Intel® ISA-L: Cryptographic Hashes for Cloud Storage

Introduction

In today's modern world, every new device will generate some kind of data that will be stored in the cloud which will put higher demand for more and faster cloud storage. Cloud storage is an easy way to have centralized storage and access everywhere. With the introduction of the Internet of Things (IoT), those devices will generate more data to upload to the cloud. Therefore, processing and encrypting your data before storing it is an important part of data processing. Intel® Intelligent Storage Acceleration Library (Intel® ISA-L) has the capabilities to generate cryptographic hashes extremely fast. In this article, a sample application along with the source code will be shared to demonstrate the utilization of the Intel® ISA-L's capabilities. The sample application has been tested on the hardware and software configuration presented in the table below. Depending on the platform capability, Intel ISA-L can run on various Intel® processor families. Improvements are obtained by speeding up the computations through the use of the following instruction sets:

- Intel® AES-NI – Intel® Advanced Encryption Standard - New Instruction
- Intel® SSE – Intel® Streaming SIMD Extensions
- Intel® AVX – Intel® Advanced Vector Extensions
- Intel® AVX2 - Intel® Advanced Vector Extensions 2

Hardware and Software Configuration

CPU and Chipset	**Intel® Xeon® processor E5-2699 v4, 2.2 GHz** • # of cores per chip: 22 (only used single core) • # of sockets: 2 • Chipset: Intel® C610 chipset, QS (B-1 step) • System bus: 9.6 GT/s Intel® QuickPath Interconnect • Intel® Hyper-Threading Technology off • Intel® Speed Step Technology enabled • Intel® Turbo Boost Technology disabled
Platform	**Platform: Intel® Server System R2000WT product family (code-named Wildcat Pass)** • BIOS: GRRFSDP1.86B.0271.R00.1510301446 ME:V03.01.03.0018.0 BMC:1.33.8932 • DIMM slots: 24 • Power supply: 1x1100W
Memory	Memory size: 256 GB (16X16 GB) DDR4 2133P Brand/model: Micron* – MTA36ASF2G72PZ2GATESIG
Storage	Brand and model: 1 TB Western Digital* (WD1002FAEX)

	Plus Intel® SSD P3700 Series (SSDPEDMD400G4)
Operating System	Ubuntu* 16.04 LTS (Xenial Xerus) Linux kernel 4.4.0-21-generic

Why Use Intel® ISA-L?

Intel ISA-L has the capability to generate cryptographic hashes fast by utilizing the Single Instruction Multiple Data (SIMD). The cryptographic functions are part of a separate collection within Intel ISA-L and can be found in the GitHub repository 01org/isa-l_crypto. To demonstrate this multithreading hash feature, this article simulates a sample "producer-consumer" application. A variable number (from 1-16) of "producer" threads will fill a single buffer with data chunks, while a single "consumer" thread will take data chunks from the buffer and calculate cryptographic hashes using Intel ISA-L's implementations. For this demo, a developer can choose the number of threads (producers) submitting data (2, 4, 8, or 16) and the type of hash (MD5, SHA1, SHA256, or SHA512). The example will produce output that shows the utilization of the "consumer" thread and the overall wall-clock time.

Prerequisites

Intel ISA-L has known support for Linux* and Microsoft Windows*. A full list of prerequisite packages can be found here.

Building the sample application (for Linux):

1. Install the dependencies:
    - a c++14 compliant c++ compiler
    - cmake >= 3.1
    - git
    - autogen
    - autoconf
    - automake
    - yasm and/or nasm
    - libtool
    - boost's "Program Options" library and headers

