White Paper

Vinodh Gopal
Jim Guilford
Erdinc Ozturk
Sean Gulley
Wajdi Feghali

IA Architects
Intel Corporation

Improving OpenSSL* Performance

October 2011

326232-001

Executive Summary

The SSL/TLS protocols involve two compute-intensive phases: session initiation and bulk data transfer.

Intel recently developed highly optimized implementations of cryptographic functions on IA. Intel worked with OpenSSL* to integrate these implementations, starting with OpenSSL 1.0.1.

This novel RSA implementation improves the OpenSSL performance for session initiation, and our stitched cryptographic algorithm implementations improve the performance of the bulk data transfer.

We measured the performance gains of OpenSSL on a dual Intel® Xeon® Processor X5680 system, that is OpenSSL performance using the built-in speed test, as well as at the system-level running an Apache web server sending HTTP over SSL to clients on a network.

Intel developed and released highly optimized cryptographic functions into OpenSSL*, and measured the performance on a dual Intel® Xeon® Processor X5680 system running the Apache Web-Server application, sending HTTP over SSL to clients on a network. This system, running the improved version of OpenSSL, was capable of ~28 Gigabits/second of large secure connections using AES256-SHA1 with RSA1024 for session setup. This version of OpenSSL was ~4.8X faster than the latest default version at the system level.

contemplated purchases, including the performance of that product when combined with other products.

Configurations: See Performance *section for detailed configurations. 2 tests used: speed test (which is part of theOpenSSL application) and a system level test (described in* System Configuration and Experimental Setup *on page 9). All testing was performed by Intel Corporation. For more information go to http://www.intel.com/performance.*

The Intel® Embedded Design Center provides qualified developers with web-based access to technical resources. Access Intel Confidential design materials, step-by step guidance, application reference solutions, training, Intel’s tool loaner program, and connect with an e-help desk and the embedded community. Design Fast. Design Smart. Get started today. www.intel.com/embedded/edc.

Contents

Overview

Intel recently developed highly optimized implementations of cryptographic functions on Intel® Architecture (IA). Intel worked with OpenSSL* to refine and integrate these implementations starting with OpenSSL 1.0.1, as well as in the libintel-accel plug-in engine for older OpenSSL versions [3].

The OpenSSL project [1] provides an open source implementation of the SSL/TLS [2] protocols, and is a commonly deployed library for SSL/TLS world-wide. The SSL/TLS protocols consist of two phases: an initial session-initiation/handshake phase, and a bulk data transfer phase.

Intel's cryptographic optimizations speed up the handshake and the bulk data transfer phases of OpenSSL. This paper describes the functions developed in Intel's cryptographic optimizations, and presents the performance gains measured on a dual Intel® Xeon® Processor X5680 [15] system, which consists of two 6-core Intel® processors based on the 32-nm microarchitecture, supporting the Intel® AES New Instructions (Intel® AES-NI) extension. We measured the performance gains for OpenSSL using the built-in speed test, as well as at the system-level running an Apache web server sending HTTP over SSL to clients on a network.

Introduction to OpenSSL, Apache

SSL/TLS

TLS (Transport Layer Security) [2] and its predecessor, SSL (Secure Sockets Layer), are cryptographic protocols that are used to provide security for communication over networks such as the internet. These protocols are widely used for applications such as secure web browsing (HTTPS), electronic mail, instant messaging, and voice-over-IP.

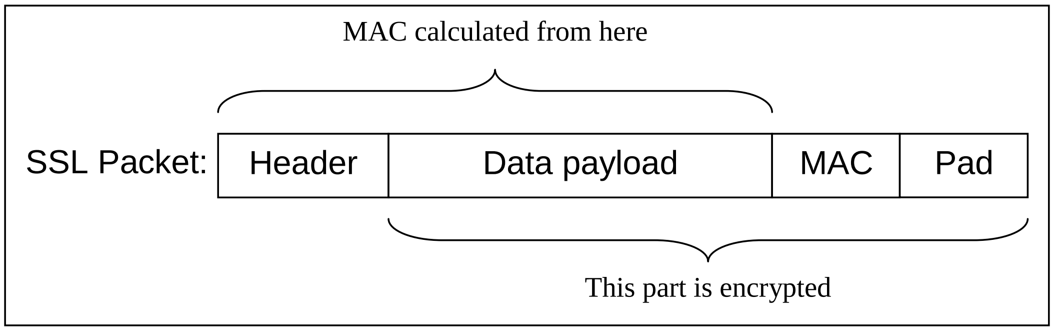
These protocols allow applications to communicate over the network while preventing eavesdropping and tampering. That is, third parties cannot read the content being transferred, and they cannot modify that content without the receiver detecting such an action.

