Swan: A tool for porting CUDA programs to OpenCL

M. J. Harvey[a,*], G. De Fabritiis[b]

[a]*High Performance Computing Service,*
Imperial College London, South Kensington, London, SW7 2AZ, UK
[b]*Computational Biochemistry and Biophysics Lab (GRIB-IMIM), Universitat Pompeu Fabra, Barcelona Biomedical Research Park (PRBB), C/ Doctor Aiguader 88, 08003 Barcelona, Spain*

Abstract

The use of modern, high-performance graphical processing units (GPUs) for acceleration of scientific computation has been widely reported. The majority of this work has used the CUDA programming model supported exclusively by GPUs manufactured by NVIDIA. An industry standardisation effort has recently produced the OpenCL specification for GPU programming. This offers the benefits of hardware-independence and reduced dependence on proprietary tool-chains. Here we describe a source-to-source translation tool, "Swan" for facilitating the conversion of an existing CUDA code to use the OpenCL model, as a means to aid programmers experienced with CUDA in evaluating OpenCL and alternative hardware. While the performance of equivalent OpenCL and CUDA code on fixed hardware should be comparable, we find that a real-world CUDA application ported to OpenCL exhibits an overall slow-down of $\approx 50\%$, a reduction attributable to the immaturity of contemporary compilers. The ported application is shown have platform independence, running on both NVIDIA and AMD GPUs without modification. We conclude that OpenCL is a viable platform for developing portable GPU applications but that the more mature CUDA tools continue to provide best performance.

Keywords: GPU, CUDA, OpenCL

1. Introduction

In recent years commodity graphical processing units (GPUs) have developed from simple fixed-function devices into generally-programmable, highly-parallel computers that are capable of very high rates of floating-point operations in comparison to contemporary traditional CPUs. As a consequence of the high performance, sustained by rapid product refresh driven by a commodity market, they have garnered much interest in the computational science field as tools for accelerating application performance, with many successes reported across a range of disciplines including computational chemistry ([1–3]), computational fluid dynamics ([4–6]), image processing ([7, 8]) and computational electrodynamics ([9, 10]).

[*]Corresponding author
Email addresses: `m.j.harvey@imperial.ac.uk` (M. J. Harvey), `gianni.defabritiis@upf.edu` (G. De Fabritiis)

There is a significant cost associated with application development for GPUs: their architecture is quite different to that of a conventional computer and code must be (re)written to explicitly expose algorithmic parallelism. A variety of GPU programming models have been proposed, from low-level, device-specific assembly languages *eg* Compute Abstraction Layer for AMD GPUs [11] to high-level data-parallel[12] abstractions such as Brook [13] along with OpenMP-style compiler directive extensions for C and Fortran[14]. The most popular development tool for scientific GPU computing has proved to be CUDA (Compute Unified Device Architecture)[15], provided by the manufacturer NVIDIA for its GPU products, which defines a C dialect for writing scalar GPU programs along with a set of C language extensions for simplifying the GPU control from the host program. Despite its accessibility to the programmer, as a proprietary vendor-controlled language and tool-chain tied to NVIDIA hardware its suitability for use as a strategic development platform for long-term software engineering projects is debatable.

The Khronos Group, a trade consortium that promotes open standards for media authoring and acceleration, has recently published the OpenCL specification[17], intended as a standard for cross-platform, parallel programming of heterogeneous processing systems. This standard proposes an abstract device model which maps well to contemporary GPU architectures and a GPU programming model that bears close similarities to CUDA. OpenCL has attracted vendor support, with implementations available from NVIDIA[1], AMD[2] , Apple[3] and IBM [4] . To scientific programmers, then, OpenCL may be an attractive alternative to CUDA as it offers a similar programming model with the prospect of hardware and vendor independence. However, OpenCL lacks CUDA's C-language extensions which greatly simplify the host program's management of GPU code, making it less immediately accessible to programmers new to GPU programming while also posing a technical engineering challenge for programmers attempting to port existing CUDA code-bases to OpenCL.

