Intel®Intelligent Storage Acceleration Library (Intel®ISA-L)

Getting Started Guide

October 17, 2014

Contents

CHAPTER 1 INTRODUCTION

The Intel®Intelligent Storage Acceleration Library includes a library of highly optimized software functions for storage applications. Intel®ISA-L functions are designed to deliver performance beyond what optimized compilers alone can deliver. The core functions are assembly language while demo/example code are POSIX C-code.

1.1 Compatibility

In general, functions optimized for current processors are expected to continue to run on newer processors, but may not take full advantages of the architectural improvements of the newer processor. Each new generation of Intel processor may include new instructions or capabilities that may be utilized to increase the efficiency and/or performance of storage based algorithms. As new processors are introduced and when there is significant performance benefit, new equivalent functions may be added to the Intel®ISA-L to take advantage of the new capabilities.

The included functions and the minimum core required to run each algorithm are recorded in the documentation accompanying each library release. First and foremost, a target processor must support the required instructions used in a specific Intel®ISA-L algorithm. The current Intel®ISA-L contains functions focused on Xeon®class processors utilizing instructions introduced in the Nehalem generation and newer.

1.2 General Properties

The Intel®Intelligent Storage Acceleration Library has the following general considerations.

- Core algorithms are 64-bit assembly code
- Demo/Usage functions are user space POSIX compatible C-code.

*For errata, see the Intel®ISA-L Release Notes.

1.3 Performance Sensitivity

The Intel®ISA-L algorithms mainly exercise the core-to-cache or core-to-memory system paths. As a result, performance is sensitive to the core algorithm code, target processor and SKU, and in some cases the memory configuration (number of memory channels, speed, etc.). See Appendix A for current system configurations used in Intel®ISA-L development and testing.

1.4 Documentation

The following documents are maintained for the Intel®Intelligent Storage Acceleration Library:

- Intel®Intelligent Storage Acceleration Library Getting Started Guide (this document)
- Intel®Intelligent Storage Acceleration Library Release Notes (per release)
- Intel®Intelligent Storage Acceleration Library API Reference Manual (per release)

Commands and code examples are in `Courier font`
All other information in this guide is in normal text (this font).

1.4.1 About this Manual

This document covers getting started with the Intel®Intelligent Storage Acceleration Library (Intel®ISA-L). Instructions are given on how to obtain, build and install the libraries and how to execute sample code included in the library package.

Note: The Intel®Intelligent Storage Acceleration Library will be referred to as the Intel®ISA-L or software library in this document.

CHAPTER 2
SYSTEM REQUIREMENTS

A system based on the Intel® Xeon® processor with Intel® 64 architecture is recommended. Library functions have been developed in user space using Linux Fedora (x86_64), the YASM modular assembler, and the GNU C Compiler (gcc) but individual functions are designed to be portable to a large variety of systems and compilers.

2.1 Software Setup to Match Intel® ISA-L Development and Testing

- The following operating systems have been tested with Intel® ISA-L:
  - Linux distribution with kernel > 2.6 x86_64
  - FreeBSD 9.0 (no AVX) and FreeBSD 9.1 (Intel® ISA-L new Hashing API function support on FreeBSD 9.0 and 9.1 requires Intel® ISA-L and its applications to be compiled with clang)
  - Windows Server 2008
- YASM Modular Assembler (at least v1.2.0) http://www.tortall.net/projects/yasm
- Compiler: either compiler listed below can be used
  - Intel® C++ Composer XE 12.0 or Intel® C++ Compiler 11 (or higher) for Linux* (ICC)
  - FreeBSD clang version 3.1 (branches/release_31 156863) (20120523)
  - GNU C Compiler (GCC) (4.4.4.20100630)
- GNU glibc
- Additional libraries, only required for running the tests:
  - OpenSSL library and headers. Intel® ISA-L has been tested with Version 1.0.0k and 1.0.1e.
    Note: The AES-XTS validation and performance tests require OpenSSL 1.0.1, which is standard on Fedora 18 systems. For Fedora 17 and earlier, FreeBSD and Windows, OpenSSL 1.0.1 library and headers need to be built and linked explicitly. See Appendix D
  - Zlib library and headers. Versions 1.2.5 and 1.2.7 have been tested with Intel® ISA-L

2.1.1 Processor Requirements

Consult the Intel® Intelligent Storage Acceleration Library API Reference Manual to understand the specific instruction set requirements and target products for a specific function. Most functions in the current library release (CRCs and SHA-1 hashing) require a Westmere or newer processor. SSE XOR/PQ RAID type functions require a Nehalem or newer processor, AVX based functions will require a Sandy Bridge or newer processor and AVX2 functions will require a Haswell processor.

