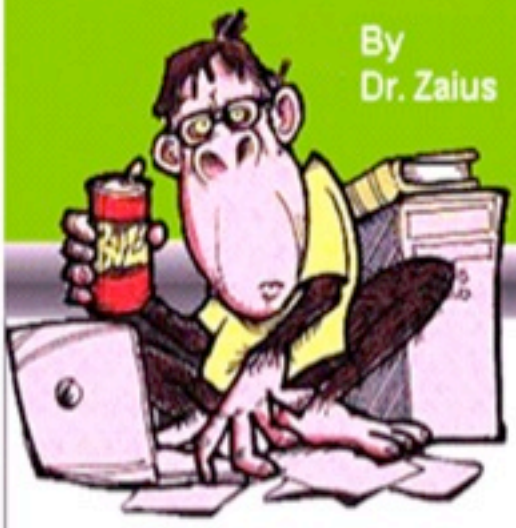
ClusterChimps.org

oclelf
oclcc
oclcrypt
OpenCL

By
Dr. Zaius



The ClusterChimps Guide to Offline OpenCL Compiling and Linking

This book is intended for developers who are writing OpenCL programs. We will introduce you to a suite of tools named OCLTools that you can use to help streamline your OpenCL development and deployment process. The first tool (oclcc) is a standalone OpenCL compiler that removes the necessity for you to compile your OpenCL programs at runtime. The second tool (oclelf) provides the ability to link your kernel source or prebuilt kernel into your binary to simplify application distribution. The last tool is oclcrypt which provides DES encryption capabilities to your OpenCL development and deployment process. Currently OCLTools is only available on Linux and has been tested with Nvidia's OpenCL implementation.

Disclaimer: While we discuss products from Nvidia we have no relationship with them other than that of fanboy.

About ClusterChimps.org

ClusterChimps is dedicated to helping bring inexpensive supercomputing to the masses by leveraging emerging technologies coupled with bright ideas and open source software. We do this because we believe it will help advance computation intensive research areas including basic research, engineering, earth science, biology, materials science, and alternative energy research just to name a few.

About the Author

Dr. Zaius is a well renowned orangutan in the field of cluster computing and GPGPU programming. He has spent most of his career in the financial industry working at exchanges, investment banks, and hedge funds. He is currently the driving force behind the site ClusterChimps.org. Originally from the island of Borneo, Dr. Zaius now resides in New York City with his wife and 3 children. He can be reached at zaius@clusterchimps.org

Table of Contents

Chapter

1

Introdukshon

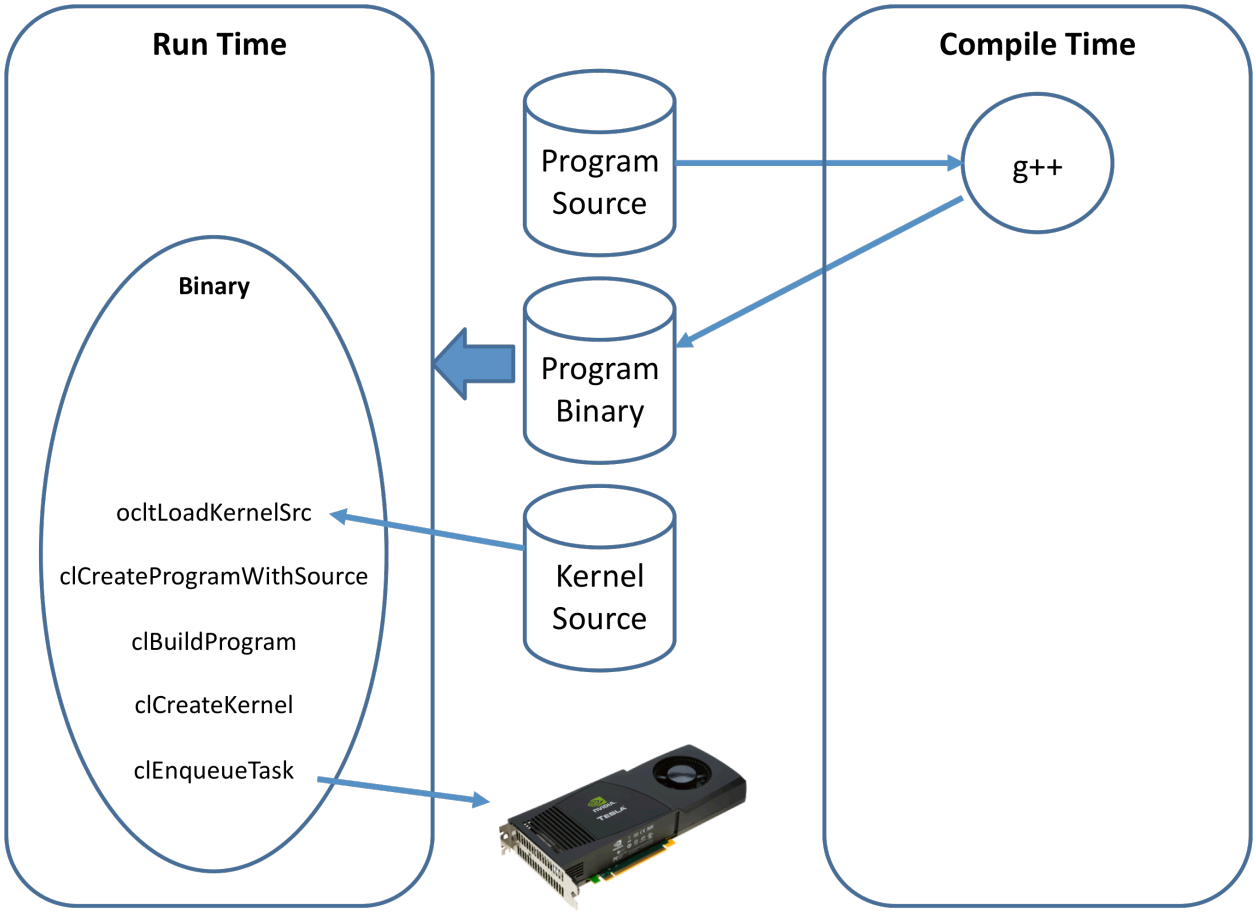
One of the more interesting features of OpenCL is the runtime compilation aspect of building and deploying OpenCL applications. By delaying compilation until runtime it is possible for your application to run on platforms that didn't even exist when you designed and implemented it. While this feature can be viewed as a positive thing for certain types of applications it also carries with it negative consequences for others. Imagine for a moment a company that designs and implements a computational finance framework in OpenCL. They spend millions of dollars on development of their models only to be forced to distribute the source code of their models with their final product. Also consider that the whole reason they used OpenCL in the first place was to speed up the execution of their models and now they have to tack on compilation time to their models runtime. To safe guard their intellectual property they are forced to come up with source code obfuscation routines that tend to have the nasty side effect of making their runtime compilation process even lengthier.

These are not insurmountable problems but as an OpenCL developer wouldn't you rather spend your time focusing on your models rather than focusing on their deployment? Enter OCLTools. OCLTools is a compact, yet effective, suite of development tools that focus on solving the issues of OpenCL kernel deployment. OCLTools come with a standalone (offline) OpenCL compiler called oclcc. With oclcc OpenCL developers now have the flexibility to precompile their OpenCL kernels, moving the lengthy compilation step from the runtime of their product back into the development phase of their product. OCLTools also comes with an OpenCL ELF file generator named oclelf that enables developers to link kernel binaries or source into their application binary. For developers worried about safeguarding their intellectual property OCLTools even provides DES encryption with oclcrypt.