In this paper, we provide a summary of the principle differences between CUDA and OpenCL and describe a code translation and runtime library, called Swan, that we have developed to facilitate CUDA to OpenCL porting exercises. Finally, we review the performance of an OpenCL port of a real-world, production molecular dynamics application, ACEMD [3] against the reference CUDA version.

2. Methods

2.1. CUDA device architecture and programming model

There have been two generations of NVIDIA GPU hardware supporting the CUDA programming model. Comprehensive coverage of the architecture and programming of these devices is readily available[15] so we shall outline only the principal features here.

[1]`http://developer.nvidia.com/object/opencl.html`
[2]`http://www.amd.com/streamopencl`
[3]`http://developer.apple.com/technologies/mac/snowleopard/opencl.html`
[4]`http://www.alphaworks.ibm.com/tech/opencl`, *(accessed 11 November 2010)*

The current generation processor ('Fermi' GF100 series[16]) processor is a general purpose, single instruction, multiple thread (SIMT) processing unit with a high degree of intrinsic parallelism and arithmetic performance: for example, over 21,000 threads may run concurrently on the Tesla C2050, and obtain a peak arithmetic rate of $\leq$ 515 GFLOPS (double precision fused multiply-add). These devices are composed of a set of *multiprocessors*, each of which contains 32 scalar arithmetic units (*cores*). A scalar program fragment, known as a *kernel* may be run concurrently in a *block* of *threads* in parallel *warps* of 32 threads. All threads within a warp execute in lockstep, with best performance obtained when all threads follow the same execution path. Each block is restricted to executing on a single multiprocessor. Multiple independent blocks may be executed concurrently across a single device in a *grid*.

Threads within a common block may intercommunicate via a small 48kB[5] region of in-core shared memory. Exploiting inter-thread co-operation though shared memory is often critical in obtaining best performance. Each multiprocessor has a large (32kB) register file, of which each thread receives a private, static allocation. The total number of threads which may execute on a multiprocessor – and thus the degree of parallelism – is dependent on the register resources required by each individual thread. Excessive register pressure can lead to inefficient operation if there are too few threads running to fully hide instruction and memory access latencies.

The CUDA programming model is a C-like language with type qualifier extensions for indicating data locality and a special function-call syntax for specifying the parallelism of a kernel invocation. An example of a high-level CUDA program is shown in Figure 1.

GPU and CPU program code exist side-by-side in the same source file, with GPU kernel code indicated by the `__global__` qualifier in the function declaration. All kernels are run under the control of the host program. Execution of a kernel is performed using a C-style function call, augmented with a launch configuration within triple chevrons, <<<>>>, that gives the number of blocks and threads per block to be created. Kernel code has access to special variables (`threadIdx` and `blockIdx`) that indicate a given thread's location within the grid of blocks. GPU address space is separate from that of the host program and allocation, initialisation and copies between GPU and host must be performed explicitly using an API (`cudaMalloc`, `cudaMemcpy`, *etc*).

Although CUDA provides a concise, convenient approch to GPU programming , it carries the traditional disadvantage of language extensions, namely that custom compiler support is required. An alternative programing mechanism is via a low-level C API known as the *driver* API. Although this increases the portability of the host-side program, from the programmer's perspective not only is more code required when using the driver API, but it is also inherently more fragile as compile-time checking of the formal arguments of the kernel is no longer possible.

[5]configurable as 16kB to increase L1 cache size

```
 1  /* GPU Kernel */
 2  __global__ void vector_sum( float *in1, float *in2, float *out, int N ) {
 3    int idx = threadIdx.x + blockIdx.x * blockDim.x;
 4
 5    if( idx < N ) out[idx]= in1[idx] + in2[idx];
 6  }
 7
 8  void vector_sum_cuda( float *in1h, float *in2h, float *outh, int N ) {
 9    int grid, block;
10    float *in1, *in2, *out;
11
12    /* Allocate buffers on the GPU */
13    cudaMalloc( (void**) &in1, sizeof( float ) * N );
14    cudaMalloc( (void**) &in2, sizeof( float ) * N );
15    cudaMalloc( (void**) &out, sizeof( float ) * N );
16
17    /* Copy input data to the GPU */
18    cudaMemcpy( in1, in1h, sizeof(float) * N, cudaMemcpyHostToDevice );
19    cudaMemcpy( in2, in2h, sizeof(float) * N, cudaMemcpyHostToDevice );
20
21    /* Launch the vector_sum kernel */
22    block = 512;
23    grid  = (N/block) + 1;
24
25    vector_sum<<< grid, block >>>( in1, in2, out, N );
26    cudaThreadSynchronize();
27
28    /* Copy results back to host */
29    cudaMemcpy( outh, out, sizeof(float) * N, cudaMemcpyDeviceToHost );
30
31    /* Free GPU buffers */
32    cudaFree( in1 );
33    cudaFree( in2 );
34    cudaFree( out );
35  }
```

