OpenCL 0.7.1

Jack Lloyd (lloyd@acm.jhu.edu)

April 29, 2001

Contents

1 Introduction

OpenCL is a class library, written in ISO C++, which attempts to provide the most common cryptographic algorithms in an easy to use and portable package. While you are reading this, you may want to refer to the files `opencl.h` and `filters.h`. These files contain the classes that form the basic interface for the library.

All declarations in the library are contained within the namespace `OpenCL`. OpenCL declares several typedef'ed types to help buffer it against changes in machine architecture. These types are used extensively in the interface, and thus it would be often be convenient to use them without the `OpenCL::` prefix. You can, by `using` the namespace `OpenCL_types` (this way you can use the type names without the namespace prefix, but the remainder of the library stays out of the global namespace). The included types are `byte` and `u32bit`, which are unsigned integer types.

2 Building The Library

These are the basic steps for compiling OpenCL on a Unix or Unix-like system:

```
$ ./configure.pl CC-OS-CPU
$ gmake
$ gmake check # optional, but recommended
# gmake INSTALLROOT=path install # default INSTALLROOT=/usr/local
```

The `./configure.pl` script is written in Perl, and the Makefile requires GNU make. It is shown above as 'gmake', which will work for most systems. To get the list of values for `CC`, `OS`, and `CPU` that `./configure.pl` supports, run it with the "`--help`" option.

Note that you should only select the 64-bit version of a CPU (like "sparc64" or "mips64") if your operating system knows how to handle 64-bit object code – a 32-bit kernel on a 64-bit CPU will generally not like 64-bit code. This restriction also applies to submodels, ie, gcc-solaris-ultra2 will not work unless you're running a 64-bit Solaris kernel (for 32-bit Solaris, you want gcc-solaris-v9).

Note that not all OSes or CPUs have specific support. If your CPU architecture isn't supported by `./configure.pl`, use 'generic'. This simply disables machine-specific optimization flags. Similarly, setting OS to 'generic' disables things which depend greatly on OS support (specifically, shared libraries).

However, it's impossible to guess which options to give to a system compiler. Thus, if you want to compile OpenCL with a compiler which `./configure.pl` does not support, fill out the bug report form (linked off the OpenCL web page), giving enough information to update the script (preferably, mail the man pages for the C and C++ compilers and the linker).

2.1 Configuration Parameters

There are some configuration parameters which you may want to tweak before building the library. These can be found in `config.h`.

`DEFAULT_BUFFERSIZE`: This constant is used as the size of buffers throughout OpenCL. A good rule of thumb would be to use the page size of your machine.

3 The Basic Interface

OpenCL has two different interfaces. The one documented in this section is meant more for implementing higher-level types (see the section on filters, below) than for use by applications.

Virtually every class listed here implements the function `void` **clear()**, which destroys any sensitive data contained within the object and returns it to it's initial state. This is called by the destructor of each class, so usually you will not have to call it, but occasionally it may be useful.

3.1 Symmetrically Keyed Algorithms

Block ciphers, stream ciphers, and MACs all handle keys in pretty much the same way. To make this similiarity explicit, all algorithms of those types are derived from the `SymmetricAlgorithm` base class. This type has only two functions:

`void` **set_key**(`const byte` *key*[], `u32bit` *length*):

Most algorithms only accept keys of certain lengths. If you attempt to call **set_key** with a key length that is not supported, the exception `InvalidKeyLength` will be thrown.

Objects of this type have a parameter KEYLENGTH, which gives one value that the algorithm supports. This will generally be the largest key length possible (or 256 bits, whichever is less). Unless you have a reason to do otherwise, using KEYLENGTH as the length of the key is recommended.

`bool` **valid_keylength**(`u32bit` *length*) const:

This function returns true if a key of the given length will be accepted by the cipher.

3.2 Block Ciphers

Block ciphers implement the inferface `BlockCipher`, found in `opencl.h`.

`void` **encrypt**(`const byte` *in*[BLOCKSIZE], `byte` *out*[BLOCKSIZE]) const
`void` **encrypt**(`byte` *block*[BLOCKSIZE]) const

These functions apply the block cipher transformation to *in* and place the result in *out*, or encrypts *block* in place (*in* may be the same as *out*). BLOCKSIZE is a constant which specifies how much data a block cipher can process at one time.

