CUDA-on-CL: a compiler and runtime for running NVIDIA® CUDA™ C++11 applications on OpenCL™ 1.2 Devices

Hugh Perkins
ASAPP Inc, 304 Hudson Street, NY 10013
hughperkins@gmail.com

ABSTRACT

In the machine learning domain, machine learning frameworks are predominantly written and maintained in NVIDIA® CUDA™ language. There have been attempts to port these frameworks to OpenCL® , notably the ports of Caffe framework by Gu et al; Tschopp; and Engel; and of Torch framework by Perkins. The authors of these frameworks found merging their work into the mainstream framework challenging, and maintain their forks as separate branches or repositories. CUDA-on-CL addresses this problem by leaving the reference implementation entirely in NVIDIA CUDA, both host-side and device-side, and providing a compiler and a runtime component, so that any CUDA C++11 application can in theory be compiled and run on any OpenCL 1.2 device. We use Tensorflow framework as a case-study, and demonstrate the ability to run unary, binary and reduction Tensorflow and Eigen kernels, with no modification to the original CUDA source-code.

Performance studies are undertaken, using the Tensorflow kernels. For buffer sizes of 1MB or more, performance is comparable between CUDA and CUDA-on-CL, across unary operations, binary operations and single-axis reductions. Full reduction is around 14 times slower on CUDA-on-CL than on CUDA. We think this may be because of the absence of the low-level hardware `shfl` operation. The asymptotic time for zero buffer sizes is double that of CUDA, possibly because of the overhead of additional kernel boilerplate needed to workaround limitations in the OpenCL 1.2 standard.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**; • **Software and its engineering** → **Compilers**; *Source code generation*;

KEYWORDS

GPGPU, OpenCL, CUDA, Machine Learning, Compilers, LLVM

ACM Reference format:
Hugh Perkins. 2017. CUDA-on-CL: a compiler and runtime for running NVIDIA® CUDA™ C++11 applications on OpenCL™ 1.2 Devices. In *Proceedings of IWOCL '17, Toronto, Canada, May 16-18, 2017,* 4 pages.
DOI: http://dx.doi.org/10.1145/3078155.3078156

IWOCL '17, Toronto, Canada
 978-1-4503-5214-7/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3078155.3078156

1 INTRODUCTION

In the machine learning community, high-performance machine learning libraries are used to fit high-capacity mathematical models to datasets, and enable predictions on previously unseen test data. The nature of the high-capacity models places high demands on the machine learning libraries in terms of runtime execution speed, and available mathematical functionality. To achieve state of the art results, the latest discrete GPUs are typically used, eg [13], although this is not the only approach possible [7].

Amongst discrete GPUs, those produced by NVIDIA are among the fastest available, and mainstream machine learning libraries, such as Tensorflow [1], Caffe [11] or Torch [4], typically target usage primarily on NVIDIA devices. Using hardware from other vendors, such as AMD or Intel, is typically not easily possible, because the core source-code is written using NVIDA® CUDA™ [12], which is NVIDIA-specific.

1.1 Source-code language

Attempts have been made to port the mainstream machine learning libraries from NVIDIA CUDA to the portable GPGPU language OpenCL [17]. Caffe was ported by Engel [5]; by Gu et al [10]; and by Tschopp [18]. Torch was ported by Perkins [15]. In each case, the OpenCL source-code was maintained as a separate fork, evolving somewhat independently of the upstream NVIDIA CUDA codebase. Upstream changes to the CUDA source-code need to be painstakingly ported across to OpenCL. Similarly, innovations within the OpenCL implementation cannot easily be merged into upstream, but need to back-ported, into NVIDIA CUDA language.

Such an approach is high-maintenance. A more general solution would be to maintain one single code-base, that can run on devices from any vendor. For example, the code could be written only in OpenCL. In practice, high-performance libraries must provide a CUDA implementation, because the OpenCL ports run significantly slower on NVIDIA devices than the original CUDA versions. Therefore, migrating the reference code-base from NVIDIA CUDA to OpenCL is not achievable realistically at this time.

A more pragmatic solution could be to leave the existing code-bases as-is, in NVIDIA CUDA, and to instead provide a compiler and runtime, to run this code on non-NVIDIA devices. AMD's HIP compiler [2] uses this approach, for AMD devices. Such an approach could be generalized to other vendors, on a vendor-by-vendor basis. The approach theoretically offers high performance because each hardware vendor can make optimal use of vendor-specific hardware optimizations. However, to date, only AMD provide an NVIDIA CUDA source-code compiler, and a more general approach might be useful.