The crc, raid, erasure_code and *_mb directories now contain test and perf code with multibinary functions. These functions determine, at runtime, what instruction sets are enabled on the processor. The appropriate function is then used for the duration of that particular test.

For example, if the code is compiled on a Westmere and there are 3 versions of a particular function (func1_sse(), func1_avx() and func1_avx2()), then the function (func1()) will determine that the appropriate function to call is func1_sse(). There is also a base function (func1_base()), which the multibinary function will call if none of the required instruction sets are enabled.

2.2 Calling Convention

Functions in the library use the 64-bit embedded and Unix standard for calling convention http://refspecs.linuxfoundation.org/elf/x86_64-abi-0.95.pdf or the Microsoft x86 calling conventions. When available, 32-bit versions use cdecl. Building of the core library functions (assembly language) require: Yasm. Building examples and tests (C-code) follow simple command line POSIX standards and should be portable to any mostly POSIX-compliant OS.

CHAPTER 3
LIBRARY ACCESS AND DOWNLOAD

Please work with your Intel sales representative to setup licensing for Intel®ISA-L. Once licensing has been completed your Intel field representative will request access for specific members of your company.

Note: The Intel®ISA-L may contain encryption technologies that may be restricted from import into some countries. It is solely your responsibility to ensure the appropriate import license approvals are in place for your company prior to importing these encryption technologies. By selecting the option "I accept the agreement" and clicking the Next button, you agree that your company, as required by the country of import, has the required import license for use of these encryption technologies. If not, please do not proceed and delete this file.

Upon access approval, your designated company contact will be able to access the Intel®ISA-L library area on the Intel Business Link (IBL) system. This area is a repository for documentation and the library versions themselves (encrypted .zip files).

The specific section is:

Storage and I/O: Intel®Intelligent Storage Acceleration Library (Intel®ISA-L)

Full Path: Design Kits ⇒ Platforms & Solutions ⇒ Storage ⇒ Storage and I/O Products: Intel®Intelligent Storage Acceleration Library (Intel®ISA-L) ⇒ Intel®Intelligent Storage Acceleration Library (Intel®ISA-L)

Download the desired library release and the <doc_id>_ISA-L_Key_2.12.pdf document. The .zip decryption key for each release may be found in the <doc_id>_ISA-L_Release_Keys_2.12.pdf document. Unzip/decryption with WinZip is recommended as the linux unzip utilities may not be compatible with the current encryption method used in Intel®ISA-L.

Note: The Intel®ISA-L and .pdf key file are Intel Confidential information and should not be transferred or shared with other company members.

Using the associated key, decrypt the .zip library package then transfer onto a compatible system which meets the requirements in section 2 above. The .zip file will unpack a top level directory with source sub-directories for each algorithm type. Within each sub-directory is the .asm of the optimized core function and .c source for the example test/demo code. Include files (.h) can be found in the /include directory. The .h files also include special commenting used in the API documentation generation for the library.

A detailed overview of the source layout is included in Appendix B.

3.1 Updating Intel®Intelligent Storage Acceleration Library (Intel®ISA-L)

The Intel®ISA-L is expected to be updated on a quarterly basis. New versions can be downloaded from the Storage and I/O: Intel®Intelligent Storage Acceleration Library (Intel®ISA-L) area of IBL and unzipped/decompressed to a

directory of your choosing.

Within a new download you will find a Release_notes.txt containing the Intel®Intelligent Storage Acceleration Library Release Notes. This summary highlights any functions that were added in the new update and any old functions that were updated (new versions). Further details on each function, and which product they will run on can be found in the Intel®Intelligent Storage Acceleration Library API Reference Manual.

The API manual with each release, provides a per function identifier (serial number and versioning) used for Intel®ISA-L tracking. The purpose is to aid tracking and update of the Intel®ISA-L core (.asm) algorithms both in distributions from Intel and within your product source code tree.