`BlockCiphers` have similar functions **decrypt**, which perform the inverse operation. All variations of **encrypt** and **decrypt** assume that **set_key** has already been called.

Block ciphers implement the `SymmetricAlgorithm` interface.

3.3 Stream Ciphers

Stream ciphers are somewhat different from block ciphers, in that encrypting data results in changing the internal state of the cipher. Also, you may encrypt any length of data in one go (in byte amounts).

`void` **encrypt**(`const byte` *in*[], `byte` *out*[], `u32bit` *length*)
`void` **encrypt**(`byte` *data*[], `u32bit` *length*):

These function encrypt the arbitrary length (well, less than 4 gigabyte) string *in* and place it in *out*, or encrypt it in place in *data*. The **decrypt** functions look just like **encrypt**.

Stream ciphers implement the `SymmetricAlgorithm` interface.

Some stream ciphers support random access to any point in their cipher stream. These ciphers have the interface `RandomAccessStreamCipher` (which is derived from `StreamCipher`). For these ciphers, calling `void` **seek**(`u32bit` *byte*) will change the ciphers state so that it as if the cipher and been keyed as normal, then encrypted $byte - 1$ bytes of data (so the next byte in the cipher stream is byte number *byte*).

3.4 Hash Functions

Hash functions take their input without producing any output, only producing anything when all input has already taken place.

`void` **update**(`const byte` *input*[], `u32bit` *length*):

Update the hash function with *length* more bytes in *input*.

`void` **final**(`byte` *hash*[HASHLENGTH]):

Complete the hash calculation and place the result into *hash*. HASHLENGTH is a public constant in each class that gives the length of the hash in bytes. After you call **final**, the hash function is reset to it's initial state, so it may be reused immediately.

3.5 Message Authentication Codes

A MAC is essentially a keyed hash function, so classes derived from `MessageAuthCode` have **update** and **final** classes just like a `HashFunction` (and like a `HashFunction`, after **final** is called, it can be used to make a new MAC right away; the key is kept around). One minor difference is that the constant called HASHLENGTH in a hash function is called MACLENGTH in a MAC object.

Additionally, a MAC has the `SymmetricAlgorithm` interface (a MAC should be keyed before being used).

3.6 Random Number Generators

The random number generators provided in OpenCL are meant for creating keys, IVs, padding, nonces, and anything else which requires 'random' data. It is important to remember that the output of these classes will vary, even if they are supplied with exactly the same seed (ie, two Randpools with similar initial states will not produce the same output).

To ensure good quality output, you need to seed the RNG with truly 'random' data, such as timing data from hardware. Preferably, you should use an `EntropySource` (see below).

To add entropy to a RNG, you can use `void` **add_entropy**(`const byte` *data*[], `u32bit` *length*) or the `EntropySource` interface.

One a RNG has been initialized, you can get a single byte of random data by calling `byte` **random()**, or get a large block by calling `void` **randomize**(`byte` *data*[], `u32bit` *length*), which will put random bytes into each member of the array from indexes $0 \ldots length - 1$.

3.6.1 Randpool

`Randpool` is based around a large pool of data and a hash function (usually, MD4). `Randpool` is slower than `X917`, but can easily satisfy any reasonable demand (on a 350 MHz Pentium II `Randpool` can produce over 100K of data per second). Because the internal state of `Randpool` is much larger than `X917`, it is more likely to be secure, and it is recommended that `Randpool` be used over `X917` in most cases.

3.6.2 X917

`X917` is based on the ANSI X9.17 standard, which makes use of a block cipher. `X917` is quite a bit faster than `Randpool` (depending on the block cipher used). Using `Square`, usually one of the fastest ciphers in the suite, `X917` is about 7 times faster than `Randpool`. The version used is a variant of the normal X9.17; most importantly, only 1/2 of the output of the block cipher is actually given to the caller (then a new block is computed), the timestamp is encrypted in CBC mode instead of ECB mode, and that after a `X917` object has generated a certain number of blocks (normally set to 32), it will automatically rekey itself using it's internal state. These alterations make any attack much harder (at the cost of reducing speed). `X917` is a template which takes a block cipher as an parameter.

