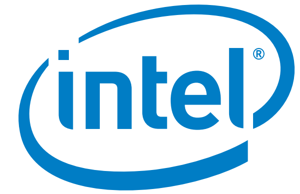
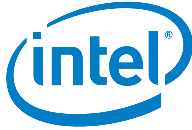


Programming Intel® QuickAssist Technology Hardware Accelerators for Optimal Performance

White Paper

April 2015

Document Number: 332125-002



You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting: http://www.intel.com/design/literature.htm.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at http://www.intel.com/ or from the OEM or retailer.

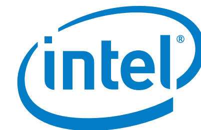
No computer system can be absolutely secure.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit http://www.intel.com/performance.

Intel, Intel Core, Xeon, and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

© 2015, Intel Corporation.



Contents

1.0 Introduction ... 6
- 1.1 Intended Audience ... 6
- 1.2 Overview of this Document ... 6
- 1.3 Terminology ... 7
- 1.4 Reference Documents ... 8

2.0 Intel® QuickAssist Technology Overview ... 10
- 2.1 Parallel Compression Benchmark ... 11

3.0 Background ... 13

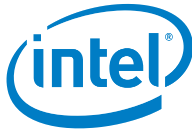
4.0 Accelerator Overview ... 14
- 4.1 Asynchronous Hardware Interface ... 14
- 4.2 Multiple Parallel Engines ... 16
- 4.3 Direct Memory Access ... 17

5.0 Parallelizability ... 19
- 5.1 Symmetric Cryptography ... 19
- 5.2 Public Key Cryptography ... 20
- 5.3 Compression ... 20

6.0 Programming Model ... 22
- 6.1 Software, Multi-Core and Multi-Threading ... 22
- 6.2 Hardware, Synchronous, Single Threaded ... 25
- 6.3 Hardware, Synchronous, Multi-Threading ... 27
- 6.4 Hardware, Asynchronous ... 29
- 6.5 Multi-Core and Hyper Threading ... 30
- 6.6 Multi-Socket and NUMA ... 30
- 6.7 Batch Submissions ... 30
- 6.8 Polling vs. Interrupts ... 30
- 6.9 Programming Model Summary ... 32

7.0 Memory Model ... 33

8.0 Methodology ... 34
- 8.1 Benchmark Application ... 34
- 8.2 Compression Framework ... 34
  - 8.2.1 Chunks ... 36
  - 8.2.2 Buffer Sizes ... 37
- 8.3 Providers ... 37
  - 8.3.1 zlib Provider ... 38



8.3.2 qat Provider....38
8.4 Inputs....40
8.5 Outputs....42
8.5.1 Compression Ratio....42
8.5.2 Compression Elapsed Time....42
8.5.3 Throughput....43
8.5.4 Compression CPU Time....43
8.6 Platform Description....44

9.0 Related Work....45

10.0 Future Work....46

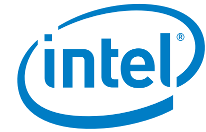
11.0 Summary of Recommendations....47

Figures

Figure 1. Parallel Compression Software Stack....12
Figure 2. Multi-Socket and NUMA....18
Figure 3. Compression Ratio vs. Chunk Size....21
Figure 4. Multi-Threading and Multi-Core....23
Figure 5. Impact of Multi-Threading on zlib Provider (SW Only, No Acceleration, No HT)....24
Figure 6. Calling Hardware Synchronously from a Single Thread....25
Figure 7. Time Taken to Compress 1 GB using Software and Hardware....26
Figure 8. Calling Hardware from Multiple Threads or Asynchronously....27
Figure 9. Hardware Offload from Multiple Threads Synchronously....28
Figure 10. Calling Hardware Asynchronously from a Single Thread....29
Figure 11. Comparison of Different Programming Models....32
Figure 12. Multi-Threading and Segments....35
Figure 13. Compressing in Chunks....37
Figure 14. Flowchart for QAT Provider - Compression Function....38
Figure 15. Flowchart for QAT Provider - Compression Callback Function....39

Tables

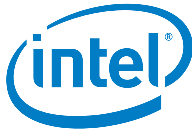
Table 1. Terminology....7
Table 2. Reference Documents....8
Table 3. Popular Cryptographic and Compression Frameworks and Libraries....10
Table 4. Number of Engines in Intel® Communications Chipset 8955....16
Table 5. Affinity Mask....41



Revision History

Date	Revision	Description
April 2015	002	Updated Table 1 Terminology.
March 2015	001	Initial release.

§



1.0 Introduction

Intel® QuickAssist Technology allows compute-intensive workloads, specifically cryptography and compression, to be offloaded from the CPU core onto dedicated hardware accelerators. Intel is working to enable software applications and frameworks to take advantage of this offload seamlessly and transparently, achieving optimal performance with minimal changes to the application. Integrating the functionality is relatively straightforward. Doing so in a way which is optimal from a performance perspective can be more challenging, however, due to differences in the programming and memory models between the hardware and traditional software applications. This document discusses some techniques for overcoming these differences. Using a worked example of a sample application which performs compression, it specifically demonstrates how to apply some of these techniques to perform parallelization under the hood of a synchronous, user space API. The results of this benchmark demonstrates that an approximate 95% reduction was achieved in elapsed time and CPU time required to compress a 1 GB file relative to a software implementation based on zlib, while achieving approximately the same compression ratio.

The document is intended to arm developers with the knowledge they need to be able to identify opportunities for parallelism in various applications; to identify the best layer at which to exploit this parallelism; and to program hardware based on Intel® QuickAssist Technology to achieve optimal performance.

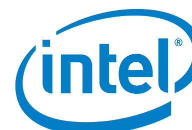
1.1 Intended Audience

The intended audience includes software designers and developers programming applications using Intel® QuickAssist Technology who are using the Intel® QuickAssist API, or who are programming adapters which allow Intel® QuickAssist Technology to be "plugged in" to other cryptographic and/or compression libraries/frameworks.

1.2 Overview of this Document

The document is structured as follows:

- Section 2.0 Intel® QuickAssist Technology Overview presents an overview of the Intel® QuickAssist Technology, and the example of an application which performs compression is introduced.
- In Section 3.0 Background, some background on cryptography and compression is provided including why these workloads are important.
- In Section 4.0 Accelerator Overview, an overview of the accelerator hardware is given, focusing on the inherently asynchronous nature of the hardware interface,



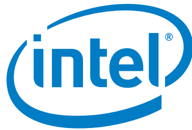
and how the hardware consists of multiple parallel engines, all of which need to be kept busy in order to achieve the maximum benefit. Also discussed is the memory model used by the hardware.

- Section 5.0 Parallelizability discusses parallelism in general, and specifically the parallelism inherent in cryptography and compression.
- Section 6.0 Programming Model discusses how software can deal with both the asynchronous nature of the hardware with the need to keep multiple parallel engines busy.
- Section 7.0 Memory Model discusses how software can deal with the hardware's memory requirements is discussed.
- Section 8.0 Methodology describes a simple framework and application which has been developed to demonstrate some of the principles described in the earlier sections, and to show the results of that benchmark.
- Section 9.0 Related Work gives a reference to some additional key design decisions that should be considered and other techniques that may be employed.
- Section 10.0 Future Work states some examples of possible future work.
- Finally, the recommendations from this document are summarized in Section 11.0.

1.3 Terminology

Table 1. Terminology

Term	Description
AES	Advanced Encryption Standard
API	Application Programming Interface
CNG	Cryptography API: Next Generation. This is a cryptographic API and framework that is part of Microsoft* Windows*.
CPU	Central Processing Unit. A CPU can contain multiple cores. A platform can contain multiple CPUs, each in its own socket.
CTR	Counter, a mode of block ciphers
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
ECB	Electronic Code Book, a mode of block ciphers
GB	Gigabyte
Gbps	Gigabits per second
HMAC	Keyed-Hash Message Authentication Code, as defined by IETF RFC2104

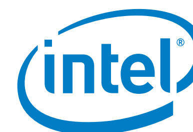


Term	Description
IPsec	IP Security, a cryptographic protocol
IV	Initial Value (or Initialization Vector), an input to block ciphers in certain modes
KB	Kilobyte
MB	Megabyte
OpenSSL*	An open source project that implements the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols, as well as a full-strength general purpose cryptography library.
SHA1	Secure Hash Algorithm, first generation, with output size 160 bits
SHA256	Secure Hash Algorithm, second generation, with output size 256 bits
SSL	Secure Sockets Layer, a cryptographic protocol
TLS	Transport Layer Security, a cryptographic protocol
XEX	Xor-Encrypt-Xor, a mode of block ciphers
XTS	XEX-based tweaked-codebook mode with ciphertext stealing, a mode of block ciphers
zlib	A software library used for data compression.

1.4 Reference Documents

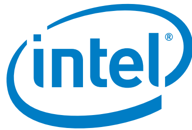
Table 2. Reference Documents

Document	Document No./Location
[1] Cisco, "Cisco Global Cloud Index: Forecast and Methodology, 2013–2018" 2014.	http://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud_Index_White_Paper.pdf
[2] ZDNet, "Microsoft to encrypt network traffic amid NSA datacenter link tapping claims" 2013.	http://www.zdnet.com/microsoft-to-encrypt-network-traffic-amid-nsa-datacenter-link-tapping-claims-7000023687/
[3] Vinodh Gopal, Jim Guilford, Wajdi Feghali, Erdinc Ozturk, Gil Wolrich, Kirk Yap, Sean Gulley, Martin Dixon, "Cryptographic Performance on the 2nd Generation Intel® Core™ processor family" January 2011.	http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/2nd-gen-core-cryptographic-performance-paper.pdf
[4] A. Polyakov, "openssl-1.0.2-beta3."	http://permalink.gmane.org/gmane.comp.encryption.openssl.announce/127



Document	Document No./Location
[5] Schmidt, Douglas C. & Cranor, Charles D., "Half-Sync/Half-Async: An Architectural Pattern for Efficient and Well-structured Concurrent I/O," in Pattern Languages of Program Design, 1995.	http://www.cs.wustl.edu/~schmidt/PDF/PLoP-95.pdf
[6] "Parallel Compression Benchmark."	-
[7] "RtlCompressBuffer function."	http://msdn.microsoft.com/en-us/library/windows/hardware/ff552127%28v=vs.85%29.aspx
[8] "The Calgary Corpus."	http://corpus.canterbury.ac.nz/resources/calgary.tar.gz
[9] " Intel® QuickAssist Technology Performance Optimization Guide."	https://01.org/sites/default/files/page/330687_qat_perf_opt_guide_rev_1.0.pdf

§



2.0 *Intel® QuickAssist Technology Overview*

Intel® QuickAssist Technology offers acceleration for two key compute intensive workloads, namely cryptography and compression. To help customer applications take advantage of this new acceleration technology, Intel is enabling certain key software frameworks and applications.

Fortunately, most software applications that use cryptography and compression today do so via a handful of well-established frameworks and libraries. Some of the most commonly used such frameworks are listed in Table 1 Popular Cryptographic and Compression Frameworks and Libraries. Many of these frameworks are extensible, i.e., they allow other implementations—including hardware-based implementations—to be "plugged in" under the hood. This enables applications to take advantage of the acceleration seamlessly and transparently.

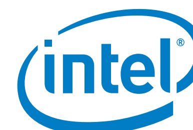
Table 3. Popular Cryptographic and Compression Frameworks and Libraries

Framework	Operating Environment	Services Offered	Plug In Support?	Asynchronous Support
OpenSSL* libcrypto	Linux* user space, also portable to other operating systems	Symmetric crypto, public key crypto	Yes	Work In Progress[1]
CNG	Windows*	Symmetric crypto, public key crypto	Yes	No
Linux Kernel Cryptographic Framework (aka scatterlist API)	Linux kernel	Symmetric crypto	Yes	Yes (for symmetric crypto)
zlib	Most operating systems supported	Compression	No	No

Integrating a hardware accelerator underneath these frameworks so that it is functional is generally relatively straightforward. It involves adapting between the existing APIs used by the frameworks/applications and those provided by Intel® QuickAssist Technology.