Figure 1: Example CUDA program. The GPU code (the *kernel*, annotated with the `__global__` qualifier) is defined alongside host program code in the same source file. GPU memory occupies a separate address space to that of the host program and must be allocated and initialised with CUDA API calls. Kernel execution (line 25) resembles a C function call, with the addition of a qualifier ($<<<$ *grid*, *block* $>>>$) that indicates the number and configuration of threads to launch.

2.2. OpenCL device and programming model

OpenCL is an open standard for parallel programming of heterogeneous processors promoted by the Khronos Group [17, 18]. The standard comprises an abstract model for the architecture and memory hierarchy of OpenCL-compliant compute devices, a C programming language for compute device code and a host-side C API. Because the standard has been designed to reflect the design of current hardware it is unsurprising that there are many correspondences between it and CUDA.

The archetypal OpenCL platform consists of a host computer to which is attached one or more *compute devices* each of which is in turn comprised of one or more *compute units*, each having one or more *processing elements*. The execution model for OpenCL consists of the controlling host program and kernels which execute on OpenCL devices. Kernels are executed as multiple instances called *work-items* which are grouped into *work-groups*, analogous with CUDA's threads and blocks.

The OpenCL memory model describes a similar hierarchy to that of CUDA: each compute device has a pool of *global memory* to which all work-items in a work-group may read and write, *local memory* which is local to a work-group, *private memory* private to each work-item and *constant memory*, a read-only region of global memory. Note that, unlike CUDA, the OpenCL specification defines these regions only in terms of their access properties; the relative speeds and physical location of these memories is strictly implementation-specific.

The CUDA nomenclature and corresponding OpenCL terms are summarised in Table 1.

Category	CUDA term	OpenCL term
Hardware	GPU	OpenCL Device
	Multiprocessor	Compute Unit
	Scalar core	Processing Element
	warp	N/A
Memory	Global memory	Global memory
	Device memory	Device memory
	Constant memory	Constant memory
	Shared memory	Local memory
	Local memory (registers or `__local__`)	Private memory
	CUDA Array	Image Object
	Texture Unit	Sampler
Execution	module	program
	kernel	kernel
	thread	work item
	block	work group
	grid	N/A
	stream	Command Queue

Table 1: A summary of the major elements of CUDA terminology and corresponding OpenCL terms.

The OpenCL kernel launch API is substantially more verbose than the higher-level CUDA API (and is very similar to the CUDA driver API). An example is shown in Figure 2. Unlike CUDA, which implicitly manages the GPU and kernels, an OpenCL program must explicitly manage a context and command queue through which it controls the GPU (see `setup()` function). Furthermore, kernels are not afforded the same "first-class" status as in CUDA and must be explicitly compiled and managed as opaque data objects (`clCreateProgramWithSource`, *et seq*).

Lacking the kernel-launch extensions of CUDA, kernel execution in OpenCL is a more verbose process: the formal arguments for a kernel must first be defined using a call to `clSetKernelArg` for each argument, followed by a call to `clEnqueueNDRangeKernel`. In addition to being less concise than CUDA, this method is inherently less robust as it prevents effective compile-time checking of argument types and counts.

2.3. Swan design & implementation

Both CUDA and OpenCL use dedicated compilers to compile kernel source code into a binary form appropriate for the target platform. For CUDA, kernel compilation is typically incorporated into the build of the parent host code. OpenCL, in contrast, performs the kernel compilation phase at runtime of the parent program. In either case, the result is that both OpenCL and driver API CUDA programs must manipulate opaque data objects representing compiled kernels via the respective API calls (Figure 2).