3.6.3 Entropy Sources

For a RNG to be secure, it has to be seeded with some amount of "truly random" data. Because OpenCL is designed to be platform independent, no routines for this are provided. However, OpenCL does have an `EntropySource` interface, which can be subclassed for each individual platform. Some OSes (such as Linux, FreeBSD, and Windows NT) provide easy ways to get random data from the OS itself, while others (like most commercial Unices) will require very specialized code. `EntropySource`s for some systems can be found on the OpenCL FTP site.

The only function in the `EntropySource` interface is `u32bit` **get_entropy**(`byte` *data*[], `u32bit` *length*). Basically you pass an `EntropySource` written for your particular system to a `RandomNumberGenerator`, by calling the function `void` **add_entropy**(`EntropySource&` *source*, `bool` *slow*). This tells the RNG to ask the `EntropySource` for some amount of random data. It passes it an array where it wants the entropy stored, and a length parameter telling it how much it wants. If *slow* is `true`, then the RNG assumes that your application doesn't have anything pressing to do, and basically tells the `EntropySource` "Go nuts, I've got all the time in the world". Doing this at least once (when your application starts up, for example) is a *highly* recommended idea, because it ensures that your RNG is in a good (ie, highly random and unpredictable) state.

The `EntropySource` shouldn't write more data into the array than was requested (otherwise memory not owned by the process will probably be written to), however, it can write less (for instance, if it cannot collect enough data, or collecting that much would take too long). For this reason, the **get_entropy** function returns how much data it actually wrote (which should never be more than the *length* field passed to the function).

Note for writers of EntropySources: it isn't necessary to use any kind of cryptographic hash on your output. The data produced by an EntropySource is only used by an application after it has been hashed by the `RandomNumberGenerator` which asked for the entropy, and thus any hashing you do will be wasteful of both CPU cycles and possibly entropy.

3.7 Miscellaneous

This section has documentation for anything that just didn't fit into any of the major categories.

3.7.1 Checksums

Checksums are very similar to hash functions, and in fact share the same interface. However, there are some significant differences, the major ones being that the output size is very small (generally in the range of 2 to 4 bytes), and is not designed to be cryptographically secure. But for their intended purpose (error checking), they perform very well. Some examples of checksums included in OpenCL are the Adler32 and CRC32 checksums.

3.7.2 Exceptions

Sooner or later, something is going to go wrong. OpenCL's behavior when something unusual occurs, like most C++ software, is to throw an exception. Exceptions in OpenCL are derived from it's `Exception` class. You can see most of the major varieties of exceptions used in OpenCL by looking at `exceptn.h`. The only function you really need to concern yourself with is `const char*` **what()**. This will return an error message relevant to the error that occured. For example:

```
try {
   // various OpenCL operations
   }
catch(OpenCL::Exception& e)
   {
   std::cout << "OpenCL exception caught: " << e.what() << std::endl;
   // error handling, or just abort
   }
```

OpenCL's exceptions are derived from `std::exception`, so you don't need to explicitly check for OpenCL exceptions.

3.7.3 Secure Memory

A major concern with mixing modern multiuser OSes and cryptographic code is that at any time the code (including secret keys) could be swapped to disk, where it can later be read by an attacker (unless, of course, you're using OpenBSD, which encrypts whatever it writes to swap `<g>`). OpenCL stores almost everything (and especially anything sensitive) in memory buffers which a) clear out their contents when their destructors are called, and b) have easy plugins for various memory locking functions, such as the **mlock**(2) call on many Unix systems.

These classes should also be used within your own code for storing sensitive data. They are only meant for primitive data types (int, long, etc): if you want a container of higher level OpenCL objects, you can just use a `std::vector`, since these objects know how to clear themselves when they are destroyed. You cannot, however, have a `std::vector` (or any other container) of `Pipe`s or `Filter`s, because these types have pointers to other `Filter`s, and implementing copy constructors for these types would be both hard and quite expensive.

These types are not described in any great detail: for more information, consult the files `secmem.h` and `secalloc.h`.

SecureBuffer is a simple array type, whose size is specified at compile time. It will automatically convert to an array of the appropriate type, and has various useful functions, including **clear()**, and u32bit **size()**, which returns the length of the array. It is a template that takes as parameters a type, and a constant integer which is how long the array is (for example: `SecureBuffer<byte, 8> key;`).

SecureVector is a variable length array. It's size can be increased or decreased as need be, and it has a wide variety of functions useful for copying data into it's buffer. Like SecureBuffer, it implements **clear** and **size**.

