [41]. A k-bit integer A is broken into $s = \lceil k/w \rceil$ words of a_i's, where $i = 1, \ldots, s$ and w is typically set to the bit-length of a machine word (e.g., 32 or 64). Here we describe our MP implementation of Montgomery multiplication and various optimization techniques.

3.3.1 Multiplication

In Algorithm 1, the multiplication of two s-word integers appear three times in lines 1, 2, and 3. The time complexity of the serial multiplication algorithm that performs a shift-and-add of partial products is $O(s^2)$ (also known as the *schoolbook* multiplication). Implementation of an $O(s)$ parallel algorithm with linear scalability is not trivial due to carry processing. We have implemented an $O(s)$ parallel algorithm on s processors (threads) that works in two phases. In Figure 2, hiword and loword are high and low w bits of a $2w$-bit product respectively, and gray cells represent updated words by s threads. This parallelization scheme is commonly used for hardware implementation.

In the first phase, we accumulate $s \times 1$ partial products in $2s$ steps (s steps for each loword and hiword), ignoring any carries. Carries are accumulated in a separate array through the processing. Each step is simple enough to be

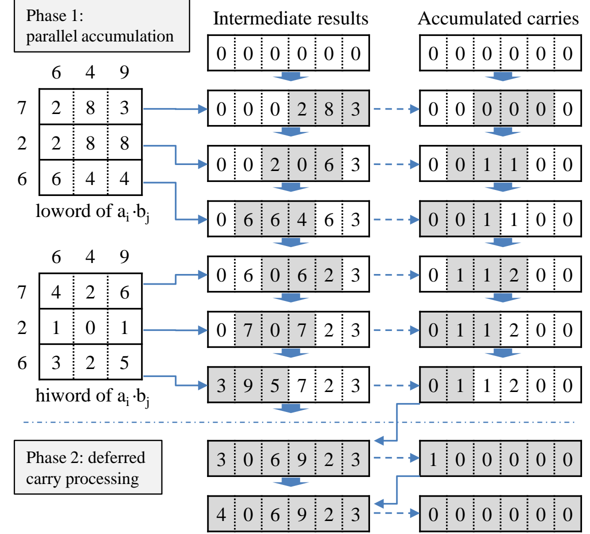


Figure 2: Parallel multiplication example of $649 \times 627 = 406,923$. For simplicity, a word holds a decimal digit rather than w-bit binary in the example.

translated into a small number of GPU instructions since it does not involve cascading carry propagation.

The second phase repeatedly adds the carries to the intermediate result and renews the carries. This phase stops when all carries become 0, which can be checked in one instruction with the `__any()` voting function in CUDA [48]. The number of iterations is $s-1$ in the worst case, but for most cases it takes one or two iterations since small carries (less than $2s$) rarely produce additional carries.

Our simple $O(s)$ algorithm is a significant improvement over the prior RSA implementation on GPUs. Harrison and Waldron parallelize $s \times s$ multiplications as follows [37]: Each of s threads independently performs $s \times 1$ multiplications in serial. Then s partial products are summed up in additive reduction in $\log n$ steps, each of which is done in serial as well. The resulting time complexity is $O(s \log s)$, and most of the threads are underutilized during the final reduction phase.

We also implemented RNS-based Montgomery multiplications. We adopt Kawamura's algorithm [40]. Even with extensive optimizations, the RNS implementation performs significantly slower than MP, and we use only the MP version in this paper. For future reference, we point out two main problems that we have encountered. First, CUDA does not support native integer division and modulo operations, on which the RNS Montgomery multiplication heavily depends. We have found that the performance of emulated operations is dependent on the size of a divisor and degrades significantly if the length of a divisor is longer than 14 bits. Second, since the number of threads is not a power of two, warps are not fully utilized and array index calculation becomes slow.

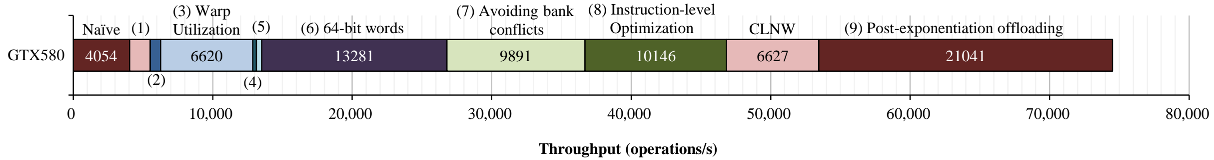


Figure 3: 1024-bit RSA performance with various optimization techniques. Sub-bars are placed in the same order as the techniques shown in Section 3.3.2, except for CLNW.

3.3.2 Optimizations

On top of CRT parallelization, CLNW, Montgomery reduction, modular exponentiation, and square-and-multiply optimization techniques, we conduct further optimizations as below. Figure 3 demonstrates how the overall throughput of the system increases as each optimization technique is applied. The *naïve* implementation includes CRT parallelization, basic implementation of Montgomery multiplication, and square-and-multiply modular exponentiation. For a 1024-bit ciphertext message with CRT, each of two 512-bit numbers (a ciphertext message) spans across 16 threads, each of which holds a 32-bit word, and those 16 threads are grouped as a CUDA block.

(1) Faster Calculation of $M \cdot n$: In Algorithm 1, the calculation of M and $M \cdot n$ requires two $s \times s$ multiplication operations. We reduce these into one $s \times 1$ and one $s \times s$ multiplication and interleave them in a single loop. This technique was originally introduced in [45], and we apply it for the parallel implementation.

(2) Interleaving $T + M \cdot n$: We interleave the calculation of $T + M \cdot n$ in a single multiplication loop. This optimization effectively reduces the overhead of loop construction and carry processing. This technique was used in the serial RSA implementation on a Digital Signal Processor (DSP) [34], and we parallelize it.

(3) Warp Utilization: In CUDA, a *warp* (a group of 32 threads in a CUDA block), is the basic unit of scheduling. Having only 16 threads in a block causes underutilization of warps, limiting the performance. We avoid this behavior by having blocks be responsible for multiple ciphertext messages, for full utilization of warps.

(4) Loop Unrolling: We unrolled the loop in Montgomery multiplication, by using the `#pragma unroll` feature supported in CUDA. Giving more optimization chances to the compiler is more beneficial than in CPU programming, due to the lack of out-of-order execution capability in GPU cores.