Integrating a hardware accelerator such that it achieves optimal performance can be more challenging. Many software applications are single-threaded and use synchronous APIs to access cryptography and/or compression. They also operate in user space using virtually addressed, virtually contiguous, paged memory. As shown

[1] At the time of writing, asynchronous support in OpenSSL is available on a development branch on http://www.openssl.org.



below, the hardware interface is inherently asynchronous, has parallel engines which need to be kept busy, and because it performs DMA, it requires a physically addressed, physically contiguous, pinned memory. Adapting between these programming and memory models can consume a lot of CPU cycles, reducing or even negating[2] the benefits of offloading the workload in the first place.

Integrating Intel® QuickAssist Technology into these frameworks for optimal performance requires that developers be familiar with how the hardware operates, and how to program to it to achieve optimal performance. This document attempts to describe some of the key concepts, and through a worked example, demonstrates some of the techniques that can be used to achieve optimal performance. It also references other documents where key performance-related learnings are captured.

2.1 Parallel Compression Benchmark

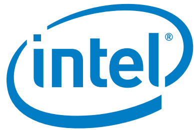
Throughout the document, the example of an application which performs compression is used. As described in Section 8.0 Methodology, a simple compression framework was developed. Similar to some of the common cryptographic and compression frameworks described in Table 3 above, the framework allows different implementations, called “providers”, to be plugged in. Two providers were implemented, to allow for comparison: one a software implementation based on zlib and the other a hardware implementation using Intel® QuickAssist Technology. The hardware provider (called “qat”) can be configured via parameters to use the hardware either synchronously or asynchronously, allowing for parallelism. The framework also supports multi-threading, which is another means to achieve parallelism. Also developed was a simple command-line application called parcomp, which uses this framework to perform parallel compression.

This benchmark acts as an example of how to overcome the performance challenges above:

- The application runs in user space.
- It operates on data buffers in virtual, paged memory.
- It offers a synchronous API to the application.

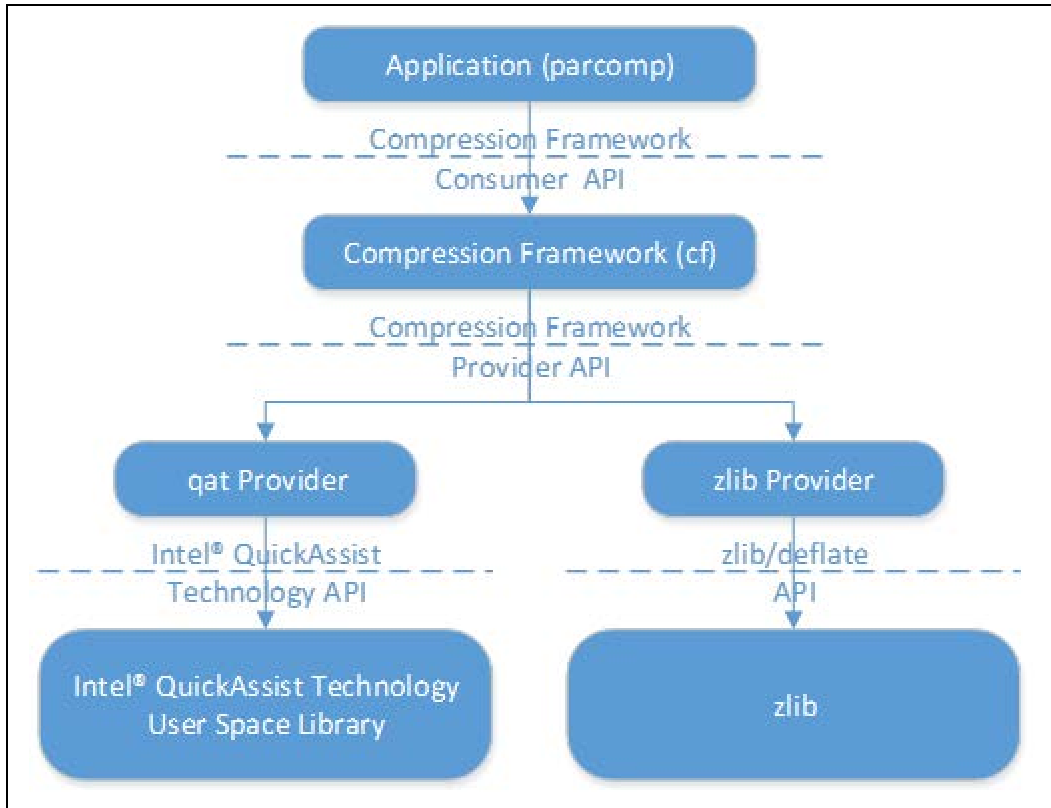
This document describes how, within these constraints, a throughput was achieved of up to over 20 Gbps of compression, cutting the time taken to compress a 1 GB file to approximately 0.7 seconds (compared to approximately 19 seconds using zlib) while achieving approximately the same compression ratio, and cutting the total CPU cycles consumed by a similar ratio.

[2] For example, for cipher algorithms such as AES which are relatively cheap in software using Intel® Advanced Encryption Standard New Instructions (Intel® AES-NI), and for small buffer sizes (e.g., less than a few hundred bytes), offloading may actually consume more cycles than performing the operation in software.

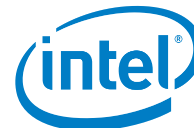


The software stack is illustrated in Figure 1. Parallel Compression Software Stack. For details of the software stack, see Section 8.0 Methodology.

Figure 1. Parallel Compression Software Stack



§



3.0 Background

The volume of data flowing into and out of the world's data centers continues to grow [1]. More and more of this data is being encrypted before it is sent across the network or stored on disk [2] to provide confidentiality. This in turn requires secret session keys to be exchanged between the parties by using public key cryptography algorithms and key agreement protocols. In addition, much of this data is compressed before being encrypted in order to reduce storage and network bandwidth requirements. Public key cryptography, data compression, and encryption are all compute-intensive workloads; performing these algorithms in software can consume significant CPU cycles on the data center servers and reduce the number of cycles available to run the applications and services for which customers pay.

Intel® QuickAssist Technology allows these compute-intensive workloads to be offloaded from the CPU to hardware accelerators, which are more efficient in terms of cost and power than general purpose CPUs for these specific workloads. A server platform that performs a lot of cryptography and/or compression, and which includes Intel® QuickAssist Technology, can therefore potentially achieve higher application performance in the same cost/power envelope—or can achieve the same performance in a lower cost/power envelope—than a platform that uses software only to perform these workloads.

§

4.0 Accelerator Overview

This section describes at a high level the hardware implementing Intel® QuickAssist Technology. The main points from a software application developer's perspective are as follows:

- The hardware interface is inherently asynchronous.
- The hardware consists of multiple parallel engines, all of which need to be kept busy in order to achieve the maximum throughput (and therefore benefit) of the hardware.
- Data sent to the accelerator must be stored in physically contiguous, pinned memory (or multiple such regions, described by a scatter-gather list).

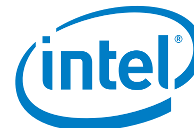
Each of these aspects are described below in more detail. In later sections, how best to develop software to take account of these aspects of the accelerator is described.

4.1 Asynchronous Hardware Interface

At the lowest level, the hardware accelerators that implement the Intel® QuickAssist Technology present a message-based request/response interface to the CPU. This is an inherently asynchronous interface, i.e., the requested data transformation is not complete immediately on returning from sending the request, but rather some time later when the response is received. Contrast this with the sort of synchronous interface that traditional software implementations provide, where the transformed data is available as soon as the called function returns.

The operation proceeds as follows:

- The host software creates a request descriptor containing all the information required by the hardware to perform the job, and writes it onto a request ring (or queue) in the DRAM. It then writes to a doorbell register to inform hardware that the request is ready to be processed. At this point, the host software is free to do some other work.
- The accelerator fetches the request from the ring, and then fetches (from the DRAM, via the DMA) all of the data required to process the request. It performs the requested transformation (e.g., encryption), and then writes the resulting transformed data back to the DRAM. Finally, it writes the response descriptor to the response ring, and writes to a doorbell to indicate to the host software that the response is ready to be processed.
- This doorbell can be configured to generate an interrupt to the host software, or the host software can poll the response ring to see when the response is ready. The host software then processes the response. Depending on the application, this



might involve sending a now-decrypted packet or SSL record to the protocol stack to be processed, or writing the now-compressed data to an output buffer or file, etc.

The above message-based request/response interface is abstracted by an application programming interface (API) which is called the Intel® QuickAssist Technology API. All of the data required in the request descriptor is passed as parameters on this API.

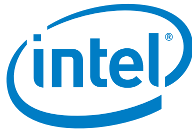
To handle the asynchronous nature of the hardware, the concept of a callback function is used. The programmer registers this callback function, which processes the response. Depending on whether the application wishes to use an interrupt-driven or polled model, the callback function is invoked as follows:

- If interrupts are enabled on the response rings, then the callback function will be invoked in the context of an interrupt handler. In the Linux kernel, for example, this is invoked in the context of a "bottom half".
- Otherwise, the user must call a defined polling function to have responses processed (see Section 6.8 Polling vs. Interrupts for more information on when this should be called). This checks for responses on the specified response rings, and for each response, calls the callback function. It then returns control to the calling function.

Note: The programming for an asynchronous API like this typically involves separating the processing of a job into two phases: what happens before the operation is invoked (the pre-processing) and what happens afterwards (the post-processing). The callback function is written to implement the post-processing, or to cause it to be invoked. Synchronous semantics can also be implemented on top of an asynchronous interface, as follows:

- If interrupts are being used, then the calling thread can pend on some synchronization primitive (e.g., a semaphore), and the callback function can post to this to unblock the thread.
- If polling is being used, then the function can periodically poll until the response is available and has been processed; or a separate thread can do the polling and use a synchronization primitive as described for the interrupt case. As above, see Section 6.8 Polling vs. Interrupts for more information on when this should be called.

Note: The Intel® QuickAssist Technology API supports both synchronous and asynchronous calling semantics. If no callback function is specified, then synchronous semantics are assumed.



4.2 Multiple Parallel Engines

In order to achieve a desired level of performance, implementations of the Intel® QuickAssist Technology include multiple hardware accelerators, or engines, which can operate in parallel. Table 4. Number of Engines in Intel® Communications Chipset 8955 shows the number of engines for each workload that exist in Intel's current generation product.

Table 4. Number of Engines in Intel® Communications Chipset 8955

Workload	Number of Parallel Engines per Device
Compression	6
Symmetric Cryptography	12
Public Key Cryptography	30

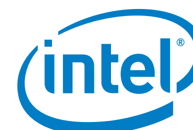
In order to achieve the full advertised throughput of the device, all of the underlying engines must be kept busy. For example, to achieve the maximum compression throughput, the software needs to keep six engines busy; to achieve the full approximately 41K operations/second of RSA2K decryption, the software needs to keep all 30 public key cryptography engines busy.

Note: The software does not need to load balance requests across these engines; this is handled in hardware.

In order to keep all of these engines busy, there should be at least as many jobs outstanding as there are hardware engines in the device. In fact, as the data later in this document will show, it is generally required to have more jobs outstanding than the number of engines, because even within these engines there is some pipelining, with different jobs in different stages of the pipeline (e.g., fetching uncompressed data from the DRAM, compressing data, writing compressed data to the DRAM). The data below in fact shows exactly this: for compression, the throughput of the device plateaus when there are 18 jobs outstanding (six engines, each operating on three jobs simultaneously – see Figure 10. Calling Hardware Asynchronously from a Single Thread, later in this document).

