Porting SLAMBench KFusion to Khronos SYCL

Keir, Paul

Published: 12/03/2016

Document Version
Peer reviewed version

Porting SLAMBench KFusion to Khronos SYCL

Paul Keir
University of the West of Scotland
paul.keir@uws.ac.uk

Abstract

The C++ SYCL for OpenCL standard was ratified last year by the Khronos Group, with early commercial and open-source implementations available already. While SYCL provides a programming model comparable to OpenCL, the use of an OpenMP-style single-source idiom, with integrated support for C++11 templates and language features, also introduces significant differences. To investigate the robustness of SYCL and its implementations, we report on the completed effort to port the SLAMBench KFusion computer vision benchmark to SYCL.

Categories and Subject Descriptors I.4.0 [*Image Processing and Computer Vision*]: General; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.1.5 [*Programming Techniques*]: Object-oriented Programming

Keywords GPGPU, OpenCL, C++, Parallelism, Computer Vision

1. Introduction

The Khronos SYCL 1.2 specification [3] defines a C++11/14 API for parallel programs targeting execution upon a wide range of heterogeneous and hierarchical processing hardware of the kind typically targeted by OpenCL. Distinct from the pending OpenCL C++ kernel language, the foremost characteristic of SYCL is the complete integration of host and device code; wherein a kernel specified by a function object can as readily be passed to a SYCL `parallel_for` function template, as called directly. So too, an arbitrary function may equally well be called within the call-graph of a kernel targeting the OpenCL device; as within common or garden host code. Another interesting aspect of SYCL concerns the transparent exchange of C++ type information between host and device; permitting novel possibilities for GPGPU template metaprograms. With no extensions to the C++ language, it is also reassuring to find that a serial execution is always at hand for debugging; to borrow the language of OpenMP, a SYCL program is *single-source*.

SYCL is an convenient solution for introducing OpenCL-style parallelism to a serial or homogeneously parallel C++ project. Also when porting legacy OpenCL codes to SYCL, the calls to OpenCL C built-in functions can be transposed directly to those of SYCL; with many types sharing the same names.

ASR-MOV 2016, March 12–16, 2016, Barcelona, Catalonia, Spain.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

```
struct vec_add {
  template <typename I, typename T>
  static void k(I ix,
                const T *a, const T *b, T *c,
                const size_t extent) {
    if (ix[0] < extent)
      c[id] = a[id] + b[id];
  }
};
```

Figure 1. A SYCL vector addition kernel for DAGR

The DAGR header library [1] developed as part of this work, supports the *cut and paste* transplant of kernels written in OpenCL C, into SYCL C++. Consequently, DAGR supports comparable function signatures. Figure 1 demonstrates the DAGR equivalent of the common vector addition kernel; note the pointer syntax of parameters 2, 3 and 4. Conventionally, SYCL `accessor` objects provide the crucial working abstraction over host memory; yet while some pointer interface operators such as `operator*` and `operator[]` *are* supported, others, such as `operator++` and `operator--` are not. A SYCL `accessor` satisfies the *dereferenceable* concept, but not the *iterator* concept. This is true also of SYCL's explicit pointer classes; such as `global_ptr`. It is no surprise then to find that a SYCL `accessor` object cannot be passed as an argument to a polymorphic pointer parameter such as `T*`; and of course the `std::is_pointer` trait also evaluates to `false`.

```
template <typename T>
void vec_add_h(const T *a, const T *b, T *c,
               const size_t sz)
{
  queue q;
  const range<1> r(sz);
  const buffer<T,1> buf_a(a, r), buf_b(b, r);
        buffer<T,1> buf_c(c, r);

  dagr::run<vec_add>(q,r,buf_a,buf_b,buf_c,sz);
}
```

Figure 2. SYCL vector addition host code for DAGR

Another common aspect of OpenCL kernel function signatures which DAGR mirrors in SYCL is in the correspondence of kernel parameters passed *by value*, with those host variables which do not originate from SYCL `buffer` objects. This is illustrated by the `sz` argument to the `dagr::run` method in the host code of Figure 2, which corresponds to the `size_t` value parameter of the kernel in Figure 1.

[1] Like "SYCL", "DAGR" misspells a sharp metal instrument in four letters.

The DAGR API is subsequently applied to port version 1.1 of the main KFusion algorithm [7, 8] of the SLAMBench computer vision benchmark suite [1] to SYCL. SLAMBench provides implementations of KFusion in a range of languages, including an OpenCL version, which naturally provides the starting point here. Nevertheless, the final result of the port is more akin to the C++ version, upon which the OpenCL version was originally based. Development was undertaken using a *trial* version of Codeplay Software's *ComputeCpp* [6]. Accordingly, performance evaluation of the SYCL port of SLAMBench KFusion is deferred.

2. SYCL SLAMBench

The SYCL version of the SLAMBench [1] KFusion algorithm was developed and tested on 64-bit Ubuntu 15.04, with GCC 4.9.2, and the 15.06 and 15.10 evaluation versions of Codeplay Software's *ComputeCpp*. The OpenCL driver was version 5.0.0.43 from the 64-bit Intel Code Builder for OpenCL. A significant first task was the development of a GCC-like compiler driver, `syclcc`, which automatically invokes both the device compiler from Codeplay's *ComputeCpp*, *and* the native C++ host compiler. So equipped, and following minor modifications to the SLAMBench CMake configuration, by setting `CXX` to `syclcc`, the GUI and benchmark versions of KFusion will build; upon the same simple `cmake` and `make` command invocations as for each supported language.

