Cougar Open CL v1.0

Users Guide for Open CL support for Delphi/ C++Builder and .NET

MtxVec version v4, rev 2.0

www.dewresearch.com

Table of Contents

1 About Open CL

Open CL is a standard designed to make it easier to write software for GPU devices and make that code portable across different GPU devices. Open CL drivers for CPU and GPU have many great features:

1.) Cross-platform support. Same code is to run on embedded devices (like mobile phones), desktop PC's and super computers across a wide range of operating systems.
2.) Support for both ATI and Nvidia GPUs.
3.) Support for CPU devices. There exists a great opportunity that an extended Open CL will become the main target for accelerated code running on CPUs. Both Intel and AMD currently offer their own drivers for Open CL code to run on CPUs.
4.) Dynamic code compilation. The compiler is included with the drivers and the code is compiled only for the target device. End users running applications have the possibility to specify expressions which (through the Open CL) can run on GPU or be compiled in to native CPU code.
5.) Open CL drivers are free for supported platforms.
6.) End user application can be distributed without any dlls.

2 Features of Cougar Open CL

1.) Uses dynamic Open CL dll loading and can be included in end user applications possibly running on machines without Open CL drivers.
2.) Automatically detects all platforms (Intel, AMD, NVidia) and devices and loads their parameters.
3.) Provides routines to store encrypted source code in to .res resource files that are embedded in to the final application.
4.) Caches binaries compiled on the first run for subsequent faster load times.
5.) Automatically detects changes to the hardware or Open CL driver versions and rebuilds the cached binaries.
6.) Loads all the kernels (functions) present in the Open CL source code, including their properties.
7.) Implements a shared context between CPU and GPU devices for more efficient heterogeneous computing. (one context per platform)
8.) Allows build options and source headers to be specified at program load time optionally requesting unconditional recompile.
9.) Can load multiple programs, but kernels are invoked from central location giving an impression of a single program.
10.) Optionally can automatically detect the device which will make the available code run in the fastest way.

3 MtxVec for Open CL

1.) Implements all standard math functions
2.) Support for real and complex numbers across all functions (where applicable)
3.) Makes use of object cache concept known from MtxVec for faster memory handling and higher performance.
4.) Implements separate kernels for CPU and GPU devices to achieve best performance on both architectures.
5.) Can run in single and double precision concurrently.

6.) Integrated debugger support for debugger visualizers allows GPU code debugging as if it would be running on the CPU.
7.) Delivers over 500 unique kernels. When considering also single/double and CPU/GPU variants, it is well over 2000.
8.) Full support for operator overloading.
9.) Supports multiple automatic code fall-back scenarios. Even when no Open CL driver is detected, the code will still run. When not using Open CL it can run with Delphi code only without external dll's or with native MtxVec using Intel IPP and MKL performance libraries. When native MtxVec is found to be faster than Open CL, it will automatically default to it.
10.) Supports execution of "micro" kernels. Micro kernels are short functions which could normally not be accelerated with Open CL.
11.) The performance penalty for micro-kernels is estimated at 50% of peak performance for GPU devices. This comes however with the benefit of utter simplicity of code writing and debugging **with programmers productivity matching the work on CPU.**

3.1 Example code:

```
Var a,b,c: clVector;
begin
     a.CopyFromArray(cDoubleArray); //copy data from CPU memory to GPU memory
     b.CopyFromArray(cDoubleArray);
     c := sin(a) + cos(b)*a;
     c.CopyToArray(cDoubleArray); //copy data from GPU memory to CPU memory
end;
```

Debugger allows you to stop on every line and examine the contents of all variables residing on GPU as if though they would be simple arrays.

3.2 Running MtxVec for Open CL on CPU devices:

1.) Reduced overhead of copying data between CPU and the driver in compare to GPU devices.
2.) Runs 5-10x faster than Delphi code, but currently achieves about 25% of speed achievable with native heavily optimized CPU code (native MtxVec). Performance is expected to increase with newer versions of drivers promised by both AMD and Intel.
3.) Makes automatic use of all available CPU cores.
4.) Requires only few (5-10) kernels (functions) to be called sequentially to overcome the Open CL overhead.
5.) Requires vector length of around 200 000 elements for optimal results (!)
6.) It is expected that in the future version(s) of Open CL drivers for CPU devices the required vector length for optimal performance will be reduced to match native MtxVec (1000 elements).

3.3 Running MtxVec for Open CL on GPU devices:

1.) Runs 2-5x faster than native heavily optimized CPU code (3GHz Core i7 with native MtxVec) when using mid range GPU (like ATI HD 6770). It runs 20-50x faster than Delphi code.
2.) Requires hundreds of kernels (functions) to be called sequentially to overcome the Open CL overhead.
3.) Requires vector length of 200 000 elements or longer for optimal results.
4.) Delivers 2x better computational economy for double precision than for single precision for many devices. Although many GPU devices report 4 or more times lower speed for processing double precision than single precision floating point data, this will remain hidden in many cases because the bottleneck is not raw computational power but memory bandwidth.