(5) Elimination of Divergency: Since threads in a warp execute the same instruction in lockstep, code-path divergency in a warp is expensive (all divergent paths must be taken in serial). For example, we minimize the divergency in our code by replacing `if` statements with flat arithmetic operations.

(6) Use of 64-bit Words: The native support for integer multiplication on GPUs, which is the basic building block of large integer arithmetic, has recently been added and is still evolving. GTX580 supports native single-cycle instructions that calculate hiword or loword of the product of two 32-bit integers.

Use of 64-bit words instead of 32-bit introduce a new trade-off on GPUs. While the multiplication of two 64-bit words takes four GPU cycles [48], it can halve the required number of threads and loop iterations depicted in Figure 2. We find that this optimization is highly effective when applied.

(7) Avoiding Bank Conflicts: The 64-bit access pattern to the intermediate results and carries in Figure 2 causes bank conflicts in shared memory between independent ciphertext messages in the same warp. We avoid this bank conflict by padding the arrays to adjust access pattern in shared memory.

(8) Instruction-Level Optimization: We have manually inspected and optimized the core code (about 10 lines) inside the multiplication loop, which is the most time-consuming part in our GPU code. We changed the code order at the CUDA C source level, until we got the desired assembly code. This includes the elimination of redundant instructions and pipeline stalls caused by Read-After-Write (RAW) register dependencies [47].

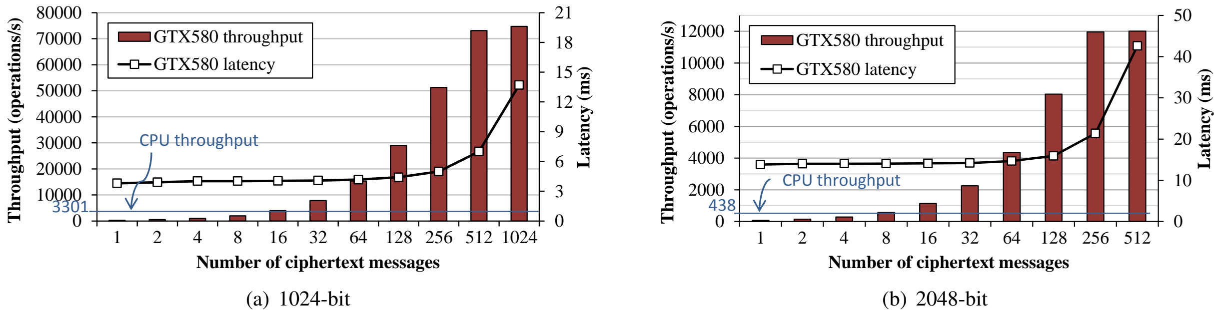
(9) Post-Exponentiation Offloading: Fusion of two partial modular exponentiation results from (4) is done on the CPU with the Mixed-Radix Conversion (MRC) algorithm as follows [27]:

$$M := M_2 + [(M_1 - M_2) \cdot (q^{-1} \bmod p)] \cdot q \qquad (6)$$

Although this processing is much lighter than modular exponentiation operations, the relative cost has become significant as we optimize the modular exponentiation process extensively. We have offloaded the above equation to the GPU, parallelizing at the message level. We also offload other miscellaneous processing in decryption such as integer-to-octet-string conversion and PKCS #1 depadding [38].

3.4 RSA Microbenchmarks

We compare our parallel RSA implementation to a serial CPU implementation. We use Intel Integrated Per-



(a) 1024-bit

(b) 2048-bit

Figure 4: RSA MP performance on a GTX580. A single core (Xeon X5650 2.66 GHz) is used for CPU performance.

	Processor	512	1024	2048	4096
Latency	CPU core	0.07	0.3	2.3	14.9
(ms)	GTX580, MP	1.1	3.8	13.83	52.46
Throughput	CPU core	13,924	3,301	438	67
(ops/s)	GTX580, MP	906	263	72	19
Peak	CPU core	13,924	3,301	438	67
(ops/s)	GTX580, MP	322,167	74,732	12,044	1,661

Table 1: RSA performance with various key sizes

formance Primitives (IPP) [8] as a CPU counterpart. IPP is the fastest implementation we have tried, outperforming other publicly available libraries for all key sizes. It performs 3,301 RSA decryption operations/s for a 1024-bit key on a 2.66 GHz CPU core. Since this number is higher than what Kounavis *et al.* recently report (2,990 operations/s on a 3.00 GHz CPU core) in [43], we believe its CPU reference implementation is a fair comparison to our GPU code.

Table 1 summarizes the performance of RSA on the CPU (a single 2.66 GHz core) and GTX580. With only one ciphertext message per launch, the GPU's performance shows an order of magnitude worse throughput (operations per second) and latency (the execution time). Given enough parallelism, however, the GPU produces much higher throughput than the CPU. The MP implementation on the GTX580 shows 23.1x, 22.6x, 27.5x, and 31.7x speedup compared with a single CPU core, for 512-bit, 1024-bit, 2048-bit, and 4096-bit RSA, respectively. The performance gains are comparable to what we expect from three hexa-core CPUs.

Figure 4 shows the correlation between latency and throughput of our MP implementation. The throughput improves as the number of concurrent messages grows, but reaches a plateau beyond 512 messages. The latency increases very slowly, but grows linearly with the number of messages beyond the point where the GPU is fully utilized. Even at peak throughput the latency stays below 7 to 13.7 ms for more than 70,000 operations/s for 1024-bit RSA decryption on a GTX580 card.

Many cipher algorithms, such as DSA [5], Diffie Hellman key exchange [33], and Elliptic Curve Cryptography (ECC) [42], depend on modular exponentiation as well as RSA. Our optimization techniques presented in Section 3 are applicable to those algorithms and can offer an efficient platform for their GPU implementation.