OCLTools Programe Compilation

Before we start getting into the usage of the OCLTools components let's take a look at the normal OpenCL program development compilation flow:

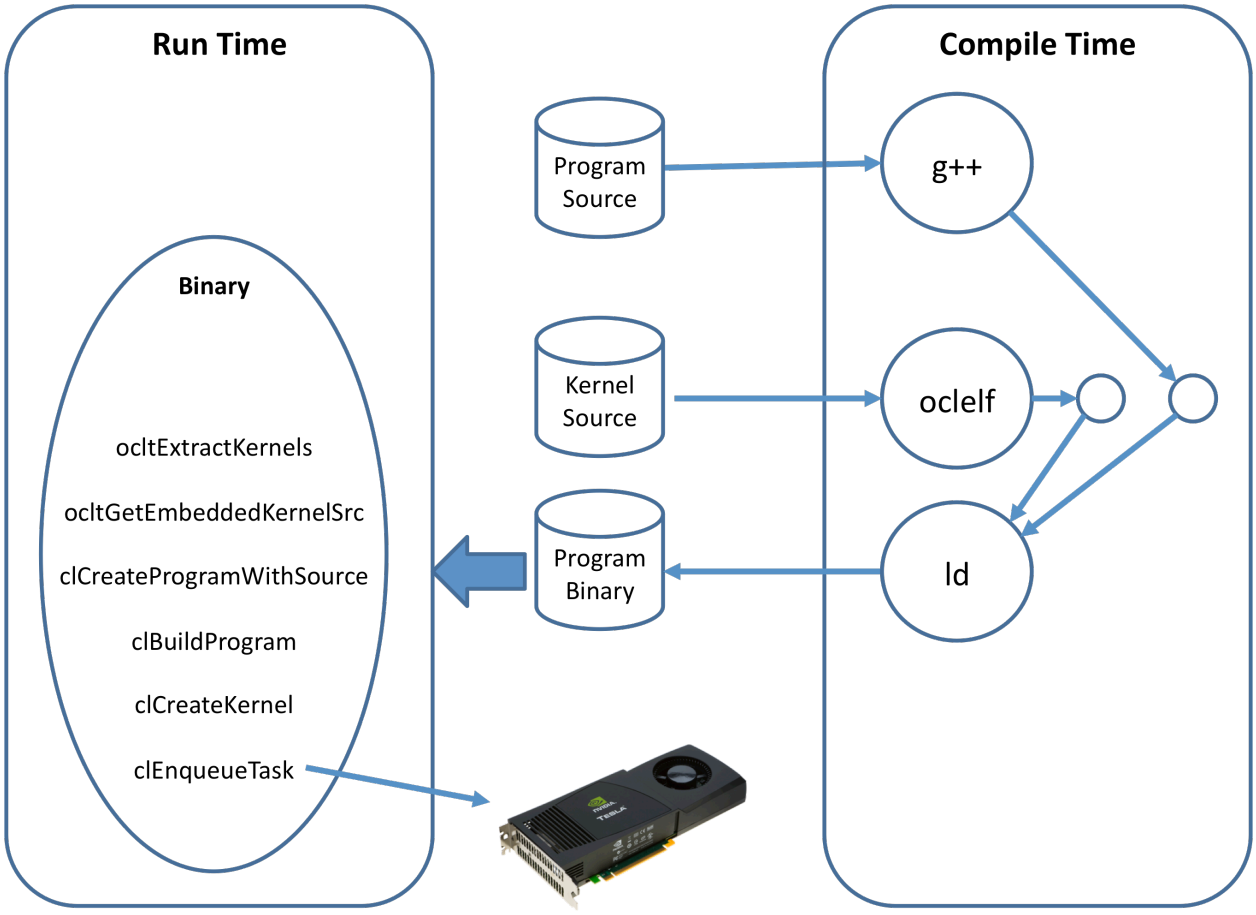
Normal Compilation Flow



The normal OpenCL program compilation starts off with g++ creating a binary from your program source. When that binary is run it must first read in your kernel source. It then makes three OpenCL API calls to build the kernel (clCreateProgramWithSource, clBuildProgram, clCreateKernel). At this point you have a kernel that is ready to be launched. This approach has a couple of drawbacks: you must ship the source to your kernel, the time required for compilation is added to the runtime of your application and the source to your kernel is sitting in a file system somewhere where it could potentially be modified, deleted or stolen.

Now that we have an understanding of the issues involved in OpenCL application deployment we will take a look at how OCLTools can be used to solve them. Let's assume that your kernel compilation time is minimal but you would rather not have to ship your source in the open. In this case you can employ OCLTools to link your kernel source into your program binary. By doing this you no longer have to worry about the location or safety of your kernel source. In this scenario your compilation flow will look like this:

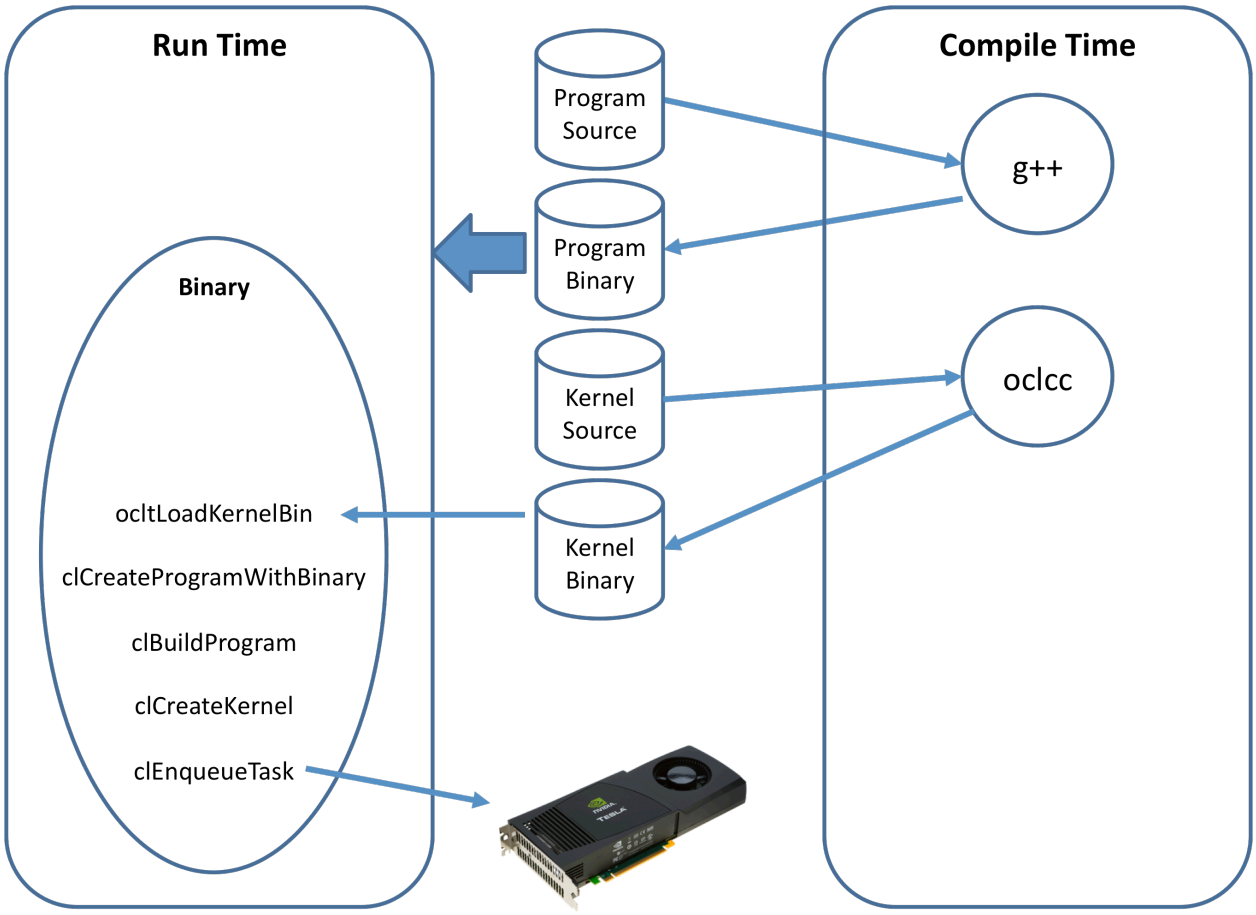
Embedded Source Compilation Flow



In this scenario you compile your program source with g++ and use oclelf to create an ELF binary with the source to your kernel embedded in it. You then use a linker to link the object files together. When your binary is run you no longer have to read the kernel source from the file system because it is already embedded in your running binary. To load the embedded kernel source you simply make calls to `ocltExtractKernels( )` and `ocltGetEmbeddedKernelSrc( )` (library routines that come with OCLTools) and then you make your three OpenCL calls to create a runnable kernel (clCreateProgramWithSource, clBuildProgram, clCreateKernel).