5.) Delivers 2x better computational economy for complex numbers than for real numbers because they require more computation per array element.

Consequently MtxVec for Open CL will deliver best results running very close to peak GPU performance with double precision complex number math.

3.4 Relation of Cougar Open CL and native MtxVec

1.) We will continue to support optimized version of native MtxVec (for CPU only) as this will continue to deliver optimal performance on CPU devices.
2.) MtxVec for Open CL may be used in the future to deliver a portable version (without dlls and running on wide range of OSes and devices) of native MtxVec running at slightly lower speed.
3.) MtxVec for Open CL allows you to make use of GPUs now with only minimum effort.

4 Kernels for 1D processing

Cougar Open CL v1.0 focuses on 1D kernels where array elements do not depend upon each other. Processing of each array element can be executed independently from the rest of the array. Such algorithms feature the highest possible economy on GPU when the math applied to one element is sufficiently large to offset the kernel call overhead. GPUs can typically perform roughly 50 simple math operations like addition and multiplication for each array element in the time it takes to load and save the value to global memory. Algorithms using many trigonometric functions thus stand to benefit the most. One such example is complex number math.

5 The GPU device information for Open CL kernel algorithm developers

1.) The declared GPU floating point power (TFlop) is based on the assumption that data is available in the GPU cores (many) registers. Typically the GPU can perform 50 additions/multiplications per load/store of one floating point value.
2.) Various reports of huge performance advantage of GPU over CPU that can be found on internet are often overblown as the CPU code being benchmarked against GPU is seriously under optimized. Nvidia for example compares raw GPU memory bandwidth with raw CPU bandwidth, when a more accurate comparison would be to compare it with the CPU cache L2 bandwidth, which in CPUs can be several Mbytes large. A high-end CPU will require a high-end GPU to make a difference for most algorithms.
3.) The relative performance of GPU against CPU in case of MtxVec and also in general relies heavily on the memory bandwidth available to both of them. Low-end GPU devices which sport up to 30 GBytes/s of memory bandwidth can be easily offset by high-end CPUs. Server chipsets can deliver today 100GBytes/s (for Intel’s E7 10 core CPU) In general though the price of bandwidth is much cheaper for GPUs.

	Main memory (global memory)	**Cache L2/L3 (CPU 1-30Mbytes)**	**Cache L1 (local mem) (32-128Kbytes)**
CPU Mobile	5GB/s	40GB/s	120GB/s
CPU Desktop	10-25GB/s	80-160GB/s	250-450GB/s
GPU Mobile	40 GB/s	Not present	(no info)
GPU Desktop	80-120 GB/s	Not present	400GB/s
GPU Highend	200 GB/s	Not present	(no info)

Table: Memory bandwidth comparison between GPU and CPU.

4.) Memory bandwidth for GPU can be 4x the spec, when kernels use local memory (~400GBytes/s). Each group of GPU cores shares the same local memory which is typically around 32Kbytes. This makes it possible to accelerate some algorithms which access the same array elements more than once. This is frequently the case for linear algebra. The program must be written specifically to make use of the local memory.
5.) Memory bandwidth for the CPU can be 3-4x higher when using CPU cache. This is applicable only to algorithms which will read data multiple times from same memory locations in small enough chunks.
6.) Only high-end GPU devices currently support double precision. This is true for both AMD (6900 series is the only one featuring double precision support) and NVidia (only Tesla and Quadro series can run double precision at meaningful speeds).

MtxVec code performance is mostly sensitive to memory bandwidth available and will scale nearly linearly with it. For GPU this is the raw memory bandwidth and for CPU this is the L2/L3 cache bandwidth. Great tool to examine the memory bandwidth on your system is SisSoftware Sandra which shows memory bandwidth as a function of block size.

5.1 Cross-Fire and SLI

Both of these technologies apply to the display of data. For Open CL developers the devices appear the distinct regardless, if they are connected or not.

6 Running the same code on CPU and GPU

This will work, but performance penalties for one or the other will be one or more orders of magnitude. The two hardware architectures have so different weaknesses and strengths that a good cross-over design is impossible. Even writing code which runs reasonably effectively on AMD GPU and NVidia GPU concurrently can be a challenge.

Recipe for fast CPU code:

1.) For-loops with consecutive array element access
2.) Few threads typically not more than CPU core count.
3.) The size of all variables working with (the working set) should fit within CPU cache most of the time. CPU Cache size however has grown to be fairly generous (megabytes).