The NVIDIA CUDA compiler, `nvcc`, combines the host program and kernel compilation phases into a single step and produces binary output that contains host object code along with kernel objects embedded as static data. Furthermore, it automatically produces the code required to invoke the kernel, thus hiding the complexity of the driver API upon which the compiled program still ultimately relies.

From the perspective of the CUDA programmer wishing to convert a program to use OpenCL it is this missing functionality in managing kernels that is the most inconvenient omission from OpenCL.

```
 1  #include "CL/cl.h"
 2  #include <string.h>
 3
 4  static const char *vector_sum_source =
 5    "__kernel void vector_sum"
 6    "( __global float *in1, __global float *in2, __global float *out, int N ) {"
 7    " int idx = get_global_id(0);"
 8    "if ( idx < N ) out[idx] = in1[idx] + in2[idx];"
 9    "}";
10
11  static int initialised          = 0;
12  static cl_context        context = NULL;
13  static cl_command_queue  cq      = NULL;
14  static cl_device_id     *devices = NULL;
15  static cl_program        program = NULL;
16  static cl_kernel         kernel  = NULL;
17
18  static void setup( void );
19
20  void vector_sum_opencl( float *in1h, float *in2h, float *outh, int N ) {
21    cl_event event;
22    cl_mem in1, in2, out;
23
24    /* On first call initialise the OpenCL context and kernels */
25    setup();
26
27    /* Allocate buffers on the GPU */
28    in1  = clCreateBuffer( context, CL_MEM_READ_WRITE, N * sizeof(float), NULL, NULL );
29    in2  = clCreateBuffer( context, CL_MEM_READ_WRITE, N * sizeof(float), NULL, NULL );
30    out  = clCreateBuffer( context, CL_MEM_READ_WRITE, N * sizeof(float), NULL, NULL );
31
32    /* Copy input to GPU */
33    clEnqueueWriteBuffer ( cq, in1, 1, 0, sizeof(float) * N, in1, 0, NULL, NULL );
34    clEnqueueWriteBuffer ( cq, in2, 1, 0, sizeof(float) * N, in2, 0, NULL, NULL );
35
36    /* Set launch configuration */
37    size_t bt[3] = {1,1,1}, gt[3] = {1,1,1};
38    bt[0] = 512;
39    gt[0] = bt[0] * (1+ (N/bt[0]));
40
41    /* Set kernel parameters */
42    clSetKernelArg( kernel, 0, sizeof(cl_mem), in1 );
43    clSetKernelArg( kernel, 1, sizeof(cl_mem), in2 );
44    clSetKernelArg( kernel, 2, sizeof(cl_mem), out );
45    clSetKernelArg( kernel, 3, sizeof(int)    , &N   );
46
47    /* Launch kernel */
48    clEnqueueNDRangeKernel ( cq, kernel, 3, 0, gt, bt, 0, NULL,  &event );
49    clWaitForEvents( 1, &event );
50
51    /* Copy data back from GPU */
52    clEnqueueReadBuffer ( cq, out , 1, 0, sizeof(float) * N, outh, 0, NULL, NULL );
53
54    /* Free GPU buffers */
55    clFree( in1 );
56    clFree( in2 );
57    clFree( out );
58  }
59
60  static void setup( void ) {
61    size_t            len;
62    if( initialised ) { return; }
63
64    /* Create context */
65    context = clCreateContextFromType( NULL, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL );
66    clGetContextInfo( context, CL_CONTEXT_DEVICES, 0, NULL, &len );
67    devices = (cl_device_id *) malloc( len );
68    clGetContextInfo( context, CL_CONTEXT_DEVICES, len, devices, NULL );
69
70    /* Create command queue associated with a GPU device */
71    cq = clCreateCommandQueue( context, devices[ 0 ], 0, NULL );
72
73    /* Build the GPU program */
74    len = strlen( vector_sum_source );
75    program = clCreateProgramWithSource( context , 1, &vector_sum_source, &len, NULL );
76    clBuildProgram( program , 0, NULL, NULL, NULL );
77    clCreateKernelsInProgram( program, 0, NULL, NULL );
78    clCreateKernelsInProgram( program, 1, &kernel, NULL );
79    initialised = 1;
80  }
```