Individual functions are given version numbers with the format mm-vv-ssss.

```
mm = Two hex digits indicating the processor a function was optimized for.
00 = Nehalem/Multibinary
01 = Westmere
02 = Sandybridge
03 = Ivy Bridge
04 = Haswell
05 = Silvermont
vv = function version number (specific function version currently in the library)
ssss = function serial number (unique number for every library member)
```

The following table is an example of what the identifier table might look like within each Intel®ISA-L API documentation release.

```
Function_Name                Version
---------------------------  ----------
crc16_t10dif_01              01-05-0010
crc32_ieee_01                01-05-0011
crc32_iscsi_simple           00-01-0012
crc32_iscsi_baseline         00-01-0013
crc32_iscsi_00               00-02-0014
crc32_iscsi_01               01-02-0015
crc16_t10dif_by4             05-01-0016
crc32_ieee_by4               05-01-0017
...
crc16_t10dif                 00-02-011a
crc32_ieee                   00-02-011b
crc32_iscsi                  00-02-011c
crc32_iscsi_base             00-01-011d
crc16_t10dif_base            00-01-011e
crc32_ieee_base              00-01-011f
...
```

In addition to providing the identifier table, the API documentation expands on the instruction set requirements for each algorithm. For example, the crc32_iscsi_01 function requires a processor with the SSE 4.2 and CLMUL instruction support.

As discussed in section 1.1, in general functions are expected to continue to run on newer processors (due to instruction set backward compatibility), but may not take full advantages of the architectural improvements of the newer processor. In some cases, new functions will be created or old functions will be re-tuned for new processors if significant performance benefit is expected.

Upon unpacking a new distribution then:

1. Review the Release_notes.txt to identify any new functions you may want to use and any updates (new versions) of functions already in your code tree.
2. Note the identifiers for any new or updated functions in the Intel®ISA-L library.
3. Query your existing source code tree for existing Intel®ISA-L identifiers.
4. Add new functions to your source code tree as desired.
5. Update old functions with new versions as needed.

CHAPTER 4
BUILDING INTEL®ISA-L

The Intel®Intelligent Storage Acceleration Library build environment has been tested on Linux, Windows 7 and FreeBSD. Here we document the main issues for each of the platforms.

4.1 GNU/Linux

On Linux platforms the Intel®ISA-L library is constructed with the GNU make facility. This follows the convention of a top level make file and additional make files with each sub-directory. The top level make targets are:

```
Makefile include for optimized libraries
make targets:
   lib  - build library of optimized functions
   slib - build shared library
   test - run unit tests of functions
   perf - run performance tests
   clean - remove object files
   other - make per function optional tests
   ex - make some optional examples
```

Note: System commands given in this chapter assume that the user is issuing commands from a bash shell. This is the default shell. Use the echo $0 command to verify use of the bash shell or run /bin/bash to switch to the bash shell.

To build the Intel®Intelligent Storage Acceleration Library and demo programs on em64t architecture open a console and navigate to the top Intel®ISA-L directory.

4.1.1 Make

Running make will build the library placing object files (.o) and the linkable library isa-l.a in the isa-l_src_2.12/bin directory. The top level make file (Makefile) defines a list of algorithm types such as:

units = crc mb_sha1 mb_sha256 mb_sha512 mb_md5 sha1 raid aes_xts_128 aes_xts_256 erasure_code igzip

For each entry in units there is a corresponding sub-directory and Makefile. Running make at the top level creates a library with all Intel®ISA-L contents (isa-l.a)

```
cd /home/<your user name>/isa-l_src_2.12
make
```

4.1.2 Building a sub-set of the library

If only a sub-set of the library is desired, descend and run make from within the desired sub-directory. For example, to make a library of only the raid functions called raid.a:

```
cd /home/<your user name>/isa-l_src_2.12
make clean
cd raid
make
```

Note: Please note you must call make clean in the top level Intel®ISA-L directory before trying to build a subset of the library.

Sample output:

```
make
mkdir bin
  ---> Building pq_gen_sse.asm
  ---> Building pq_check_sse.asm
  ---> Building xor_gen_sse.asm
  ---> Building xor_check_sse.asm
  ---> Creating Lib raid.a
ar: creating raid.a
```

Note: It is necessary to then run a 'make clean' command in this sub-directory before attempting any subsequent make commands in the top-level directory.

4.1.3 Make Clean

At any time make clean can be run from the top level to start over.

```
make clean
```