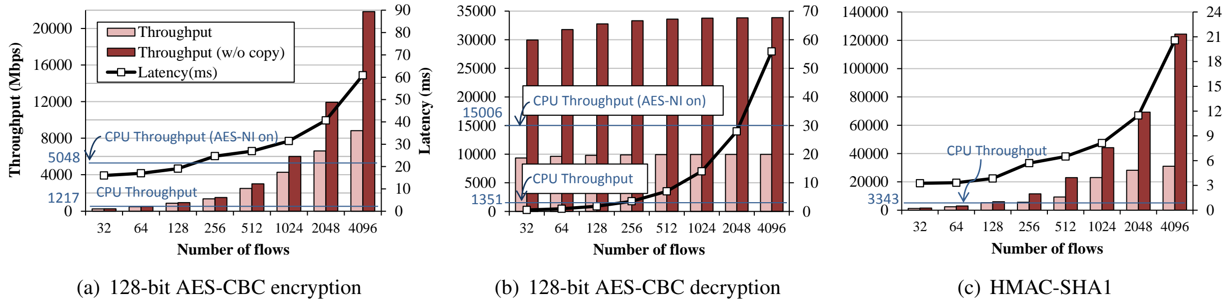
We summarize our RSA implementation on GPUs. First, the parallel RSA implementation on a GPU brings significant throughput improvement over a CPU. Second, we need many ciphertext messages in a batch for full utilization of GPUs with enough parallelism, in order to take a performance advantage over CPUs. In Section 5.4 we introduce the concept of *asynchronous concurrent execution*, which allows smaller batch sizes and thus shorter queueing and processing latency, while yielding even better throughput. Lastly, while the GPU implementation shows reasonable latency, it still imposes perceptible delay for SSL clients. This problem is addressed in Section 5.2 with *opportunistic offloading*, which exploits the CPU for low latency when under-loaded and offloads to the GPU for high throughput when a sufficient number of operations are pending.

4 Accelerating AES and HMAC-SHA1

4.1 GPU-accelerated AES

Since CBC mode encryption has a dependency on the previous block result, AES encryption in the same flow is serialized. On the other hand, decryption can be run concurrently as the previous block result is already known at decryption time. Therefore, AES-CBC decryption in a GPU runs much faster than AES-CBC encryption.

We have implemented AES for a GPU with the following optimizations. As shown in [36], on-chip shared memory offers two orders of magnitude faster access time than global memory on GPU. To exploit this fact, at the beginning of the AES cipher function, each thread copies part of the lookup table into shared memory. Additionally, we have chosen to derive the round key at each round, instead of using pre-expanded keys from global memory. Though it incurs more computation overhead, it avoids expensive global memory access and reduces the total latency.



(a) 128-bit AES-CBC encryption (b) 128-bit AES-CBC decryption (c) HMAC-SHA1

Figure 5: AES and HMAC-SHA1 performance on GTX580. A single core (Xeon X5650 2.66 GHz) is used for CPU performance.

4.2 AES-NI

Intel has recently added the AES instruction set (AES-NI) to the latest lineup of x86 processors. AES-NI runs one round of AES encryption or decryption with a single instruction (AESENC or AESDEC), and its performances for AES-GCM and AES-CTR are 2.5 to 6 times faster than a software implementation [7, 39]. AES-NI is especially useful for handling large files since data transfer overhead between host and device memory quickly becomes the bottleneck for GPU-accelerated AES. However, GPU-based symmetric cipher offloading still provides a benefit, if (i) CPUs do not support AES-NI, (ii) CPUs become the bottleneck handling the network stack and other server code, or (iii) other cipher functions (such as RC4 and 3DES) are needed.

4.3 GPU-accelerated HMAC-SHA1

The performance of HMAC-SHA1 depends on SHA1. Thus, we focus on the SHA1 algorithm. SHA1 takes 512 bits at each round and generates a 20-byte digest. Each round uses the previous round's result; thus SHA1 can not be run in parallel within a single message. Our SHA1 optimization in a GPU is divided into two parts: (i) reducing memory access by processing data in the register, and (ii) reducing required memory footprint to fit in the GPU registers.

Each round of SHA-1 is divided into four different steps, and at each step it processes 20 32-bit words; in total, 80 intermediate 32-bit values are used. A typical CPU implementation pre-calculates all 80 words before processing, which requires a 320-byte buffer. However, the algorithm only depends on the previous 16 words at any time. We calculate each intermediate data on demand, thus reducing the memory requirement to 64 bytes, which fits into the registers.[5]

To avoid global memory allocation, we unroll all loops and hardcode the buffer access with constant indices. This way the compiler register-allocates all the necessary 16 words. With this approach we see about 100% performance improvement over the naïve implementation.

[5]The idea to reduce the memory footprint is from a Web post in an NVIDIA forum: `http://forums.nvidia.com/index.php?showtopic=102349`

4.4 Microbenchmarks

Figure 5 compares the throughput and latency results of AES and HMAC-SHA1 with one GTX580 card and one 2.66 GHz CPU core. For the CPU implementations, we use Intel IPP, which shows the best performance of AES and SHA-1 as of writing this paper. We fix the flow length to 16 KB, the largest record size in SSL, and vary the number of flows from 32 to 4,096. Our AES-CBC implementation shows the peak performance of 8.8 Gbps and 10.0 Gbps for encryption and decryption respectively when we consider the data transfer cost, but the numbers go up to 21.9 Gbps and 33.9 Gbps without the copy cost. AES-NI shows 5 Gbps and 15 Gbps even with a single CPU core and thus one or two cores would exceed our GPU performance. Our GPU version matches 6.5 and 7.4 CPU cores without AES-NI support for encryption and decryption. For HMAC-SHA1, our GPU implementation shows 31 Gbps with the data transfer cost and 124 Gbps without, and matches the performance of 9.4 CPU cores.

Our findings are summarized as follows. (i) AES-NI shows the best performance per dollar, (ii) the data transfer cost in GPU reduces the performance by a factor of 3.39 and 4 in AES and HMAC-SHA1, and (iii) the GPU helps in offloading HMAC-SHA1 and AES workloads when CPUs do not support AES-NI. Since a recent hardware trend shows that the GPU cores are being integrated into the CPU [16, 54], we expect the impact of the data transfer overhead will decrease in the near future.

5 SSLShader

We build a scalable SSL reverse proxy, SSLShader, to incorporate the high-performance cryptographic operations using a GPU into SSL processing. Though proxying generally incurs redundant I/O and data copy overheads, we choose transparent proxing because it provides the SSL acceleration benefit to existing TCP-based

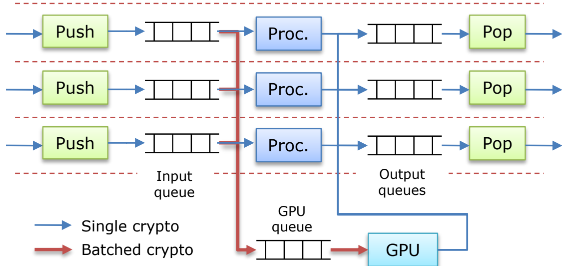


Figure 6: Overview of SSLShader

servers without code modification. SSLShader interacts directly with the SSL clients while communicating with the back-end server in plaintext. We assume that the SSLShader-to-server communication takes place in a secure environment, but one can encrypt the back-end traffic with a shared symmetric key in other cases.

