OpenCL Evaluation for Numerical Linear Algebra Library Development

Peng Du*, Piotr Luszczek*, Jack Dongarra*†‡
*University of Tennessee Innovative Computing Laboratory
†Oak Ridge National Laboratory
‡University of Manchester

I. Portable GPU Programming

With the help of of CUDA [7], [6], many applications improved their performance by using GPUs. In our project called Matrix Algebra on GPU and Multicore Architectures (MAGMA) [10], we mainly focus on dense linear algebra routines similar to those from LAPACK [1]. Other than CUDA, there exist other frameworks that allow platform-independent programming for GPUs. The main three frameworks are:

1) DirectCompute from Mircosoft,
2) OpenGL Shading Language (GLSL), and
3) OpenCL

The first one allows access to graphics cards from multiple vendors. However, it is specific to Microsoft Windows and therefore it is not portable between host Operating Systems (OS).

OpenGL Shading language [8] is portable across both GPU hardware and the host OS. However, it is specifically geared towards programming new graphics effects – GLSL does not have the scientific focus.

OpenCL [3] has been designed for general purpose computing on GPUs (GPGPU). It is an open standard maintained by the Khronos group with the backing of major graphics hardware vendors as well as large computer industry vendors interested in off-loading computations to GPUs. As a result, there exist working OpenCL implementations for graphical cards and, in addition, there is an implementation that works without a GPU by off-loading computations to multi-core processors. As a result, OpenCL offers portability across GPU hardware, OS software, as well as multicore processors. Therefore OpenCL is our choice of implementing a portable numerical linear algebra library.

II. Comparison between CUDA and OpenCL

CUDA and OpenCL have many conceptual similarities but they diverge on terminology. Table I shows the corresponding terms in both frameworks. In addition, Table II shows the platform details in between two different NVIDIA GPUs and one GPU from ATI/AMD.

Figure 1 shows side-by-side differences of the kernel codes for triangular inversion routine (`TRTI2`) for OpenCL and CUDA. The changes are in the lines annotated in red. They belong to the following categories:

CUDA term	OpenCL term
host CPU	host
streaming multiprocessor (SM)	compute unit (CU)
scalar core	processing element (PE)
host thread	host program
thread	work-item
thread block	work-group
grid	NDRange
shared memory	local memory
constant memory space	constant memory
texture memory space	constant memory

TABLE I
Comparison of terms used by CUDA and OpenCL to describe very similar concepts.

- Obtaining the ID for the thread/work-item and block/work-group.
- The definition of shared memory in CUDA is replaced in OpenCL by local memory: `__shared__` is replaced with `__local`
- OpenCL makes explicit differentiation between global memory addresses (device memory address space) and local memory addresses (register variable or pointer to shared memory) whereas CUDA treats all pointers as simply `double *`.
- Syntax for synchronization primitives.

III. Design for a Math Library based on OpenCL

To make an assessment about developing linear algebra library using OpenCL, this work is based on an algorithm for triangular solvers on GPU. The basic idea is to invert the diagonal blocks in parallel and use matrix multiplication (GEMM) to update the solution.

Figure 2 shows the breakdown of run time of the initialization procedure. `oclDtrsm` (triangular solver written in

GPU Device	GTX 280	C2050 (Fermi)	Radeon 5870
Compute Units	30	32	20
Processing elements	8	16	16