Recipe for fast GPU code:

1.) Allow the GPU to launch as many threads as possible. GPUs have many cores and they can all run nearly concurrently. The more threads the better can be the core usage economy. Avoid for-loops to allow more threads, if at all possible.
2.) The size of memory you work with does not matter because there is no cache, except when making use of the “local memory”. Contents of local memory however is not preserved between kernel calls and the size is about equal to CPU L1 cache (32KB) which is shared between groups of dozens of cores.

Of course, main memory is much faster on average GPU than on average CPU, but it is also the main bottleneck for GPU and memory is frequently cached on CPU.

7 Getting started

Add clMtxVec to the uses clause of your application. The Open CL features can be accessed through the clPlatform global variable. The variable contains a list of available platforms and devices. MtxVec for Open CL is shipped with many ready to use Open CL kernels (functions), which can be loaded from the apps resources.

```
clPlatform[k].Device[i].Programs.Add.LoadProgram(DEW_1D_KERNELS_CPU_SINGLE,      '',
BuildOptions)
```

Note that the flag specifies, if we want to load CPU or GPU code and for which precision. To load program form a text file:

```
clPlatform[k].Device[i].Programs.Add.LoadProgram(cList, '', BuildOptions, true);
```

The last parameter specifies if the rebuild is mandatory, or precompiled cached binary can be used. cList is a string list with list of file names containing our program(s).

Before the app is ready to run Open CL code a work space needs to be defined for the selected device:

```
clPlatform.UnmarkThreads;
sld := clPlatform[PlatformListBox.ItemIndex].Device[DeviceListBox.ItemIndex];
sld.Cache.SetBlockPoolSize(100,Round(Exp2(18)));
sld.Cache.SetVecCacheSize(100);
sld.Cache.SetMatrixCacheSize(100);
sld.CommandQueue[0].MarkThread;
```

Here we defined the number of memory regions and their maximum size which we plan to use with our program. This is similar as setting cache size for MtxVec. It has been determined that typical programs use only a very limited number of vector variables at the same time. Usually this number is around 40. We have set block pool size to 100 and the size of each block to 262144 of single precision elements. The blocks can be shared between different object types like vectors and matrices. The call to “MarkThread” is best explained with an example. How could code like this:

c := a + b;

know on which device to execute? We do this by associating a Command Queue of the selected device with a thread. This association is achieved through the call to “MarkThread”. You call this function from inside of the thread on which you want to run the code. This of course implies that in order to run the code on multiple devices concurrently, multiple threads will need to be started. It is also possible to associate the device with the variable by passing it as a parameter to the constructor:

```
var a: clVector;
begin
        a := clVector.Create(myDevice);
```

Data can be copied between Open CL devices directly only if they are part of the same platform.

7.1 Command queues

Each time a function is to be called, the corresponding OpenCL kernels needs to be added to queue associated with the device. CommandQueue property of the TOpenCLDevice object contains a list of queues to which we can enqueue our Kernels. For some devices only one queue is allowed. By default one queue is created when the object is initialized. Multiple queues can be useful when there is a need to achieve overlap between data copy and data compute. While one queue is running computation on the device, the other can be busy copying data from or to the device. To add additional queues write:

```
SelectedDevice.CommandQueue.Add;
```

All Kernels enqueued to all command queues on the given device share the cache of that device. Only one queue can be associated with one thread and vice versa.

The associated threads will swift through all code submitting possibly many hundreds Kernels to the queues and will then block on the line where the resulting data is to be copied from the device until that data will be ready. While debugging however, the queues will block on every breakpoint or step to allow inspection of the data being processed.

When not debugging our threads will reach end of program long before the actual computation on the device will finish and will wait there for the device to finish as well and to return our desired results.

7.2 Writing the first program

The open CL functions are accessible via clVector and clMatrix record types:

```
Var a,b,c: clVector;
begin
     a.CopyFromArray(cDoubleArray); //copy data from CPU memory to GPU memory
     b.CopyFromArray(cDoubleArray);
     c := sin(a) + cos(b)*a;
     c.CopyToArray(cDoubleArray); //copy data from GPU memory to CPU memory
end;
```

A part from supporting overloaded multiplication, addition, division and subtraction there are also many methods available on records themselves.

7.3 Copying data to and from the GPU

This is achieved with the call to the CopyFromArray/CopyToArray and Copy/CopyTo functions. They can take considerable time and may make sense only if the number of operations on GPU will take long enough time to offset for this overhead. For orientation see the following table of copy timings for array of 262144 single precision elements:

Copy operation	Time
From CPU to CPU	180us
From CPU to Open CL CPU	360us
From CPU to Open CL GPU	2000us

Times will scale proportionally with the size of the array. Depending on hardware, the GPU copy can sometimes take much longer (20ms with older hardware).

7.4 Kernel call overhead

Calling a kernel has a fixed overhead depending on the device type and model. This is the time that the kernel execution will take, even if it does nothing.