4.1.4 Make Test

Running make test will build and run core algorithms and example/test code giving a quick indication if the code compiles and runs properly on the system. The associated test executables (have _test in their name) will be placed in the top level directory and object (.o) files will be placed in isa-l_src_2.12/bin.

```
cd /home/<your user name>/isa-l_src_2.12
make clean
make test
```

```
Sample Output:

make test
mkdir bin
  ---> Building crc/crc16_t10dif.asm DEBUG
  ---> Building crc/crc32_ieee.asm DEBUG
  ---> Building mb_sha1/mb_mgr.c DEBUG
  ---> Building mb_sha1/sha1_mult.asm DEBUG
  ---> Building mb_sha1/mb_mgr_sha1_submit.asm DEBUG
  ---> Building mb_sha1/mb_mgr_sha1_flush.asm DEBUG
  ---> Building sha1/sha1.asm DEBUG
  ---> Building raid/pq_gen_sse.asm DEBUG
```

```
  ---> Building raid/pq_check_sse.asm DEBUG
  ---> Building raid/xor_gen_sse.asm DEBUG
  ---> Building raid/xor_check_sse.asm DEBUG
  ---> Building Test crc16_t10dif_test DEBUG

./crc16_t10dif_test
Test crc16_t10diff.....................Test done: Pass
  ---> Building Test crc32_ieee_test DEBUG
./crc32_ieee_test
Test crc32_ieee..............................Test done: Pass
  ---> Building Test multi_buffer_sha1_test DEBUG
./multi_buffer_sha1_test
test done
  ---> Building Test multi_buffer_sha1_rand_test DEBUG
./multi_buffer_sha1_rand_test
mb_sha1 test, 10 sets of 100x1048576 max: .......... mb_sha1 rand: Pass
  ---> Building Test sha1_openssl_test DEBUG
./sha1_openssl_test
Test sha1_openssl ..................................................
..........sha1_test: Pass
  ---> Building Test sha1_reference_test DEBUG
./sha1_reference_test
Test sha1 ..........................................................
...sha1_test: Pass
  ---> Building Test pq_gen_sse_test DEBUG
./pq_gen_sse_test
Test pq_gen_sse_test ................................ done: Pass
  ---> Building Test xor_gen_sse_test DEBUG
./xor_gen_sse_test
Test xor_gen_sse_test ............................done: Pass
  ---> Building Test pq_check_sse_test DEBUG
./pq_check_sse_test
Test pq_check_sse_test 16 sources X 1024 bytes.........Pass
  ---> Building Test xor_check_sse_test DEBUG
./xor_check_sse_test
Test xor_check_sse_test 16 sources X 1024 bytes.............................Pass
Finished running tests
```

Note: Some of the tests require certain instruction sets to be enabled on the platform; e.g. one or more of AVX2, AVX, AES-NI, SSE 3, SSE 4.1, SSE 4.2 or PCLMULQDQ may need to be enabled. If any of these instruction sets are not enabled, some tests will fail with a message similar to the following:

```
make: *** [multi_buffer_sha1_avx_test.run] Illegal instruction (core dumped)
```

To avoid all of the tests failing if some instruction sets are missing, running make with the "-k" option (`make -k test`) will continue to attempt to run subsequent tests if a test fails.

4.1.5 Make Perf

Running make perf builds and runs basic performance measurement examples (having _perf in their names) with default arguments.

```
Sample output:
```

```
make perf
  ---> Building Test multi_buffer_sha1_vs_ossl_perf DEBUG
./multi_buffer_sha1_vs_ossl_perf
sha1_openssl: runtime =     473834 usecs, bandwidth 156 MB in 0.4738 sec = 345.78 MB/s
sha1_mb     : runtime =     209611 usecs, bandwidth 156 MB in 0.2096 sec = 781.64 MB/s
Multi-buffer sha1 test complete 10 buffers of 16384 B with 1000 iterations
Pass functional check
  ---> Building Test sha1_perf DEBUG
./sha1_perf
Test sha1 performance
Make rand data
sha1_reftest: runtime =    2560755 usecs, bandwidth 256 MB in 2.5608 sec = 104.83 MB/s
sha1_openssl: runtime =     780229 usecs, bandwidth 256 MB in 0.7802 sec = 344.05 MB/s
sha1_devtest: runtime =     590878 usecs, bandwidth 256 MB in 0.5909 sec = 454.30 MB/s
Pass func check
  ---> Building Test pq_gen_sse_perf DEBUG
./pq_gen_sse_perf
Test pq_gen_sse_perf 16 sources X 2097152 bytes
pq_gen_sse_cold: runtime =    4409242 usecs, bandwidth 36000 MB in 4.4092 sec = 8561.28 MB/s
  ---> Building Test xor_gen_sse_perf DEBUG
./xor_gen_sse_perf
Test xor_gen_sse_perf 16 sources X 2097152 bytes
xor_gen_sse_cold: runtime =    3445463 usecs, bandwidth 34000 MB in 3.4455 sec = 10347.40 MB/s
```

Note: The library performance applications assume 1MB = 1,000,000 bytes.