What if your kernel compilation time is excessive but you don't care about your kernel binary being out in the open? You can use OCLTools to precompile your kernel source eliminating the lengthy compilation time. In this scenario your compilation flow will look like this:

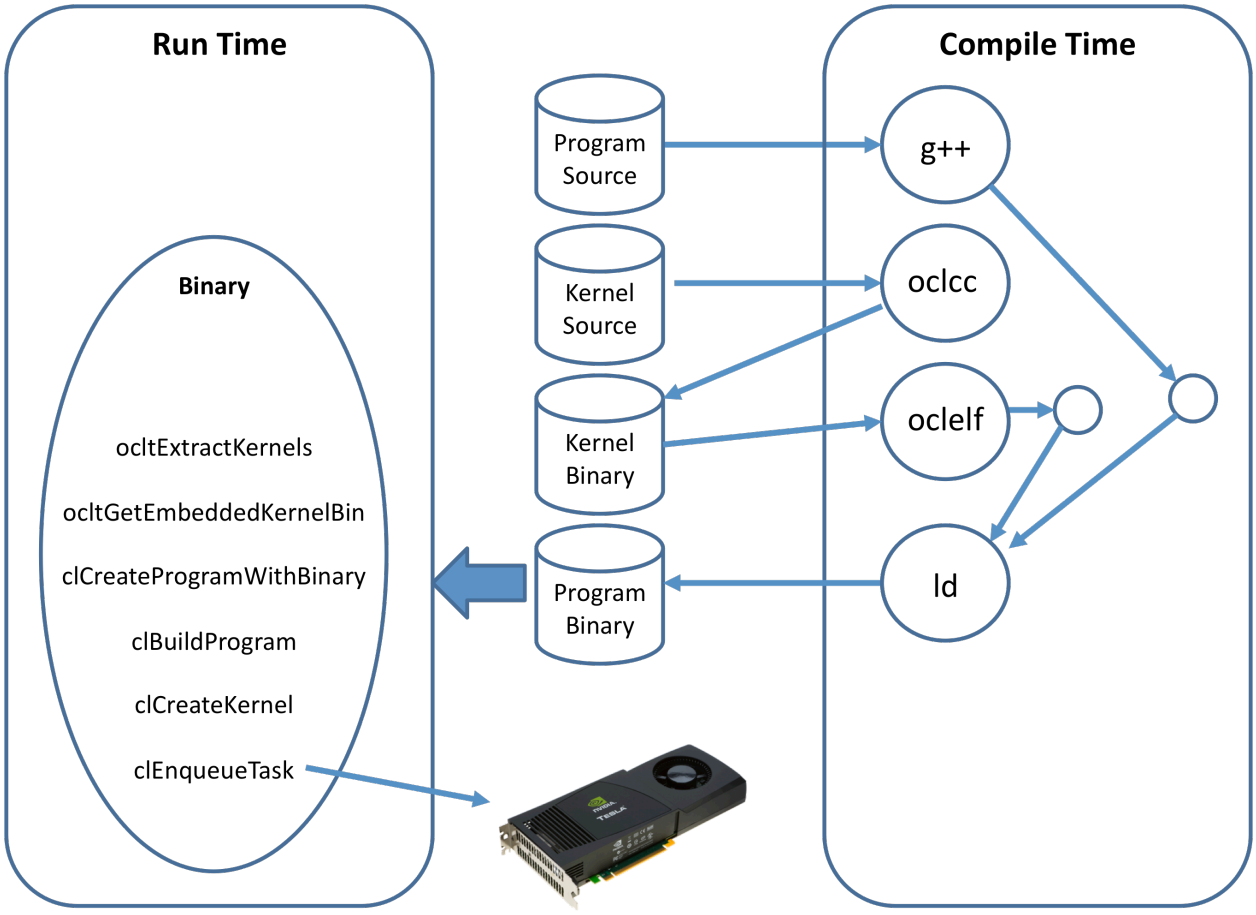
Precompiled Source Compilation Flow



You still start off by compiling your program source with g++ but now you also compile your kernel source with oclcc. When your binary is run it must read in the kernel binary produced by oclcc using `ocltLoadKernelBin( )` (a library routine provided by OCLTools) and make three OpenCL API calls to produce a runnable kernel. If you have precompiled your kernel source the OpenCL API calls that you need to make will change somewhat. You now need to call `clCreateProgramWithBinary( )` instead of `clCreateProgramWithSource( )`. Bizarrely enough you still need to make a call to `clBuildProgram( )` even though you have precompiled your kernel. According to the OpenCL specification OpenCL implementers can either compile to an intermediate format (such as PTX assembler in Nvidia's case) or compile directly to a binary format capable of executing on the device. This is why you still need to call `clBuildProgram( )` even though you precompiled your kernel source.

What if your kernel takes a long time to compile and you don't want to be dependant on having precompiled kernel binaries laying around at runtime. In this case you use OCLTools to precompile your kernel source to a binary using oclcc and you link your kernel binary into your application using oclelf. In this scenario your compilation flow will look like this:

Embedded Precompiled Source Compilation Flow

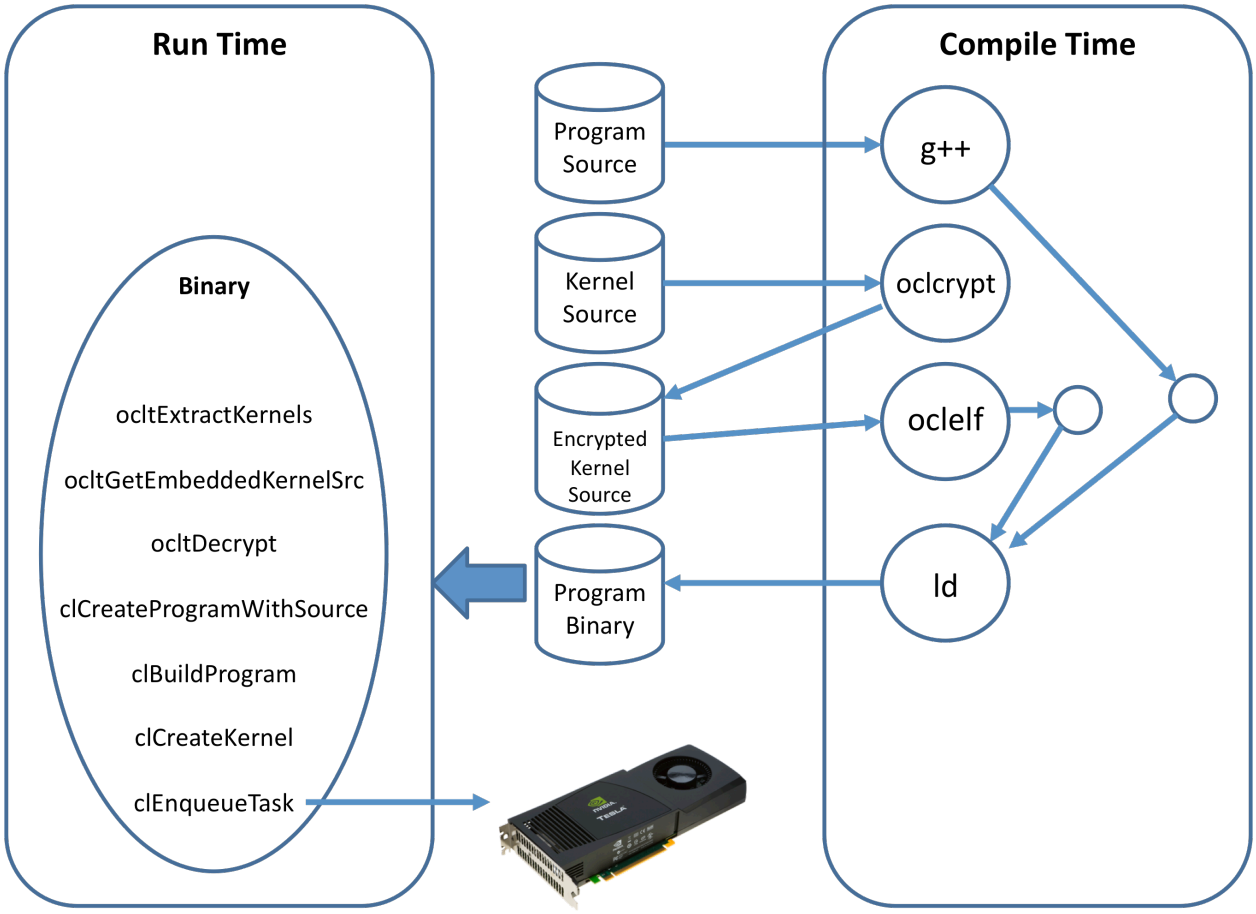


In this scenario you use g++ to compile your program source and you use oclcc to precompile your kernel source. You then use oclelf to embed your precompiled binary into an ELF object file and your linker finishes up by linking everything into one application binary. When your binary is run you don't need to read your kernel in from the file system because it is already imbedded into your running binary. To load the embedded precompiled kernel you simply make calls to `ocltExtractKernels( )` and `ocltGetEmbeddedKernelBin( )` (library routines that come with OCLTools) and then you make your three OpenCL calls to create a runnable kernel (clCreateProgramWithBinary, clBuildProgram, clCreateKernel).

For developers wishing to safeguard their intellectual property OCLTools helps on this front also. Even if you use OCLTools to embed your kernel source code into your application binary it

would not be that difficult to reverse engineer your kernel source. By simply running the 'strings' command on the resulting embedded binary you can extract all strings from it. An OpenCL developer will quickly be able to recognize the kernel source and can easily extract it. To guard against this reverse engineering you can make use of OCLTools encryption. In this scenario your compilation flow will look like this:

Encrypted Embedded Source Compilation Flow



In this scenario you use g++ to compile your program source and you use OCLTools oclcrypt tool to encrypt your kernel source. You then use oclelf and a linker to link the encrypted source into your application binary. When your binary is run you use OCLTools utility routines to extract your kernel source (`ocltExtractKernels( )`) from your application binary and decrypt it (`ocltDecrypt( )`). Then you make your three OpenCL calls to create a runnable kernel (clCreateProgramWithSource, clBuildProgram, clCreateKernel). Now if you run the "strings" command on your application binary you will no longer find the kernel source in the output so extracting the kernel source becomes significantly more difficult.

Chapter

2

Biulding OCLTools

OCLTools is built using the GNU build system, also known as Autotools, which is a suite of programming tools designed to assist in making source-code packages portable to many Unix-like systems. The GNU build system is part of the GNU toolchain and is widely used in many free-software and open-source packages. The tools comprising the GNU build system are free-software-licensed under the GNU General Public License with special license exceptions permitting use of the GNU build system with proprietary software.

The first step in the build process is to download the ocltools tarball (ocltools.tar.gz) from the ClusterChimps.org website. The gzipped tarball can be found at http://www.clusterchimps.org/ocltools.html. Once you have downloaded the file uncompress it:

```
zaius> gunzip ocltools-1.0.tar.gz
```

and then extract the source from the tar file:

```
zaius> tar -xvf ocltools-1.0.tar
```

Prerequisites

There are two prerequisites to being able to successfully build the OCLTools package. First you must have an OpenCL implementation installed on the system that you are building on. We have specifically avoided dependencies on Nvidia specific headers and libraries, so in theory OCLTools should work with any OpenCL implementation, however we have only tested against Nvidia's OpenCL implementation. You can find Nvidia's implementation at http://developer.nvidia.com/opencl. OCLTools also has a dependency on the boost program options library so the second prerequisite is having a version of boost installed. If you do not have boost installed you can download it at http://www.boost.org. We built against boost

1.44.1 but any version should do as the program options component has been stable for some time.

Contents

After you have extracted the source from the tarball take a look at the ocltools-1.0 directory. You will find the standard Autotools files and 6 directories containing usage examples. The example directories are provided to illustrate proper usage of OCLTools. Each example is the same OpenCL program (simple matrix multiplication) just built in a different manner. The OCLTools compiler, ELF file generator, encryption tool, and utility library all live in the src directory.

`example1`
is a plain vanilla OpenCL build that does not make use of OCLTools… a base line.

`example2`
shows you how you can use OCLTools to embed the kernel source into your application binary

`example3`
shows you how you can precompile your kernel source

`example4`
shows you how to precompile your source *and* embed it into your application binary

`example5`
shows you how you can use OCLTools to link multiple kernels into a single binary

`example6`
shows you how to use oclcrypt to better safeguard your intellectual property with encryption

`src/oclelf`
generates ELF files containing OpenCL source or precompiled binaries suitable for linking into an executable

`src/oclutil`
helper library that simplifies ocltools usage

`src/oclcc`
offline OpenCL compiler

`src/oclcrypt`
DES encryption tool

Building

To build OCLTools you must first run configure to generate your makefiles. Given the somewhat inconsistent library naming conventions between different versions of boost, you may need to change the name of the boost program options library in the configure.ac file to match what you have installed on your system. If you end up modifying the configure.ac file be sure to rerun autogen.sh. When you run configure you must provide the installation location and the location of boost on your system. In my case:

```
./configure –prefix=/usr/local/ocltools \
--with-boostinc=/usr/local/boost/1.44.0/include/boost-1_44_0/ \
--with-boostlib=/usr/local/boost/1.44.0/lib
```

At this point just type `make install` and be sure to add the location that you installed into to your PATH environment variable.

You will end up with two oclutil libraries in the installation lib directory: libocltoolscrypt.so and libocltools.so. The first library has dependencies on ssl and the second one does not. OCLTools relies on ssl for its encryption routines. We provide a version of the library that supports encryption and one that does not. This way if you don't want to encrypt your source you don't need to have a dependency on libssl.so and all of the baggage that comes with it.

The makefiles in the example directories are not part of the autotools build. These are just plain ordinary makefiles. You will need to edit each makefile and set install-dir to the location that you installed OCLTools into. Once you have made this adjustment you will be able to build the example programs.

Chapter

3

Usage Exampels

This chapter contains simple OCLTools usage examples illustrating the various functionalities provided by the OCLTools suite of tools.