Version 1.1 of SLAMBench includes 14 significant kernels. Of these, 12 are implemented on GPU using both CUDA and OpenCL. The two remaining are *acquire*, the IO-heavy acquisition of another RGB-D frame; and *solve*, a singular value decomposition, too small to be offloaded. The SYCL version contains the same 12 kernels.

The focus for the original SLAMBench project is portability; with a nevertheless competitive performance profile. From this perspective a goal in the development of the SYCL version was in replicating the high-level structure of the KFusion algorithm implementation; shared by each of the language implementations included with SLAMBench. Due to the Khronos specified compatibility between OpenCL and SYCL, the OpenCL implementation was the foundation in the development of the SYCL equivalent. This meant that OpenCL `cl_mem` variables declared at global scope, became SYCL `buffer` pointers; also declared at global scope. Calls to `clCreateBuffer` became calls to the C++ `new` operator, with some care required as the relevant read/write access permission is not requested by the SYCL `buffer` constructor; using a subsequent SYCL `accessor` instead. Hence, access permission requests were transferred to the site of each kernel enqueue. The declaration; creation; and release of kernels are implicit in SYCL, and were thus elided in this version. Calls to `clSetKernelArg` and `clEnqueueNDRangeKernel` are rendered especially concise with a single call to `dagr::run`. Ultimately, the aim in porting the host component in this fashion is readability, so allowing a fruitful comparison of the SYCL and existing versions; and parity of performance by replicating the same algorithm.

Figure 3's invocation of the *bilateralFilter* kernel in SYCL, uses the DAGR API, with the `ro` wrapper function utilised twice to request `access::mode::read` access. The single call to `dagr::run` shown occurs in the `Kfusion::preprocessing` method, and is directly comparable to the single call to `bilateralFilterKernel` in the C++ version; albeit with additional SYCL `queue` and `range` arguments. In the more verbose OpenCL version, `clSetKernelArg` is called for each of five arguments, before the call to `clEnqueueNDRangeKernel`.

3. Related Work

Regarding purely C++ interfaces to GPU acceleration, a published ISO standard in consideration for inclusion in the next iteration of the C++ standard [4] provides an API for the parallel execution of a range of common STL algorithms. The API relies on the addition of tag-like *execution policy* objects as first arguments of relevant algorithm functions. Sequential, parallel and vectorised (including GPU) implementations are specified. A SYCL implementation [5] also exists; with 8 STL algorithms implemented so far.

```
dagr::run<bilateralFilterKernel>(q, r,
  *ocl_ScaledDepth[0], ro(*ocl_FloatDepth),
  ro(*ocl_gaussian), e_delta, radius);
```

Figure 3. DAGR enqueue of the bilateral filter kernel

Another example of a DSL using SYCL is described in Potter [2]. That research contrasts with the current work in its use of a *deep* embedding, to allow the composition of kernels. ViennaCL [9] provides a C++ API targeting a range of backends including OpenMP, OpenCL and CUDA. The main focus for ViennaCL is the provision of common linear algebra operations, with OpenCL strings facilitating custom kernels via runtime compilation.

4. Conclusion

The SYCL DAGR API has been introduced, along with its deployment to the task of porting the SLAMBench computer vision benchmark suite [1] to SYCL, for compatibility with Codeplay Software's implementation: *ComputeCpp*. Notable features of DAGR include a concise end-user API; support for both value and pointer kernel parameters, as in OpenCL C; and the provision of lightweight wrapper functions to convey information regarding access permissions and shared memory quotas to the DAGR implementation. The API is designed as a header-only library, and utilises the Khronos SYCL standard to provide a concise interface for developers less interested in C++ lambda functions per se, than in maintaining parity between an OpenCL and a SYCL backend. While it is understood that the DAGR API offers only a portion of the configuration space of SYCL, it is hoped that this is nevertheless a focused and useful interface.

References

[1] M. Z. Zia, L. Nardi, A. Jack, E. Vespa, B. Bodin, P. H. J. Kelly and A. J. Davisonr, *Comparative Design Space Exploration of Dense and Semi-Dense SLAM*, IEEE Intl. Conf. on Robotics and Automation (ICRA 2016), Stockholm, Sweden, May 2016.

[2] R. Potter, P. Keir, R. J. Bradford and A. Murray. *Kernel Composition in SYCL*. In ACM Proceedings of the International Workshop on OpenCL, 2015.

[3] L. Howes and M. Rovatsou (Eds.) *The Khronos SYCL Specification version 1.2*, 2015.

[4] J. Hoberock, editor. *Technical Specification for C++ Extensions for Parallelism*, ISO 2015

[5] R. Reyes. *SyclParallelSTL github*. `https://github.com/KhronosGroup/SyclParallelSTL`, 2015.

[6] *Codeplay - ComputeCpp*. `https://www.codeplay.com/products/computecpp`, 2015.

[7] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. Fitzgibbon. *KinectFusion: Real-time dense surface mapping and tracking*. In ISMAR, 2011.

[8] G. Reitmayr and H. Seichter. *KFusion github*. `https://github.com/GerhardR/kfusion`, 2011.

[9] K. Rupp, F. Rudolf and J. Weinbub. *ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs*, Proceeedings of the International Workshop on GPUs and Scientific Applications, 2010.