These protocols operate in two phases. In the first phase, a session is initiated. The server and client negotiate together to select algorithms for encryption and authentication, and a shared secret key. In the second phase, the bulk data is transferred. The protocols use encryption of the data packets to ensure that third parties cannot read the contents of the data packets. They use a message authentication code (MAC), based on a cryptographic hash of the data, to ensure that the data is not modified in transit.

During session initiation, before a shared secret key has been generated, the client must communicate private messages to the server using a public key encryption method. The most popular method is RSA, which is based on integer modular exponentiation. Modular exponentiation is a compute-intensive operation and accounts for a majority of the session initiation cycles. A faster modular exponentiation implementation directly translates to a lower session initiation cost.

Under SSL, the bulk data being transferred may be broken into multiple records as there is a maximum size limit of 16 Kbytes on the payload (referred to as a fragment in SSL).

Figure 1: SSL Packet Format



A header is added, and a message authentication code (MAC) is computed over the header and data using a cryptographic hash function. The MAC is appended to the end of the message, and the message is padded. Then everything other than the header is encrypted with the chosen cipher.

The key point here is that all of the bulk data buffers have two algorithms applied to them: encryption and authentication. In many cases, computation of these two algorithms can be combined (“stitched”) [7] together to increase overall performance. During the initial handshake phase, the client and server select an algorithm pair (encryption and authentication) to be used for the session. As RC4-MD5 and AES-SHA1 are very widely used, we focused our efforts on stitching these algorithm pairs.

OpenSSL*

OpenSSL* [1] provides an open-source implementation of the SSL and TLS protocols. OpenSSL is used by many applications and large companies.

For these users, the most interesting figure of merit is the number of secure connections that a server can handle (per second), as this translates directly to the number of servers these companies need in order to service their client base. The way to maximize the number of connections is to minimize the cost of each connection. This can be done by minimizing the cost of initiating a session and by minimizing the cost of transferring the data for that session.

Apache Web Server

Apache is an open source web server [4]. Since 1996, it has been the most popular HTTP server in use, now serving ~70% of all websites [5].

Components of Intel's cryptographic optimizations

Intel's cryptographic optimizations have the following optimized implementations:

- RSA 1024-bit
- Stitched RC4-HMAC-MD5 for both Encrypt and Decrypt
- Stitched AES-128-CBC-HMAC-SHA1 Encrypt
- Stitched AES-256-CBC-HMAC-SHA1 Encrypt
- Standalone RC4
- Standalone SHA1
- Standalone AES-128-CBC and AES-256-CBC

"RSAX" is an optimized RSA-1024 bit implementation and the only component targeted at the session initiation phase. The rest improve speed of the bulk data transfer phase. We developed optimized implementations of the most widely used single ("standalone") encryption and authentication algorithms, as well as even *more efficient* combined ("stitched") implementations of their pairs. We did not develop a decrypt version of the AES-CBC-SHA1 pair, due to Intel® AES-NI based parallel mode implementations being extremely efficient standalone.

The Intel® 64 and IA-32 instruction set architectures have two distinct instruction subsets that can be used by cryptographic algorithms: General Purpose instructions and Single Instruction Multiple Data (SIMD) instructions [13]. SIMD instructions include 128-bit extensions Intel® SSE through SSE4.2 and are utilized in our implementations. Also, Intel® processors based on Intel® 32-nm microarchitecture (such as Intel® Xeon® X5680 used in this study) introduced Intel® AES-NI instruction set extension that allows significantly faster implementation of AES cipher modes [14].

Finally, the 2nd generation Intel® processor based on 32-nm microarchitecture, introduces the 256-bit Intel® AVX extension [9], and some of our implementations have dedicated AVX code paths. However the system used in this study does not support AVX. It is known that the AVX versions give additional performance gain on Intel® processors that support the AVX instruction set [8].