SYCL [16] implements much of the functionality of NVIDIA CUDA, in a similar language. Modifications required to port from

NVIDIA CUDA are fewer, compared to OpenCL. In addition, SYCL is an open standard. However, NVIDIA CUDA code cannot be compiled directly: host-side api calls to the NVIDIA CUDA API need to be replaced by SYCL equivalents. On the device-side, API calls, such as `threadIdx.x`, need to be migrated.

1.2 Communication with GPU Driver

SYCL standard provides a solution to reading NVIDIA CUDA source-code. It does not address the question of how to provide the resulting parse-tree, or bytecode, to the GPU driver. Two approaches are:

- convert the code into vendor-specific bytecode
- convert into SPIR[8], or SPIR-V[9], bytecode

AMD's HIP employs the first approach, which is likely to give excellent performance, but is vendor-specific.

SPIR and SPIR-V are open standards for providing bytecode directly to a compatible GPU driver, in a cross-platform, portable way. This is a general approach, whilst efficient, and potentially providing access to high-performance low-level optimizations. Two implementations of SYCL using SPIR are Codeplay's ComputeCpp [3], and Keryell et al's triSYCL implementation [6].

SPIR-V is a new standard, and driver support is just beginning. ComputeCpp, using SPIR-1.2, supports AMD Fiji and Hawaii GPUs, and Intel Gen 9 CPUs/GPUs (personal communication, 20 January 2017). An opportunity could exist, in the short-term, for a more general approach.

1.3 NVIDIA CUDA API

In order to be able to run NVIDIA CUDA source-code unmodified, it is not sufficient to be able to compile the C++11 language. The source-code contains calls to the NVIDIA CUDA API, on both the host-side and the device-side. In addition, in the machine-learning domain, extensive use is made by libraries of cuBLAS API. SYCL does not provide a solution to this challenge. In a pure SYCL implementation, one would need to migrate each of the CUDA API calls to SYCL-specific API calls. An additional challenge with this is that some NVIDIA CUDA functionality, such as GPU buffer pointer arithmetic, is not supported by either OpenCL or SPIR-V standards, and a trivial migration, eg by updating function names, is not in general possible.

Neither triSYCL, ComputeCpp, nor HIP address the issue of the NVIDIA CUDA API: migrating the API needs to be handled by modifying the original NVIDIA CUDA source-code.

2 CUDA-ON-CL

CUDA-on-CL has the following goals:

- Run NVIDIA CUDA source-code unchanged, both host-side and device-side
- Portable across a broad range of devices
- Provide cuBLAS API

Therefore, CUDA-on-CL is implemented as follows:

- Uses an existing open-source CUDA parser - CLANG - to parse the NVIDIA CUDA source-code files into LLVM bytecode
- Writes the device-side LLVM bytecode as OpenCL 1.2

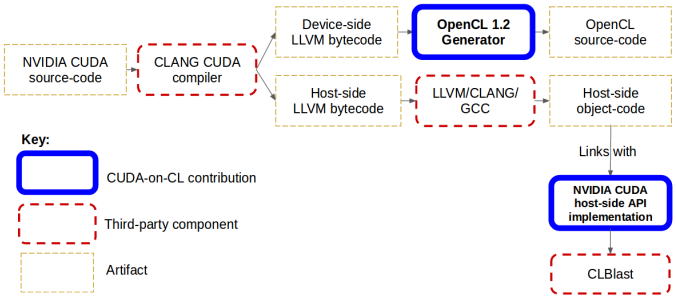

Figure 1: CUDA-on-CL Architecture

- Expose an implementation of the NVIDA CUDA host-side API

In the future, as SPIR-V support becomes more wide-spread, the OpenCL 1.2 generation component can be dropped, and the LLVM bytecode can be injected directly into the GPU drivers, as SPIR-V bytecode.

CUDA-on-CL provides a partial implementation of the cuBLAS API, using Nugteren's CLBlast [14]. CLBlast is a portable C++11 implementation of BLAS for OpenCL.

On the host-side, a virtual memory management module is provided, to allow GPU buffer pointer arithmetic. Across the host-side/device-side boundary, kernel method parameters are modified and extended as required to handle limitations of OpenCL 1.2 standard. On the device-side, shims are provided as necessary to handle any missing low-level hardware optimizations.