4.1.6 Make Other

Running make other builds optional test/demo code. Currently this builds a few useful demo functions for XOR/PQ algorithms and igzip compression: pq_load_test, xor_load_test, pq_gen_sse_compare_perf, igzip_file_perf and igzip_sync_flush_file_perf.

```
Sample output:
make other
  ---> Creating Lib bin/isa-l.a
ar: creating bin/isa-l.a
  ---> Building Test pq_load_test DEBUG
  ---> Building Test xor_load_test DEBUG
  ---> Building Test pq_gen_sse_compare_perf DEBUG
  ---> Building Test igzip_file_perf DEBUG
  ---> Building Test igzip_sync_flush_file_perf DEBUG
```

4.1.7 Make Ex

Running make ex builds some optional example code.

```
Sample output:
make ex
  ---> Building Test crc_simple_test DEBUG
  ---> Building Test multi_buffer_sha1_example DEBUG
```

```
---> Building Test xor_example DEBUG
---> Building Test igzip_example DEBUG
---> Building Test igzip_sync_flush_example DEBUG
---> Building Test igzip_bytes_consumed_example DEBUG
---> Building Test igzip_bytes_consumed_example_with_init DEBUG
```

4.1.8 Make Parameter Overrides

The make file structure accepts other arguments for building library variations.

Note: It is important to make clean between builds when using overrides.

To override the default TEST_SEED value (0x1234) with 0xFFFF (used as a seed for the random number generator in some of the unit tests), execute the following:

```
make clean
make D='TEST_SEED=0xFFFF' test
```

By default, the library code is built using cache cold data blocks. To build and run the performance tests with warm cache data, execute the following:

```
make clean
make D=CACHED_TEST perf
```

The following is specific to raid and erasure_code. By default, the code in these directories is built using non-temporal instructions (to reduce cache-pollution). To build and run the performance tests excluding (not using) non-temporal instructions, execute the following:

```
make clean
make D=NO_NT_LDST perf
```

At this time, only the raid and erasure_code functions accept NO_NT_LDST overrides, so see the Intel®Intelligent Storage Acceleration Library API Reference Manual for your specific release for supported make arguments per supported function.

Specifically for erasure_code, the base functions for GF multiply and divide can be sped up at the expense of memory size by defining GF_LARGE_TABLES. To build and run the erasure_code performance tests using GF_LARGE_TABLES, execute the following:

```
make clean
make D=GF_LARGE_TABLES perf
```

4.1.9 Building and running on 32-bit systems

Some of the Intel®ISA-L code can be built and executed on 32-bit Linux / FreeBSD operating systems. You must affix "arch=32" to the make commands that you execute. For example:

```
make arch=32 slib
make arch=32 test
make arch=32 perf
```

Note: `make arch=32 ex` and `make arch=32 other` are not supported on 32-bit systems.

4.2 FreeBSD

The Intel®ISA-L library has been tested on FreeBSD 9.0 and FreeBSD 9.1. Building the library on FreeBSD proceeds similarly to the GNU/Linux platforms, documented in section 4.1, however there are some important differences to be noted:

- The makefiles present in the Intel®ISA-L library must be built with GNU make, as the make command present by default on FreeBSD will not build the library. GNU make can be added using:

  ```
  pkg_add -r gmake
  ```

- The code and tests can then be built and run using:

  ```
  gmake
  gmake test
  gmake perf
  ```

- To build the new Hashing API and its applications on FreeBSD 9.0 and 9.1 please add the CC=clang prefix as follows:

  ```
  gmake CC=clang
  gmake CC=clang test
  gmake CC=clang perf
  ```

  Note: Please note that if you are building the applications in the Intel®ISA-L top level directory the CC=clang prefix is required to build the new Hashing API. Individual units, excluding the new Hashing API can be built without the prefix.

Note: Support for AVX has only been introduced in FreeBSD release 9.1. Library functions that use AVX instructions will not work on earlier FreeBSD versions. Intel®ISA-L new Hashing API functions support on FreeBSD 9.0 and 9.1 is limited to the use of clang compiler rather than gcc. As with GNU/Linux, some of the tests require other instruction sets to be enabled on the platform as well as AVX, such as AVX2. To avoid all of the tests crashing if some instruction sets are missing, running gmake with the "-k" option (`gmake -k test`) will continue to attempt to run subsequent tests if a test fails.

Note: You must affix "arch=32" to the gmake commands that you execute in order to build and run the 32-bit compatible code.

```
gmake arch=32 slib
gmake arch=32 test
gmake arch=32 perf
```

Note: `make arch=32 ex` and `make arch=32 other` are not supported on 32-bit systems.