You can change the allocation functions used by replacing (or editing) the file secalloc.h. This file contains the template SecureAllocator which is used by SecureBuffer and SecureVector to allocate and deallocate their memory (the idea is quite similiar to the STL allocators).

Memory locking can be implemented by editing the functions **lock_mem** and **unlock_mem** in util.cpp. These functions take a void* and a length argument; the intent is that if memory locking is available, the block of memory starting at the pointer and extending for the specified number of bytes should be locked, if possible (they're also useful fo profiling memory use).

3.7.4 Timers

OpenCL includes a pair of functions, **system_time** and **system_clock**, which are used extensively in some areas of the code (especially in the random number generators). These functions by default use **std::time** and **std::clock**, but often you can do better with system-dependant functions, especially for the system clock (for example, returning the microseconds value from **gettimeofday** is far superior).

4 Filters

4.1 Basic Filter Usage

Up until this point, using OpenCL would be very tedious; to do anything you would have to bother with putting data into arrays, doing whatever you want with it, and then sending it someplace. The filter metaphor (defining a series of operations which take some amount of input, process it, then send it along to the next filter) works very well in this situation. If you've ever used a Unix system, the usage of filters in OpenCL should be very intuitive (and even if you haven't, don't worry, it's pretty easy). For instance, here is how you encrypt a file with Blowfish in CBC mode with padding, then encode it with Base64 and send it to standard output (we assume you have already created key and iv, probably using one of OpenCL's RandomNumberGenerator types, and that file is an open istream):

```
Pipe encryptor(new CBC_wPadding_Encryption<Blowfish>(key, iv),
               new Base64Encoder);
file >> encryptor;
encryptor.close(); // flush buffers, complete computations
cout << encryptor;
```

Pipe works in conjunction with the Filter class (for example, the CBC_wPadding_Encryption and Base64Encoder types used above are Filters), but you should never have to deal with them directly (in fact, it's a distinctly bad idea, as Pipe and Filter are closely tied and rely on knowing a great deal about each other to work correctly).

Some useful functions in `Pipe` not shown above are `u32bit` **remaining()**, which returns how many bytes are available for immediate reading, and several different I/O functions. Using the `istream/ostream` operators above, all data available (either stored in the pipe or until EOF from an istream) is taken out. Sometimes, you want only a small amount at a time, or you are getting your input in small chunks, in which case you can use `u32bit` **read**(`byte` *output*[], `u32bit` *len*), which will read up to *len* bytes into *output* and return how many bytes were actually written into the array, and `void` **write**(`byte` *input*[], `u32bit` *len*), which writes *len* bytes from *input* into the pipe. There are also versions of **read** and **write** which take a single byte as their argument, as convenience functions. You can see the complete declaration for `Pipe` in `filebase.h`

You can reuse a `Pipe` by calling it's **reset** function, which restores a pipe to it's initial state of no filters (writing into the `Pipe` and then reading will give you back your data unchanged). This usually isn't too useful, so you can use `void` **attach**(`Filter*` *filter*) to attach a new filter onto the pipe again. You can call **attach** as many times as you like; each filter added will be attached to the end of the current set of filters (note that you can use **attach** even if you haven't called **reset**). Calling **reset** will also destroy any output currently stored in the pipe.

One last point: if you call **attach**, everything which has previously been processed by the `Pipe` remains as-is. To prevent various problems, if you are calling **attach** on a `Pipe` which already has had input written into it (and you haven't reset the `Pipe`), you should call **close** first (this may not be exactly what you're looking for either: different filters have different semantics when they are closed).

4.2 Fork

It's fairly common that you might recieve some data and want to perform more than one operation on it (ie, encrypt it with DES and calculate the MD5 hash of the plaintext at the same time). That's where `Fork` comes in. `Fork` is a filter that takes it's input and passes it on to *one or more* `Filter`s which are attached to it.

`Fork` changes the nature of the pipe system completely. Instead of being a linked list, it becomes a tree. Each "leaf" of this tree has it's own output buffer. When you read data from the pipe, your request for a read passes through all the `Filter`s in the pipe until it reaches the end, whereupon your data is retrieved from an output buffer. Obviously, if `Fork` forwarded your request to read to all it's "children", confusion would result: `Filter`s would be stepping on each other's toes (and output), as they all try to write into your buffer. Also, what value should **remaining()** return?