Figure 2: Example OpenCL program. The CUDA program from Figure 1 recast as an OpenCL program. Unlike the CUDA high-level API, device contexts and program objects must be explicitly managed (`setup()` function). Kernel source is included as a string which is passed to the OpenCL compiler at runtime. Kernel execution (lines 42-49) require explicit marshalling of the formal arguments.

```
 1  /* #Include swan-generated kernel entry point*/
 2  #include "vector_sum.kh"
 3
 4  void vector_sum_swan( float *in1h, float *in2h, float *outh, int N ) {
 5    dim3 grid, block;
 6    float *in1;
 7    float *in2;
 8    float *out;
 9
10    /* Allocate buffers on the GPU */
11    in1 = (float*) swanMalloc( sizeof(float) * N );
12    in2 = (float*) swanMalloc( sizeof(float) * N );
13    out = (float*) swanMalloc( sizeof(float) * N );
14
15    /* Copy input data to the GPU */
16    swanMemcpyHtoD( in1h, in1, sizeof(float) * N );
17    swanMemcpyHtoD( in2h, in2, sizeof(float) * N );
18
19    /* Launch the vection_sum kernel */
20    swanDecompose( &grid, &block, N, 512 );
21    k_vector_sum( grid, block, 0, in1, in2, out, N );
22
23    /* Copy results back to the host */
24    swanMemcpyDtoH( out, outh, sizeof(float) * N );
25
26    /* Free GPU buffers */
27    swanFree( in1 );
28    swanFree( in2 );
29    swanFree( out );
30  }
```

Figure 3: Example SWAN program. The CUDA program from Figure 1 recast as a Swan program. The kernel code (unchanged from the CUDA form) is compiled by the `swan` tool which produces a C header file that contains embedded kernel object code and generated C function entry points for each kernel. Kernel execution is thus via a conventional C function call (line 21) with the launch parameters included in the formal parameters.

The Swan tool was developed to facilitate a port of an existing CUDA code base to OpenCL by minimising the amount of code refactoring and rewriting required. A principal design goal was to reproduce the convenience and compile-time checking robustness of the CUDA language extensions within a standard C framework. An example of the CUDA program in Figure 1 re-cast to use Swan (Figure 3) gives a flavour of how this is achieved.

Swan is composed of two components: a source-code processing tool, `swan`, and a runtime library `libswan`. `swan` performs a similar function to that of `nvcc`: it analyses and compiles CUDA kernel source-code and outputs generated source-code that includes both a data block containing the compiled kernel binaries and C functions that serve as entry-points for calling them. `swan` also performs a source-to-source translation when compiling CUDA kernels for an OpenCL target.

`libswan` is a library that provides a complete abstraction of CUDA or OpenCL. It provides a minimal C API modelled after the CUDA driver API, along with the CUDA vector data-types (OpenCL defines vector types only within the kernel language). There are separate implementations of the library for CUDA and OpenCL, with the appropriate one selected by the user at link time.

We describe below the major implementation features of these two components:

2.3.1. Kernel compilation

The CUDA kernel source code is first passed through the C pre-processor to resolve any macros and then passed to the appropriate compiler to be compiled to byte-code. If the target is CUDA, `nvcc` is invoked with the `--cubin` switch so that emits compiled kernel object code. The OpenCL specification does not define any command line compiler, so Swan includes an auxiliary program `swan-oclc` that acts as a wrapper to the OpenCL kernel compilation API and outputs compiled binary object.

Some contemporary OpenCL tool-chains do not support the generation of binary representations of program objects, so Swan also provides the facility to embed the (post-processed, compilation-validated) source and perform run-time compilation.