Example1

Example1 is a baseline application. The only OCLTools usage that example1 has is in the calls to `ocltGetPlatformId( )` and `ocltLoadKernelSource( )`. Making calls to these functions simplifies your OpenCL program. It eliminates a bit of mundane coding required to get a simple OpenCL program up and running. First let's look at the makefile:

Example1 Makefile

```
.PHONY: all

install-dir := [ocltools-installation-directory]
all: example1

example1: main.c
    gcc -o $@ $^ -I$(install-dir)/include -L$(install-dir)/lib -locltools \
    -L/usr/lib64 -lOpenCL

clean:
    rm -f *.o
    rm -f example1
    rm -f *.so*
```

Before you can build this example you need to edit the makefile and replace the "[ocl-tools-installation-directory]" with the directory that you installed OCLTools into. The host source code is pretty straight forward:

Example1 main.cpp

```
// Multiply two matrices A * B = C
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <oclutil.h>
#include <CL/cl.h>

#define WA 3
#define HA 3
#define WB 3
#define HB 3
#define WC 3
#define HC 3
#define TRUE 1

// Initializes a matrix with random float entries.
void randomInit(float* data, int size)
{
   int i;
   for (i = 0; i < size; ++i)
   data[i] = rand() / (float)RAND_MAX;
}

////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////
int
main(int argc, char** argv)
{
   // set seed for rand()
   srand(2006);

   // 1. allocate host memory for matrices A and B
   int size_A = WA * HA;
   int mem_size_A = sizeof(float) * size_A;
   float* h_A = (float*) malloc(mem_size_A);

   int size_B = WB * HB;
   int mem_size_B = sizeof(float) * size_B;
   float* h_B = (float*) malloc(mem_size_B);
```

```
// 2. initialize host memory matrices
randomInit(h_A, size_A);
randomInit(h_B, size_B);

// 3. print out A and B
printf("\n\nMatrix A\n");
int i;
for(i = 0; i < size_A; i++)
{
   printf("%f ", h_A[i]);
   if(((i + 1) % WA) == 0)
   printf("\n");
}

printf("\n\nMatrix B\n");
for(i = 0; i < size_B; i++)
{
   printf("%f ", h_B[i]);
   if(((i + 1) % WB) == 0)
   printf("\n");
}

// 4. allocate host memory for the result C
int size_C = WC * HC;
int mem_size_C = sizeof(float) * size_C;
float* h_C = (float*) malloc(mem_size_C);

// 5. OpenCL specific variables
cl_context       clGPUContext;
cl_command_queue clCommandQue;
cl_program       clProgram;
cl_kernel        clKernel;
cl_device_id     cdDevice;

size_t kernelLength;
cl_int errcode;

// OpenCL device memory for matrices
cl_mem d_A;
cl_mem d_B;
cl_mem d_C;
```

```
// Initialize OpenCL
cl_platform_id   cpPlatform;

errcode = oclltGetPlatformID(&cpPlatform, "NVIDIA");
checkError(errcode, CL_SUCCESS);

// Get a GPU device
errcode = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice,
                         NULL);
checkError(errcode, CL_SUCCESS);

clGPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &errcode);
checkError(errcode, CL_SUCCESS);

//Create a command-queue
clCommandQue = clCreateCommandQueue(clGPUContext, cdDevice, 0,
                                    &errcode);
checkError(errcode, CL_SUCCESS);

// Setup device memory
d_C = clCreateBuffer(clGPUContext, CL_MEM_READ_WRITE, mem_size_A, NULL,
                     &errcode);
d_A = clCreateBuffer(clGPUContext, CL_MEM_READ_WRITE |
                     CL_MEM_COPY_HOST_PTR, mem_size_A, h_A, &errcode);
d_B = clCreateBuffer(clGPUContext, CL_MEM_READ_WRITE |
                     CL_MEM_COPY_HOST_PTR, mem_size_B, h_B, &errcode);

// 6. Load and build OpenCL kernel
char *clMatrixMul =
   oclltLoadKernelSrc("[/location/of/kernel/matrixMul.cl]",
   &kernelLength);
checkError(clMatrixMul != NULL, TRUE);

clProgram = clCreateProgramWithSource(clGPUContext, 1, (const char **)
                   &clMatrixMul,&kernelLength, &errcode);
checkError(errcode, CL_SUCCESS);

errcode = clBuildProgram(clProgram, 0, NULL, NULL, NULL, NULL);
checkError(errcode, CL_SUCCESS);

clKernel = clCreateKernel(clProgram, "matrixMul", &errcode);
checkError(errcode, CL_SUCCESS);

// 7. Launch OpenCL kernel
size_t localWorkSize[2], globalWorkSize[2];
```

```
    int wA = WA;
    int wC = WC;
    errcode  = clSetKernelArg(clKernel,0,sizeof(cl_mem),(void *)&d_C);
    errcode |= clSetKernelArg(clKernel,1,sizeof(cl_mem),(void *)&d_A);
    errcode |= clSetKernelArg(clKernel,2,sizeof(cl_mem),(void *)&d_B);
    errcode |= clSetKernelArg(clKernel,3,sizeof(int),(void *)&wA);
    errcode |= clSetKernelArg(clKernel,4,sizeof(int),(void *)&wC);
    checkError(errcode, CL_SUCCESS);

    localWorkSize[0] = 3;
    localWorkSize[1] = 3;
    globalWorkSize[0] = 3;
    globalWorkSize[1] = 3;

    errcode = clEnqueueNDRangeKernel(clCommandQue, clKernel, 2, NULL,
              globalWorkSize, localWorkSize, 0, NULL, NULL);
    checkError(errcode, CL_SUCCESS);

    // 8. Retrieve result from device
    errcode = clEnqueueReadBuffer(clCommandQue, d_C, CL_TRUE, 0,
              mem_size_C, h_C, 0, NULL, NULL);
    checkError(errcode, CL_SUCCESS);

    // 9. print out the results
    printf("\n\nMatrix C (Results)\n");
    for(i = 0; i < size_C; i++)
    {
       printf("%f ", h_C[i]);
       if(((i + 1) % WC) == 0) printf("\n");
    }
    printf("\n");

    // 10. clean up memory
    free(h_A); free(h_B); free(h_C);

    clReleaseMemObject(d_A);
    clReleaseMemObject(d_C);
    clReleaseMemObject(d_B);

    free(clMatrixMul);
    clReleaseContext(clGPUContext);
    clReleaseKernel(clKernel);
    clReleaseProgram(clProgram);
    clReleaseCommandQueue(clCommandQue);

}
```

As you can see from the source this is a pretty standard OpenCL host program. There are, however, two differences. The first is the call to `ocltGetPlatfromId( )`, which searches through all of the OpenCL platforms installed on your system and returns the one with a CL_PLATFORM_NAME that matches "NVIDIA". The second difference is the call to `ocltLoadKernelSrc( )` which does all of the mundane file IO to read in the kernel. In order to successfully run the example you will need to edit the location of the kernel source in the call to `ocltLoadKernelSrc( )`. The kernel itself is a trivial matrix multiplication kernel that you have probably seen a hundred times.

Example1 Kernel

```
// kernel.cl
// Multiply two matrices A * B = C
// Device code.


// OpenCL Kernel
__kernel void
matrixMul(__global float* C,
          __global float* A,
          __global float* B,
          int wA, int wB)
{

   // 2D Thread ID
   int tx = get_local_id(0);
   int ty = get_local_id(1);

   // value stores the element
   // that is computed by the thread
   float value = 0;
   for (int k = 0; k < wA; ++k)
   {
      float elementA = A[ty * wA + k];
      float elementB = B[k * wB + tx];
      value += elementA * elementB;
   }

   // Write the matrix to device memory each
   // thread writes one element
   C[ty * wA + tx] = value;
}
```

When you make the necessary changes in the Makefile and main.c and type make you should get a binary. When you run it you should see something like:

```
zaius> example1

Matrix A
0.389147 0.108601 0.087847
0.112299 0.136472 0.945850
0.178413 0.288748 0.291875

Matrix B
0.821486 0.459928 0.748167
0.999626 0.830676 0.648438
0.009072 0.794227 0.115912

Matrix C (Results)
0.429036 0.338962 0.371751
0.237254 0.916234 0.282148
0.437851 0.553728 0.354549
```

Example2

Example2 shows you how you can use OCLTools to embed the source to your kernel into your application binary. Doing this only requires two simple changes to the files in example1. First you need to make a couple of changes to your Makefile.