The solution to this dilemma is that you have to inform `Fork` what you want it to do. You do this by calling the function `void` **set_port**(`u32bit` *port*). The *port* specifies which `Filter` it is that `Fork` should pass read requests on to. You can find out how many ports there are by calling `u32bit` **total_ports()** (valid port numbers range from $0 \ldots n-1$, where n is the return value of **total_ports()**), and the currently selected port from `u32bit` **current_port()**. Generally, after you have finished entering input, you will iterate through all ports reading the output.

Since you will have to call **set_port** when using `Fork`, you need to keep a pointer to the `Fork`s you are using, rather than simply calling `new` in the call to the constructor. See the section "A Filter Example" for an example of using `Fork`.

4.3 Chain

`Chain` is about as simple as it gets. `Chain` creates a chain of `Filter`s and encapsulates them inside a single filter (itself). This is primarily useful for passing a sequence of filters into something which is expecting only a single Filter (most notably, Fork). You can call Chain's constructor with up to 4 Filter*s (they will be added in order), or with an array of Filter*s and a u32bit which tells Chain how many Filter*s are in the array (again, they will be attached in order).

See the next section for an example of using Chain.

4.4 A Filter Example

Here is some code which takes one or more filenames as it's argument and calculates the result of several hash functions for each file. This code can be found as hasher.cpp in the OpenCL distribution.

```
#include <fstream>
#include <string>
#include <opencl/md5.h>
#include <opencl/sha1.h>
#include <opencl/rmd160.h>
#include <opencl/encoder.h>
using namespace OpenCL_types;

int main(int argc, char* argv[])
{
   if(argc < 2)
      {
      std::cout << "Usage: hasher <filenames>" << std::endl;
      return 1;
      }

   const u32bit COUNT = 3;
   OpenCL::Filter* hash[COUNT] = {
      new OpenCL::Chain(new OpenCL::HashFilter<OpenCL::MD5>,
                        new OpenCL::HexEncoder),
      new OpenCL::Chain(new OpenCL::HashFilter<OpenCL::SHA1>,
                        new OpenCL::HexEncoder),
      new OpenCL::Chain(new OpenCL::HashFilter<OpenCL::RIPEMD160>,
                        new OpenCL::HexEncoder) };
   std::string name[COUNT] = { "MD5", "SHA-1", "RIPE-MD160" };

   OpenCL::Fork* fork = new OpenCL::Fork(hash, COUNT);
   OpenCL::Pipe pipe(fork);

   for(u32bit j = 1; argv[j] != 0; j++)
      {
      std::ifstream file(argv[j]);
      if(!file)
         {
         std::cout << "ERROR: could not open " << argv[j] << std::endl;
         continue;
```

```
        }
      file >> pipe;
      file.close();
      pipe.close();
      for(u32bit k = 0; k != COUNT; k++)
        {
        fork->set_port(k);
        std::cout << name[k] << "(" << argv[j] << ") = " << pipe << std::endl;
        }
      }
   return 0;
}
```

4.5 Rolling Your Own

Well, now that you know how filters work in OpenCL, you probably want to write your own. Lucky for you, all of the hard work is done by the `Filter` base class, leaving you to handle the details of what your filter is supposed to do. The first thing is to make sure to do is derive your filter from OpenCL's `Filter` class. Remember that if you get confused about any of this, you can always look at the implementation of OpenCL's filters to see exactly how everything works.

There are basically only three functions that a filter need worry about:

`void` **write**(`byte` *input*[], `u32bit` *length*):

The **write** function is what is called when a filter receives input for it to process. The filter is *not* required to process it right away; many filters buffer their input before producing any output. A filter will generally have **write** called many times during it's lifetime.

`void` **send**(`byte` *output*[], `u32bit` *length*):

Eventually, a filter will want to produce some output to send along to the next filter in the pipeline. It does so by calling **send** with whatever it wants to send along to the next filter.

`void` **final()**:

Implementing the **final** function is optional. It is called when it has been requested that filters finish up their computations. Note that they should *not* deallocate their resources; this should be done by their destructor. They should simply finish up with whatever computation they have been working on (for example, a compressing filter would flush the compressor and **send** the final block), and empty any buffers in preparation for processing a fresh new set of input.