The design goal of SSLShader is twofold. First, the performance should scale well to the number of CPU and GPU cores. Second, SSLShader should curb the latency to support interactive environments while improving the throughput at high load. In this section we outline the key design features of SSLShader.

5.1 Basic Design

Figure 6 depicts the overall architecture of SSLShader. SSLShader is implemented in event-driven threads. To scale with multi-core CPUs, it spawns one worker thread per CPU core and each thread accepts and processes client-side SSL connections independently. Each connection is accepted and processed by the same thread to avoid cache bouncing between CPU cores. SSLShader also creates one GPU-interfacing thread per GPU that launches GPU kernel functions to offload cryptographic operations.

Each cryptographic operation type (RSA, AES, HMAC-SHA1) has its own request input queue per worker thread. Cryptographic operations of the same type are inserted into the same queue, and are moved to a queue in the GPU-interfacing thread when the input queue size exceeds a certain threshold value. GPU-interfacing threads simply offload the requested operations in a batch by launching GPU kernels. The results are placed back on a per-worker thread output queue so that the worker thread can resume the execution of the next step in SSL processing.

5.2 Opportunistic Offloading

In order to fully exploit the parallelism, we should batch multiple cryptographic operations and offload them to the GPU. On the GTX580, the peak 1024-bit RSA performance is achieved when batching 256-512 operations, that is, handling 256-512 concurrent SSL connections.

Cryptographic operation	Minimum	Maximum
RSA (1024-bit)	16	512
AES128-CBC Decryption	32 (2,048)	2,048
AES128-CBC Encryption	128 (2,048)	2,048
HMAC-SHA1	128	2,048

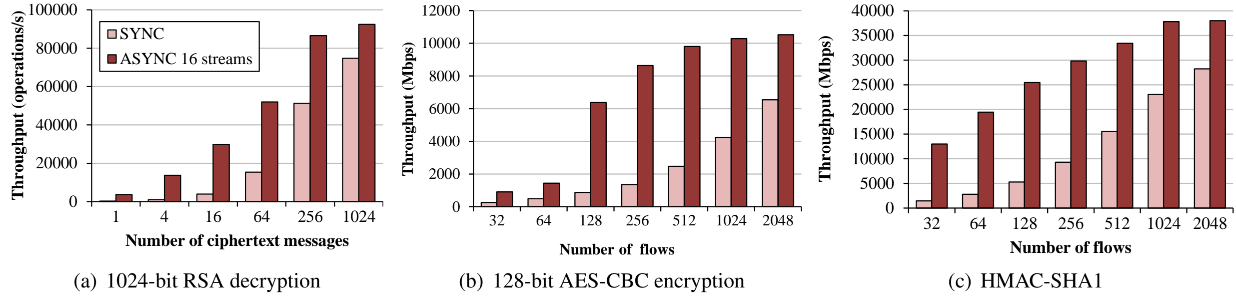
Table 2: Thresholds for GPU cryptographic operations per single kernel launch. Numbers in parenthesis are thresholds when AES-NI is used.

While batching generally improves the GPU throughput, a naïve approach of batching a fixed number of operations would increase processing latency when the load level is low.

We propose a simple GPU offloading algorithm that reduces response latency when lightly loaded and improves the overall throughput at high load. When a worker thread inserts a cryptographic request to an input queue, it first checks the number of the same type of requests in all workers' queues, and its minimum batching threshold (the number of queued requests required for GPU offloading). If the number of requests is above the threshold, SSLShader moves all the requests in the worker thread queue to a GPU-interfacing thread queue. The batching thresholds are determined based on the GPU's throughput. The minimum threshold is set when the GPU performs better than a single CPU core, and the maximum is set when the maximum throughput is achieved. We limit maximum batch size since pushing too many requests into a queue in the GPU-interfacing thread could result in long delay without throughput improvement. The thresholds can be drawn automatically from benchmark tests at configuration time. For AES, thresholds are different when AES-NI is enabled. If AES-NI is available, we set the minimum threshold to be the same as the maximum threshold, hoping to benefit from extra processing power only when the CPU is overloaded. Table 2 shows the thresholds we use with the GTX580.

For low latency and high parallelism, the worker thread prioritizes I/O events, and processes cryptographic operations when it has no I/O event. Worker threads handle cryptographic operations in the first-in first-out (FIFO) manner. We put a timestamp on each cryptographic request as it arrives, and use the timestamp to find the earliest arrived operation. The GPU also uses FIFO scheduling for processing cryptographic operations. The GPU-interfacing thread looks at the head timestamp of requests by the type, and processes the earliest request's type in a batch. Sometimes it takes too long for the worker thread to drain the cryptographic operations in its queue and this can lead to I/O starvation. To prevent this, we have worker threads periodically check for I/O events while processing cryptographic operations.

We also tested priority-based scheduling by having the



(a) 1024-bit RSA decryption (b) 128-bit AES-CBC encryption (c) HMAC-SHA1

Figure 7: Performance improvement from asynchronous concurrent execution with 16 streams, independent CUDA contexts of commands that execute in order asynchronously. Each flow size is 16KB for (b) and (c).

CPU prioritize HMAC-SHA1 and AES encryption, and the GPU prioritize RSA and AES decryption. This strategy often improves the peak throughput, but we reject this idea because lower-priority cryptographic operations could suffer from starvation, and we noticed unstable throughput and longer latency in many cases.

5.3 NUMA-aware GPU Sharing

NUMA systems are becoming commonplace in server machines. In NUMA systems, the communication cost between CPU cores varies greatly, depending on the number of NUMA hops. For high scalability, it is necessary to reduce inter-core communication by careful NUMA-aware data placement.