To be able to perform multiple jobs in parallel, the jobs need to be independent. There can be no data dependency between the output of one job and the input of another. See Section 5.0 Parallelizability for a discussion on parallelism.

For those cases where parallelism is possible, there are various mechanisms software applications can use to keep the hardware busy. The preferred mechanisms will be described in Section 6.0 Programming Model.



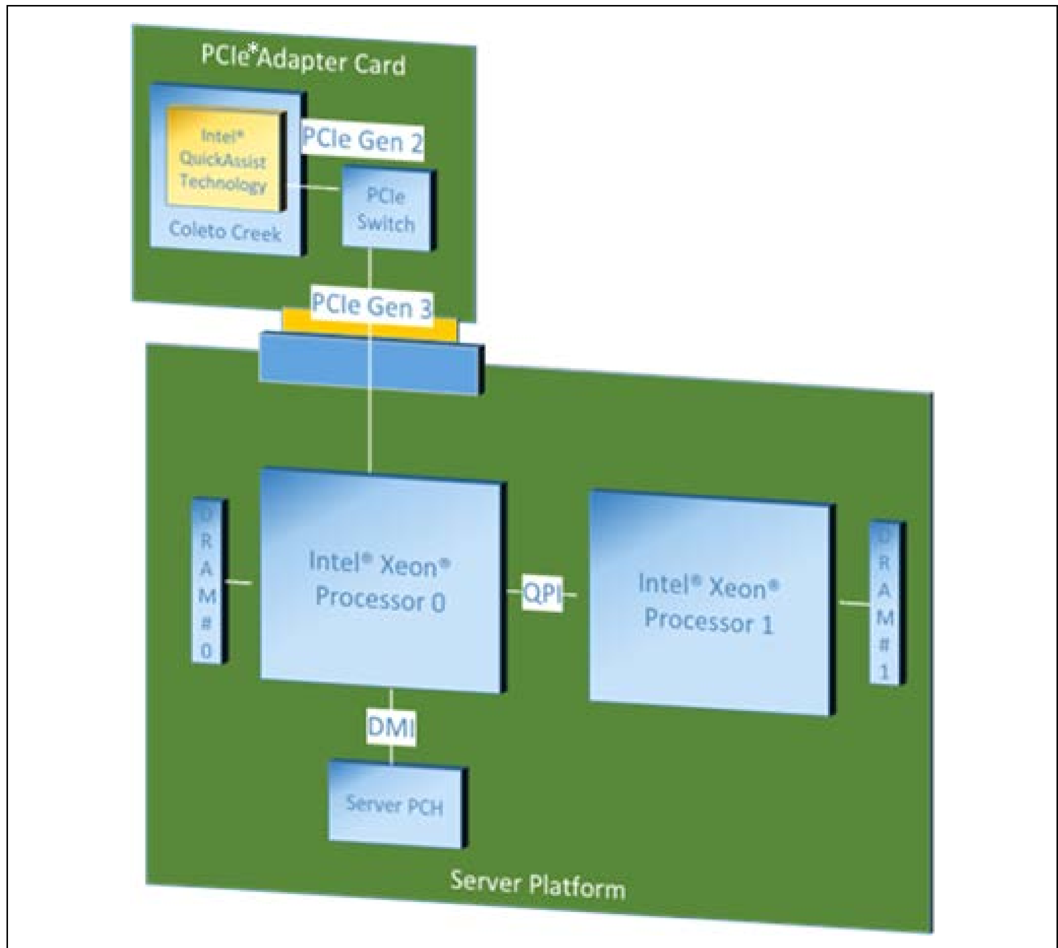
4.3 Direct Memory Access

Intel® QuickAssist Technology is implemented in the hardware as a device which uses Direct Memory Access (DMA) to access data in the DRAM. As such, the data to be operated on must be in the DMA-able memory. In the current generation of hardware, this means that the data must be stored in pages which are pinned, and the pages be physically contiguous. Alternatively, software can pass data in multiple such regions, described by a scatter-gather list.

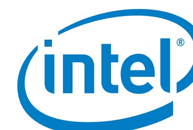
An additional consideration in the multi-processor (MP) or multi-socket platforms is the locality of both the accelerator device, and of the data in memory, to the processor on which the software is running. Many servers consist of multiple processors or CPU sockets, each with its own memory controller and local DRAM. Such systems are called Non-Uniform Memory Architecture (NUMA) systems. When a device performs DMA to read from or write to memory, the time taken to complete that operation will depend on whether the memory is local to the socket to which the device is physically connected, or is located in a remote socket. This is because requests to and responses from remote memory incur additional latency across the QPI bus which connects the sockets.

Figure 2. Multi-Socket and NUMA shows the case of a platform in which there are two processors (sockets), each with its own local memory controller, and a single Intel® QuickAssist Technology-enabled PCIe* card connected via the PCIe to Processor #0. If software running on Processor #1, or with data located in DRAM #1, wishes to perform acceleration, then the request, response, and data all have to incur additional latency across the QPI bus. The impact of this additional latency to the performance of an application is described later.

Figure 2. Multi-Socket and NUMA



§



5.0 Parallelizability

As noted earlier, in order to take advantage of the parallel engines in the hardware, they need to be operating on independent data. Some workloads and applications are easily parallelizable; these are sometimes called "embarrassingly parallel". In other cases, it is not as easy, typically because of data dependencies.

Note: As described later, software implementations can also take advantage of parallelizability. For example, they can distribute the work across multiple cores/threads, or operate on multiple data with a single instruction (SIMD).

Another consideration is the granularity (size) of the parallelizable task. Creating a request and sending it to hardware, and then processing the response later, consumes cycles, called the "cost of offload". To make offload worthwhile, the cost of offload should be less than the cost of simply performing the task in software.

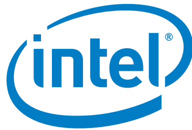
Consider each of the three workloads in turn in terms of their parallelizability, and the granularity thereof.

5.1 Symmetric Cryptography

For block ciphers (e.g., AES), certain modes (e.g., ECB, CTR) are parallelizable in both the encrypt and decrypt directions, meaning that each individual block (16 bytes, in the case of AES) can be encrypted or decrypted independently of the others. Other modes are parallelizable only in one direction, e.g., CBC is parallelizable in the decrypt direction only, because of a data dependency in the encrypt direction, whereby the output of the encryption of a block is used as input to the next block.

In terms of granularity, even for those modes/directions which are parallelizable, offloading individual 16B blocks is almost certain to cost more in terms of "cost of offload" than the job being offloaded. This is especially true for algorithms such as AES, where the Intel® AES-NI instruction allows this to be implemented on the CPU in a small number of cycles. However, given a sufficiently large buffer to be encrypted or decrypted, and a mode/direction which is parallelizable, it would be possible to break this up into smaller chunks (but which are sufficiently large to be worthwhile offloading), offload them as separate independent jobs, and then reassemble the outputs into a single buffer.

Moving up the software stack, some cryptographic protocols allow parallelism, while others do not. For example, in versions of TLS/SSL prior to 1.1, when using AES in the CBC mode, multiple records from a single SSL session cannot be encrypted in parallel, because the CBC residue of one record is used as the IV for the next record, which again requires the implementation to be serialized. In this case, to achieve parallelism, one would have to operate on records from different SSL connections/sessions; this is



feasible assuming the application's programming model allows this (see Section 6.0 Programming Model). In contrast, for TLS 1.1 and beyond, as well as in IPsec, the IV is explicit in each record (or packet, for IPsec), which breaks the dependency between records and allows multiple records from the same session to be encrypted in parallel.

Note: As noted earlier, software implementations can also take advantage of this kind of parallelizability. For example, as described in reference [3], the AESENC instruction (part of the Intel® AES-NI instruction set), which does one round of AES encryption, has a latency of several cycles, but a throughput of one block per cycle. In the case of the CBC-encrypt, one cannot start encrypting a block until the previous block has been encrypted, which leaves the AES engine in the CPU core under-utilized; performance is limited by the latency, rather than the throughput. However, if encrypting multiple independent buffers in parallel, the data dependencies can be broken and ideal performance is limited only by the throughput. Intel has worked with the OpenSSL Software Foundation to take advantage of this in OpenSSL* [4]. When an application writes a data buffer to an SSL socket (via the function ssl3_write_bytes), if certain constraints are met (e.g., the buffer is sufficiently large, the cipher suite is AES-128-CBC+HMAC-SHA1 or -SHA256 and uses an explicit IV, etc.), then the implementation will create multiple independent SSL/TLS records and perform the encryption for up to eight such records in parallel.

For storage applications, these typically operate on disk blocks of 4 KB. They also tend to use cipher modes such as XTS, in which each disk block can be encrypted and decrypted independently, which allows for parallel processing of blocks.

5.2 Public Key Cryptography

Public key cryptography is not inherently parallelizable; each individual request to perform an RSA decryption or other primitive operation cannot be parallelized. As in the case of symmetric crypto, one must rely on the application to submit multiple such independent jobs in parallel.

5.3 Compression

Compression does lend itself to being parallelized, albeit this involves a tradeoff of some compression ratio. A sufficiently large input buffer can be broken into multiple segments or chunks, each of which can be compressed “statelessly” (independently), each yielding a complete deflate block that can also be decompressed independently. Some compression opportunities may be lost in doing this (i.e., the compression cannot take advantage of patterns repeated in different segments), but for sufficiently large chunks, this is not typically significant. The parcomp application described in this document, and shown in Figure 3. Compression Ratio vs. Chunk Size, shows that using Intel® QuickAssist Technology, a chunk size of 64 KB gives an optimal compression ratio. For software (using zlib), larger buffer sizes give marginally better compression ratio, but above 64 KB, the gains are modest.

Note: The compression ratio is heavily dependent on the data being compressed. This data was gathered using the Calgary corpus, as described in Section 8.0 Methodology.

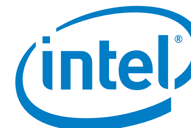
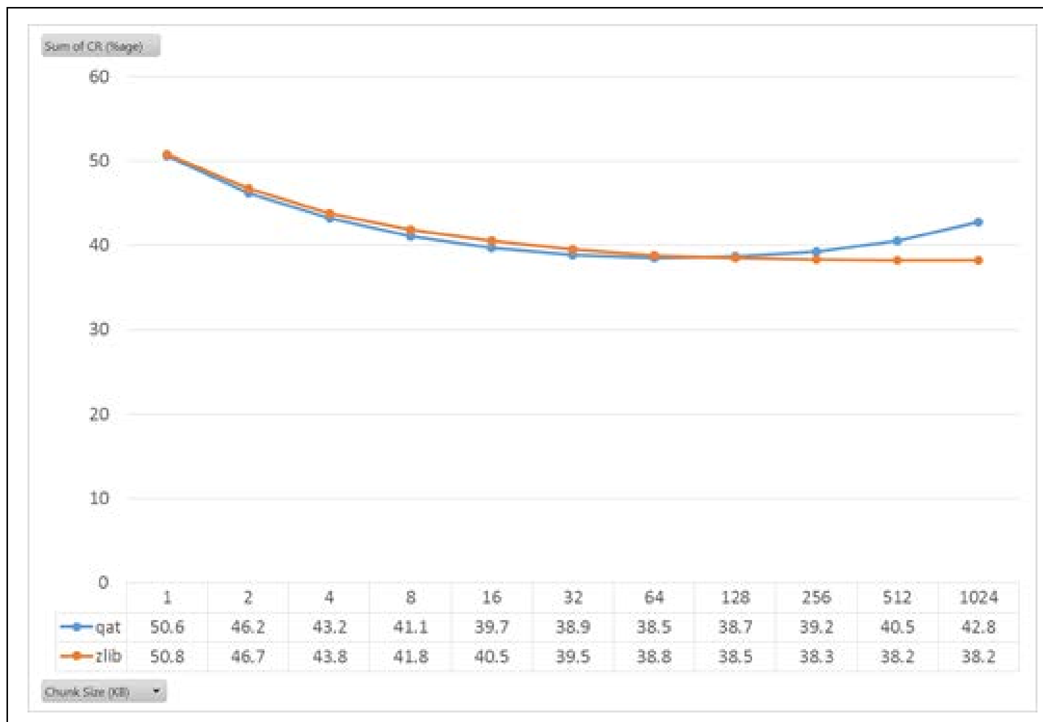


Figure 3. Compression Ratio vs. Chunk Size

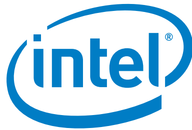


	1	2	4	8	16	32	64	128	256	512	1024
qat	50.6	46.2	43.2	41.1	39.7	38.9	38.5	38.7	39.2	40.5	42.8
zlib	50.8	46.7	43.8	41.8	40.5	39.5	38.8	38.5	38.3	38.2	38.2

The main points from this are (at least for the Calgary corpus – see Section 8.0 Methodology):

- If the buffer to be compressed is sufficiently large (e.g., in the order of MB or larger), then it is easily parallelized by breaking it into chunks of about 64 KB, with minimal loss in compression ratio.
- Using Intel® QuickAssist Technology, the optimal buffer size to offload is found to be around 64-128 KB. Larger buffers (e.g., >128 KB) can yield sub-optimal Huffman trees, leading to a poorer compression ratio.