Thus, by comparison with the existing ecosystem, CUDA-on-CL:

- is very general, running theoretically on any OpenCL 1.2 device
- can compile and run NVIDIA CUDA source-code unchanged
- provides a partial implementation of cuBLAS API

Figure 1 provides a schematic of the CUDA-on-CL high-level architecture.

Having presented the high-level challenges of CUDA-on-CL, and how CUDA-on-CL solves these, let's look at detailed, low-level challenges.

2.1 Detailed Challenges

Some key challenges associated with compiling and running NVIDIA CUDA source-code on OpenCL 1.2 devices are:

- Need to replace device-side CUDA calls, such as `ThreadIdx.x` with OpenCL device-side calls, such as `get_local_id(0)`
- OpenCL 1.2 forbids pointer arithmetic on `cl_mem` buffers
- OpenCL 1.2 forbids passing structs by-value to kernels
- OpenCL 1.2 forbids passing pointers in structs to kernels
- OpenCL 1.2 requires all pointers to be explicitly labeled with their address-space, ie global, local, or implicitly private
- OpenCL 1.2 lacks many low-level device-side instructions such as shfl

Conceptually, the solution to many of these challenges is similar:

- We control the source-code and /or bytecode on both the host-side and the device-side

- Therefore we are free to modify the code as we wish

In general each challenge is solvable, though there might be a cost, in terms of CUDA-on-CL internal development time, and application execution speed. We will describe the solution implemented by CUDA-on-CL for four representative low-level challenges.

2.2 Pointer arithmetic

One of the challenges faced during the manual porting of Caffe to OpenCL was the use of host-side pointer offsets. A single pointer in NVIDIA CUDA represents both a GPU-allocated buffer, and an offset into that buffer. In OpenCL, two values are needed, one to hold the offset. This change needs to be threaded through all methods and objects that process the pointer.

CUDA-on-CL handles this issue by implementing its own virtual memory management. The pointer provided to host-code is not the actual cl_mem buffer pointer, but a virtual pointer. By comparing a received virtual pointer with a table of allocated objects, CUDA-on-CL deduces the underlying `cl_mem` object and the offset.

2.3 Address space declarations

OpenCL requires all pointers to have their address-space statically declared, ie global, local, or implicitly private. This requires walking the execution path to determine the address-space of each pointer and function parameter. In general, the address-space cannot be determined prior to execution. Therefore the OpenCL generation occurs at runtime. We note that each function might be called multiple times, with different combinations of parameter address-spaces. Therefore, CUDA-on-CL clones each function, for each required combination of address-space assignments, and assigning appropriately mangled names.

2.4 Low-level hardware operations

Low-level hardware operations, such as `shfl`, are not available in OpenCL. Therefore these are implemented via shims. Theoretically, on hardware that provides an appropriate hardware level implementation, eg available via inline assembler in the case of NVIDIA, or via OpenCL 2.0 sub-groups on certain other drivers, we could conditionally substitute in a more performant implementation.

2.5 Kernel parameter structs containing pointers

OpenCL forbids passing by-value structs containing global pointers as kernel parameters. However, this is allowed by NVIDIA CUDA. A solution implemented by CUDA-on-CL is to:

- clone such struct definitions into a struct with no pointers
- on the hostside, copy the values from the old struct to the pointer-free struct
- pass the pointer-free struct into the kernel, along with the pointers as separate kernel parameters
- on the kernel side, allocate a struct of the original type, copy across the data from the pointer-free struct, and assign the global pointer parameters to the struct

This produces quite lengthy boilerplate, however it only needs to occur once per kernel launch.