When we use a GPU, we should consider the following issues: (i) GPUs are known to perform badly when used by multiple threads simultaneously due to high context switching overhead [48]; (ii) gathering cryptographic operations from multiple cores brings more parallelism and helps to exploit the full GPU capacity; and (iii) memory access or synchronization across NUMA nodes is much more expensive than intra-NUMA node operation. For these reasons, we limit the sharing of GPUs to the threads in the same NUMA node.

For intra-NUMA node communication we choose threads over processes for faster sharing of the queues as offloading cryptographic operations requires data movement between worker and GPU-interfacing threads. For inter-NUMA node communication, we choose processes for ease of connection handling without kernel lock contention at socket creation and closure.

5.4 Asynchronous Concurrent Execution

The most recent CUDA device with Compute Capability 2.0 provides concurrent GPU kernel execution and data copy for better utilization of the GPU. On the GTX580, up to sixteen different kernels can run simultaneously within a single GPU, and copies from device to host and host to device can overlap with each other as well as with kernel execution To benefit from concurrent execution and copy, SSLShader launches all GPU transactions asynchronously. With asynchronous concurrent execution, we see up to $1,399\%$, 731%, and 890% performance improvements over synchronous execution in RSA, AES encryption and HMAC-SHA1, respectively. Figure 7 depicts the effect of asynchronous concurrent execution by varying the batch size. When the batch size is small, asynchronous concurrent execution improves performance greatly as idle GPU cores can be better utilized. But even for a large batch size such as $2,048$, we see $30 \sim 60\%$ performance improvement in HMAC-SHA1 and AES. The overlap of DMA data copy and kernel execution improves the performance even when all cores in the GPU are already utilized. In RSA, the performance improvement in the batch size of 1024 is fairly small compared to those of AES or HMAC-SHA1 because the data copy time in RSA is relatively smaller than the execution time and the GPU is sufficiently utilized with large batch sizes.

We believe our design and implementation strategies in this section are not limited to only SSLShader, and can be applied to any applications that want to exploit the massive parallelism of GPUs. While none of the techniques are ground-breaking, their combination brings a drastic difference in the utilization of GPUs, latency reduction, and throughput improvement.

6 Evaluation

In this section we evaluate the effectiveness of SSLShader using HTTPS, the most popular protocol that uses SSL. We show that SSLShader achieves high performance in connection handling and large-file data transfer with small latency overhead.

6.1 System Configuration

Our server platform is equipped with two Intel Xeon X5650 (hexa-core 2.66 GHz) CPUs, 24 GB memory, and two NVIDIA GTX580 cards (512 cores, 1.5 GHz,

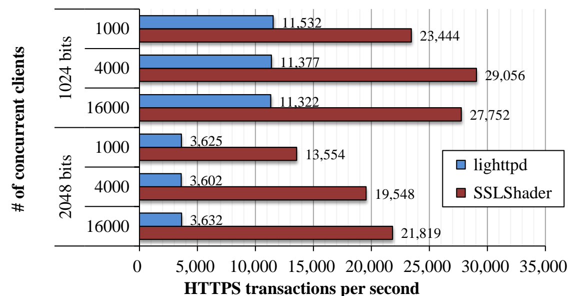


Figure 8: Transactions per second

Image Name	CPU Usage (%)
Kernel NIC device driver	2.32
Kernel (including TCP/IP stack)	60.35
SSLShader	5.31
libc (memory copy and others)	9.88
IPP + libcrypto (cryptographic operations)	12.89
lighttpd (back-end web server)	4.90
others	4.35

Table 3: CPU usage breakdown using oprofile

1.5 GB RAM). We install Ubuntu Linux 10.04, NVIDIA CUDA Driver v256.40, and Intel ixgbe[6] driver v2.1.4 on the server. As back-end web server software, we run lighttpd[7] v1.4.28 with 12 worker processes to match the number of CPU cores. In all experiments, we run lighttpd and SSLShader on the same machine.

We compare SSLShader against lighttpd with OpenSSL. For fair comparison, we spent a fair amount of time to patch OpenSSL 1.0.0 to use IPP 7.0 which has AES-NI support as well as the latest RSA and SHA-1 optimizations. We find that IPP 7.0 improves the RSA, AES, and HMAC performance by 55%, 10%, and 22% respectively from the OpenSSL 1.0.0 default implementation. As our goal is to offload SSL computation overhead, we focus on static content to prevent the back-end web server from becoming a bottleneck. To generate HTTP requests, we run the Apache HTTP server benchmark tool (ab) [1] on seven 2.66 GHz Intel Nehalem quad-core client machines. We modified ab to support rate-limiting and to report latency for each connection.

6.2 SSL Handshake Performance

To evaluate the performance of connection handshake, we measure the number of SSL transactions per second (TPS) for a small HTTP object (43 bytes including HTTP headers). Figure 8 shows the maximum TPS achieved by varying the number of concurrent connections. For 1024-bit RSA keys, SSLShader achieves 29K TPS, which is 2.5 times faster than 11.2K TPS for lighttpd with OpenSSL. SSLShader achieves 21.8K TPS, for 2048-bit RSA, which is 6 times higher than 3.6K TPS of lighttpd. Given that 768-bit RSA was cracked early in 2010 [12] and that NIST recommends 2048-bit RSA for secure operations as of 2011 [46], the large performance improvement with 2048-bit RSA is significant. In SSLShader, the throughput increases as the concurrency increases because the GPU can exploit more parallelism. In 2048-bit RSA, 21.8K is close to the peak throughput of 24.1K msg/s with two GTX580s, meaning that the GPUs are almost fully utilized. However, the performance of RSA 1024-bit is much less than the peak throughput of a single GPU, which implies that the GPUs have idle computing capacity.

We run oprofile to analyze the bottleneck for the RSA 1024-bit case with 16,000 concurrent clients. Table 3 summarizes where the CPU cycles are spent. We see that more than 60% of CPU time is spent in the kernel for accepting connections and networking I/O; 13% of the CPU cycles are spent for cryptographic operations, mostly for the Pseudo Random Function (PRF) used for generating session keys from the master secret in the handshake step. We chose not to offload PRF to GPUs because it is run only once in the handshake step and its computation overhead is less than 1/10th of the RSA decryption overhead. We conclude that the performance bottleneck is in the Linux kernel that does not scale to multi-core CPUs for connection acceptance, as is also observed in [57].