4.3 Windows

Building Intel®ISA-L on Windows requires either:

- Visual Studio 2010 or 2012 with Intel Compiler **or**
- Visual Studio 2008 with Intel Compiler and the Microsoft Visual C++ 2008 Redistributable Package (x64) (Redistributable Package can be obtained at http://www.microsoft.com/en-us/download/details.aspx?id=15336)

The Intel 64-bit Compiler Development Environment **must** be used for building the Intel®ISA-L library, otherwise fatal errors may be observed during the build. To build the library:

- Use the "Intel 64 Visual Studio <*year*> mode" command prompt to build and run the code. **Do not use** normal Visual Studio command prompt or the Windows command prompt. Do not open the library in Visual Studio.
- Alternatively, the Intel 64-bit Compiler Development environment can be enabled by invoking the compilervars_arch.bat script in Intel Composer's \bin directory with the appropriate architecture and Visual Studio version arguments, e.g.

```
C:\> "C:\Program Files (x86)\Intel\Composer XE 2013\bin\compilervars_arch.bat"  intel64 vs2012
```

Note: The YASM executable in the YASM installation directory should be renamed from yasm(version)win64.exe to yasm.exe and the containing directory should be added to the system path.

The Intel®ISA-L library must be built with nmake. A makefile for building the Intel®ISA-L library with nmake (Makefile.nmake) is provided in the top-level directory. The library file, isa-l.lib, can be built at the top-level directory in the "Intel 64 Visual Studio <*year*> mode" command prompt window as follows:

```
cd C:\Users\<your user name>\isa-l_src_2.12
nmake -f Makefile.nmake clean
nmake -f Makefile.nmake lib
```

The validation tests and examples can be built as follows:

```
nmake -f Makefile.nmake tests
nmake -f Makefile.nmake ex
```

To run a validation test or an example simply call the built .exe, for example:

```
c:\Users\<your user name>\isa-l_src_2.12>aes_xts_128_rand.exe
aes_xts_128 enc/dec rand test, 10 sets of 1048576 max:  ..........Pass
```

Note: Some of the validation tests require Zlib or OpenSSL to be installed. Prior to building the validation tests, the variables: libpath, libinc, zlibpath and zlibinc need to be modified in Makefile.nmake with the appropriate paths. For instructions on building 64-bit Zlib on Windows please see Appendix C.

Note: Performance tests are not currently supported on Windows platforms.

CHAPTER 5
RAID DEMO FUNCTIONS

While the optimized core functions take arguments, most of the Intel®ISA-L demo functions do not. So while the core optimized (.asm) algorithms take parameters, the demo C-code that shows how to call and use the core functions are fixed examples. Refer to the Intel®Intelligent Storage Acceleration Library API Reference Manual for complete parameters for all functions.

The current exceptions are the xor_load_test and pq_load_test functions related to RAID processing. Command line arguments for these functions are designed to make it easy to experiment with a wide variety of cases that come up in RAID processing. Executing these functions with -h will print out the command line options available.

```
./xor_load_test -h

Sample output:

./xor_load_test: [options]
-q                    - more quiet (can be repeated)
-p <secs>           - set pause time
-d <duty cycle>     - set duty cycle ratio
-b <bw/s> [G,M,K] - adjust for set memory bandwidth instead
-s <sources>        - number of sources
-n <len> [M,K]      - length of each source
-l <loops>          - initial number of loops
```

5.1 Pin a Test to a Core

On linux, the taskset command can be used to pin the test to a specific core. For example, to launch xor_load_test on core 2:

```
taskset -c 2 ./xor_load_test -p 0
```

5.2 Usage Examples:

By default ./xor_load_test and ./pq_load_test adjust duty cycle to load the core it is on at the 50% level. Command line parameters can be used to control target bandwidth (how much cpu to use) and other important parameters such as number of sources or size of the data block to use in the RAID calculation.

5.2.1 Bandwidth Control

Use the -p, -d, and -b parameters to select a variety of additional bandwidth control options:

```
./xor_load_test  -p 0              ;Runs max bandwidth on the core (no limit)
```

```
./xor_load_test  -p 1 -d 0.25      ;Attempt a 25\% duty cycle on 1 second period
./xor_load_test  -b 1 G            ;Adjust duty to about 1 GB/s bandwidth
```

5.2.2 Alter Number of Source buffers and Data Block Size

Us the -s parameter to vary the number of source buffers and -n to vary the size of the data block processed. A greater number of sources requires more processing.