2.3.2. Entry-point generation

`swan` inspects the kernel source code and extracts the formal definition of each kernel. From this, it generates the C source code for a function that represents an entry point for invoking the kernel from the host code. This function has formal arguments which match those of the kernel, along with additional arguments that represent the kernel launch parameters (*ie* those passed via the CUDA <<<>>> notation). The function body contains setup code to initialise `libswan` with the kernel binary followed by a call to the `libswan` function `swanRunKernel()`. Each argument to the entry-point function is passed to `swanRunKernel()` along with a flag indicating its type.

Entry points are systematically named by appending `k_` to the original kernel name.

2.3.3. Source translation

When processing kernel code for an OpenCL target, `swan` performs a source-to-source translation. This conversion does not require a complete C parser: because of the close similarities between the CUDA and OpenCL C dialects, it is possible to perform this conversion using a set of regular expression substitutions. CUDA templated kernels are not supported.

Unlike CUDA, OpenCL does not support global (file-level) variables. Any that are present in the CUDA source are removed and identically-named pointers that are passed as additional formal parameters to each kernel. At initialisation of the program, storage for each global variable is then allocated dynamically. As with the CUDA driver API, the host program uses a C string containing the variable name as a handle to refer to a global variable. Similarly, a dummy variable with `__local` placement is allocated and a pointer passed to represent any dynamic shared memory used by the CUDA kernel. CUDA textures are re-written as OpenCL Images.

2.3.4. `libswan` API

The `libswan` API is modelled after the CUDA driver API. Memory allocation and copy functions have a direct correspondence to CUDA API functions. For texture operations, higher level functions are defined to better hide the differences between CUDA texture and OpenCL image and sampler setup. The vector data-types provided by CUDA are re-implemented in Swan using the OpenCL vector types defined for use within the kernel C language.

There are two implementations of `libswan`, one for CUDA and one for OpenCL. The latter is appropriate for any compliant OpenCL implementation. The choice of OpenCL or CUDA target is made at compile time.

Kernel	% of runtime (CUDA)	relative speed (OpenCL/CUDA)
Particle binning	2.3	3.6
Bonded forces	3.6	1.6
Non-bonded forces	93.8	1.4
Integration	0.3	2.2
Total		1.5

Table 2: Relative timings of the four principal kernel-groups of ACEMD (20 kernels total). Test hardware: HP xw6600 dual Xeon 5430 workstation, NVIDIA Tesla c1060 GPU, Cuda 3.0, Red Hat 5.4. Model system was Gramicidin-A (29042 atoms) with a direct electrostatic cutoff of $12\AA$, VdW switching at $10.5\AA$.

3. Results and Discussion

Swan was developed to simplify an experimental port of ACEMD from CUDA to OpenCL. ACEMD is a classical molecular dynamics code that achieves very high performance on NVIDIA hardware[3]. It is used in the GPUGRID distributed computing project[19] and there is a strong incentive to be able to run on AMD hardware (roughly half of discrete GPU market) if acceptable performance could be achieved. The kernel code has been extensively optimised for the NVIDIA G200 and GF100 architectures.

The strategy followed in the porting exercise was as follows: Firstly the code was converted to use Swan to manage all kernels, whilst maintaining CUDA as the compilation target using the following steps:

- The starting CUDA source-code was re-organised to segregate kernel and host code into separate files.
- All CUDA API calls replaced with Swan equivalents. As ACEMD had already wrappered these, these changes were localised to only a few files. Code continued to be compiled with `nvcc` but linked against `libswan`
- The Makefile was changed to compile kernel source with `swan` and the source files to include the output Swan kernel headers.
- Kernel invocations in host source code were changed to use Swan entry-points.
- `nvcc` replaced with the system C compiler for host-side code compilation.

Once done, the conversion of the code from CUDA to OpenCL could be achieved by simply changing the Swan build target to OpenCL and recompiling and re-linking against the appropriate version of `libswan`.