6.3 Response Time Distribution

Naïvely using a GPU for cryptographic operations could lead to high latency when the load level is low. Opportunistic offloading guards against this problem, minimizing the latency when the load is light and maximizing the throughput when the load is high. To evaluate the effectiveness of our opportunistic offloading algorithm, we measure the response time for both heavy and light load cases. We control the load by rate-limiting the clients. For lighttpd, we set the limits to 1K TPS for light load and 11K TPS for heavy load. For SSLShader, we further increase the heavy load limit to 29K TPS. For heavy load experiments, we vary the maximum number of clients from 1K to 4K. Clients repeatedly request the small HTTP objects as in the handshake experiment.

Figure 9 shows the cumulative distribution functions (CDFs) of response times. When the load level is low, both lighttpd and SSLShader handle most of the connections in a few milliseconds (ms), which shows that the opportunistic offloading algorithm intentionally uses the CPU to benefit from its low latency. SSLShader shows a slight increase in response time (2 ms vs. 3 ms on median) due to the proxying overhead. At heavy load

[6] http://sourceforge.net/projects/e1000/files/ixgbe%20stable/
[7] http://www.lighttpd.net/

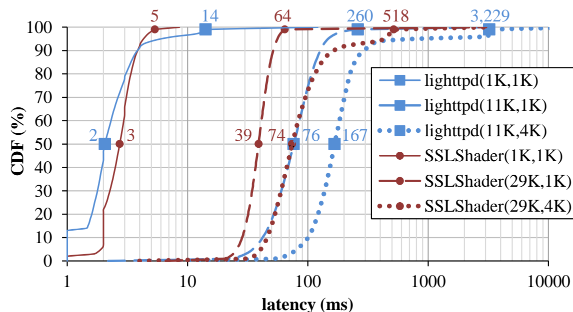


Figure 9: Latency distribution in the overloaded case. Numbers in parenthesis represent the maximum requests rate and the maximum concurrency.

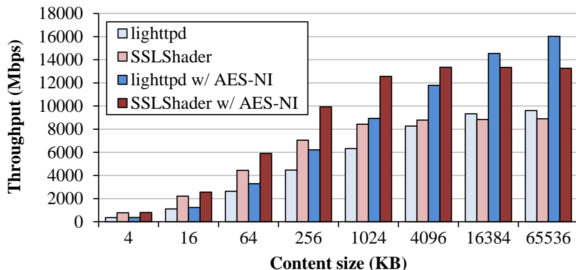


Figure 10: Bulk transfer throughput

with 1K concurrent connections, SSLShader's latency is lower than that of `lighttpd` because CPUs are overloaded and `lighttpd` produces longer response times. In contrast, SSLShader reduces the CPU overhead by offloading the majority of cryptographic operations to GPUs. SSLShader shows 39 ms and 64 ms for 50th and 99th percentiles while `lighttpd` shows 76 ms and 260 ms each even at the much lower TPS load level. Even if we increase the load with 4K concurrent clients, 70% of SSLShader response times remain similar to those of `lighttpd` with 1K clients at the 11K TPS level.

6.4 Bulk Data Encryption Performance

We measure bulk data encryption performance by varying the file size from 4 KB to 64 MB with and without AES-NI support, and show the results in Figure 10. When AES-NI is enabled, the SSLShader throughput peaks at 13 Gbps while `lighttpd` peaks at 16.0 Gbps. We note that increasing the content size above 64 MB does not increase `lighttpd`'s throughput. For contents smaller than 4 MB, SSLShader performs 1.3 to 2.2x better than `lighttpd` while `lighttpd` shows 1.1x to 1.2x better performance for contents larger than 4 MB. As the content size grows and throughput increases, the proxying overhead increases accordingly, and eventually becomes the performance bottleneck. With oprofile, we find that 30% of CPU time is spent on data copying, and 20% is spent on handling interrupts for SSLShader, leaving only 50% for use in cryptographic operation and other processing. Without AES-NI, SSLShader achieves 8.9 Gbps, while `lighttpd` achieves 9.6 Gbps. The peak throughput of SSLShader is slightly lower due to the copying overhead as well.

Considering typical Web objects and email contents are smaller than 100 KB [21, 23], we believe that the performance gain in small content size and the benefit of transparent proxying outweigh the small performance loss in large files in many real-world scenarios. Also, the GPU is starting to be integrated into the CPU as in AMD's Fusion [16], and we expect that such technology will mitigate the performance problem by eliminating the data transfer between GPU and CPU.

7 Discussion & Related Work

SSL Performance: SSL performance analysis and acceleration have drawn much attention in the context of secure Web server performance. Earlier, Apostolopoulos *et al.* analyzed the SSL performance of Web servers using the SpecWeb96 benchmark tool [22]. They observe that the SSL-enabled Web servers are up to two orders of magnitude slower than plaintext Web servers. For small HTTP transactions, the main bottleneck lies in SSL handshaking while data encryption takes up significant CPU cycles when the content gets larger. Later, Coarfa *et al.* reported similar results and estimated the upper bound in the performance improvement with each SSL operation optimization [29]. To improve the SSL handshake performance, Boneh *et al.* proposed client-side caching of server certificates to reduce the SSL round-trip overhead [25]. Another approach is to process multiple RSA decryptions in a batch using Fiat's batch RSA [55]. They report a 2.5x speedup on their Web server experiments by batching four RSA operations.

Recently, Kounavis *et al.* improve the SSL performance with general-purpose CPUs [43]. They optimize the schoolbook big number multiplication and benefit from AES-NI for symmetric cipher. To reduce the CPU overhead for MAC algorithms, they use the Galois Counter Mode (GCM) which combines the AES encryption with the authentication. In comparison, we argue that GPUs bring extra computing power in a cost-effective manner, especially for RSA and HMAC-SHA1. By parallelizing the schoolbook multiplication and various optimizations, our 1024-bit RSA implementation on a GPU shows 30x improvement over their 3.0 GHz CPU core. Further, we focus on TLS 1.0 which is widely used in practice, whereas GCM is only supported in TLS 1.2, which is not popular yet.

AES Implementations on GPU: Modern GPUs are attractive for computation-intensive AES operations [30, 31, 36, 44, 49, 58]. Most GPU-based implementations ex-

ploit shared memory and on-demand round key calculation to reduce the global memory access. However, we find few references that evaluate the AES performance in the CBC mode. Unlike electronic codebook (ECB) mode or counter (CTR) mode, the CBC mode is hard to parallelize but is most widely used. Also, most of them report the numbers without data copy overhead, but we find the copy overhead severely impairs the AES performance.