Example2 Makefile

```
.PHONY: all

install-dir := [ocltools-installation-directory]
all: example2

example2: main.cpp matrixMul.o
    g++ -o $@ $^ -I$(install-dir)/include -L$(install-dir)/lib -locltools \
    -L/usr/lib64 -lOpenCL

matrixMul.o: matrixMul.cl
    $(install-dir)/bin/oclelf $@ $^

clean:
    rm -f *.o
    rm -f example2
    rm -f *.so*
```

In this makefile we use oclelf to create an ELF formatted object file that contains the source to your OpenCL kernel. We add the matrixMul.o to the example2 target so that it will be linked into the example2 binary by g++. Make note of the naming convention used with the OpenCL kernel. We made the name of the kernel file match the name of the kernel function housed in the file. THIS IS IMPORTANT. Let's take a look at the changes that we need to make to the host code and you will see why it is important.

Example2 Host Source

```
.
.
.
// 6. Load and build OpenCL kernel
oclExtractKernels();
size_t lSize;
unsigned char *buffer = oclGetEmbeddedKernelSrc("matrixMul", &lSize);

clProgram = clCreateProgramWithSource(clGPUContext, 1, (const char **)
                                      &buffer, &lSize, &errcode);

errcode = clBuildProgram(clProgram, 0, NULL, NULL, NULL, NULL);
checkError(errcode, CL_SUCCESS);

clKernel = clCreateKernel(clProgram, "matrixMul", &errcode);
checkError(errcode, CL_SUCCESS);
.
.
.
```

The only changes necessary in the host code are in section 6. We add a call to `oclExtractKernels( )` which builds a kernel database from the matrixMul.o that was linked in. This DB is indexed by the name of the OpenCL kernel file in the makefile. We also need to replace the call to `oclLoadKernelSrc( )` with a call to `oclGetEmbeddedKernelSrc( )`. We no longer need to load the kernel source from the file system because it is linked into our application now. When we call `oclGetEmbeddedKernelSrc( )` we pass in the name of the kernel file that we want to load (we could have multiple kernel files linked into the application binary as you will see in example5). This is why it was important to make the name of the OpenCL kernel source file match the name of the kernel function inside it. If you happen to have more than one kernel

function in the kernel file just use the first one as the file name. Build and run this example and you should see something similar to:

```
Zaius> example2

Matrix A
0.389147 0.108601 0.087847
0.112299 0.136472 0.945850
0.178413 0.288748 0.291875

Matrix B
0.821486 0.459928 0.748167
0.999626 0.830676 0.648438
0.009072 0.794227 0.115912

Matrix C (Results)
0.429036 0.338962 0.371751
0.237254 0.916234 0.282148
0.437851 0.553728 0.354549
```

Example3

Example3 uses oclcc to precompile the kernel source. To do this we only need to make a couple small changes to the host code and the makefile. Let's start with the makefile:

Example3 Makefile

```
.PHONY: all

install-dir := [ocltools-installation-directory]

all: example3 matrixMul.ptx

example3: main.cpp
    g++ -o $@ $^  -I$(install-dir)/include -L$(install-dir)/lib -locltools \
    -L/usr/lib64 -lOpenCL

matrixMul.ptx: matrixMul.cl
    $(install-dir)/bin/oclcc -o $@ --cl-fast-relaxed-math --cl-nv-verbose $^

clean:
    rm -f *.o
    rm -f example3
    rm -f *.so*
    rm -f *.ptx
```

We added a new target for matrixMul.ptx that compiles the matrixMul.cl file with oclcc. We added a couple of OpenCL compiler flags into the mix for no other reason than to show you how to do it. In this example we are not embedding the precompiled source into the application binary therefore we did not need to make any changes to the example3 target. The host code changes are:

Example3 Host Code

```
.
.
.
// 6. Load and build OpenCL kernel
char *compilerFlags;
size_t lSize;
unsigned char *buffer =
   oocltLoadKernelBin("[/location/of/kernel/matrixMul.ptx]",
                      &compilerFlags, &lSize);

cl_int status;
clProgram = clCreateProgramWithBinary(clGPUContext, 1,
              (const cl_device_id *)&cdDevice, &lSize,
              (const unsigned char**)&buffer, &status, &errcode);

errcode = clBuildProgram(clProgram, 0, NULL, compilerFlags, NULL, NULL);
checkError(errcode, CL_SUCCESS);

clKernel = clCreateKernel(clProgram, "matrixMul", &errcode);
checkError(errcode, CL_SUCCESS);
.
.
```

In this example we replace the call to `ocltLoadKernelSrc( )` with a call to `ocltLoadKernelBin( )`. Note that we are no longer passing in the name of the kernel source file but the name of the precompiled kernel file that is being built by the new target in the makefile. Now, instead of making a call to `clCreatProgramWithSource( )`, we make a call to `clCreateProgramWithBinary( )`. This is because we are no longer dealing with OpenCL source but a precompiled binary. You can see that we still need to make a call to `clBuildProgram( )` even though we have precompiled the source. We do this because that is what the OpenCL specification says we need to do. If we omit the `clBuildProgram()` call, our application will not run correctly (at least not with Nvidia's OpenCL implementation). We also grab the compiler flags out of our precompiled kernel binary

so that they can be passed to the `clBuildProgram( )` call. I don't believe that this is actually necessary (at least not with Nvidia's implementation) because if you run oclcc on the same kernel source but pass in different compiler flags the ptx assembler that is generated is different and I'm willing to bet that it doesn't matter whether we pass the compiler flags into the call to `clBuildProgram( )` or not. Since we have no way of knowing this for sure we pass along the compiler flags just to be safe.

If you build and run the example you will see something like:

```
zaius> example3

Matrix A
0.389147 0.108601 0.087847
0.112299 0.136472 0.945850
0.178413 0.288748 0.291875


Matrix B
0.821486 0.459928 0.748167
0.999626 0.830676 0.648438
0.009072 0.794227 0.115912


Matrix C (Results)
0.429036 0.338962 0.371751
0.237254 0.916234 0.282148
0.437851 0.553728 0.354549
```

Example4

Example4 not only precompiles the kernel source but it links in the resulting kernel binary into your application binary. Doing this effectively moves the time required to compile your kernel out of your applications runtime and back into the compilation phase of development. Again, we only need to make a couple of simple changes to our host code and the makefile. Let's start with the makefile:

Example4 Makefile

```
.PHONY: all

install-dir := [ocltools-installation-directory]
all: example4

example4: main.cpp matrixMul.o
    g++ -o $@ $^ -I$(install-dir)/include -L$(install-dir)/lib -locltools \
    -L/usr/lib64 -lOpenCL

matrixMul.ptx: matrixMul.cl
    $(install-dir)/bin/oclcc -o $@ --cl-fast-relaxed-math --cl-nv-verbose $^

matrixMul.o: matrixMul.ptx
    $(install-dir)/bin/oclelf $@ $^

clean:
    rm -f *.o
    rm -f main
    rm -f *.so*
    rm -f *.ptx
```

In this example we add a target for matrixMul.ptx that causes matrixMul.cl to be compiled by oclcc and we add a target for matrixMul.o that causes oclelf to create an ELF object file that contains the precompiled binary. We also add matrixMul.o to the example4 target so that g++ links in the ".o" file with the embedded precompiled binary into our application. Here are the minimal changes to the host code that are necessary:

Example4 Host Code

```
.
.
.
// 6. Load and build OpenCL kernel
ocltExtractKernels();

size_t lSize;
char *compilerFlags;
unsigned char *buffer = ocltGetEmbeddedKernelBin("matrixMul",
                        &compilerFlags, &lSize);

cl_int status;
clProgram = clCreateProgramWithBinary(clGPUContext, 1,
              (const cl_device_id *)&cdDevice, &lSize,
              (const unsigned char**)&buffer, &status, &errcode);

errcode = clBuildProgram(clProgram, 0, NULL, compilerFlags, NULL, NULL);
checkError(errcode, CL_SUCCESS);

clKernel = clCreateKernel(clProgram, "matrixMul", &errcode);
checkError(errcode, CL_SUCCESS);
.
.
.
```