We now briefly describe some of the components.

RSAX

RSAX is Intel's novel implementation of the integer modular exponentiation, which is the basis of the RSA algorithm. Integer modular exponentiation is the operation of raising an integer number to a power and then reducing it by a modulus (taking the remainder when divided by an integer modulus). This is computationally very intensive when the numbers in question have many hundreds of bits.

The standard approaches for modular exponentiation involve a series of squaring or multiplication steps, each of which is followed by a reduction step. The RSAX implementation features a novel reduction method based on folding, coupled with an extensively optimized key size specific assembler implementation. The net result is significantly better performance than the prior OpenSSL implementation. Details on RSAX can be found in [6].

Function Stitching

Function stitching is a technique used when two different algorithms are used primarily in combination with each other, such as hashing and encrypting the contents of a data buffer.

This section presents just a brief overview of Stitching. A more detailed description of Stitching may be found in [7], which gives results for several pairs of stitched algorithms in isolation. This paper extends those results by applying stitching to an important practical application (OpenSSL) and measuring the performance results at the application/system level.

Stitching is a fine-grained interleaving of the instructions from each algorithm so that both algorithms are executed simultaneously. The advantage of doing this is that execution units which would otherwise be idle when executing a single algorithm (due to either data dependencies or instruction latencies) can be used to execute instructions from the other algorithm, and vice versa.

In summary, function stitching improves overall efficiency by achieving higher instruction level parallelism and consequently better utilization of underlying execution resources of a super-scalar microarchitecture. [7]

System Configuration and Experimental Setup

We used OpenSSL versions 1.0.0d and a snapshot of the upcoming 1.0.1 version containing all of Intel's cryptographic optimizations. Version 1.0.0d was released on February 8, 2011, and was the latest available stable release at the time of our experiments. We refer to this release as the Default version. Version 1.0.1, which at the time of this writing was still in development (with nightly snapshots available at the ftp://ftp.openssl.org/snapshot/), contains all of Intel's cryptographic optimization code. We refer to this version as the Intel version. In addition to the 1.0.1 version integration, all of Intel's cryptographic optimizations have been published as a plug-in "engine" available at http://www.openssl.org/contrib/. It can be used with unmodified older OpenSSL versions' binaries, with all improved standalone codes active. With a limited update patch, provided inside, stitched codes are activated too.

Speed tests

We used the built-in 'openssl speed' command line application to test the raw performance of all components of Intel's cryptographic optimizations. Each test was run a series of times and an average was taken for each data point. Command line options used were only those specifying algorithm's name, e.g.: '`openssl speed rc4`'.

Apache with OpenSSL System Tests

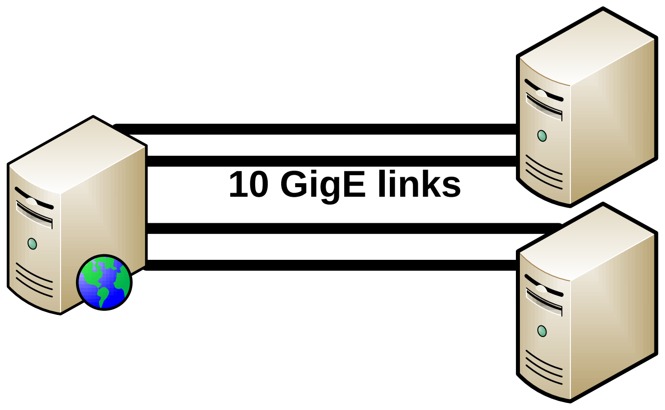
We used the httpd-2.2.17 version of the Apache web-server for the system experiments. Two installations were done, one configured to use the Default version and the other to use the 1.0.1 OpenSSL version previously described. The client machines always used the fastest 1.0.1 OpenSSL version. This was done to maximize the CPU Utilization on the server as we only measured the compute time required to process connections on the server under heavy load.

No effort was made to tune the Apache or OS configuration options for better performance. The httpd server configurations were left mostly unmodified beyond the minimal localized changes such as ServerName and the particular cipher suite we were testing, e.g. `SSLCipherSuite AES256-SHA`.