§



6.0 Programming Model

As noted above, in order to achieve the maximum throughput from the hardware, it is necessary to keep its multiple parallel engines busy, which in turn requires to have multiple jobs outstanding at one time. This is analogous to the way in which a multi-core CPU requires multiple software threads (or processes) to keep the multiple cores busy.

To illustrate this, the next section briefly describes how to optimize a software application for multi-core. Subsequent sections describe how to optimize for hardware offload using the Intel® QuickAssist Technology.

6.1 Software, Multi-Core and Multi-Threading

A single-threaded application does not run any faster on a multi-core CPU than it does on a single-core CPU; it just ends up under-utilizing the CPU. In order to make it run faster, the application needs to be modified to make it multi-threaded.

Note: Clearly, other software can be run on the other cores. In fact, many computers rely on this: individual applications are single-threaded, but the operating system schedules different processes to run on different cores, so that the system feels responsive overall. However, to make individual applications run faster, multi-threading is needed.

Consider the example of the sample application which compresses files. The basic steps required in such a program are:

- Read the uncompressed file from disk to memory.
- Compress the data in memory.
- Write the compressed data from memory to disk.

For larger files, the bulk of the CPU cycles will be spent performing the compression. (Note that applications like this, where the bulk of the cycles are spent performing the workload which can be offloaded using Intel® QuickAssist Technology, will clearly benefit most from the technology.) Figure 4 (a) shows what this looks like in terms of what the CPU utilization on a multi-core CPU. The green bars represents cycles spent reading and writing the file to/from disk, processing command line parameters, etc.; the blue bar represents the time spent actually compressing the data (e.g., in zlib). This single-threaded application does not run any faster on a multi-core CPU; rather, the remaining cores simply remain unused.

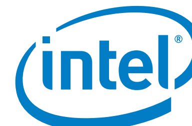
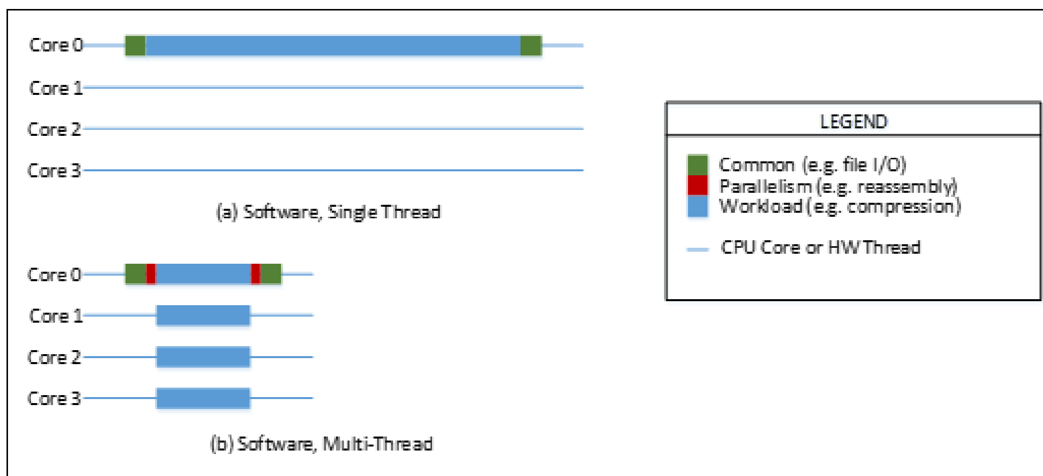


Figure 4. Multi-Threading and Multi-Core



To speed up the compression time on a multi-core CPU, the application can be reworked to be multi-threaded. As described earlier, the file can be broken into N segments, and these can be compressed (statelessly) in parallel in separate software threads, which the operating system will schedule on the different cores, as shown in Figure 4 (b). The main thread then waits for all the other threads to complete, and reassembles the output into a single compressed file.

By spreading the load across N cores, the compression phase now operates up to N times faster. The total CPU cycles increase marginally, as the software now has some extra work to do in setting up, tearing down, and context switching between the threads, reassembling the output, etc. This is represented by the red bars in Figure 4. Again, for the case where there is a lot of data to be compressed, this cost is relatively small. For smaller files, however, the benefit of parallelizing the compression will be more modest.

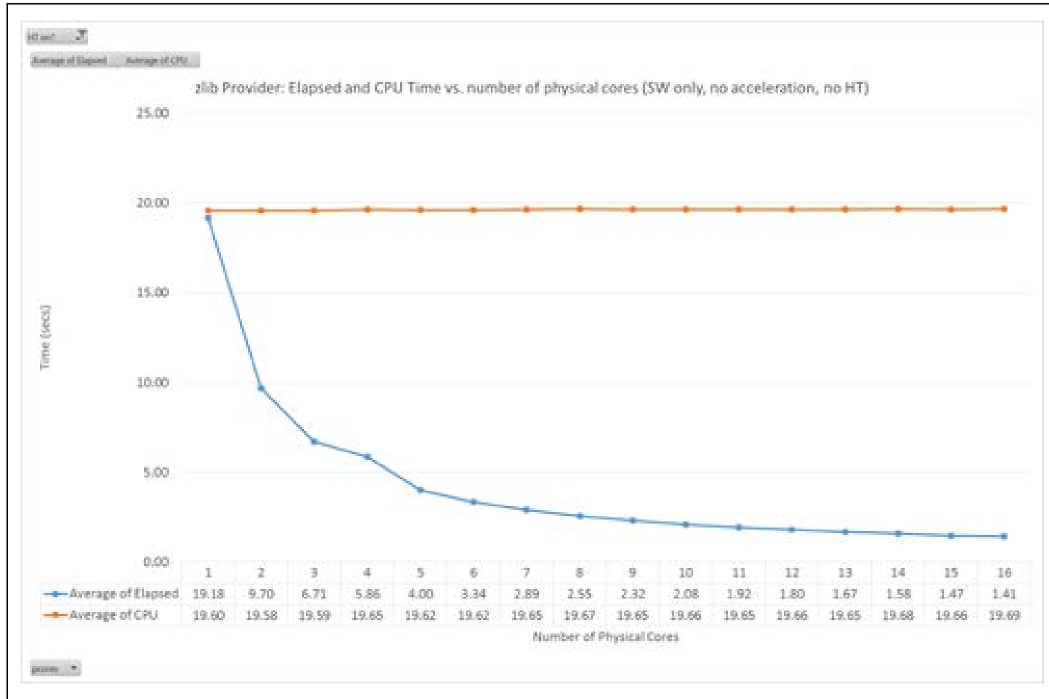
To see what this looks like in practice, consider the parallel compression benchmark using the zlib provider, running on various numbers of threads. Figure 5. Impact of Multi-Threading on zlib Provider (SW Only, No Acceleration, No HT) shows the results of compressing a 1 GB file using the zlib provider, in terms of both the elapsed time and the CPU time.

Note: The data was generated using the Calgary corpus, a compression level of 1, a single chunk per thread (by specifying a chunk size of 1 GB), and an affinity mask that caused threads to be scheduled on only one logical core (lcore) per physical core (pcore), meaning that hyperthreading was effectively disabled. See Section 8.0 Methodology for details. The actual command line used was as follows:

```
parcomp -pzlib -icalgary.1G -otest1 -l1 -c1000000 –yffff –tn
```

where $1 \le n \le 16$.

Figure 5. Impact of Multi-Threading on zlib Provider (SW Only, No Acceleration, No HT)



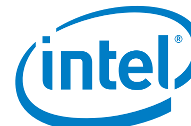
Key data points to note:

- The elapsed time decreases as more threads are added. For example, it reduces from 19.18 seconds with a single thread to 1.41 seconds with 16 threads.
- The CPU time (see Section 8.5.4 Compression CPU Time) stays almost constant, as the same CPU cycles are simply spread across multiple cores. The CPU time increases very marginally as threads are added, probably due to the thread management and context switching, and reassembly of the output.
- In all cases, the compression ratio (not shown) was constant.

The impact of enabling hyper-threading was also measured. This led to a reduction of approximately 20% in the time taken to compress the data (e.g., running two software threads on the two lcores of the same pcore led to a reduction in time of approximately 20% vs. a single software thread on one lcore).

Note: The reported CPU utilization in this case increased by approximately 50%. However, this may be ignored since hyperthreading essentially uses otherwise unused cycles of the physical core.

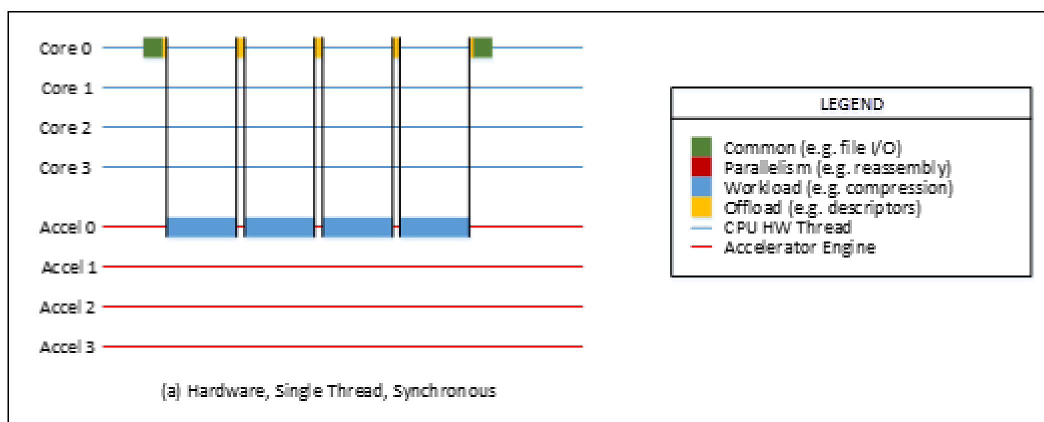
Now, again consider the offload case below.



6.2 Hardware, Synchronous, Single Threaded

The naive way to introduce acceleration is to offload each request from a single-threaded application, using synchronous semantics (sometimes called blocking mode). In this case, the application waits for each request to complete before sending the next request. However, this does not make optimal use of the hardware (only one engine is being used), nor does it make optimal use of the CPU cores (only one core is being used, and that is idle for much of the time waiting for the response). This is illustrated in Figure 6. Calling Hardware Synchronously from a Single Thread. The yellow bars represent cycles spent doing the offload to hardware, e.g., creating descriptors, writing to doorbells, handling responses, etc.