The only changes necessary in the host code are in section 6. We add a call to `ocltExtractKernels( )` which builds a kernel database from the matrixMul.o that was linked in. This DB is indexed by the name of the OpenCL kernel file in the makefile. We also need to replace the call to `ocltLoadKernelBin( )` with a call to `ocltGetEmbeddedKernelBin( )`. We no longer need to load the kernel source or binary from the file system because it is now linked into our application. When we call `ocltGetEmbeddedKernelBin( )` we pass in the name of the precompiled kernel file that we want to load (we could have multiple kernel files linked into the application binary as you will see in example5). This is why it is important to make the name of the OpenCL kernel source file match the name of the kernel function inside it. If you happen to have more than one kernel function in the kernel file just use the first one as the file name. Now instead of making a call to `clCreatProgramWithSource( )` we make a call to

`clCreateProgramWithBinary( )` because we are no longer dealing with OpenCL source but a precompiled binary. You can see that we still need to make a call to `clBuildProgram( )` even though we have precompiled the source. We do this because that is what the OpenCL specification says that we need to. If we omit the `clBuildProgram( )` call, our application will not run correctly (at least not with Nvidia's OpenCL implementation). We also grab the compiler flags out of our precompiled kernel binary so that they can be passed to the `clBuildProgram( )` call. If we build and run this application we get something like:

```
zaius>

Matrix A
0.389147 0.108601 0.087847
0.112299 0.136472 0.945850
0.178413 0.288748 0.291875

Matrix B
0.821486 0.459928 0.748167
0.999626 0.830676 0.648438
0.009072 0.794227 0.115912

Matrix C (Results)
0.429036 0.338962 0.371751
0.237254 0.916234 0.282148
0.437851 0.553728 0.354549
```

Examples

Example5 is practically identical to example4 except that example5 links in multiple precompiled OpenCL kernel files into your application. All we really change here is the makefile.

Example5 Makefile

```
.PHONY: all

install-dir := [ocltools-installation-directory]
all: example5

example5: main.cpp kernels.o
    g++ -o $@ $^  -I$(install-dir)/include -L$(install-dir)/lib -locltools \
    -L/usr/lib64 -lOpenCL

matrixMul.ptx: matrixMul.cl
  $(install-dir)/bin/oclcc -o $@ --cl-fast-relaxed-math --cl-nv-verbose $^
```

```
matrixMUL.ptx: matrixMUL.cl
    $(install-dir)/bin/oclcc -o $@ --cl-fast-relaxed-math --cl-nv-verbose $^

kernels.o: matrixMul.ptx matrixMUL.ptx
    $(install-dir)/bin/oclelf $@ $^

clean:
    rm -f *.o
    rm -f main
    rm -f *.so*
    rm -f *.ptx
```

In this example we add two ptx targets to our makefile. We have the same target from example4 and a new target that causes the kernel named matrixMUL.cl to be compiled by oclcc. Open up matricMUL.cl and you will see that the kernel function is named matricMUL but other than that the file is identical to the matrixMul.cl file. Our kernels.o target causes oclelf to create an ELF file containing the precompiled kernels from both ".cl" files. The changes to the example5 target cause g++ to link in the file containing the precompiled kernels to our application binary.

We don't need to make any changes to our host code because as you recall our call to `ocltGetEmbeddedKernelBin( )` passes in the name of the kernel that we are interested in. Build and run example5 and you will see something like:

```
zaius>

Matrix A
0.389147 0.108601 0.087847
0.112299 0.136472 0.945850
0.178413 0.288748 0.291875

Matrix B
0.821486 0.459928 0.748167
0.999626 0.830676 0.648438
0.009072 0.794227 0.115912

Matrix C (Results)
0.429036 0.338962 0.371751
0.237254 0.916234 0.282148
0.437851 0.553728 0.354549
```

Edit the host code for example5 and change the code to use the matrixMUL kernel binary instead of the matrixMul kernel binary. You can do this by changing two lines of code:

```
.
.
.
unsigned char *buffer = oclltGetEmbeddedKernelBin("matrixMUL",
                        &compilerFlags, &lSize);

cl_int status;
clProgram = clCreateProgramWithBinary(clGPUContext, 1,
              (const cl_device_id *)&cdDevice, &lSize,
              (const unsigned char**)&buffer, &status, &errcode);

errcode = clBuildProgram(clProgram, 0, NULL, compilerFlags, NULL, NULL);
checkError(errcode, CL_SUCCESS);

clKernel = clCreateKernel(clProgram, "matrixMUL", &errcode);
.
.
.
```

Now when you build and run the program you will get the same output but you will be running the matrixMUL kernel instead of the matrixMul kernel.

Example6

Example6 explores the cryptography features of OCLTools. This example is conceptually identical to example2 except that instead of embedding the clear text OpenCL kernel source code into our application binary we are embedding encrypted OpenCL kernel source into our application binary. Why would you want to encrypt the embedded source? Well if you go back to example 2 and run the "strings" command against the example2 binary you will see that the "strings" command can easily be used to extract the embedded kernel source from your application binary. After spending considerable resources on developing your kernel models you may not want someone to be able to reverse engineer your kernels so easily. By making use of OCLTools encryption routines you can make it significantly more difficult for someone to steal your intellectual property. I will be the first to admit that OCLTools encryption does not provide a 100% safeguard against intellectual property theft but it does make it pretty damn difficult.

To encrypt your embedded kernel source we need to make a couple of changes to our makefile and to our application host code. Let's look at the makefile first:

Example6 Makefile

```
.PHONY: all

install-dir := [ocltools-installation-directory]
all: example6

example6: main.cpp matrixMul.o
    g++ -o $@ $^ -I$(install-dir)/include -L$(install-dir)/lib \
    -l ocltoolscrypt -L/usr/lib64 -lOpenCL

matrixMul.crypt: matrixMul.cl
    $(install-dir)/bin/oclcrypt -o $@ -c -k password $^

matrixMul.o: matrixMul.crypt
    $(install-dir)/bin/oclelf $@ -c $^

clean:
    rm -f *.o
    rm -f example6
    rm -f *.so*
    rm -f *.crypt
```

In order to encrypt the embedded kernel source we need to add a target to cause our OpenCL kernel source to be run through oclcrypt. Make note of the key used for the encryption (-k on the oclcrypt command line) because you will need to make use of this key in your application source. This key must be eight characters. Instead of passing the clear text kernel source into oclelf we pass in the encrypted kernel source. Make note of the –c flag on the oclelf command line. This is necessary for the resulting ELF object file to be created correctly. Also take note of the change we made to the example6 target. We replaced the dependency on the ocltools library with a dependency on the ocltoolscrypt library. This is required because the ocltools library does not include the encryption routines. There is no reason for your applications to have a dependency on ssl unless you want one.

Now let's take a look at the changes necessary in your application code to decrypt your kernel source.

Example6 Host Code

```
.
.
.
// 6. Load and build OpenCL kernel
oclExtractKernels();
size_t lSize;
unsigned char *buffer = oclGetEmbeddedKernelSrc("matrixMul", &lSize);

buffer = (unsigned char *)oclDecrypt("password", (char*)buffer,
                                     lSize);
clProgram = clCreateProgramWithSource(clGPUContext, 1, (const char **)
                                     &buffer, &lSize, &errcode);

errcode = clBuildProgram(clProgram, 0, NULL, NULL, NULL, NULL);
checkError(errcode, CL_SUCCESS);
.
.
.
```

We add a call to `oclExtractKernels( )` to build our kernel database out of the linked in embedded ELF file. We then make a call to `oclGetEmbeddedKernelSrc( )` to pull the kernel source from the database. Since we did not link in clear text kernel source we add an extra step to decrypt the kernel source by calling `oclDecrypt( )`. This is where we need the eight character key that we used in our makefile to encrypt the source. In the case of this example we used "password" as the key in our makefile so we must use "password" as the key in our call to `oclDecrypt( )`.