Additionally, if necessary filters should define a constructor that takes any needed arguments, and a destructor to deal with deallocating memory, closing files, etc.

4.6 The Filter Directory

This section contains descriptions of every `Filter` included in OpenCL.

4.6.1 Keys

Key handling is a little different for `Filters`. Instead of a (very low level) interface of an array of bytes and a length parameter, they use a (slightly higher level) `SymmetricKey`. The major functions of this type are it's constructors:

SymmetricKey(`RandomNumberGenerator&` *rng*, `u32bit` *length*):
This constructor takes *length* bytes of output from *rng* and uses it for a key.

SymmetricKey(`const byte` *input*[], `u32bit` *length*):
This constructor simply copies it's input.

SymmetricKey(`const std::string` *str*)
The argument *str* is assumed to be a hex string; it is converted to binary and stored in the key.

Synonyms for this type include `BlockCipherKey`, `StreamCipherKey`, `MACKey`, and `BlockCipherModeIV` (they're all exactly the same thing, the different names just makes it clear what the bytes are being used for).

4.6.2 Basic Wrappers

Stream ciphers, hash functions, and MACs don't need anything special when it comes to filters. Stream ciphers take their input, encrypt it, and send it along to the next `Filter`. Hash functions and MACs both take their input and produce no output until **final()** is called, at which time they complete the hash and send that as output.

StreamCipherFilter(`const StreamCipherKey&` *key*)

The constructor for a `StreamCipherFilter` takes the a key, which it will pass pretty much directly on to the `StreamCipher` being used, so if the key length is inappropriate for the cipher used, `InvalidKeyLength` will be thrown.

HashFilter(`u32bit` *outlength*):

The constructor for a `HashFilter` takes only one argument, which is a number specifying how much of the hash should be produced as output. Sometimes, you will only want to use (say) half of the hash, and this mechanism lets you do this easily. It defaults to the full size of the hash. If *outlength* is greater than the size of the output, the full hash is used.

MACFilter(`const MACKey&` *key*, `u32bit` *outlength*):

The constructor for a `MACFilter` takes a key, used in calculating the MAC, and a length parameter, which has semantics exactly the same as the one passed to `HashFilter`s constructor.

All three of `StreamCipherFilter`, `HashFilter`, and `MACFilter` are templates, which take the desired type as an parameter. These filters can be found in `filters.h`.

4.6.3 Cipher Modes

For block ciphers, the situation is more complicated. Because ECB mode is dangerous (a message may be easily altered without detection and similar plaintext encrypts to similar ciphertext), block ciphers must be used in a different mode. The modes provided with OpenCL are CBC with padding (see RFC 2040), CFB, OFB, and Counter. All are templates, taking a `BlockCipher` as a parameter. They are presented in `modes.h`, and derive from `CipherMode`, which is a subclass of `Filter` (this allows you to enforce that a particular filter is, in fact, a block cipher mode, without having special cases for each mode).

Only their constructors are interesting; other than that they are just like any other filter, and are used like any other discussed in this documentation.

CBC_wPadding_Encryption(`const BlockCipherKey&` *key*, `const BlockCipherModeIV&` *iv*):

This is quite simple: a key, of a length suitable for the cipher, and an IV, which is the size of the cipher block (if it is not an exception will be thrown). The constructor for `CBC_wPadding_Decryption` is exactly the same.

CFB_Encryption(`const BlockCipherKey&` *key*, `const BlockCipherModeIV&` *iv*, `u32bit` *feedback*):

This is just like CBC, but it takes an (optional) additional argument, the size of the feedback (it will default to the full blocksize of the cipher being used). This value is given in bytes, and can range from 1 to BLOCKSIZE. If the *feedback* is not a valid amount, `InvalidArgument` will be thrown. `CFB_Decryption`, the inverse opertaion, takes similar arguments.

OFB(`const BlockCipherKey&` *key*, `const BlockCipherModeIV&` *iv*):

This is the usual OFB mode. Variable feedback sizes are not supported, as it has be shown that they are insecure. `Counter` is a variant of `OFB`, whose constructor takes the same arguments. Remember that a block cipher operating in either OFB or counter modes is like a stream cipher, and thus you should never encrypt 2 messages with the same key without changing the IV.