The performance of the CUDA and OpenCL versions ACEMD is compared in Table 2. Overall, the runtime of the OpenCL version is seen to be 50% greater than that of the CUDA version, with the performance of the the primary computational step – the non-bonded force computation – a factor of 1.4 slower. By analysing the intermediate (PTX) code produced by the CUDA and OpenCL compilers, it was apparent that the CUDA compiler was able to emit more efficient code, both in terms of instruction counts and register usage. Since increasing the register usage affects the number of threads that may be executed concurrently this can have a significant effect on performance. In their work on optimising the Netlib linear algebra library for GPUs, Dongarra *et al*[20] also report that the contemporary NVIDIA OpenCL compiler produces less optimised PTX (assembly) code in comparison to that of the CUDA compiler.

The OpenCL port of ACEMD was fully-featured with respect to the CUDA version and produced numerically identical results when run on the same hardware platform.

The OpenCL version of ACEMD was also tested successfully on AMD hardware (Radeon HD5850, OpenCL SDK 2.2). Significantly, the same program executable (with kernels embedded as source-code) was able to run on both NVIDIA and AMD hardware without recompilation. Performance (not shown) was significantly worse than a prediction based on the relative specifications of the NVIDIA and AMD GPUs would have suggested and clearly indicated that further optimisation would need to be undertaken to optimise kernel code for the VLIW architecture of the HD5850's Evergreen processor[21, 22]. Nevertheless, numerical results were found to be in agreement with those obtained with the reference CUDA version. An analysis of the performance of ACEMD on AMD hardware will be presented in a separate publication.

4. Conclusions

We have described a tool designed to facilitate the porting of a CUDA code to OpenCL by minimising the source-code changes that must be made manually.

Development of this tool was motiviated by the desire to evaluate the performance of an OpenCL version of the authors' real-world CUDA application, ACMED. ACEMD is ≈ 100 thousand lines of source code, of which $\approx 10\%$ is GPU code in ≈ 120 kernels. The conversion of ACEMD from high-level CUDA to Swan took a matter of days.

Through analysis of the resulting ported code, we observe that a performance reduction may be expected when converting a performance-optimised CUDA code to OpenCL, as a result of the current NVIDIA OpenCL compiler producing less efficient code. The performance discrepancy between CUDA and OpenCL varies with the task and is most significant for kernels which are highly resource-optimised. It is expected that the performance of OpenCL code will approach parity with CUDA as compilers mature.

The ported ACEMD was found to be able to run on both NVIDIA and AMD hardware without changes. Such hardware-independence constitutes a significant benefit for the scientific programmer as it reduces the cost of the software engineering required to port or to maintain multi-platform support within an application.

We conclude that OpenCL is a viable platform for developing portable GPU programs, but that for performance-sensitive code, CUDA remains the most mature and effective programming tool for NVIDIA GPUs. Using a tool like Swan to abstract the differences between CUDA and OpenCL, the programmer may continue to develop a single code-base using CUDA programming techniques but selectively compile for either platform. This flexibility offers an approach to developing portable GPU code that does not compromise performance.

Software availability Swan is freely available from `http://www.multiscalelab.org/swan` and licensed for academic use under the GNU General Public License (GPL) version 2.

Acknowledgements MJH gratefully acknowledges support from the HPC-EUROPA2 project (project number 228398) with the support of the European Commission Capacities Area - Research Infrastructures Initiative, and from the Imperial College High Performance Computing Service. GDF acknowledges support from the Ramòn y Cajal scheme and from the EU Virtual Physiological Human Network of Excellence. The authors thank Acellera Ltd (`http://www.acellera.com`) for use of its resources.

5. Program Summary

*Manuscript Title:*Swan: A tool for porting CUDA programs to OpenCL

Authors: M. J. Harvey and G. De Fabritiis

Program Title: Swan

Journal Reference:

Catalogue identifier:

Licensing provisions: GNU Public License version 2

Programming language: C

Computer: PC

Operating system: Linux

RAM: 256 Mbytes

Number of processors used: 1

Supplementary material:

Keywords: GPU CUDA OpenCL

Classification: 6.5 Software including Parallel Algorithms

External routines/libraries: NVIDIA CUDA, OpenCL

Subprograms used: NVIDIA CUDA compiler, Perl

Nature of problem: Graphical Processing Units (GPUs) from NVIDIA are preferentially programed with the proprietary CUDA programming toolkit. An alternative programming model promoted as an industry standard, OpenCL, provides similar capabilities to CUDA and is also supported on non-NIVIDA hardware (including multicore x86 CPUs, AMD GPUs and IBM Cell processors). The adaptation of a program from CUDA to OpenCL is relatively straightforward but laborious. The *Swan* tool facilitates this conversion.