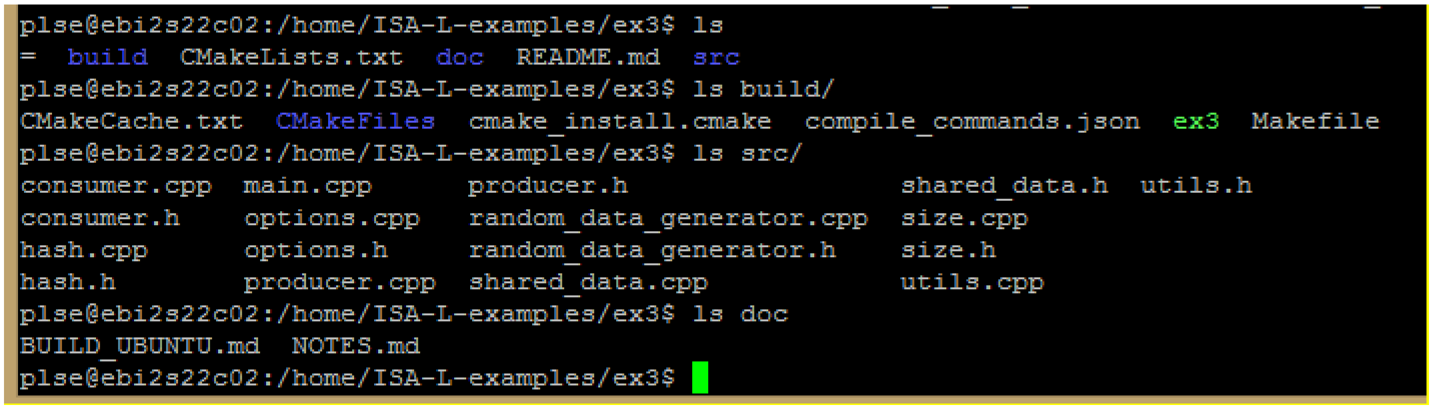
```
>sudo apt-get update
>sudo apt-get install gcc g++ make cmake git autogen autoconf automake yasm
nasm libtool libboost-all-dev
```

2. Also needed is the latest versions of isa-l_crypto. The get_libs.bash script can be used to get it. The script will download the library from its official GitHub repository, build it, and install it in `./libs/usr`.
    - `bash ./libs/get_libs.bash`

3. Build from the `ex3` directory:
    - `mkdir <build-dir>`
    - `cd <build-dir>`
    - `cmake -DCMAKE_BUILD_TYPE=Release $OLDPWD`
    - `make`

Getting Started with the Sample Application

The download button for the source code is provided at the beginning of the article. The sample application contains the following:



```
plse@ebi2s22c02:/home/ISA-L-examples/ex3$ ls
=  build  CMakeLists.txt  doc  README.md  src
plse@ebi2s22c02:/home/ISA-L-examples/ex3$ ls build/
CMakeCache.txt  CMakeFiles  cmake_install.cmake  compile_commands.json  ex3  Makefile
plse@ebi2s22c02:/home/ISA-L-examples/ex3$ ls src/
consumer.cpp  main.cpp     producer.h                 shared_data.h  utils.h
consumer.h    options.cpp  random_data_generator.cpp  size.cpp
hash.cpp      options.h    random_data_generator.h    size.h
hash.h        producer.cpp shared_data.cpp            utils.cpp
plse@ebi2s22c02:/home/ISA-L-examples/ex3$ ls doc
BUILD_UBUNTU.md  NOTES.md
plse@ebi2s22c02:/home/ISA-L-examples/ex3$ 
```

This example will go through the following steps at a high level work flow and only focus in detail on the consumer code found inside “consumers.cpp and the “hash.cpp” files:

Setup

1. In the “main.cpp” file, we first parse the arguments coming from the command line and display the options that are going to be performed.

```
 int main(int argc, char* argv[])
{
     options options = options::parse(argc, argv);
     display_info(options);
```

2. From the “main.cpp” file, we call the `shared_data` routine to process the options from command line.

```
shared_data data(options);
```

In the “shared_data.cpp”, we create the `shared_data` that is the shared buffer that is going to be written to by the producers and read by the consumer, as well as the means to synchronize those reads and writes.

Parsing the option of the command line

3. In the options.cpp file, the program parses the command line arguments using: `options::parse()`.

Create the Producer

4. In the “main.cpp” file, we then create the producers and then call their `producer::run()` method in a new thread (`std::async` with the `std::launch::async` launch policy is used for that).

```
for (uint8_t i = 0; i < options.producers; ++i)

        producers_future_results.push_back(

              std::async(std::launch::async, &producer::run, &producers[i]));
```

In the “producer.cpp” file, each producer is assigned one chunk 'id' (stored in m_id) in which it will submit data. On each iteration, we:

- wait until our chunk is ready_write , then fill it with data.

- sleep for the appropriate amount of time to simulate the time it could take to generate data.

The program generates only very simple data: each chunk is filled repeatedly with only one random character (returned by random_data_generator::get()). See the "random_data_generator.cpp" file for more details.