The intent of the system level study was to understand the real-life system level performance improvement realized on a multi-processor multi-core server handling a high throughput of HTTPS connections from a multitude of clients via a high-speed network connection. As one of the major components we measure is function stitching that speeds up the bulk transfer phase, we chose a large uniform size for the data transfer – 16 Mbytes, that is representative of the growing amount of data transferred in a typical HTTPS session.

The system test environment comprised three dual Intel® Xeon® Processor X5680 based EP class systems, with 12 cores per system (6 cores per processor). Each system ran Fedora* Core 13 with the Linux* Kernel 2.6.33. One system was designated as the server running Apache, and the other two ran the OpenSSL clients. There were four 10 Gigabit Ethernet (10 GigE) links directly connecting the server to the two client machines, with two links per client machine.

Figure 2: Apache System Test Configuration



Our system experiments comprised three tests – one for each of the cipher-authentication algorithm pairs of interest. Each test was set up as a series of two-minute runs. The result for the test, in terms of connections per second, was calculated by averaging across all runs. For each test we used an index.html file of a fixed 16 Mbytes size.

We developed a script to launch the server and client applications, co-ordinate between them at certain points before and after a run, and calculate performance statistics. For each test the httpd server was started, and then the clients used the `openssl s_time` application to connect to the server. For each run, we started 100 s_time instances on each client machine. We set up each client machine to map 50 instances to one 10 GigE link and 50 to the other. This is an example of a typical s_time command line:
`openssl s_time -connect 192.168.0.1:443 -new -www /index.html -time 120`.

Each s_time instance was set up to run for 120 seconds, and would repeatedly make as many connections as possible during that time. Our script was designed to start all the client s_time instances at approximately the same time; specifically, there were 200 s_client processes active making connections to the server repeatedly. We picked the number of s_client instances to get good system throughput for large transfers.

Once the s_time instances had completed for a run (~120 seconds of wall-clock time), our script collected all the results, and then launched the clients again for another run. The server was only restarted when moving to another test. We observed that Apache tightly controls its resources, and in between runs kills most of the server processes and then restarts them as the load increases from the clients requesting again with a new run. A sample s_time output for one instance (for an AES256 test run) looks like:

```
Collecting connection statistics for 120 seconds

128 connections in 6.38s; 20.06 connections/user sec, bytes read
2147521536

128 connections in 121 real seconds, 16777512 bytes read per
connection
```

We summed up the number of connections recorded across all s_time instances and divided by the number of seconds run, to establish the runs connections per second that were supported by the server. Since each connection transferred 16 Mbytes of payload data, we used the connections per sec and the payload size to determine the Gigabits/second throughput performance of the server.

As an example, the instance above reported 128 connections. So for 200 instances, we get a total of 200*128 = 25,600 connections at the system-level, assuming for simplicity that all instances are identical. We get system-level connections/second as 25,600/121 = 211.6 connections/second. The throughput can be computed as 211.6 * 16777512 bytes/second, or 3.55 GigaBytes/second, which translates to 28.4 Gigabits/second that can be supported by the server.

The `-new` option was added to create a new session ID for each connection. Once an SSL connection was established, the s_time application retrieved the index.html file through the command `GET /index.html HTTP/1.0`. As the clients were requesting a file from the server, the tests were measuring the performance of the encryption flows.

The cost of starting up Apache, loading and initializing the OpenSSL library is not factored in the experiments, but we expect this to be a small overhead in real systems where the server is running in steady-state for extended durations of time. As the runs that we measured were for a fixed time, there were a number of partial connections that were terminated, and we did not use these partial connections in our calculations of performance of the server. We expect these errors to be insignificant.

Performance

The performance results provided in this section were measured on Intel® Xeon® Processor X5680 systems, supporting the Intel® AES-NI instruction-set and running at a frequency of 3.33GHz.

We first present the results of the OpenSSL built-in speed test application, which was run on a single core with Intel® Hyper-Threading Technology OFF.

We then present the Apache Server performance on the same system with all cores active and Intel® Hyper-Threading Technology ON.

Speed test Results

The following tables have the results of the OpenSSL speed tests. The *speedup* row is a ratio of the performance of the default and the Intel version and represents the speedup achieved with Intel's cryptographic optimizations. The RSA test has a single column, as the size of the operand is fixed to be the 1024-bit key.