Figure 6. Calling Hardware Synchronously from a Single Thread



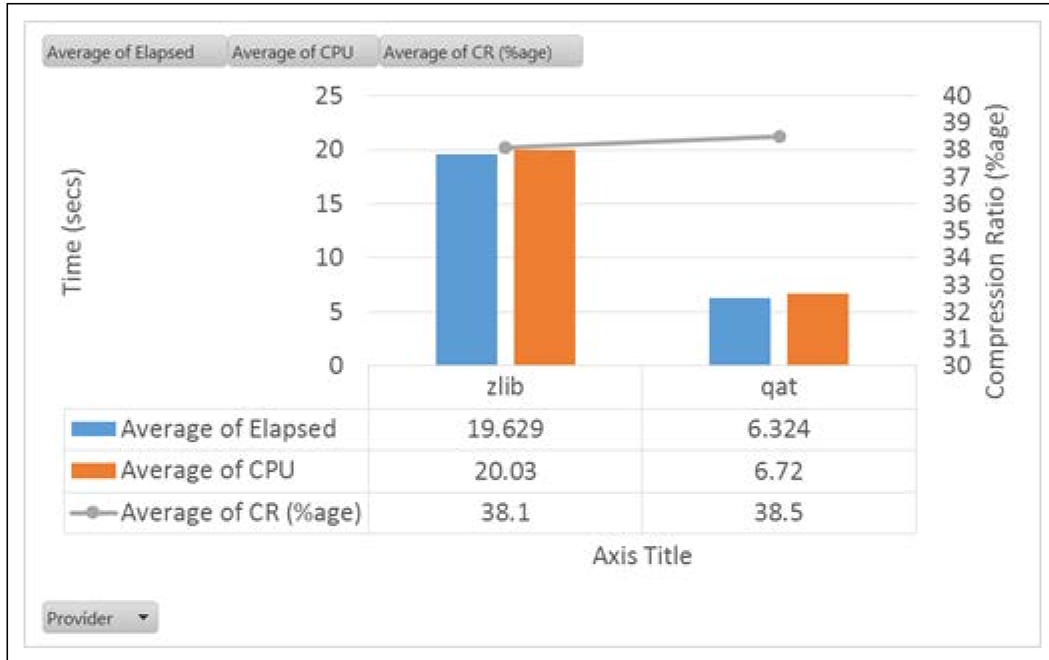
Note: The diagram illustrates the case where the job is broken up into multiple chunks, rather than offloading a single larger job. There are several reasons why this is done, aside from parallelism. For example, the application may wish to minimize the amount of physically contiguous, pinned memory required; or be able to decrypt or decompress subsets of the data independently, e.g., at a disk block boundary; or the hardware may operate optimally at a given buffer size, as seen in the earlier Figure 3. Compression Ratio vs. Chunk Size.

To see how this compares to the zlib-based software implementation, in terms of both compression ratio, elapsed time, and CPU time, see Figure 7. Time Taken to Compress 1 GB using Software and Hardware.

Note: The data was generated using the Calgary corpus, and a compression level of 1. The chunk size was chosen to be optimal for each case, i.e., for zlib, a single chunk (chunk size 1 GB) was used, and for hardware, a chunk size of 64 KB was chosen. See Section 8.0 Methodology for details. The actual command lines used were as follows:

```
parcomp -pzlib -icalgary.1G -otest1 -l1 -c1000000 –j1 –t0

parcomp -pqat -icalgary.1G -otest1 -l1 -c64 -j1 -t0
```

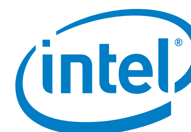
Figure 7. Time Taken to Compress 1 GB using Software and Hardware



Key data points to note:

- Even using the hardware synchronously and single-threaded, it runs faster in this "single-stream" mode than software (6.3 vs. 19.6 seconds), for essentially the same compression ratio (38.5% vs. 38.1%).
- The CPU is mostly idle (waiting for the job to complete), so it consumes less power than the case where it is 100% utilized implementing compression in software. Alternatively, other processes can be scheduled to run on the core during this idle time without impacting on the compression time.

By taking advantage of parallelism, optimizing further the utilization of both the accelerator and the CPU can be done. There are at least two ways to take advantage of parallelism: using multi-threading or using asynchronous (non-blocking) calling semantics. Both methods can improve the elapsed time taken to compress the file, but the asynchronous mechanism does so at a much lower cost in terms of CPU cycles. Each model is reviewed below.



6.3 Hardware, Synchronous, Multi-Threading

As shown in Figure 8 (a), multi-threading can be used as done for the software case above. These threads may run on separate cores, as shown here, or since the CPU thread was under-utilized they may run on a single core [in which case it looks more like Figure 8 (b)].

Figure 8. Calling Hardware from Multiple Threads or Asynchronously

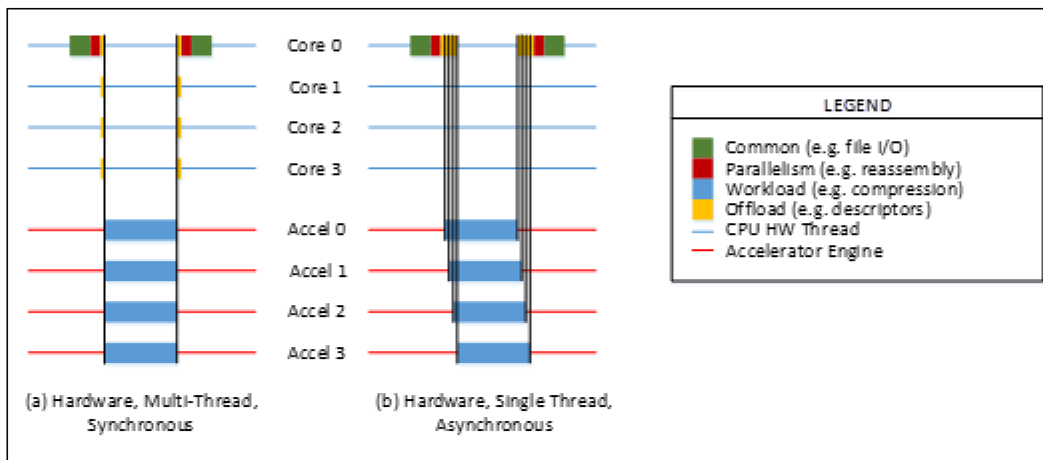


Figure 9. Hardware Offload from Multiple Threads Synchronously shows the data for various numbers of threads.

Note: The data was generated using the Calgary corpus, a compression level of 1, and a chunk size of 64 KB. See Section 8.0 Methodology for details. The actual command line used was as follows:

parcomp -pqat -icalgary.1G -otest1 -l1 -c64 -j1 –tn

where 1 ≤ n ≤ 16.

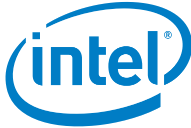
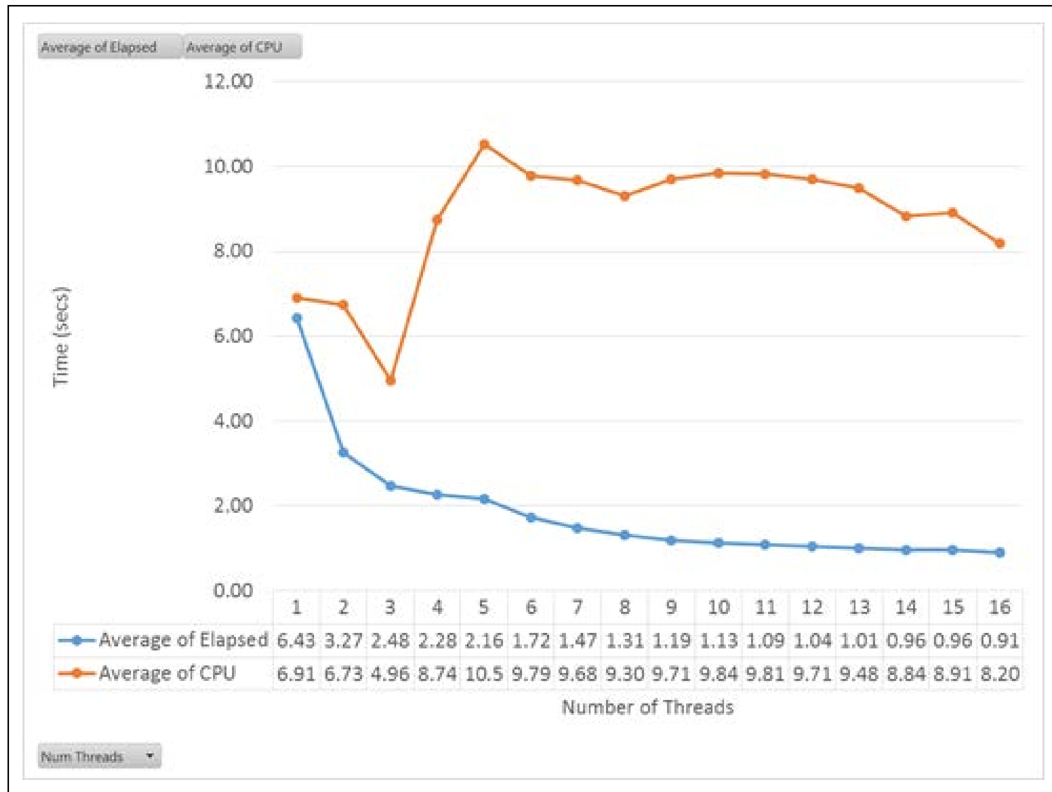


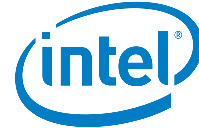
Figure 9. Hardware Offload from Multiple Threads Synchronously



Key data points to note:

- The elapsed time drops as the number of threads increases, as expected. For example, using 16 threads, it drops from approximately 6.4 to 0.9 seconds.
- The total CPU utilization, however, increases due to the context switching and other costs, from about 6.9 seconds (at one thread) to nearly 10 seconds (at 11 threads).

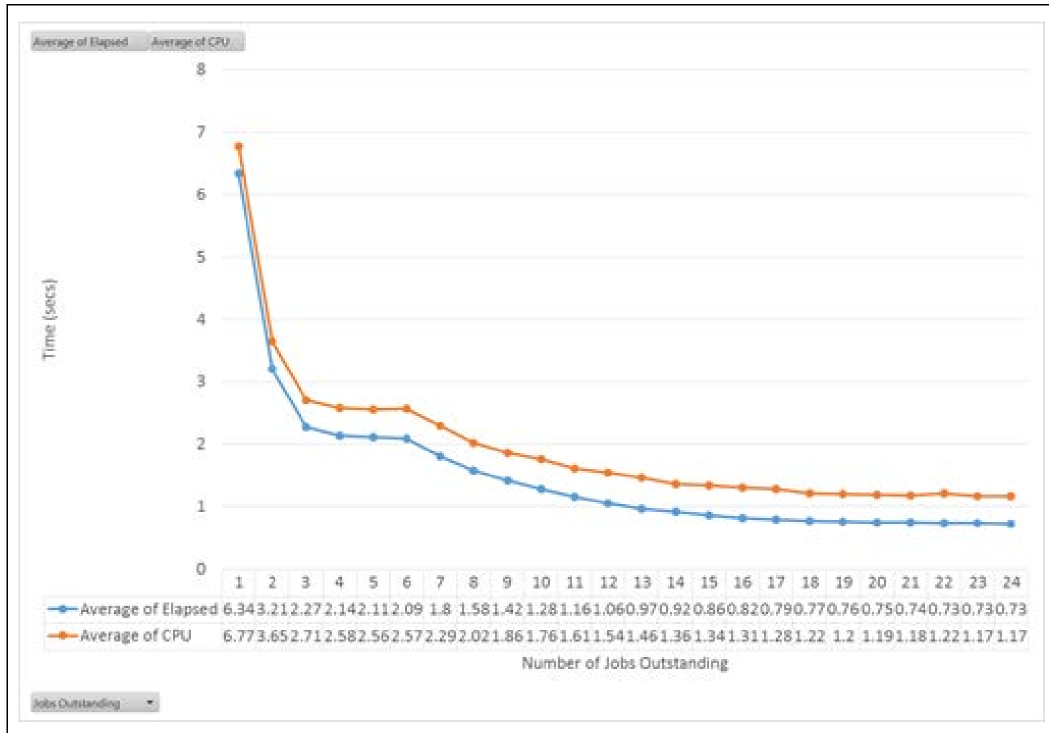
Note: The peak in CPU utilization for five threads, and the drop at three curves, are unexplained, but are reproducible; this is an area for future investigation.