Note: The default build of these functions makes sure the data blocks are cache cold (cache warm builds are also possible, see section 4.1.8).

```
./pq_load_test  -p 0  -s 8               ;8 sources, no bandwidth limit
./pq_load_test  -p 0  -s 16 -n 8 K  ;16 sources, 8 K blocks
```

CHAPTER 6
INTEGRATING THE INTEL®ISA-L INTO YOUR CODE BASE

After becoming familiar with the demo code for the library you may move on to integrating functions into your own code tree. There are some key considerations to keep in mind as you integrate the algorithms into your environment.

1. Users are typically not expected to alter the core algorithm code (.asm) as it is hand tuned assembly language and alteration often will likely destroy the optimization. If your compiler requires a unique calling convention, contact your Intel field representative to help get consultation from the Intel®ISA-L technical team.
2. The C-language demo code (and core functions) are run in user space for demonstration purposes and are a useful starting point to see how to call and use the core (.asm) functions . The demo code is not a complete production solution. Your code that will call the core functions will need to deal with OS/application specific issues such as memory allocation, exception/fault handling, memory barriers, etc.

APPENDIX A
INTEL®ISA-L DEVELOPMENT SYSTEMS

The purpose of this section is to provide an overview of the systems used inside Intel for Intel®ISA-L development and test. The intent is as a point of reference for library users such that similar performance results can be recreated. The essential factors for the Intel®ISA-L algorithms are processor speed/SKU, number and speed of memory channels, and basic OS/system software. Of course, observed performance will decrease when other concurrent system activity consumes or competes for resources such as cache or memory bandwidth.

Jasper Forest:

- OS: Linux 3.9.10-100.fc17.x86_64
- Intel®Xeon®CPU C5528 @ 2.13GHz
- 4 Core (8 Thread), 8M LLC, 4.8 GT/s QPI
- 3x DDR3 memory channels, each populated with 1x 4GB 1333 DIMM/channel
- Total memory: 12 GB

Westmere-EP:

- OS: Linux 3.9.10-100.fc17.x86_64
- Intel®Xeon®CPU X5680 @ 3.33GHZ
- 6 Core (12 Thread), 12M LLC, 6.4 GT/s QPI
- 3x DDR3 memory channels, each populated with 1x 8GB 1600 DIMM/channel
- Total memory: 24 GB

Sandy Bridge-EP:

- OS: Linux 3.9.10-100.fc17.x86_64
- Intel®Xeon®CPU E5-2680 @ 2.70GHz
- 8 Core (16 Thread), 20M LLC, 8 GT/s QPI
- 4x DDR3 memory channels, each populated with 1x 8GB 1600 DIMM/channel
- Total memory: 32GB

Ivy Bridge-EP:

- OS: Linux 3.9.10-100.fc17.x86_64

- Intel®Xeon®CPU E5-2680v2 @ 2.80GHz
- 10 Core (20 Thread), 25M LLC, 8 GT/s QPI
- 4x DDR3 memory channels, each populated with 1x 8GB 1600 DIMM/channel
- Total memory: 32GB

Haswell:

- OS: Linux 3.7.6-201.fc18.x86_64
- Intel®Core™i7-4700EQ CPU @ 2.40GHz
- 4 core (8 Threads), 6MB LLC
- 2x DDR3 memory channels, each populated with 1x 4GB 1600 DIMM/channel
- Total memory: 8GB

APPENDIX B
CODE LAYOUT

The top level directory of the code contains the makefiles, and subdirectories for each class of function. Each subdirectory contains the .asm files with the optimized core functions, and the .c source files with the test and demo code. The general directory structure is shown below:

```
aes_xts_128/
aes_xts_256/
erasure_code/
crc/
igzip/
include/
mb_md5/
mb_sha1/
mb_sha256/
mb_sha512/
md5_mb/
raid/
sha1/
sha1_mb/
sha256_mb/
sha512_mb/
```

APPENDIX C
BUILDING 64-BIT ZLIB FOR WINDOWS

This section provides a step by step guide to building a 64-bit Zlib 1.2.5 library on Windows. Please replace anything in angle brakets <> with the appropriate path or number. This Guide was written with reference from http://blog.sinzy.net/hjk41/entry/22891.