Figure 3: OpenSSL Speed Test for RSA1024 (1 Thread)

	RSA 1024 sign/s
openssl 1.0.0d speed	**2435.3**
openssl 1.0.1 speed	**3770.5**
speedup	*1.55*

The other components relating to bulk data transfer have varying performance in many columns depending on the size of the data. The size of the data is in bytes, whereas the reported performance is in the unit of Kbytes/second. The following table shows the performance for the non-stitched algorithms.

Figure 4: OpenSSL Speed Test for Bulk Non-Stitched (1 Thread)

Algorithm	OpenSSL Version	Performance in Kbytes/second				
Size in Bytes		16	64	256	1024	8192
SHA1	openssl 1.0.0d speed	32,746.6	107,033.8	253,540.4	387,296.6	458,398.6
	openssl 1.0.1 speed	35,965.9	122,658.0	311,909.5	507,423.1	622,252.3
	speedup	*1.1*	*1.1*	*1.2*	*1.3*	*1.4*
RC4	openssl 1.0.0d speed	368,563.7	412,746.0	356,534.5	358,573.1	360,363.1
	openssl 1.0.1 speed	302,988.4	594,378.4	759,759.2	824,110.8	844,305.8
	speedup	*0.8*	*1.4*	*2.1*	*2.3*	*2.3*
AES 128 ENC	openssl 1.0.0d speed	101,056.6	113,460.6	115,176.2	116,231.9	116,682.6
	openssl 1.0.1 speed	793,718.7	912,838.1	935,134.9	940,856.0	942,830.7
	speedup	*7.9*	*8.0*	*8.1*	*8.1*	*8.1*
AES 256 ENC	openssl 1.0.0d speed	73,780.7	81,861.5	82,445.4	82,701.3	83,073.5
	openssl 1.0.1 speed	588,006.9	661,034.2	672,661.9	675,686.1	677,360.6
	speedup	*8.0*	*8.1*	*8.2*	*8.2*	*8.2*
AES 128 DEC	openssl 1.0.0d speed	72,114.7	81,593.4	83,374.3	83,959.5	84,317.3
	openssl 1.0.1 speed	778,171.1	1,938,286.4	2,447,218.7	2,644,311.4	2,717,694.6
	speedup	*10.8*	*23.8*	*29.4*	*31.5*	*32.2*
AES 256 DEC	openssl 1.0.0d speed	52,638.0	58,220.5	59,306.2	59,326.5	59,389.3
	openssl 1.0.1 speed	580,025.0	1,528,491.4	1,829,379.3	1,936,121.2	1,968,417.5
	speedup	*11.0*	*26.3*	*30.8*	*32.6*	*33.1*

The huge speedup factors observed for the non-stitched AES Algorithms above are primarily due to AES-NI; the default version of OpenSSL performs the AES Algorithm with software routines that do not leverage the special instruction-set. The SHA1 speedup is based on improved implementation described in [12].

For the stitched algorithms, we do not have a corresponding speed test to run in the default version, and therefore just show the absolute performance of the Intel version. For speedup achieved by stitching for these algorithm pairs, compared to the best standalone implementations, the reader is referred to [7].

Figure 5: OpenSSL Speed Test for Bulk Stitched (1 Thread)

Algorithm	OpenSSL Version	Performance in Kbytes/second				
Size in Bytes		16	64	256	1024	8192
AES128-SHA1	openssl 1.0.1 speed	246,515.1	360,791.9	459,731.6	510,836.1	527,444.7
AES256-SHA1	openssl 1.0.1 speed	222,560.9	311,682.9	415,506.8	469,066.8	483,678.7
RC4-MD5	openssl 1.0.1 speed	162,647.1	296,692.6	397,061.8	473,039.9	506,468.4

These are the commands used to generate the data:

1. openssl speed rsa1024
2. openssl speed sha1
3. openssl speed rc4

4. openssl speed -evp aes-128-cbc
5. openssl speed -evp aes-256-cbc
6. openssl speed -evp aes-128-cbc –decrypt
7. openssl speed -evp aes-256-cbc –decrypt
8. openssl speed -evp aes-128-cbc-hmac-sha1 (OpenSSL 1.0.1 only)
9. openssl speed -evp aes-256-cbc-hmac-sha1 (OpenSSL 1.0.1 only)
10. openssl speed -evp rc4-hmac-md5 (OpenSSL 1.0.1 only)

Apache with OpenSSL System Test Results

We present the results of running the Apache server using 3 different cipher-authentication algorithm pairs transferring a file of 16 Mbytes size from the server to clients (i.e. an encryption flow). The performance is reported in Gigabits/second, with all 12 Cores active on the server, whereas the speedup numbers are a ratio and represent the gains achieved by Intel's cryptographic optimizations.