4.6.4 Encoders

Often you want your data to be in some form of text (for sending over channels which aren't 8-bit clean, printing it, etc). The filters `HexEncoder` and `Base64Encoder` will convert arbitrary binary data into Hex or Base64 formats. Not suprisingly, you can use `HexDecoder` and `Base64Decoder` to convert it back into it's original form. You can find the declarations for these types in `encoder.h`.

5 More Information

5.1 Portability

A fair amount of effort was made into making this library portable to a wide variety of platforms, assuming an implementation of the C++ standard. OpenCL has been successfully compiled, tested, and used with the following systems:

OS	CPU	Compiler(s)
Linux	x86	egcs 1.1.2, gcc 3.0 (20010327), KAI C++ 3.4g, PGI C++ 3.2
Linux	IA-64	gcc 2.9 (20000216)
Linux	Alpha	gcc 2.96 (20000731)
Linux	PowerPC	gcc 2.95.2
Tru64	Alpha	Compaq C++ 6.2
Solaris	SPARC	gcc 2.95.2
IRIX	MIPS	MIPSProC++ 7.2.1

5.2 Compatibility

Generally, cryptographic algorithms are well standardized, and thus compatibilty between implementations is relatively simple (of course, not all algorithms are supported by all implementations). However, there are a few algorithms which are poorly specified, and these should be avoided if you wish your data to be proccessed in the same way by another implementation (including future version of OpenCL).

The block cipher GOST has a particularly poor specification: there are no standard Sboxes, and the specification does not give test vectors even for sample boxes, which leads to issues of endian conventions, etc. Other algorithms including in OpenCL suffering from these problems (though to a less serious degree) include HAVAL, ISAAC, and EMAC.

If you wish maximum portability between different implementations of an algorithm, it's best to stick to strongly defined and well standardized algorithms: TripleDES, Blowfish, CAST5, Rijndael, Serpent, HMAC, MD5, SHA-1, and RIPE-MD160 all being good examples.

5.3 Patents

Some of the algorithms implemented by OpenCL are covered by patents. Algorithms known to be patented in the United States or other countries (where the patent holder has not granted royalty-free use for any purpose) include: CS-Cipher, IDEA, MISTY1, RC5, RC6, and SEAL. Note that just because an algorithm is not listed here, you should not assume that it is not encumbered by patents.

5.4 Further Reading

It's probably a very good idea if you have some knowledge of cryptography prior to trying to use OpenCL. It is recommended you read one or more of these books before seriously using the library:

Handbook of Applied Cryptography, by Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone; CRC Press

Cryptography: Theory and Practice, by Douglas R. Stinson; CRC Press

Applied Cryptography, 2nd Ed., by Bruce Schneier; John Wiley & Sons

5.5 Acknowledgements

The implementation of DES is based off a public domain implementation by Phil Karn from 1994 (he, in turn, credits Richard Outerbridge and Jim Gillogly).

Matthew Skala's public domain `twofish.c` (as given in GnuPG 0.9.8) provided the basis for Twofish.

Rijndael and Square are based on the reference implementations written by the inventors, Joan Daemon and Vincent Rijmen.

ThreeWay is based on reference code written by Joan Daemon.

The Serpent S-Boxes used were discovered by Dag Arne Osvik and detailed in his paper "Speeding Up Serpent".

The design of Randpool takes some of it's design principles from those suggested by Eric A. Young in his SSLeay documentation and Peter Guttman's paper "Software Generation of Practically Strong Random Numbers".

X917's design was changed from the X9.17 standard in response to the attacks presented in the paper "Cryptanalytic Attacks on Pseudorandom Number Generators", by Kelsey, Schneier, Wagner, and Hall.

5.6 Contact Information

A DSA key with a fingerprint of `33E3 9768 1D13 E7B4 1A01 BBCE A63F 2CBD FA02 FBCC` is used to sign all OpenCL releases. This key can be found in the file `doc/pgpkeys.asc`; PGP keys for the developers are also stored there.

Email: `opencl@acm.jhu.edu`

OpenCL Web Site: `http://opencl.sourceforge.net`

OpenCL File Distribution Site: `ftp://prdownloads.sourceforge.net/opencl`

5.7 Developers

Name: Jack Lloyd
Email: <lloyd@acm.jhu.edu>
PGP Key Fingerprint: `2DD2 95F9 C7E3 A15E AF29 80E1 D6A9 A5B9 4DCD F398`
Credits: Initial design and coding