TABLE II
Comparison of computational resources available on NVIDIA's GTX 280

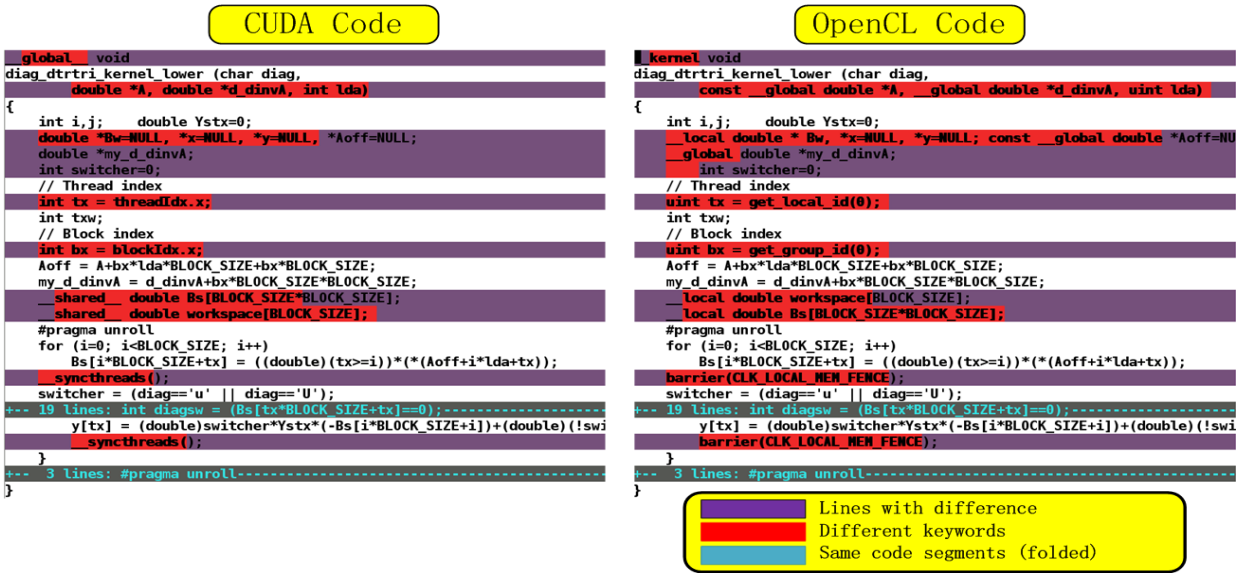

Fig. 1. Comparison of Device Kernel Code Between OpenCL and CUDA.

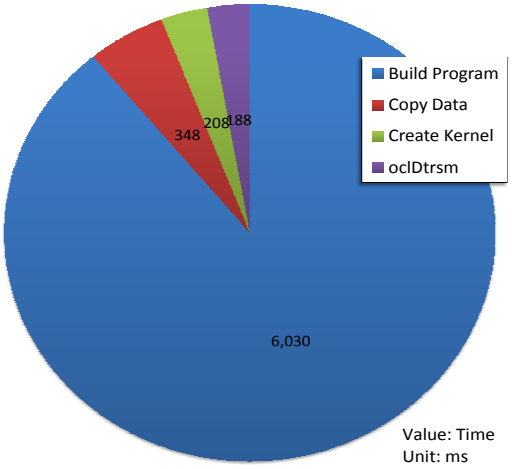

Fig. 2. Breakdown of initialization time.

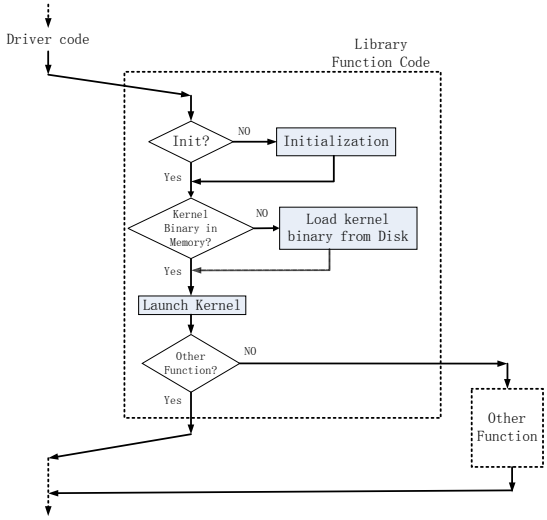

Fig. 3. Flow chart of loading procedure for the library kernels.

OpenCL using double precision) is run with M=10240 and NRHS=128. Each of the three major parts of the initialization run slower than the GPU kernel of oclDtrsm. Compiling OpenCL source code into a binary form (PTX assembly on CUDA) takes the longest time among all parts. Similar effect is also observed on an ATI HD 5870 GPU with the same OpenCL code using ATI STREAM SDK [2]. into 7000+ lines of PTX assembly code takes just over 6 seconds, One solution to reduce this overhead is to separate compiling and execution – source code compilation could just be done once during the library installation process. As of this writing, there is no off-line kernel compiler in the implementation of OpenCL from NVIDIA. It is suggested [4] for this to be implemented by the end users. On NVIDIA GPU, we used clGetProgramInfo to obtain the PTX assembly and write it to disk. During the initialization phase, the assembly is read from the disk and processed by a call to clBuildProgram. This method reduced the time to prepare the PTX assembly from 6 seconds to under one second (0.8 s). Still, the time to create a kernel from a pre-built program takes more time than the computation. For example, the GEMM kernel is invoked by oclDtrsm more than 600 times. We used static pointer variables in our implementation to reuse the compiled kernels.

For software libraries that target one specific device, we propose a mechanism shown in Figure 3 that does way with portability. In this design, device kernels are compiled and stored on disk during the installation and transformed into kernel executables as needed.

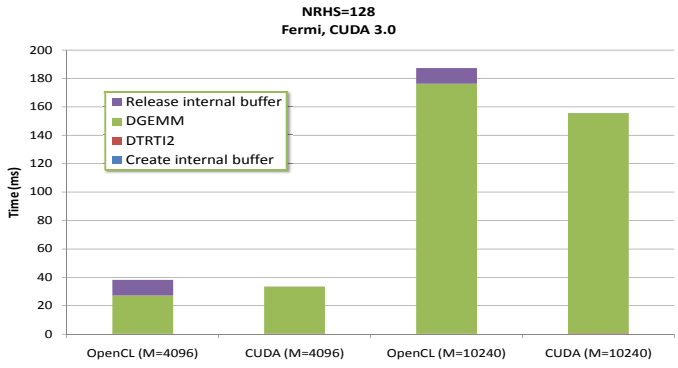

Fig. 4. Analysis of time spent in the complete kernel.