5. In the "main.cpp" file, the program stores data to the `std::future` object for each producer's thread. Each std::future object holds a way to access the results of the thread once it's done and wait synchronously for the thread o be done. The thread does not return any data.

```
std::vector<std::future<void>> producers_future_results;
```

Create the Consumer and start the hashing for the data

6. In the "main.cpp" file, the program then creates only one consumer and calls it's `consumer::run()` method

```
consumer consumer(data, options);
    consumer.run();
```

In the "consumer.cpp" file, the consumer will repeatedly:

- wait for some chunks of data to be ready_read (m_data.cv().wait_for).
- submit each of them to be hashed (m_hash.hash_entire).
- mark those chunks as ready_write (m_data.mark_ready_write).
- wait for the jobs to be done (m_hash.hash_flush).
- unlock the mutex and notify all waiting threads, so the producers can start filling the chunks again

When all the data has been hashed we display the results, including the thread usage. This is computed by comparing the amount of time we waited for chunks to be ready and read to the amount of time we actually spent hashing the data.

```
consumer::consumer(shared_data& data, options& options)
    : m_data(data), m_options(options), m_hash(m_options.function)
{
}

void consumer::run()
{
    uint64_t hashes_submitted = 0;

    auto start_work    = std::chrono::steady_clock::now();
    auto wait_duration = std::chrono::nanoseconds{0};

    while (true)
    {
        auto start_wait = std::chrono::steady_clock::now();

        std::unique_lock<std::mutex> lk(m_data.mutex());

        // We wait for at least 1 chunk to be readable
        auto ready_in_time =
            m_data.cv().wait_for(lk, std::chrono::seconds{1}, [&] { return
m_data.ready_read(); });

        auto end_wait = std::chrono::steady_clock::now();
        wait_duration += (end_wait - start_wait);

        if (!ready_in_time)
        {
            continue;
        }

        while (hashes_submitted < m_options.iterations)
        {
            int idx = m_data.first_chunck_ready_read();

            if (idx < 0)
                break;

            // We submit each readable chunk to the hash function, then mark that
chunk as writable
            m_hash.hash_entire(m_data.get_chunk(idx), m_options.chunk_size);
            m_data.mark_ready_write(idx);
            ++hashes_submitted;
        }

        // We unlock the mutex and notify all waiting thread, so the producers
can start filling the
        // chunks again
        lk.unlock();
        m_data.cv().notify_all();

        // We wait until all hash jobs are done
        for (int i = 0; i < m_options.producers; ++i)
            m_hash.hash_flush();

        display_progress(m_hash.generated_hashes(), m_options.iterations);

        if (hashes_submitted == m_options.iterations)
        {
            auto end_work      = std::chrono::steady_clock::now();
            auto work_duration = (end_work - start_work);

            std::cout << "[Info   ] Elasped time:          ";
            display_time(work_duration.count());
            std::cout << "\n";
            std::cout << "[Info   ] Consumer thread usage: " << std::fixed <<
std::setprecision(1)
                      << (double)(work_duration - wait_duration).count() /
work_duration.count() *
                             100
                      << " %\n";

            uint64_t total_size = m_options.chunk_size * m_options.iterations;
            uint64_t throughput = total_size /
std::chrono::duration_cast<std::chrono::duration<double>>(
                                      work_duration - wait_duration)
                                      .count();

            std::cout << "[Info   ] Hash speed:            " <<
size::to_string(throughput)
                      << "/s (" << size::to_string(throughput, false) << "/s)\n";

            break;
        }
    }
}
```

The “hash.cpp” file provides a simple common interface to the md5/sha1/sha256/sha512 hash routines.

```
hash::hash(hash_function function) : m_function(function), m_generated_hashes(0)
{
    switch (m_function)
    {
        case hash_function::md5:
            m_hash_impl = md5(&md5_ctx_mgr_init, &md5_ctx_mgr_submit,
&md5_ctx_mgr_flush);
            break;
        case hash_function::sha1:
            m_hash_impl = sha1(&sha1_ctx_mgr_init, &sha1_ctx_mgr_submit,
&sha1_ctx_mgr_flush);
            break;
        case hash_function::sha256:
            m_hash_impl =
                sha256(&sha256_ctx_mgr_init, &sha256_ctx_mgr_submit,
&sha256_ctx_mgr_flush);
            break;
        case hash_function::sha512:
            m_hash_impl =
                sha512(&sha512_ctx_mgr_init, &sha512_ctx_mgr_submit,
&sha512_ctx_mgr_flush);
            break;
    }
}


void hash::hash_entire(const uint8_t* chunk, uint len)
{
    submit_visitor visitor(chunk, len);
    if (boost::apply_visitor(visitor, m_hash_impl))
        ++m_generated_hashes;
}
```

```
void hash::hash_flush()
{
    flush_visitor visitor;
    if (boost::apply_visitor(visitor, m_hash_impl))
        ++m_generated_hashes;
}


uint64_t hash::generated_hashes() const
{
    return m_generated_hashes;
}
```