Solution method: Swan performs a translation of CUDA kernel source code into an OpenCL equivalent. It also generates the C source code for entry point functions, simplifying kernel invocation from the host program. A concise host-side API abstracts the CUDA and OpenCL APIs. A program adapted to use *Swan* has no dependency on the CUDA compiler for the host-side program. The converted program may be built for either CUDA or OpenCL, with the selection made at compile time. *Restrictions:* No support for CUDA C++ features

Unusual features: N/A

Additional comments: N/A

Running time: Nominal

References

[1] Ufimtsev, I. S. and Martínez, T. J., J. Chem. Theory Comput. **4** (2008) 222.

[2] Anderson, J. A., Lorenz, C. D., and Travesset, A., J. Comp. Phys. **227** (2008) 5342.

[3] Harvey, M., Giupponi, G., and De Fabritiis, G., J. Chem. Theory and Comput. (2009) 1632.

[4] Elson, E., LeGresley, P., and Darve, E., J. Comp. Phys. **227** (2008) 10148.

[5] Tölke, J. and Krafczyk, M., International Journal of Computational Fluid Dynamics **22** (2008) 443.

[6] Simek, V., Dvorak, R., Zboril, F., and Kunovsky, J., Towards accelerated computation of atmospheric equations using CUDA, in *UKSim 2009: 11th International Conference on Computer Modelling and Simulation*, pages 449–454, 2009.

[7] McGraw, T. and Nada, M., IEEE Transactions on Visualization and Computer Graphics **13** (2007) 1504.

[8] Wein, W., Brunke, S., Khamene, A., Callstrom, M. R., and Navab, N., Medial Image Analysis **12** (2008) 577.

[9] Balevic, A. et al., Accelerating simulations of light scattering based on finite-difference time-domain method with general purpose GPUs, in *Proceedings of the 2008 11th IEEE International Conference on Computational Science and Engineering*, pages 327–334, 2008.

[10] Dziekonski, A., Sypek, P., Kulas, L., and Mrozowski, M., Implementation of matrix-type FDTD algorithm on a graphics accelerator, in *Microwaves, Radar and Wireless Communications, 2008. MIKON 2008. 17th International Conference on*, pages 1–4, 2008.

[11] Advanced Micro Devices, Compute abstraction layer technology intermediate language, Technical report, Advanced Micro Devices, Inc, 2010.

[12] Daniel Hillis, W. and Steele Jr., G. L., Communications of the ACM **29** (1986).

[13] Buck, I. et al., Brooks for GPUs: stream computing on graphics hardware, in *ACM Transactions on Graphics, Proceedings of ACM SIGGRAPH 2004*, volume 23, pages 777–786, 2004.

[14] The Portland Group, PGI Fortran & C accelerator programming model, Technical report, The Portland Group, 2009.

[15] NVIDIA, NVIDIA CUDA programming guide version 2.3, Technical report, NVIDIA Corporation, 2009.

[16] NVIDIA, NVIDIA's next generation CUDA compute architecture: Fermi, Technical report, NVIDIA Corporation, 2008.

[17] Group, K. O. W., The OpenCL specification, version 1.0 revision 48, Technical report, The Khronos Group, 2009.

[18] Stone, J. E., Gohara, D., and Shi, G., Computing Science and Engineering **May/June** (2010) 66.

[19] Buch, I., Harvey, M. J., Giorgino, T., Anderson, D. P., and De Fabritiis, G., J. Chem. Inf. Mod. **50** (2010) 397.

[20] Du, P. et al., UT-CS-10-656, from CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming., Technical report, University of Tennessee, 2010.

[21] Advanced Micro Devices, Evergreen family instruction set architecture instructions and microcode, Technical report, Advanced Micro Devices, 2010.

[22] Advanced Micro Devices, R600/r700/evergreen assembly language format, Technical report, Advanced Micro Devices, 2010.