1. Download zlib125.zip from: http://www.winimage.com/zLibDll/index.html
2. Extract the .zip file.
3. Open command prompt.
4. cd c:\<*path to Visual Studio installation directory*>\Microsoft Visual Studio <*version number*>\VC\bin\amd64
5. Execute vcvars64.bat.
6. Add c:\<*path to Visual Studio installation directory*>\Microsoft Visual Studio <*version number*>\VC\bin\amd64 to the System Path environment variable.
7. cd <*zlib 1.2.5 directory*>\zlib-1.2.5\contrib\masmx64
8. Execute bld_ml64.bat to compile 64-bit binaries for the assembly files:inffasx64.asm and gvmat64.asm.
9. cd <*zlib 1.2.5 directory*>\zlib-1.2.5\contrib\vstudio\vc10
10. Open the solution file zlibvc.sln with Visual Studio. If you are not immediately prompted to update the project, right-click on "solution 'zlibvc'" in the "solution explorer" pane and select to update the project.
11. If the version number in "zlibvc.def" is in the format "VERSION 1.2.5", modify it to read "VERSION 1.25". Otherwise the compilation will fail, as Visual Studio seems to support only Major.Minor numbers.
12. Right-click on "solution 'zlibvc'" in the "solution explorer" pane and select "Properties".
13. Under "Configuration Properties", ensure that the configuration is set to "Release" and the platform is set to "x64".
14. Build the solution.
15. All the dll and lib files should now be in <*zlib 1.2.5 directory*>\zlib-1.2.5\contrib\vstudio\vc10\x64\ZlibDllRelease.
16. Modify Makefile.nmake in the top-level Intel®ISA-L directory with the following (quotes around the zlibpath and zlibinc paths are needed if there is any whitespace in the paths):

    zlibpath = "<*zlib 1.2.5 directory*>\zlib-1.2.5\contrib\vstudio\vc10\x64\ZlibDllRelease"

    zlibinc = "<*zlib 1.2.5 directory*>\zlib-1.2.5"

    Replace references to zlib.lib with zlibwapi.lib

APPENDIX D
INCLUDING OPENSSL VERSION 1.0.1

The AES-XTS random and performance tests require OpenSSL 1.0.1, which will need to be built from source and explicitly included if not available on the system. Here we outline how this may be carried out.

D.1 Linux

On Fedora 17 and earlier versions, OpenSSL 1.0.1 is not available on the system by default. It can be built by obtaining the source code and building from a suitable location as follows:

```
$ wget -N http://www.openssl.org/source/openssl-1.0.1e.tar.gz
$ tar xvfz openssl-1.0.1e.tar.gz
$ cd openssl-1.0.1e
$ ./config shared
$ make
$ make install                          ; root privileges required
```

This will install the OpenSSL libraries and headers to the default location /usr/local/ssl. It may be necessary on some Fedora systems to update the dynamic linker with the location of the shared libraries as follows (root privileges required):

```
$ echo /usr/local/ssl/lib > /etc/ld.so.conf.d/openssl.conf
$ ldconfig
```

The new version of OpenSSL will need to be included by the AES-XTS test code by modifying the dependency rules for the affected tests in the aes_xts_128/Makefile and aes_xts_256/Makefile. This can be done by replacing the following lines

```
aes_xts_128_enc_ossl_perf: LDLIBS += -lcrypto
aes_xts_128_dec_ossl_perf: LDLIBS += -lcrypto
aes_xts_128_rand_ossl_test: LDLIBS += -lcrypto
```

with the following

```
aes_xts_128_enc_ossl_perf: LDLIBS += -I/usr/local/ssl/include -L/usr/local/ssl/lib -lcrypto
aes_xts_128_dec_ossl_perf: LDLIBS += -I/usr/local/ssl/include -L/usr/local/ssl/lib -lcrypto
aes_xts_128_rand_ossl_test: LDLIBS += -I/usr/local/ssl/include -L/usr/local/ssl/lib -lcrypto
```

The 'make test' and 'make perf' commands will then build these tests using the new OpenSSL version.

D.2 FreeBSD

A similar approach to that above can be followed on FreeBSD by including either a source-built implementation as above, or by including the location of the BSD ports installation (e.g. /usr/local/include and /usr/local/lib) instead.

D.3 Windows

For Windows, binaries for OpenSSL are available via http://www.openssl.org/related/binaries.html. Intel®ISA-L has been tested using Win64 OpenSSL v1.0.1e binary, which will install the libraries and headers to a directory of choice on your system (**Note:** A message may appear during the OpenSSL installation, warning that Visual C++ 2008 Redistributables are missing - this can be disregarded).

In order to include OpenSSL it is then necessary to replace the libpath and libinc variables in Makefile.nmake to specify these locations, e.g.

```
libpath  = C:\OpenSSL-Win64\lib                          #set to ossl path for tests
libinc   = C:\OpenSSL-Win64\include
```

The tests can then be built according to the process in Section 4.3.

Note: Performance tests are not currently supported on Windows platforms.