6.4 Hardware, Asynchronous

Alternatively, as shown in Figure 8 (b) above, the Intel® QuickAssist Technology API can be used asynchronously. As shown by the data in Figure 10. Calling Hardware Asynchronously from a Single Thread, this tends to be more efficient than multi-threading, in that no context switching between OS threads is required.

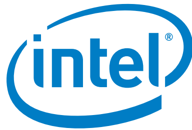
Figure 10. Calling Hardware Asynchronously from a Single Thread



Key data points to note:

- The elapsed time drops as the number of jobs outstanding increases. For example, by having up to 24 jobs outstanding, it drops from approximately 6.3 to 0.73 seconds. With 16 jobs outstanding, the elapsed time is marginally less than using 16 threads.
- The CPU utilization also drops correspondingly, from about 6.8 seconds (at one thread) to about 1.7 seconds (at 24 jobs). This is significantly less than the CPU time using multi-threading. The likely explanation is that in the multi-threading case, each thread spends lots of time polling for responses, with just a single response expected. In the async. case, there is a single thread polling, and this is mostly when there are multiple jobs outstanding, which is more efficient.

Note: The fact that both elapsed time and CPU time remain flat in going from 3-6 jobs outstanding is unexplained, but is reproducible; this is an area for future investigation.



6.5 Multi-Core and Hyper Threading

It was noted earlier for the zlib provider (i.e., a software implementation of compression), hyper-threading led to an improvement of in the order of 20% in compression time.

For the hardware offload case, this was not measured since the asynchronous offload is more efficient than multi-threading.

6.6 Multi-Socket and NUMA

As described in Section 8.0 Methodology, the system under test is a two-socket system with a single Walnut Hill card. The impact of NUMA on the performance of the sample application was not measured for this document; this is an area for future investigation.

6.7 Batch Submissions

Another mechanism to achieve multiple outstanding jobs is to submit multiple jobs in a single request. An application which has multiple jobs to submit can submit them all in a single call. This is called batch submission, and it is supported on the Intel® QuickAssist Technology API (specifically the so-called “data plane” API for crypto and compression).

Note: This is supported in asynchronous mode only.

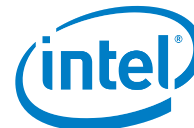
A variation on this theme, also supported on the data plane API, is the ability to submit a single job to the request ring but to defer updating the tail pointer (i.e., ringing the “doorbell”) until after a number of requests have been submitted. The benefit of this is that it allows the cost of the MMIO operation (ringing the doorbell) to be amortized across multiple requests. This is also supported on the Intel® QuickAssist Technology data plane API. Intel has used this technique in an IPsec implementation.

The impact of these techniques was not measured for this document; this is an area for future investigation.

6.8 Polling vs. Interrupts

As described earlier, when the hardware has completed a job, it writes the output data to the DRAM, writes a response descriptor to ring memory (which is also in the DRAM), and then writes to a doorbell register to indicate that the response is available. Applications can decide whether to have this doorbell trigger an interrupt or whether they wish to poll. A detailed comparison of the approaches is outside the scope of this document; however, some of the key pros and cons of each approach are highlighted.

- Latency-sensitive applications may wish to enable interrupts, which can be configured to fire as soon as a response is available, minimizing latency.



- Throughput-oriented applications may wish to use polling to avoid excessive interrupts.
- Real-time systems tend to prefer polling, as interrupts can make it difficult to guarantee meeting deadlines.

For applications that do use polling, there are two additional design decisions that need to be made:

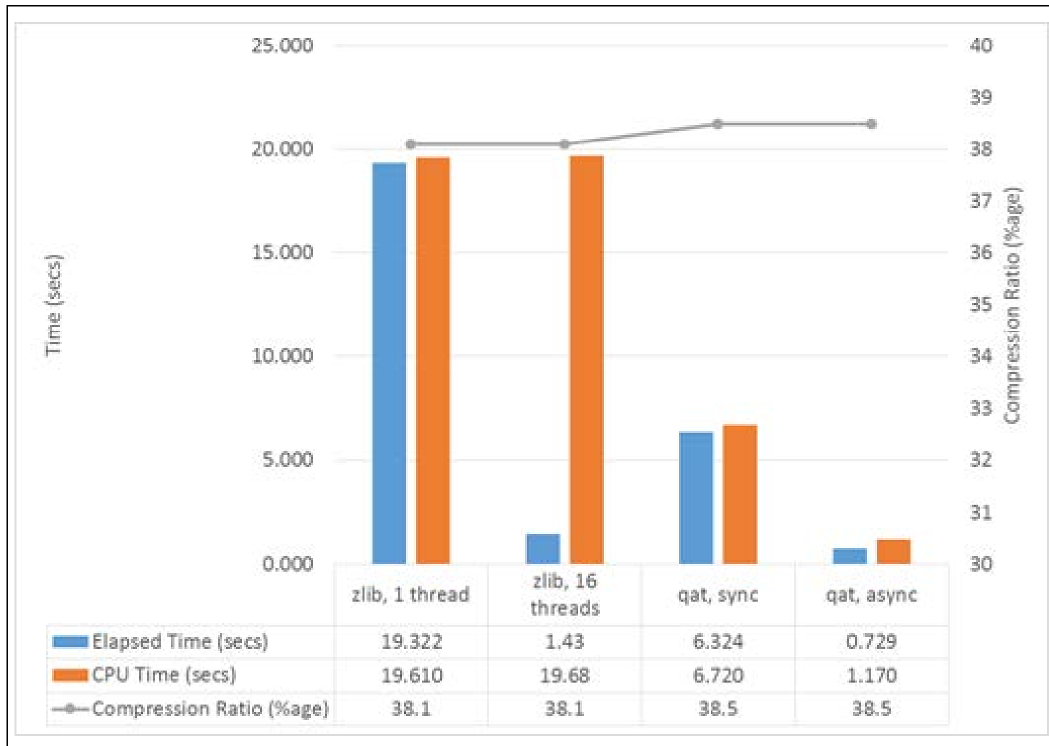
- Polling frequency: A higher polling frequency will reduce the latency but at the cost of potentially higher CPU utilization. Ideally, an application should poll only when there are known to be outstanding requests, and then at the lowest frequency while maintaining acceptable latency. Some tuning may be required to identify the optimal polling frequency for your application.
- Inline polling vs. separate polling thread: Inline polling (i.e., polling from the same thread that submits jobs) will typically avoid context switching overhead.

Note: The parallel compression application described in this document uses polling (specifically inline polling) at a high frequency. For this document, the impact of polling vs. interrupts were not measured; this is an area for future investigation.

6.9 Programming Model Summary

The different programming models are compared side-by-side in Figure 11.

Figure 11. Comparison of Different Programming Models



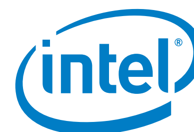
	zlib, 1 thread	zlib, 16 threads	qat, sync	qat, async
Elapsed Time (secs)	19.322	1.43	6.324	0.729
CPU Time (secs)	19.610	19.68	6.720	1.170
Compression Ratio (%age)	38.1	38.1	38.5	38.5

Key data points to note:

- Using hardware, a reduction in elapsed time and CPU cycles was demonstrated to compress a 1 GB file by more than 20x. This allows jobs requiring compression to run in a shorter elapsed time, and frees up the CPU cores for value added applications.
- An asynchronous programming model was implemented underneath a synchronous API at the compression framework level. This means that existing software applications can use this without having to change to an asynchronous programming model.

Note: Implementing a synchronous API on top of an asynchronous implementation is an example of the so-called "half-sync/half-async" design pattern as described in reference [5].
The API defined for compression has simple stateless semantics, similar to the Windows* RtlCompressBuffer API. The zlib API has more complex semantics. For applications which specifically depend on behavioral compatibility with zlib, other changes may be required. These are outside the scope of this document.

§



7.0 Memory Model

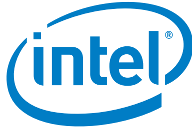
Software running in a user space application typically uses “virtual memory”, that is, memory which is virtually addressed, virtually contiguous and paged. Hardware, on the other hand, requires “physical memory”—memory which is physically addressed, physically contiguous, and pinned (i.e., guaranteed to be resident in memory, at least while the hardware is operating on it).

Note: This is because the hardware needs to be able to perform Direct Memory Access (DMA). Future instantiations of the hardware may be able to perform DMA on virtual memory, but for now, the above requirements apply.

Before sending a request to hardware to operate on data, that data needs to be either copied to physical memory, or the pages need to be pinned and an SGL of physically contiguous regions needs to be created. Both of these approaches add to the offload cost.

In the sample application, the buffer copying approach was chosen. The time spent performing buffer copying was measured. In the case of the Intel data, a total of approximately 1.385 GB of data was copied (1 GB from the uncompressed buffer to the pinned source buffers and approximately 385 MB of data from the pinned destination buffers to the compressed buffer) in an average of about 0.2 to 0.25 seconds (or approximately 0.4 to 0.5 cycles per byte, on a 2.7 GHz CPU core). Compared to the cost of compression in software (which was measured at in the order of 40 cycles/byte), this is relatively insignificant, although it may be more significant for smaller files or other workloads.

§



8.0 Methodology

Intel wanted to be able to compare the Intel® QuickAssist Technology-based implementation of compression to software implementations on various metrics (compression ratio, time to compress a given file, CPU time, etc.). To that end, a simple compression framework was created with the ability to plug in both hardware and software implementations under its API. Then, a simple benchmark application was created to call these implementations with various input parameters to measure the impact of changing certain variables. The code is available from reference [6].

The software is illustrated in Figure 1. Parallel Compression Software Stack. A brief description of the various layers is included below.

8.1 Benchmark Application

The benchmark consists of a program which reads a file from the disk into the DRAM and compresses it using the compression framework. It then writes the output buffer to the output file.

It takes various input parameters which allow us to measure the impact of varying certain variables, as described in Section 8.4 Inputs.

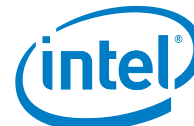
It also measures various outputs and reports them to standard output, as described in Section 8.5 Outputs.

8.2 Compression Framework

A simple compression "framework" is defined that consists of a simple "consumer API" for an application, with the ability to register different implementations of that API, called providers, via a separate "provider API". Calls to perform compression are passed on directly to the specified provider.