If you build and run this example you will get something like:

```
zaius>

Matrix A
0.389147 0.108601 0.087847
0.112299 0.136472 0.945850
0.178413 0.288748 0.291875
```

```
Matrix B
0.821486 0.459928 0.748167
0.999626 0.830676 0.648438
0.009072 0.794227 0.115912

Matrix C (Results)
0.429036 0.338962 0.371751
0.237254 0.916234 0.282148
0.437851 0.553728 0.354549
```

Chapter

4

OCLTools Refrence Guid

oclcc

```
oclcc 1.0
Copyright (C) 2011 ClusterChimps.org
This software is free (GPLv3). There is NO warranty;not even
for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

This program is a stand alone (offline) OpenCL compiler.  It accepts all
OpenCL compiler flags and Nvidia extensions.  This compiler is intended to
be used as part of the OCLTools suite of development tools and may not
function
correctly outside of the OCLTools suite of tools.

Usage: oclcc [options] input-file
Options:
  -o [ --output-file ] arg        File for compilation output.

  -I [ --includes ] arg           Directories to search for headers.

  -D [ --define ] arg             Defines a macro.

  --input-file arg                Input file to compile.

  --platform arg (=NVIDIA)        Platform to compile for

  --cl-single-precision-constant  Treat floating-point constant as single
                                  precision constant instead of implicitly
```

	converting it to double precision constant. This is valid only when the double precision extension is supported. This is the default if double precision floating-point is not supported.
--cl-denorms-are-zero	This option controls how single precision and double precision denormalized numbers are handled.
--cl-opt-disable	This option disables all optimizations. The default is optimizations are enabled.
--cl-strict-aliasing	This option allows the compiler to assume the strictest aliasing rules.
--cl-mad-enable	Allows a * b + c to be replaced by a mad. This will result in reduced accuracy.
--cl-no-signed-zeros	Allow optimizations for floating-point arithmetic that ignore the signedness of zero.
--cl-unsafe-math-optimizations	Allow optimizations for floating-point arithmetic that (a) assume that arguments and results are valid, (b) may violate IEEE 754 standard and (c) may violate the OpenCL numerical compliance requirements
--cl-finite-math-only	Allow optimizations for floating-point arithmetic that assume that arguments and results are not NaNs or +/- Inf
--cl-fast-relaxed-math	Sets the optimization options --cl-finite-math-only and --cl-unsafe-math-optimizations.
--cl-nv-maxrregcount arg	Nvidia specific: The max number of registers a GPU function can use.
--cl-nv-opt-level arg	Nvidia specific: optimization level (0-3).
--cl-nv-verbose	Nvidia specific: turns on verbose build output.
-w [--warnings-off]	Turn off warnings.
--Werror	Turn warnings into errors.
--device-query	Query OpenCL devices and exit.

```
  --version                     Print out version info and exit.

  --help                        Produce help message.
```

oclelf

```
Usage:
  oclelf outfile [-c] file1 [[-c] file2 ...]

Options:
        outfile ELF file suitable for linking into your application.

        -c      Indicates that the next input file is encrypted.

        file1   Kernel source or binary to embed in ELF file.
```

oclcrypt

```
oclcrypt 1.0
Copyright (C) 2011 ClusterChimps.org
This software is free (GPLv3). There is NO warranty; not even
for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

This program [En|De]crypts an input file creating an [En|De]crypted output
file.  While this encryption is not strong enough to stand up against
crackers it will keep people from poking around your program binaries with
'strings' and pulling out your kernel source.

Usage: crypt [options] input-file
Options:
  -k [ --key ] arg         Key for [en/de]cryption.
  -c [ --crypt ]           Encryption flag.
  -d [ --decrypt ]         Decryption flag.
  -o [ --output-file ] arg File for crypt output.
  --input-file arg         Input file to crypt.
  --version                Print out version info and exit.
  --help                   Produce help message.
```

Libocltools [libocltoolscrypt]

The libocltools.so and libocltoolscrypt.so libraries are identical with the exception of the `ocltDecrypt( )` function. The decrypt function is only present in libocltoolscrypt.so. If you don't care about encryption link in libocltools.so which does not have dependencies on ssl and all of the baggage that comes with it.

cl_int ocltGetPlatformID(cl_platform_id* clSelectedPlatformID, const char* name)

This function finds the cl_platform_id that matches the `name` passed in. If it can not find a cl_platform_id that matches the one requested it will print out a warning and return the zeroth cl_platform_id. The cl_platform_id is returned via the cl_platform_id pointer that is the first argument to the function (`clSelectedPlatformID`). If this function is successful it returns CL_SUCCESS otherwise it will return an error code.

char* ocltLoadKernelSrc(const char* filename, size_t* length)

This function reads in the source for an OpenCL kernel from the file system. If it can not open and read the file it will return NULL, otherwise it returns a char* containing the source. It then sets the value of the length argument to the length of the char* returned.

unsigned char* ocltLoadKernelBin(const char* filename, char compilerFlags, size_t* length)**

This function reads in an OpenCL kernel binary from the file system. This binary must be created with oclcc because oclcc will embed the compiler flags used at compilation time into the binary file it creates. This function extracts the compiler flags (if any) and sets the compilerFlags argument with them. It also sets the length argument to the length of the kernel binary. It returns a char* containing the kernel binary. If it can not open the file it returns NULL.

void oclt ExtractKernels()

This function extracts embedded kernels from the object file generated by oclelf that were linked into the application. If there is a problem with kernel extraction it prints out an error and exits. If it is successful, it builds an internal DB containing all kernels extracted. The DB is indexed by the output file name (minus suffix) supplied to oclelf. This function MUST be called prior to calling `otltGetEmbeddedKernelBin( )` or `otltGetEmbededKernelSrc( )` otherwise the calls to these functions will fail.

unsigned char* oclt GetEmbeddedKernelBin(char* kernelName,
char compilerFlags,**
size_t* length)

This function queries the internal kernel DB based on the `kernelName` passed in and returns the binary. It also sets the `compilerFlag` argument to match what was passed to oclcc at compile time. The `length` argument is set to the length of the binary that is returned. If the kernel can not be found this function prints an error and returns NULL. Note: You MUST call `ocltExtractKernels()` before calling this function.

unsigned char* oclt GetEmbeddedKernelSrc(char* kernelName, size_t* length)

This function queries the internal kernel DB based on the `kernelName` passed in and returns the source. The `length` argument is set to the length of the source that is returned. If the source can not be found this function prints an error and returns NULL. Note: You MUST call `ocltExtractKernels()` before calling this function.

char* oclt Decrypt(char *key, char *kernel, int length)

This function decrypts kernels encrypted by oclcrypt. You must pass in the 8 character `key` that you used in your makefile to encrypt the kernel, the encrypted `kernel` (source | binary) that you got from either `ocltLoadKernelSrc()`, `ocltLoadKernelBin()`, `ocltGetEmbeddedKernelSrc()`, or `ocltGetEmbeddedKernelBin()`, and the length of the encrypted kernel (source | binary). This function returns the decrypted kernel.

Epilogue

I hope you enjoyed the preceding pages and found them informative. If you think you have found any inaccuracies please drop me a note at zaius@clusterchimps.org. I may not get back to you immediately (I do have a day job to pay the bills) but I appreciate any and all feedback.

If you would like to be notified of new releases to OCLTools just follow us on Facebook. Any updates to all ClusterChimps tools and publications will posted to our Facebook page. We will be releasing a publication on how to build and program a Virtual Supercomputer soon. If vast amounts of inexpensive computational capacity is something that interests you, be on the lookout for it.

ClusterChimps is dedicated to helping bring inexpensive supercomputing to the masses by leveraging emerging technologies coupled with bright ideas and open source software. We do this because we believe it will help advance computation intensive research areas including basic research, engineering, earth science, biology, materials science, and alternative energy research just to name a few.

Dr. Zaius