Device Driver	Optimal Time	Optimal kernel count	Time for 10 kernels
AMD ATI GPU	15us	200+	100us
AMD CPU	45us	200	100us
INTEL CPU	45us	6	60us
NVIDIA GPU	30us	200+	100us

These times imply the shortest vector or the smallest job size for which it is still meaningful to call a kernel. On the CPU these times are related to the overhead associated with starting and stopping the threads by the driver and may be significantly reduced in the future when more optimization is applied by driver makers (by avoiding threading).

8 Adding Open CL source code as a resource to your application

The source code will be encrypted and stored in to a .res file of specified name by calling:

```
clPlatform.SaveSourceFileToRes(ResFileName,DEW_1D_KERNELS_CPU_SINGLE,
programListCPUs);
```

The second parameter is the unique ID that we will use to load the resource and the third parameter is a string list of source code file names. Finally, you need to add {$R ResFileName} to your application so that the resulting .res file will be linked in to the executable.

9 Calling custom kernels

After the program(s) has been loaded, you can obtain the desired Kernel object or the function to be called like this:

```
Kernel := SelectedDevice.Kernels.Locate('vdSin');
```

Finally, the function parameters need to be set:

```
Kernel.SetArgBuffer(0,clB.Data.SData);
Kernel.SetArgInt32(1,clB.Data.SDataIndex(0));
Kernel.SetArgBuffer(2,clC.Data.SData);
Kernel.SetArgInt32(3,clC.Data.SDataIndex(0));
```

And the kernel executed:

```
Kernel.Enqueue(SelectedDevice.CommandQueue[0].clCommandQueue, VectorLen);
```

The Kernel object has many other methods and properties and also gives access to the Open CL kernel pointer.

9.1 Writing 1D kernels

The basic 1D kernel code:

```
__kernel void vdSin(__global const int *Src, const int SrcIdx,
                              __global int *Dst, const int DstIdx,
                               const int Length)
{
    size_t i = get_global_id(0);
    Dst[DstIdx + i] = sin(Src[Src1Idx + i]);
}
```

This kernel is expected to be called once for each array element. When calling Kernel.Enqueue the global work size specified must therefore match the length of the vector. Local work size will be set by the driver. Although it is possible to sometimes obtain higher performance with a local work size set to other value than chosen by the driver, we keep the flexibility of arbitrary vector length by not doing so. Open CL driver requests that global work size is divisible with local work size but can internally work around this limitation when local work size is not set explicitly. The syntax of Open CL code is roughly equal to C.

The presented kernel code uses indices in to the buffers to allow working with sub-buffers. This indices are required, if you want to use clVector and clMatrix types with the kernel. For more information check the clMtxVec.pas unit source code.

9.2 Debugging kernels

Open CL Kernel debugging support is improving with efforts coming from both Intel and AMD. Until June 2011 there was no integrated debugging environment available. Open CL kernels also need special care because a memory overrun can easily result in computer deadlocking instead of an access violation. For this reason the kernels are first debugged on CPU devices and then deployed to GPU devices.

10 Deciding which code path to run

When the application is deployed it is best to do a test run of the algorithm on all devices available on customer's computer and remember the best choice for subsequent runs. The test algorithm can be the actual algorithm or just the most important subset to give faster results. Trying to make use of all available computing devices requires careful planning and is hard to solve in general way. It can easily happen that the final result would be worse than with a single device. You can switch between devices simply by calling "MarkThread" on a different queue.

11 Selecting GPU for Open CL dedicated double precision machine

NVidia Geforce cards run double precision math about 10-12x slower than single precision. NVidia Tesla or Quadro cards are required with to reduce this ratio to 2x. Tesla and Quadro cards however come with roughly 10x higher price. ATI offers a ratio of 4x for its main stream 6900 series.

Best results for given money for MtxVec can thus be had with 2 or 4 ATI HD 6970 cards in Crossfire configuration. The power consumption and noise levels also remain reasonable. This comes at a small fraction of the cost required for CPU based solutions to deliver the same. Comparing to Nvidia this is roughly 10x cheaper than using Nvidia Tesla with only 2x lower performance.

Because MtxVec design is bandwidth limited, it is expected that double precision math will run only 2x slower (rather than 4x) in compare to single precision in many cases on ATI 6900 series. MtxVec will deliver performance much closer to the peak GPU performance in this case.

12 Requirements to run Open CL based program on PC

Both Nvidia and AMD graphics card with latest drivers will serve the purpose. Alternatively both AMD and Intel offer separate drivers for Open CL addressing CPU only, which do not require a big download. In its current state, AMD requires a minimum SSE2 capable CPU and Intel requires a minimum SSE4.x. This makes the AMD driver a preferred choice especially because no major performance differences could be measured between the two up until July 2011. The current state of Open CL compilers for CPUs is not yet up to the standards of Intel C++ compiler performance wise.