Note: An alternative would have been to use the existing zlib library as the framework, using a zlib shim; that too would have allowed comparison of hardware and software implementations. There were a few reasons why Intel chose not to do that for the purposes of this document:

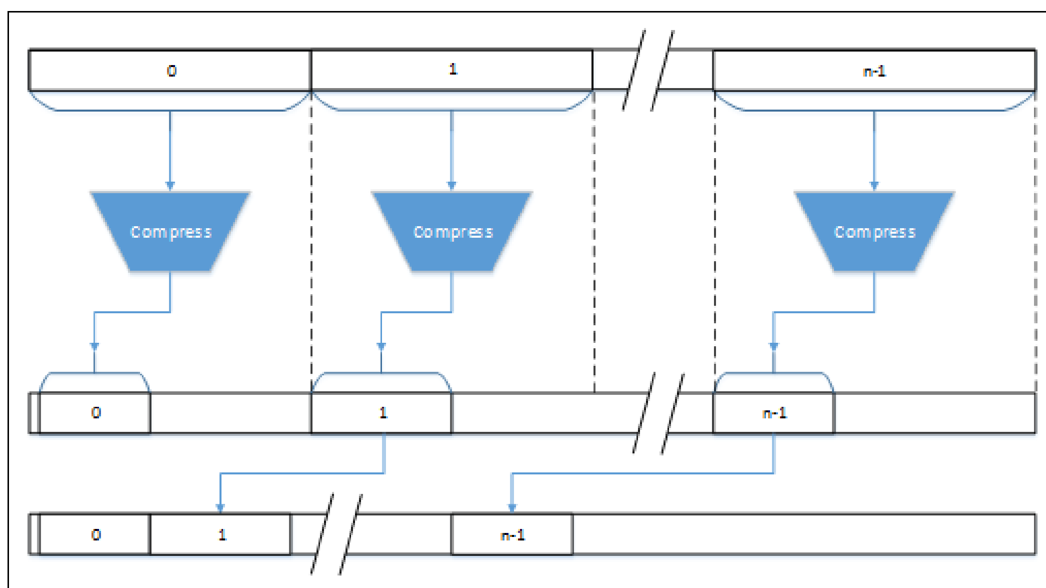
- The zlib software implementation and API have some unique behaviors/semantics which add complexity to the zlib shim. The implementation for this document was kept as clean and simple as possible.
- Intel wanted the flexibility to pass certain other parameters to the implementations (chunk size, etc.) which were not easily accommodated on the zlib API.



- Intel wanted to emphasize that the principles described herein apply not just to zlib, but also to other compression implementations. In fact, many of the principles apply equally to crypto libraries, as well.
- The zlib library does not provide a framework for adding providers (implementations) under the hood; the zlib shim is therefore implemented as a patch. For the purposes of this document, a clean separation was maintained between the API and the implementations.

The providers above are called in the context of a software thread. The framework also supports multi-threading. In this case, the framework will create the specified number of threads (using the pthread library) and divide the input buffer into that number of segments of approximately equal size[3]. Each thread is then given one of the segments to compress, in parallel. The main thread then waits for all the child threads to complete, reassembling the output buffers as they complete, as shown in Figure 12. Multi-Threading and Segments.

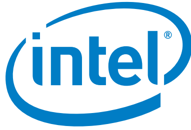
Figure 12. Multi-Threading and Segments



The API was chosen to map fairly closely to the APIs of several typical compression libraries, specifically the zlib deflate() API and the Microsoft* Windows RtlCompressBuffer() API (see reference [7]). The API is synchronous. However, it takes an arbitrarily long input buffer, which allows implementations the freedom to compress in parallel under the hood.

---

[3] The last segment may be slightly larger than the others if the input file size was not an integral multiple of the number of threads.



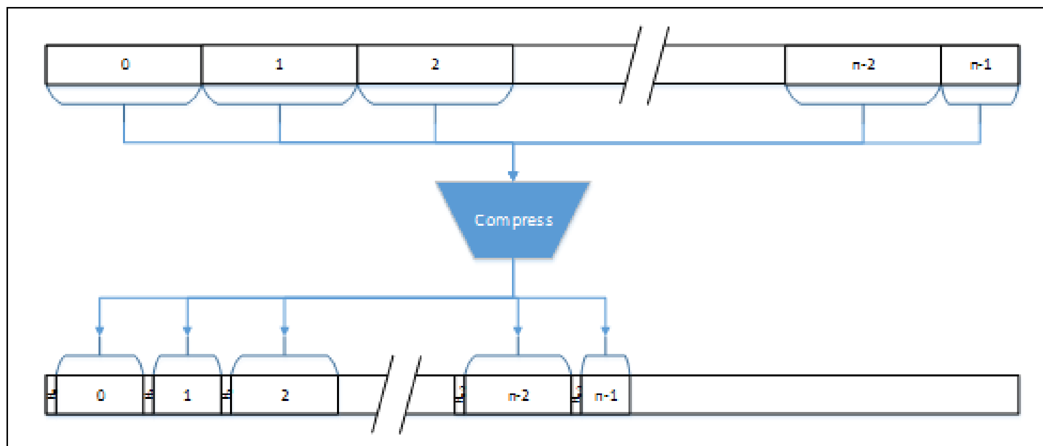
The API is as follows:

```
// declare anonymous structure
struct CfProvider;

struct CfParams {
        int level;
        int deflateWindowSize;
        int bStatic;
        int chunkSize;
        int maxJobsOutstanding;
        int bDebug;
        int numThreads;
        int coreAffinity;
};

int cfGetProvider(
                 char* name,
                 struct CfProvider **ppProvider);

int cfCompressBuffer(
                 struct CfProvider *pProvider,
                 struct CfParams *pParams,
                 unsigned char* uncompressedBuffer,
                 int uncompressedBufferSize,
                 unsigned char* compressedBuffer,
                 int compressedBufferSize,
                 int* compressedDataSize);

int cfDecompressBuffer(
                 struct CfProvider *pProvider,
                 struct CfParams *pParams,
                 unsigned char* compressedBuffer,
                 int compressedBufferSize,
                 unsigned char* uncompressedBuffer,
                 int uncompressedBufferSize,
                 int* uncompressedDataSize);
```

8.2.1 Chunks

The compression provider (within a given thread) operates on chunks, the size of which is an input parameter. Each chunk is compressed statelessly, yielding a deflate block. These deflate blocks are concatenated in the output buffer. Each deflate block is preceded by a header, which in this implementation (for simplicity) is simply a 4B length indicating the length of the deflate block, as shown in Figure 13. Compressing in Chunks.

Figure 13. Compressing in Chunks



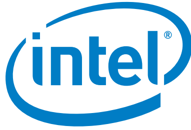
8.2.2 Buffer Sizes

When compressing a data buffer, the size of the compressed (output) buffer is not known in advance. On most compression APIs, if the output buffer is not big enough to contain the compressed data, then the function returns with a status indicating that the compression is not complete, and the application is expected to provide additional output buffer space and call the API again. Handling this so-called overflow condition, while supporting both multi-threading and asynchronous APIs, would have added significant complexity. Therefore, the above API makes the simplifying assumption that the output buffer is big enough, otherwise it fails and exits. The sample benchmark application allocates sufficiently large output buffers to ensure that this does not happen for the input data used in the test cases (it currently assumes the compression ratio will be in the range 20% to 200%). For different corpus data, with different compression ratios, this may need to be modified.

Note: To see how a production-ready implementation might deal with overflows, see Section 8.3.2 qat Provider.

8.3 Providers

The framework defines the concept of a "provider" or implementation of the API. Two providers of this API were developed and can be selected from the command line using the –p (provider) flag, one based on zlib and one based on Intel® QuickAssist Technology.



8.3.1 zlib Provider

The zlib provider uses zlib version 1.2.8.

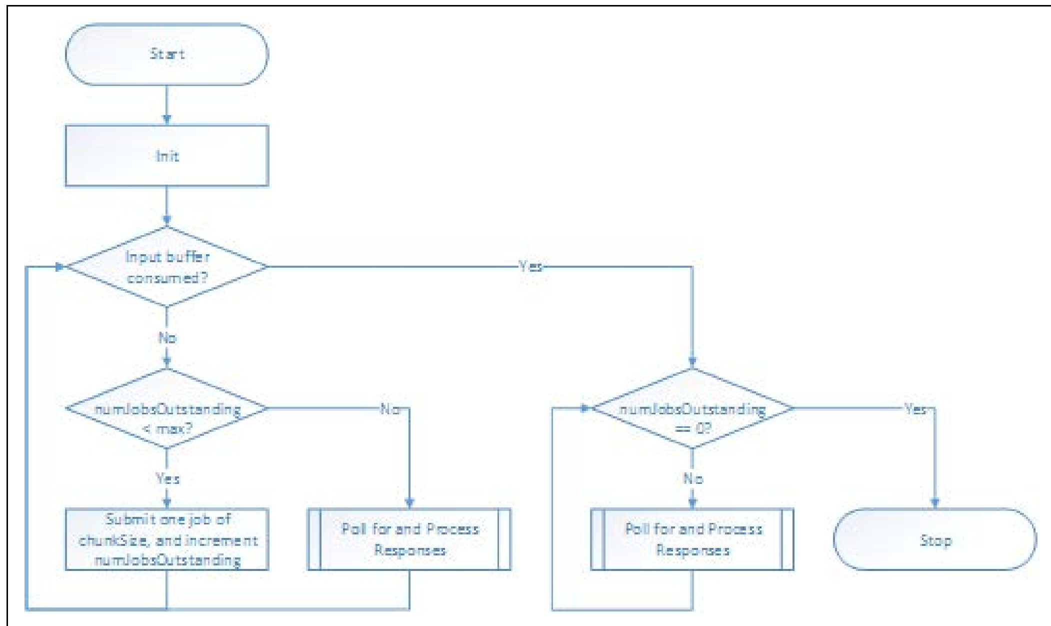
The deflate API is called once for each chunkSize bytes, with the flush flag set to Z_FINISH.

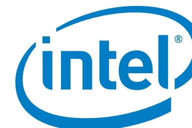
Note: This was a simplification, which comes with a small penalty in both performance and compression ratio; it keeps decompression simpler, so that each deflate block can be decompressed identically regardless of whether it was the last deflate block within a thread in the multi-threaded case.
Intel also experimented with different values for the flush flag, but with chunk sizes in the order of 64 KB and for the Calgary corpus which was tested, the impact on performance and compression ratio was not significant.

8.3.2 qat Provider

The qat provider implements the CF API using Intel® QuickAssist Technology. Specifically, it uses the "traditional" data compression API in an asynchronous fashion in the Linux user space. In the compression direction, it operates as illustrated in Figure 14. Flowchart for QAT Provider - Compression Function, and as described below. The callback function which is invoked when a response is received is described in Figure 15. Flowchart for QAT Provider - Compression Callback Function.

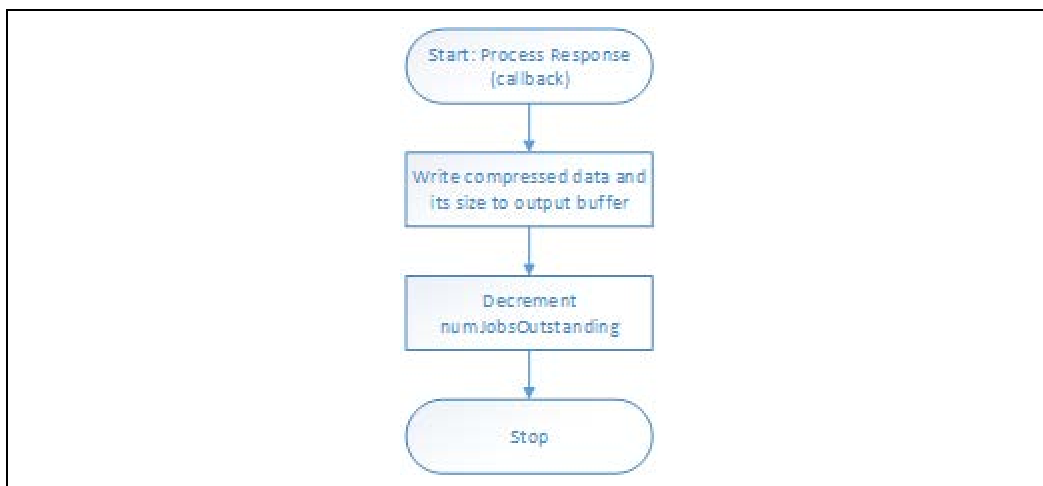
Figure 14. Flowchart for QAT Provider - Compression Function



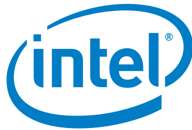


- The Init stage consists of the following steps:
  - It first allocates a compression instance, which maps a set of request and response rings into the address space of the process. (In the multi-threaded case, a separate instance is mapped for each thread, so that each thread can poll its corresponding response ring.) It also allocates memory for the intermediate buffers to be used by the instance when performing dynamic compression.
  - It then creates a session object.
  - It then allocates a set of up to `maxJobsOutstanding` pinned buffers for each of the source and destination buffers, one for each outstanding request. (Again, in the case of multi-threading, this is done per thread.)
- It then goes into a loop which does the following until the entire input buffer is consumed:
  - If the number of jobs outstanding is less than `maxJobsOutstanding`, it sends a request to the accelerator to compress a buffer of size `chunkSize`, and increments the number of jobs outstanding. The request flushes the compressed data and creates a complete deflate block on a byte boundary.
  - Otherwise, it polls for responses. If one or more responses are available, the corresponding callback function is invoked, which writes the number of produced to the output buffer preceded by the size of that deflate block. It then decrements the number of jobs outstanding, and updates pointers into the output buffer so that subsequent output data is written to the correct part of the output buffer. After each poll for responses, the CPU is yielded to allow other threads to run.
- Once the entire input buffer has been consumed, it polls for responses until there are no more jobs outstanding.