Manavski *et al.* report 8.28 Gbps AES performance on GTX 280 (240 cores, 1.296 GHz) [44], while Osvik *et al.* report 30.9 Gbps on half of a GTX 295 (2 x 240 cores, 1.242 GHz) [49]. Both of them use the ECB mode without data copy overhead. In the same setting, our implementation shows 32.8 Gbps on GTX 285 (240 cores, 1.476 GHz). Direct comparison is hard due to different GPUs, but our number is comparable to these results (3.48x that of Manavski's, 0.89x that of Osvik's) by the cycles per byte metric.

RSA Implementations on GPU: Szerwinski and Güneysu made the first implementation of RSA on the general-purpose GPU computation framework [56]. They reported two orders of magnitude lower performance than ours, but it should not be directly compared because they used a relatively old NVIDIA 8800GTS card with a different GPU architecture.

Harrison and Waldron report on 1024-bit key RSA implementation on an NVIDIA GPU [37], and to the best of our knowledge theirs is the fastest implementation before our work. They compare serial and MP parallel approaches in Montgomery multiplication and conclude that the parallel implementation shows worse performance at scale due to GPU thread synchronization overhead. We have run their serial code on an NVIDIA GTX580 card, and found that their peak throughput reaches $31,220$ operations/s at a latency of 131 ms with $4,096$ messages per batch. Our throughput on the same card shows $74,733$ operations/s at a latency of 13.7 ms with 512 messages per batch, 2.3x improvement in throughput and 9.6x latency reduction.

Comparison with H/W Acceleration Cards: Many SSL cards support OpenSSL engine API [11] so that their hardware can easily accelerate existing software. Current hardware accelerators support 7K to 200K RSA operations/s with 1024-bit keys [10, 14]. Our GPU implementation is comparable with these high-end hardware accelerators, running at up to 92K RSA operations/s at much lower cost. Moreover, GPUs are flexible for adoption of new cryptographic algorithms.

Other Protocols for Secure Communication: Bittau *et al.* propose *tcpcrypt* as an extension of TCP for secure communication [24]. *Tcpcrypt* is essentially a clean-slate redesign of SSL that shifts the decryption overhead by private key to clients and that allows a range of authentication mechanisms. Their evaluation reports 25x better connection handling performance when compared with SSL. Moreover, *tcpcrypt* provides forward secrecy by default while SSL leaves that as an option. While fixing the SSL protocol is desirable, we focus on improving the current practice of SSL in this work. IPsec [17] provides secure communication at the IP layer, which is widely used for VPN. IPsec can be more easily parallelized compared to SSL since many packets can be processed in parallel [35].

Performance per $ Comparison: In Table 4, we show the price and relative performance to price for two CPUs, GTX580, and a popular SSL accelerator card. Intel Xeon X5650 and GTX580 are choices for our experiments. i7 920 has four CPU cores with the same clock speed as the X5650 without AES-NI support. We choose the CN1620[8] because it is one of the most cost-effective accelerators that we have found. Though it is difficult to compare the performance per dollar directly (e.g., GPU cannot be used without CPU), we present the information here to get the sense of cost effectiveness for each hardware.

	Price ($)	RSA (ops/sec/$)	AES-ENC (Mbps/$)	AES-DEC (Mbps/$)	SHA1 (Mbps/$)
X5650	996	19.9	30.6	92.2	20.2
i7 920	288	45.8	18.9	19.0	46.5
GTX580	499	185.3	21.3	25.1	62.3
CN1620	2,129	30.5	2.8	2.8	2.8

Table 4: Performance per $ comparison (price as of Feb. 2011)

GTX580 shows the best performance per dollar for RSA and HMAC-SHA1. For AES operations, X5650 is the best with its AES-NI capability, and GTX580 shows a slightly better number compared to i7 920. CN1620 is inefficient in terms of performance per dollar for all operations. SSL accelerators typically have much better power efficiency compared to general purpose processors and it is mainly used in high-end network equipment rather than on server machines.

8 Conclusions

We have enjoyed the security of SSL for over a decade and it is high time that we used it for every private Internet communication. As a cheap way to scale the performance of SSL, we propose using graphics cards as high-performance SSL accelerators. We have presented a number of novel techniques to accelerate the cryptographic operations on GPUs. On top of that, we have built SSLShader, an SSL reverse proxy, that opportunistically offloads cryptographic operations to GPUs and achieves high throughput and low response latency.

[8] Model name is CN1620-400-NHB4-E-G and more details are on `http://www.scantec-shop.com/cn1620-400-nhb4-e-g-375.html`

Our evaluation shows that we can scale 1024-bit RSA up to 92K operations/s with a single GPU card by careful workload pipelining. SSLShader handles 29K SSL TPS and achieves 13 Gbps bulk encryption throughput on commodity hardware. We hope our work pushes SSL to a wider adoption than today.

We report that inefficiency in the Linux TCP/IP stack is keeping performance lower than what SSLShader can potentially offer. Most of the inefficiency in the Linux TCP/IP stack comes from mangled flow affinity and serialization problems in multi-core systems. We leave these issues to future work.

9 Acknowledgment

We thank Geoff Voelker, anonymous reviewers, and our shepherd David Mazières for their help and invaluable comments. This research was funded by NAP of Korea Research Council of Fundamental Science & Technology, MKE (Ministry of Knowledge Economy of Repbulic of Korea, project no. 10035231-2010-01), KAIST ICC, and KAIST High Risk High Return Project (HRHRP).