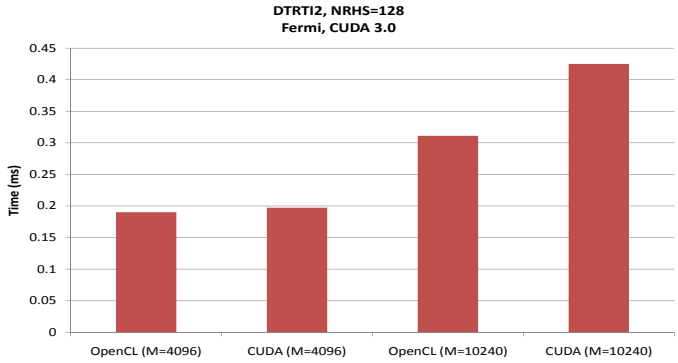

Fig. 5. Analysis of time spent in DTRTI2 kernel.

IV. Performance Results

Figure 4 shows the run time breakdown of the kernel in algorithm 1 in double precision (hence a 'D' in front of GEMM). M=4096 and M=10240 represent small and large problem sizes respectively, and the number of right hand side (NRHS) is 128 which creates sufficient amount of calls to DGEMM. The time to allocate and release an internal buffer is also recorded. This internal buffer in the algorithm is used to keep the inverted diagonal blocks. The size of this buffer is M by 32. Since the run time of DTRTI2 in all cases is too small compared to the other parts, its run time is shown in Figure 5. The time is small because all diagonal blocks are inverted in parallel. This task is on the critical path of the triangular solvers and it could be the bottleneck if done one by one interleaved with GEMM. It is worth observing that the allocation of internal buffer takes negligible time, but releasing the memory of internal buffer takes much longer time than in CUDA. The opposite is true for the ATI OpenCL SDK: it appears that this is an implementation issue. Also, CUDA and OpenCL profilers show conflicting timing of DTRTI2: device kernel in DTRTI2 runs faster with CUDA than OpenCL. We attribute this to different overhead for both profilers. For more accurate conclusions, we based our analysis mainly on the TAU profiling [9].

The chart in Figure 6 shows the results on the NVIDIA card with the most current CUDA 3.0 driver and the corresponding SDK. On the same hardware, OpenCL code is able to reach 60%-70% performance of CUDA code. The combination of our diagonal block inversion algorithm with reduction of overhead (due to the use of precompiled kernels) for OpenCL allows the algorithm on both CUDA and OpenCL achieve at least 2-fold improvement over the cublasDtrsm from CUBLAS [5].

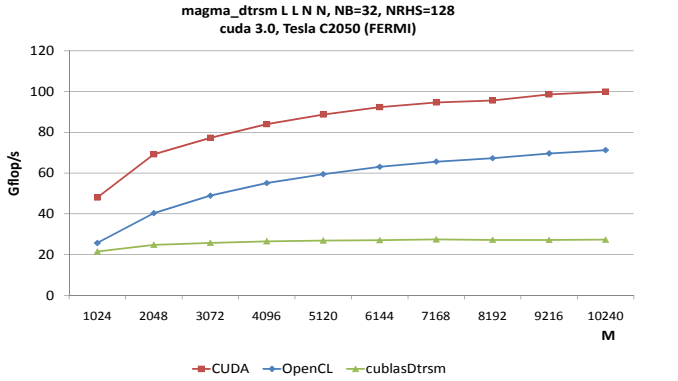

Fig. 6. Performance comparison between CUDA and OpenCL on NVIDIA's Fermi.

Algorithm 1: oclDtrsm Algorithm

```
Create internal buffer to store the inverted diagonal blocks;
Invert the diagonal blocks (TRTI2);
while there are still blocks of X to solve do
    Update solution in X (GEMM);
    Update the right hand side (GEMM);
end
Release the internal buffer;
```

References

[1] E. Anderson, Z. Bai, C. Bischof, Suzan L. Blackford, James W. Demmel, Jack J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and Danny C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition, 1999.

[2] ATI. ATI Stream Software Development Kit (SDK) v2.1, 2010. Available at: http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx.

[3] Aaftab Munshi, editor. *The OpenCL Specification*. Khronos OpenCL Working Group, October 6 2009. Version: 1.0, Document Revision:48.

[4] NVIDIA OpenCL JumpStart Guide: Technical Brief, April 2009. Version 0.9.

[5] CUBLAS library, February 2010. Version PG-00000-002_V3.0.

[6] NVIDIA. *NVIDIA CUDA™ Best Practices Guide Version 3.0*. NVIDIA Corporation, February4 2010.

[7] NVIDIA. *NVIDIA CUDA™ Programming Guide Version 3.0*. NVIDIA Corporation, February20 2010.

[8] Randi J. Rost, Dan Ginsburg, Bill Licea-Kane, John M. Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen. *OpenGL Shading Language*. Addison-Wesley Professional, 3rd edition, July 13 2009.

[9] S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[10] S. Tomov, R. Nath, P. Du, and J. Dongarra. MAGMA version 0.2 User Guide. http://icl.cs.utk.edu/magma, 11/2009.