7. Once `consumer::run` is done and returned to the main program, the program waits for each producer to be done, by calling `std::future::wait()` on each `std::future` object xx.

```
   for (const auto& producer_future_result : producers_future_results)

        producer_future_result.wait();
```

Execute the Sample Application

In this example, the program generated data in N producer threads, and hashed the data using a single consumer thread. The program will show if the consumer thread can keep up with N producer threads.

Configuring the tests

Speed of data generation

Since this is not a real-world application, the data generation can be almost as fast or slow as we want. The "—speed" argument is used to choose how fast each producer is generating data.

If "`--speed 50MB`", each producer thread would take 1 seconds to generate a 50MB chunk.

The faster the speed, the less time the consumer thread will have to hash the data before new chunks are available. This means the consumer thread usage will be higher.

Number of producers

The "`—producers`" argument is used to choose the number of producer threads to concurrently generate and submit data chunks.

Important note: On each iteration, the consumer thread will submit at most that number of chunks of data to be hashed. So, the higher the number, the more opportunity there is for "isa-l_crypto" to run more hash jobs at the same time. This is because of the way the program measures the consumer thread usage.

Chunk size

The size of the data chunks is being defined by each producer for each iteration and then the consumer submits the data chunk to the `hash_function.`

The “--chunk-size” argument is used to choose that value.

This is a very important value, as it directly affects how long each hash job will take.

Total size

This is the total amount of data to be generated and hashed. Knowing this and the other parameters, the program knows how many times chunks will need to be generated in total, and how many hash jobs will be submitted in total.

Using the “ --total-size” argument, it is important to pick a large enough value (compared to the chunk-size) that we will submit a large enough number of jobs, in order to cancel some of the noise in measuring the time taken by those jobs.

The results

```
[Info   ] Elasped time:          2.603 s
[Info   ] Consumer thread usage: 42.0 %
[Info   ] Hash speed:            981.7 MB/s (936.2 MiB/s)
```

Elapsed time

This is the total time taken by the whole process

Consumer thread usage

We compare how long we spent waiting for chunks of data to be available to how long the consumer thread has been running in total.

Any value lower than 100% shows that the consumer thread was able to keep up with the producers and had to wait for new chunks of data.

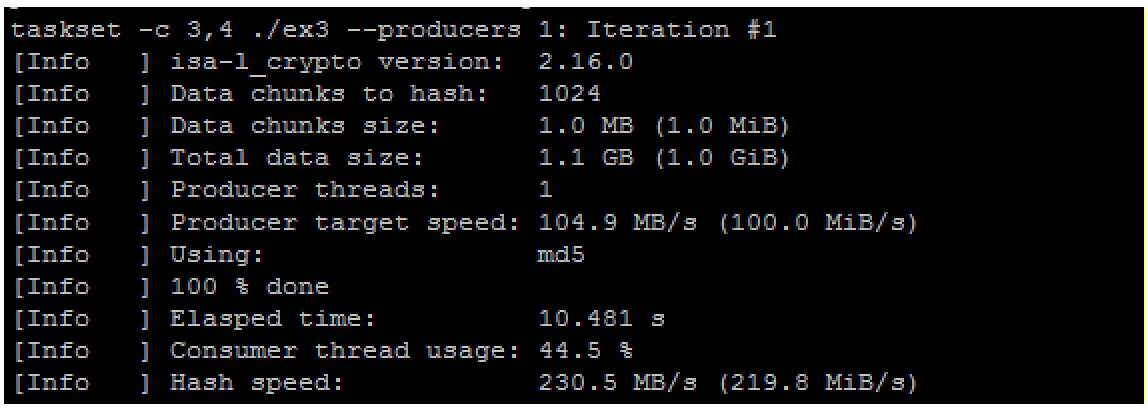
A value very close to 100% shows that the consumer threads were consistently busy, and were not able to outrun the producers.

Hash speed

This is the effective speed at which the isa-l_crypto functions hashed the data. The clock for this starts running as soon as at least one data chunk is available, and stops when all these chunks have been hashed.