References

[1] ab - Apache HTTP Server Benchmarking Tool. http://httpd.apache.org/docs/2.2/en/programs/ab.html.
[2] Alexa Top 500 Global Sites. http://www.alexa.com/topsites.
[3] Application Delivery Controllers, Array Networks. http://www.arraynetworks.net/?pageid=365.
[4] Content Services Switches, Cisco. http://www.cisco.com/web/go/css11500.
[5] Digital Signature Standard. http://csrc.nist.gov/fips.
[6] F5 BIG-IP SSL Accelerator. http://www.f5.com/products/big-ip/feature-modules/ssl-acceleration.html.
[7] Intel Advanced Encryption Standard Instructions (AES-NI). http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni/.
[8] Intel Integrated Performance Primitives. http://software.intel.com/en-us/intel-ipp/.
[9] nFast Series, Thales. http://iss.thalesgroup.com/Products/.
[10] NITROX security processor, Cavium Networks. http://www.caviumnetworks.com/processor_security_nitrox-III.html.
[11] OpenSSL Engine. http://www.openssl.org/docs/crypto/engine.html.
[12] Researchers crack 768-bit RSA. http://www.bit-tech.net/news/bits/2010/01/13/researchers-crack-768-bit-rsa/1.
[13] ServerIron ADX Series, Brocade. http://www.brocade.com/products-solutions/products/application-delivery/serveriron-adx-series/index.page.
[14] Silicom Protocol Processor Adapter. http://www.silicom-usa.com/default.asp?contentID=676.
[15] SSL Acceleration Cards, CAI Networks. http://cainetworks.com/products/ssl/rsa7000.htm.
[16] The AMD Fusion Family of APUs. http://sites.amd.com/us/fusion/APU/Pages/fusion.aspx.
[17] Security Architecture for the Internet Protocol. RFC 4301, 2005.
[18] Netcraft SSL Survey. http://news.netcraft.com/SSL-survey, 2009.
[19] Netcraft Web Server Survey. http://news.netcraft.com/archives/2010/04/15/april_2010_web_server_survey.html, 2009.
[20] NVIDIA's Next Generation CUDA™Compute Architecture: Fermi™. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009.
[21] S. Agarwal, V. N. Padmanabhan, and D. A. Joseph. Addressing email loss with suremail: Measurement, design, and evaluation. In *USENIX ATC*, 2007.
[22] G. Apostolopoulos, V. Peris, and D. Saha. Transport Layer Security: How much does it really cost? In *IEEE Infocom*, 1999.
[23] A. Badam, K. Park, V. Pai, and L. Peterson. Hashcache: Cache storage for the next billion. In *NSDI*, 2009.
[24] A. Bittau, M. Hamburg, M. Handley, D. Mazières, and D. Boneh. The case for ubiquitous transport-level encryption. In *USENIX Security Symposium*, 2010.
[25] D. Boneh, H. Shacham, and E. Rescrola. Client side caching for TLS. In *Network and Distributed System Security Symposium (NDSS)*, 2002.
[26] J. Bos and M. Coster. Addition chain heuristics. In *Advances in Cryptology (CRYPTO)*, 1989.
[27] Ç. K. Koç. High-speed RSA implementation. Technical Report, 1994.
[28] Ç. K. Koç. Analysis of sliding window techniques for exponentiation. *Computer and Mathematics with Applications*, 30(10):17–24, 1995.
[29] C. Coarfa, P. Druschel, and D. S. Wallach. Performance Analysis of TLS Web Servers. In *Network and Distributed System Security Symposium (NDSS)*, 2002.
[30] D. L. Cook, J. Ioannidis, A. D. Keromytis1, and J. Luck. CryptoGraphics: Secret Key Cryptography Using Graphics Cards . In *RSA Conference, Cryptographers Track (CT-RSA)*, 2005.
[31] N. Costigan and M. Scott. Accelerating SSL using the Vector processors in IBMs Cell Broadband Engine for Sonys Playstation 3. In *Cryptology ePrint Archive, Report*, 2007.
[32] J. Daemen and V. Rijmen. AES Proposal: Rijndael. http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf, 1999.
[33] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.
[34] S. Dussé and B. Kaliski. A cryptographic library for the Motorola DSP56000. In *Advances in Cryptology—EUROCRYPT 1990*.
[35] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. In *ACM SIGCOMM*, 2010.
[36] O. Harrison and J. Waldron. Practical Symmetric Key Cryptography on Modern Graphics Hardware. In *USENIX Security Symposium*, 2008.
[37] O. Harrison and J. Waldron. Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware. In *International Conference on Cryptology in Africa*, 2009.
[38] J. Jonsson and B. Kaliski. Public-key cryptography standards (PKCS) #1: RSA cryptography specifications version 2.1, 2003.
[39] E. Kasper and P. Schwabe. Faster and timing-attack resistant aes-gcm. In *Cryptographic Hardware and Embedded Systems (CHES)*. 2009.
[40] S. Kawamura, M. Koike, F. Sano, and A. Shimbo. Cox-rower architecture for fast parallel montgomery multiplication. In *Advances in Cryptology—EUROCRYPT 2000*, pages 523–538. Springer, 2000.
[41] D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 3th edition, 1997.
[42] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.
[43] M. E. Kounavis, X. Kang, K. Grewal, M. Eszenyi, S. Gueron, and D. Durham. Encrypting the internet. *SIGCOMM Comput. Commun. Rev.*, 40(4):135–146, 2010.
[44] S. A. Manavski. CUDA compatible gpu as an efficient hardware accelerator for aes cryptography.
[45] P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.
[46] National Institute of Standards and Technology (NIST). *Recommendation for Key Management Part 1: General (Revised)*. 2007.
[47] NVIDIA Corp. *NVIDIA CUDA: Best Practices Guide, Version 3.1*. 2010.
[48] NVIDIA Corp. *NVIDIA CUDA: Programming Guide, Version 3.1*. 2010.
[49] D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright. Fast software aes encryption. In *Foundations of Software Engineering (FSE)*, 2010.
[50] H. Park, K. Park, and Y. Cho. Analysis of the variable length nonzero window method for exponentiation. *Computers & Mathematics with Applications*, 37(7):21–29, 1999.
[51] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18(21):905–907, 1982.
[52] E. Rescorla, A. Cain, and B. Korver. SSLACC: A Clustered SSL Accelerator. In *USENIX Security Symposium*, 2002.
[53] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
[54] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics (TOG)*, 27(3):1–15, 2008.
[55] H. Shacham and D. Boneh. Improving SSL Handshake Performance via Batching. In *RSA Conference*, 2001.
[56] R. Szerwinski and T. Gneysu. Exploiting the Power of GPUs for Asymmetric Cryptography. In *International Workshop on Cryptographic Hardware and Embedded Systems*, 2008.
[57] S. B. Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *USENIX OSDI*, 2008.
[58] J. Yang and J. Goodman. Symmetric Key Cryptography on Modern Graphics Hardware. In *ASIACRYPT*, 2007.