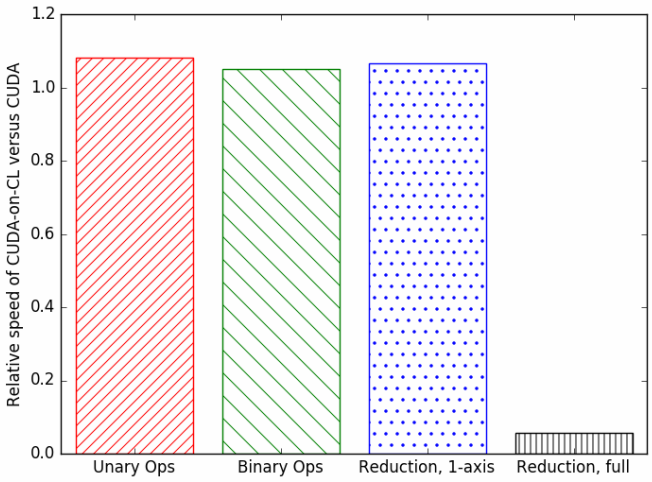

Figure 2: Relative execution speed, using Tensorflow

3 EXECUTION SPEED

A key question is, if we are generating OpenCL 1.2 code, which is then being recompiled again, to what extent does this affect performance? To provide a partial answer to this question, Tensorflow was compiled for OpenCL using CUDA-on-CL and various operations were executed. The operations are of three categories: unary operations, binary operations, and reductions. The reductions are further broken down into single-axis reductions, and full, two-axis, reductions. Figure 2 shows the ratio of the CUDA-on-CL execution speed to the CUDA implementation execution speed, for operations taken from these categories. The code for these tests is available at https://github.com/hughperkins/tensorflow-cl/tree/tensorflow-cl/tensorflow/stream_executor/cl/test . These tests are using Tensorflow 0.9, running on an NVIDIA K80 GPU.

We can see that for all of unary operations, binary operations, and single-axis reduction, the performance of CUDA-on-CL and CUDA is comparable. However, for the full, two-axis, reduction performance of CUDA-on-CL is about 14 times slower than that of CUDA. The performance equivalence for unary operations, binary operations, and single-axis reduction is comforting. The performance of full reduction is hypothesized to be related to some combination of weaker optimizations, and missing low-level hardware operations. The NVIDIA `nvcc` compiler is highly optimized, and contains several proprietary optimization routines which are not publically available. As far as low-level hardware operations, we note that the underlying Eigen implementation of reduction uses the `shfl` operation, which is available via CUDA, but must be shimmed in the OpenCL version. The shim for OpenCL 1.2 `shfl` requires writing to and from shared memory, whereas the NVIDIA hardware `shfl` implementation can directly rotate the values, at the warp level.

An additional nuance is that these timings are for a single large buffer. We can vary the buffer size, and examine the effect on performance, and this is shown in Figure 3. Figure 3a shows the effect of batch size for the unary op tanh. We can see that for batch sizes

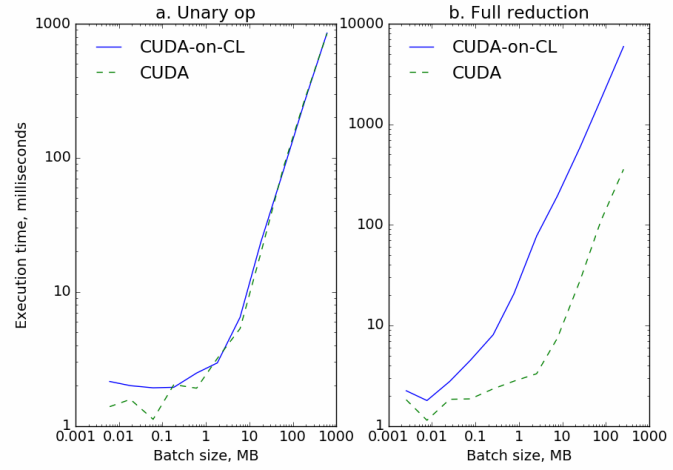

Figure 3: Effect of batch size

of around 1MB or more, the execution time of CUDA and CUDA-on-CL implementations of Tensorflow are comparable. However, for smaller batch sizes, we can see that the asymptotic zero batch size limit for the CUDA-on-CL implementation is around twice as high as that of CUDA: around 2 milliseconds, compared to around 1 millisecond, on this hardware. We hypothesize that this is a result of the overhead of copying the by-value structs, containing global pointers, at the start of each kernel.

Looking at the second chart, Figure 3b, this shows the effect of batch size for full reduction. As noted earlier, full reduction is around 14 times slower on CUDA-on-CL implementation than on CUDA implementation, possibly because of the requirement to use a shim to implement the low-level shfl operation. Similarly to the unary op case, the asymptotic zero-sized buffer sized overhead is about twice for CUDA-on-CL compared to CUDA. In addition, for larger batch sizes, the CUDA-on-CL version is consistently around 14 times slower than CUDA, independent of batch size. This suggests that this is caused by a fairly low-level operation, applied to each data item, or to each data row. This is consistent with the hypothesis that the disparity is the result of needing to use a shim for the `shfl` low-level hardware operation.