Running the example

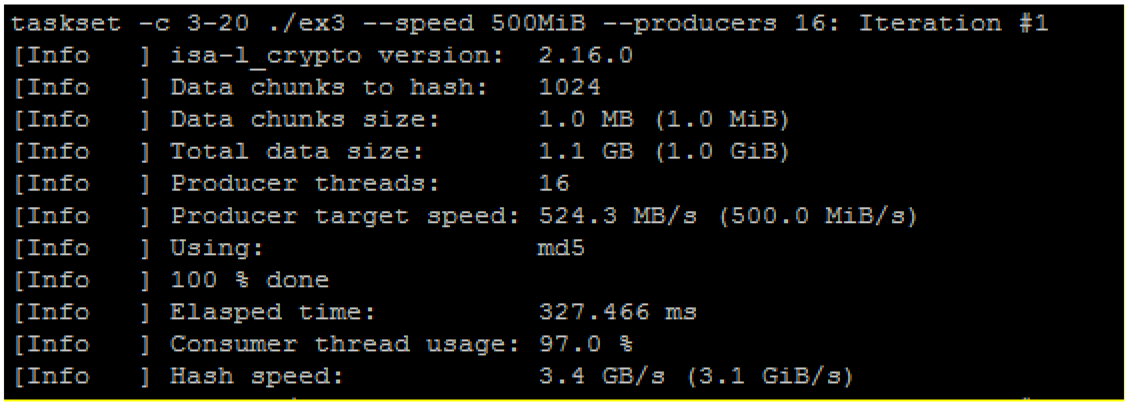
Running this example “ex3” with the taskset command to core number 3 and 4 should give the following output:



```
taskset -c 3,4 ./ex3 --producers 1: Iteration #1
[Info   ] isa-l_crypto version:  2.16.0
[Info   ] Data chunks to hash:   1024
[Info   ] Data chunks size:      1.0 MB (1.0 MiB)
[Info   ] Total data size:       1.1 GB (1.0 GiB)
[Info   ] Producer threads:      1
[Info   ] Producer target speed: 104.9 MB/s (100.0 MiB/s)
[Info   ] Using:                 md5
[Info   ] 100 % done
[Info   ] Elasped time:          10.481 s
[Info   ] Consumer thread usage: 44.5 %
[Info   ] Hash speed:            230.5 MB/s (219.8 MiB/s)
```

The program runs as a single thread on core number 3. ~55% of its time is waiting for the producer to submit the data.

Running the program with the taskset command for core 3 to 20 for the 16 threads (producers) should give the following output:



```
taskset -c 3-20 ./ex3 --speed 500MiB --producers 16: Iteration #1
[Info   ] isa-l_crypto version:  2.16.0
[Info   ] Data chunks to hash:   1024
[Info   ] Data chunks size:      1.0 MB (1.0 MiB)
[Info   ] Total data size:       1.1 GB (1.0 GiB)
[Info   ] Producer threads:      16
[Info   ] Producer target speed: 524.3 MB/s (500.0 MiB/s)
[Info   ] Using:                 md5
[Info   ] 100 % done
[Info   ] Elasped time:          327.466 ms
[Info   ] Consumer thread usage: 97.0 %
[Info   ] Hash speed:            3.4 GB/s (3.1 GiB/s)
```

The program runs as sixteen threads on core numbers 3 to 19. Only ~2% of its time is waiting for the producer to submit the data.

Notes: 2x Intel® Xeon® processor E5-2699v4 (HT off), Intel® Speed Step enabled, Intel® Turbo Boost Technology disabled, 16x16GB DDR4 2133 MT/s, 1 DIMM per channel, Ubuntu 16.04 LTS, Linux kernel 4.4.0-21-generic, 1 TB Western Digital* (WD1002FAEX), 1 Intel® SSD P3700 Series (SSDPEDMD400G4), 22x per CPU socket. Performance measured by the written sample application in this article.*

Conclusion

As demonstrated in this quick tutorial, the hash function feature can be applied to any storage application. The source code for the sample application is also for provided for your reference. Intel ISA-L has provided the library for storage developers to quickly adopt to your specific application run on Intel® Architecture.

Other Useful Links

- Accelerating your Storage Algorithms using Intelligent Storage Acceleration Library (ISA-L) video
- Accelerating Data Deduplication with ISA-L blog post
- Optimizing Storage Solutions using the Intel® Intelligent Storage Acceleration Library

Authors

Thai Le is a Software Engineer who focuses on cloud computing and performance computing analysis at Intel.

Steven Briscoe is an Application Engineer focusing on Cloud Computing within the Software Services Group at Intel Corporation (UK).