Figure 6: Dual Intel® Xeon® Processor X5680 System Throughput Performance (Gigabits/second) running Apache with OpenSSL

	RC4 MD5 RSA1024	AES 128 SHA1 RSA1024	AES 256 SHA1 RSA1024
openssl 1.0.0d 12 Core	**18.75**	**7.48**	**5.83**
openssl 1.0.1 12 Core	**30.22**	**28.06**	**27.92**
speedup	*1.61*	*3.75*	*4.79*

Note: *Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.*

Configurations: See Performance section for detailed configurations. 2 tests used: speed test (which is part of the OpenSSL application) and a system level test (described in System Configuration and Experimental Setup on page 9). All testing was performed by Intel Corporation. For more information go to http://www.intel.com/performance.

Conclusion

This paper shows that stitching of the cipher and authentication code can be applied to the TLS/SSL protocols as implemented by OpenSSL*. Coupled with the Intel® AES-NI instruction-set/code and an improved implementation of modular exponentiation, this results in a significant performance improvement (up to ~5X in some cases) as measured at the application level on a dual Intel® Xeon® Processor X5680 system.

Note: *Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.*

Configurations: See Performance *section for detailed configurations. 2 tests used: speed test (which is part of theOpenSSL application) and a system level test (described in* System Configuration and Experimental Setup *on page 9). All testing was performed by Intel Corporation. For more information go to* *http://www.intel.com/performance.*

Contributors

We thank Chengda Yang, Maxim Locktyukhin, Maxim Perminov, Martin Dixon, Gil Wolrich, Kirk Yap of Intel Corporation and Andy Polyakov of OpenSSL project for their substantial contribution to this work.

References

[1] OpenSSL: http://www.openssl.org/

[2] The TLS Protocol http://www.ietf.org/rfc/rfc2246.txt

[3] OpenSSL Release with Intel's cryptographic optimizations http://cvs.openssl.org/chngview?cn=21265

[4] Apache Web Server http://www.apache.org/

[5] Web server usage statistics http://greatstatistics.com/

[6] Fast and Constant-time implementation of Modular exponentiation http://www.cse.buffalo.edu/srds2009/escs2009_submission_Gopal.pdf

[7] Fast Cryptographic computation on IA processors via Function Stitching
http://download.intel.com/design/intarch/PAPERS/323686.pdf

[8] Cryptographic Performance on the 2nd Generation Intel Core Processor
http://download.intel.com/design/intarch/PAPERS/324952.pdf

[9] Intel AVX programming reference
http://softwarecommunity.intel.com/isn/downloads/intelavx/Intel-AVX-Programming-Reference-31943302.pdf

[10] Openssl speed http://www.openssl.org/docs/apps/speed.html

[11] Openssl s_time http://www.openssl.org/docs/apps/s_time.html

[12] Improving the Performance of the Secure Hash Algorithm (SHA-1)
http://software.intel.com/en-us/articles/improving-the-performance-of-the-secure-hash-algorithm-1/

[13] Intel® 64 and IA-32 Architectures Software Developer Manual Volume 1

[14] Breakthrough AES performance with Intel AES New Instructions
http://software.intel.com/file/26898

[15] Intel® Xeon® X5680 http://ark.intel.com/products/47916/Intel-Xeon-Processor-X5680-(12M-Cache-3_33-GHz-6_40-GTs-Intel-QPI)

Authors

Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Sean Gulley and **Wajdi Feghali** are IA Architects with the IAG Group at Intel Corporation.

Acronyms

IA	Intel® Architecture
SIMD	Single Instruction Multiple Data
SSE	Streaming SIMD Extensions
AVX	Advanced Vector Extensions