4 CONCLUSION

NVIDIA CUDA cards are amongst the fastest, and migrating machine learning library core codebases away from NVIDIA CUDA, to OpenCL, might not be achievable in the short-term. Khronos SYCL standard improves the end-user developer experience for non-CUDA cards. However, a pragmatic near-term solution might be to leave the existing source-code in NVIDIA CUDA, and provide compilers and runtimes, to run this code directly on non-NVIDIA devices.

AMD's HIP enables NVIDIA CUDA code to be compiled and run on AMD cards, in alignment with this approach. CUDA-on-CL extends this work, by enabling NVIDIA CUDA applications to run on any OpenCL 1.2 compatible GPU.

Looking at runtime execution speed, using Tensorflow: for unary operations, binary operations, and single-axis reduction, the performance of CUDA-on-CL is indistinguishable from that of CUDA, when the batch-size is 1MB or more. Tensorflow full reduction uses the low-level hardware operations `shfl`, and runs 14 times more slowly than native CUDA, on an NVIDIA K80 GPU. The overhead of additional kernel boilerplate in the CUDA-on-CL kernels increases the kernel launch time overhead from 1 millisecond, for native CUDA, to 2 milliseconds, on this hardware.

We believe that CUDA-on-CL is a significant step forward for being able to run arbitrary machine learning libraries on non-CUDA devices, and specifically on any OpenCL 1.2 compliant GPU.

REFERENCES

[1] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, and others. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] AMD. 2016. HIP : Convert CUDA to Portable C++ Code. https://github.com/GPUOpen-ProfessionalCompute-Tools/HIP. (2016). Accessed: 2017-03-31.

[3] Codeplay. 2016. ComputeCpp: Accelerate Complex C++ Applications on Heterogeneous Compute Systems using Open Standards. https://www.codeplay.com/products/computesuite/computecpp. (2016). Accessed: 2017-03-31.

[4] R. Collobert, K. Kavukcuoglu, and C. Farabet. 2011. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*.

[5] Robert Engel. 2015. Caffe Cl Engel. https://github.com/BVLC/caffe/pull/2195. (2015). Accessed: 2016-05-14.

[6] Keryell et al. 2014. triSYCL: an open source implementation of OpenCL SYCL from Khronos Group. (2014). Accessed: 2017-03-31.

[7] Google. 2016. https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html. (2016). Accessed: 2017-03-31.

[8] Khronos Group. 2014. Khronos SPIR™ Registry. https://www.khronos.org/registry/SPIR/. (2014). Accessed: 2017-03-31.

[9] Khronos Group. 2016. Khronos SPIR-V Registry. https://www.khronos.org/registry/spir-v/. (2016). Accessed: 2017-03-31.

[10] Junli Gu, Yibing Liu, Yuan Gao, and Maohua Zhu. 2016. OpenCL caffe: Accelerating and enabling a cross platform machine learning framework. In *Proceedings of the 4th International Workshop on OpenCL*. ACM, 8.

[11] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*. ACM, 675–678.

[12] David Kirk and others. 2007. NVIDIA CUDA software and GPU parallel computing architecture. In *ISMM*, Vol. 7. 103–104.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[14] Cedric Nugteren. 2015. CLBlast. https://github.com/CNugteren/CLBlast. (2015). Accessed: 2016-05-31.

[15] Hugh Perkins. 2016. cltorch: a Hardware-Agnostic Backend for the Torch Deep Neural Network Library, Based on OpenCL. *arXiv preprint arXiv:1606.04884* (2016).

[16] Andrew Richards. 2015. Update on the SYCL for OpenCL Open Standard to Enable C++ Meta Programming on Top of OpenCL. In *Proceedings of the 3rd International Workshop on OpenCL (IWOCL '15)*. ACM, New York, NY, USA, Article 9, 1 pages. DOI:https://doi.org/10.1145/2791321.2791330

[17] John E Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12, 1-3 (2010), 66–73.

[18] Fabian Tschopp. 2015. Efficient Convolutional Neural Networks for Pixelwise Classification on Heterogeneous Hardware Systems. *arXiv preprint arXiv:1509.03371* (2015).