Figure 15. Flowchart for QAT Provider - Compression Callback Function



Note: If `maxJobsOutstanding` is set to 1, this is equivalent to using the API synchronously; the application sends one request, and then waits for this to complete before sending



the next request. However, by setting `maxJobsOutstanding` to larger values, up to that number of jobs can be in progress at any time, all from a single software thread.

As noted earlier, the current implementation makes the simplifying assumption that the output buffer is big enough to store the output of the compression. A production-ready implementation may want to deal with this in a more robust fashion, for example as follows:

- Over-provision the output buffers to ensure overflow is an exception case. If it happens, it will impact performance.
- If and when an overflow is received on the response to request number N, then it can be dealt with as follows:
  - Write the compressed data produced thus far from the pinned destination buffer to the output buffer.
  - Re-submit the request to process the remaining (unconsumed) uncompressed data.
  - As each of the next `maxJobsOutstanding` -1 responses arrives, the response should be "held", i.e., do not write the produced data to the output buffer−the location to which to write will not be known until the earlier request has completed and the size of that response is known.
  - When the response to the overflow request arrives, write the produced data to the output buffer, and then process each of the "held" responses.

8.4 Inputs

The input files were all based on the Calgary corpus, downloaded from reference [8]. This was uncompressed and untarred into a directory, and the individual files were then concatenated into a single file (using cat * > calgary). The size of this file is ~3.25 MB (3251493 bytes).

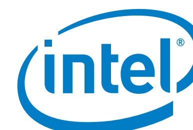
For the purposes of benchmarking, Intel wanted a file of ~1 GB in size, to allow the tests to run for a sufficiently long time. Intel also wanted files with different compressibility. Therefore, two 1 GB files created from the calgary file above:

- calgary.1G: For this, multiple copies of the calgary file were appended to one another until the file was ~1 GB in size (i.e., repeated invocations of cat calgary >> calgary.1G).

Note: Using this file achieved up to 11.7 Gbps of throughput and a compression ratio of ~38.5%.

- cal4K.1G: For this, the first 4 KB of the calgary file was taken (using head –c 4096 calgary > cal4K), and appended multiple copies of this until the output file was ~1 GB in size (i.e., repeated invocations of cat cal4K >> cal4K.1G).

Note: Using this file achieved up to 22.8 Gbps of throughput and a compression ratio of ~13.7%.



Other inputs to vary include:

- The provider (zlib or qat), to allow comparison of SW and HW
- Chunk size
- Compression level
- Direction (compression vs. decompression)
- Maximum number of jobs outstanding (for qat provider only; a value of 1 means synchronous, while larger values allow for greater degrees of asynchronous behavior)
- Deflate window size (for qat provider only)
- Static (vs. dynamic) compression (supported on qat provider only)

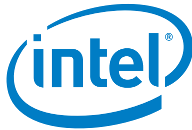
Note: Dynamic compression involves creating a dynamic Huffman tree; in hardware, this requires sending the output of the compression engine through a separate translator stage. This can improve the compression ratio, while increasing the latency of each job. With sufficient jobs outstanding, however, the impact on throughput is minimal.

- Number of threads (for multi-threading)
- Affinity mask to specify the cores to which software threads should be affinitized.

Note: On the system under test, there were two CPU sockets, each with eight physical cores and with Intel® Hyper-Threading Technology/SMT enabled, for a total of 32 logical cores. Some examples of the affinity mask required to identify, for example, thread 0 of all physical cores on socket 0, can be seen in Table 5. Affinity Mask.

Table 5. Affinity Mask

Affinity Mask (–y)	Socket 0		Socket 1	
	Thread 0	Thread 1	Thread 0	Thread 1
All 8 Physical Cores	0x000000ff	0x00ff0000	0x0000ff00	0xff000000



The full set of command line parameters are described by invoking parcomp with the –h flag:

```
-p providerName        Specifies the provider (implementation). Should be
                       one of qat (default) or zlib.
-c chunkSizeInKB       Chunk size, in KB.  Default is 64.
                       Files will be divided into chunks of this size
                       (the last chunk may be smaller), and each chunk
                       is compressed separately. State is not maintained
                       between chunks.
-l compressionLevel    Compression level.  Default is 1.
-d                     Decompress the input file. Default is to compress.
-v (or -g)             Verbose (or debug)
-x numLines            Print a summary of the inputs and outputs in a
                       comma-separated variable (CSV) format for easy
                       importing into a spreadsheet. Specify numLines as 1
                       for data only, or 2 to also include a header
                       summary
-t numThreads          Creates specified number of threads, splits input
                       file into numThreads (near-)equal chunks, and
                       performs the operation in each thread.  The maximum
                       number of threads is 32.
-f cpuFreqInMHz        Specifies the CPU frequency in MHz. If not
                       specified, this will be measured (takes approx. 1
                       second)
-n numIterations       Specifies the number of iterations (allows you to
                       run same compression N times.  Default is 1.
-y coreAffinityInHex   Specifies the cores to which threads should be
                       affinitized, as a bitmask, using hexadecimal
                       notation, e.g. -af means the first four cores.
                       Default value is 0xFF, which on a platform with
                       2-sockets x 8 pcores/socket x 2 lcores/pcore
                       affinitizes to socket 0, pcores 0-8, lcore 0
-a coreAffinityInDec   Specifies the affinity in decimal; see -y for more
                       info
-h                     Print this help message
```

The following options are applicable for the qat provider only:

```
-j maxOutstandingJobs Maximum number of outstanding jobs (requests) that
                      may be outstanding at any one time.  Default is 1.
                      The maximum number of jobs outstanding is 256.
-w windowSize         Deflate window size.  Default is 5.
-s                    Static compression. Default is dynamic.
```

8.5 Outputs

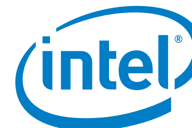
For each run of the benchmark application, the following was measured.

8.5.1 Compression Ratio

This is computed as the compressed file size divided by the uncompressed file size.

8.5.2 Compression Elapsed Time

The time taken to compress the buffer is computed by taking a timestamp (using rdtscp) before and after the compression. The difference in cycles is translated into



seconds by dividing by the CPU clock frequency. This was measured on the platform used for testing as being ~2694 MHz.

Note: The time was measured at the cfCompressBuffer() API.

It *does not* take account of time taken for the following:

- Allocation/freeing of DRAM memory buffers to hold the input and output files
- Reading/writing of the input/output files into/out of DRAM
- Parsing command line parameters

It *does take* account of time taken for the following:

- For multi-threaded cases, creation/deletion/synchronization of threads
- For hardware (qat provider):
  - Allocation of instances (i.e., mapping rings to user space) and sessions
  - Allocation/freeing of pinned buffers for data
  - Copying of the data, one chunk at a time, between the input/output buffers and the pinned buffers passed to hardware
- For software (zlib provider):
  - Stream initialization (deflateInit()) and termination (deflateEnd())

Note: As noted below, the throughput can be derived from the elapsed time. For hardware, the focus is on its throughput. However, the graphs in this document typically chart the elapsed time (seconds per GB) rather than the throughput (Gbps), because in many cases, the hardware is not being fully utilized (e.g., when the number of jobs outstanding is less than about 18). Focusing on the throughput in these cases would be misleading.

8.5.3 Throughput

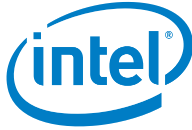
Throughput, measured in Mbps, is computed as the uncompressed file size divided by the elapsed time taken to compress the file, as described above.

8.5.4 Compression CPU Time

This is the total CPU time measured across all cores in a multi-core system. It was measured by reading from /proc/<pid>/stat just before and just after calling the cfCompressBuffer API, and extracting the user and system times from that. (This was correlated with the output of top to validate the correctness of the methodology.)

Note: CPU time is measured rather than CPU utilization because it takes into account the number of cores on which it applied, and the duration. For example, a value of 1 second can be made up in infinitely many ways, including:

- 100% of one core for 1 second
- 100% of two cores for 0.5 seconds



- 50% of two cores for 1 second

8.6 Platform Description

The platform on which the benchmark was performed was as follows:

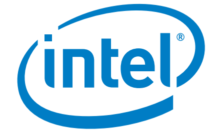
- Hardware configuration
  - Platform: Crown Pass (two CPU sockets)
  - CPU: Each CPU is an 8-core Intel® Xeon® CPU E5-2680 (formerly codenamed Sandy Bridge) clocked @ 2.70 GHz

Note: There are a total of 32 hardware threads (2 sockets x 8 cores/socket x 2 HW threads/core)

  - DRAM: 32GB of DDR-3 @ 1333 MHz
  - Accelerator: Intel® QuickAssist Adapter 8950, featuring the Intel® Communication Chipset 8955 controller (formerly codenamed Coleto Creek), SKU 2

- Software Configuration
  - BIOS: Disabled speed step (EIST) and turbo
  - OS: Fedora* Core 16 (3.6.11-4.fc16.x86_64)
  - Intel® QuickAssist Technology Driver/Firmware Version: 2.1.0

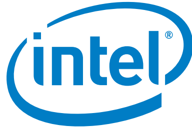
§



9.0 Related Work

In addition to the recommendations in this document, also consult this reference [9], which describes some additional key design decisions that should be considered, and other techniques that may be employed, in order to achieve optimal performance when integrating applications with Intel® QuickAssist Technology.

§

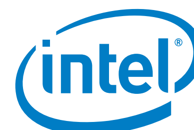


10.0 Future Work

Future work may include the following:

- Measure the performance impact of using buffer copy vs. dynamic pinning of pages.
- Measure the performance impact of polling vs. interrupts.
- Measure the performance impact of polling from the main thread (as is done in the example used in this document) vs. polling from a separate thread (as is done by much of the sample code shipped with the driver).
- Measure the performance impact of NUMA.
- Measure the performance impact of batch submission.
- Understand some of the anomalies in the data mentioned earlier in this document.
- Develop similar sample benchmarks for public key crypto and symmetric crypto, illustrating the principles from this document.

§



11.0 Summary of Recommendations

As demonstrated by the measurements taken using the parallel compression application developed as part of this document, the following recommendations should be followed in order to achieve optimal performance when programming Intel® QuickAssist Technology based accelerators:

- Identify opportunities for parallelism in your application, as described in Section 5.0 Parallelizability.
- To take advantage of this parallelism, ensure to keep the hardware busy by having sufficient jobs outstanding.
  - Ideally, use an asynchronous API for accessing Intel® QuickAssist Technology, since this is the most efficient way to have multiple jobs outstanding.
  - As an alternative, use multi-threading.
- To minimize the impact to the application, consider "hiding" the parallelization "under the hood" of a synchronous API, as demonstrated in the parallel compression example used in this document.
- Aim for NUMA locality as far as possible. Design the application so that data stored in memory associated with a given socket is processed by software running on cores on that socket, and by accelerators which are physically connected to that socket.

§