Early experiments using SYCL single-source modern C++ on Xilinx FPGA

Extended Abstract of Technical Presentation

Ronan Keryell
Lin-Ya Yu
rkeryell@xilinx.com
yu810226@gmail.com
Xilinx Research Labs
Dublin, Ireland

Abstract

Heterogeneous computing is required in systems ranging from low-end embedded systems up to the high-end HPC systems to reach high-performance while keeping power consumption low. Having more and more CPU and accelerators such as FPGA creates challenges for the programmer, requiring even more expertise of them. Fortunately, new modern C++-based domain-specific languages, such as the SYCL open standard from Khronos Group, simplify the programming at the full system level while keeping high performance.

SYCL is a single-source programming model providing a task graph of heterogeneous kernels that can be run on various accelerators or even just the CPU. The memory heterogeneity is abstracted through buffer objects and the memory usage is abstracted with accessor objects. From these accessors, the task graph is implicitly constructed, the synchronizations and the data movements across the various physical memories are done automatically, by opposition to OpenCL or CUDA.

triSYCL is an on-going open-source project used to experiment with the SYCL standard, based on C++17, OpenCL, OpenMP and Clang/LLVM. We have extended this framework to target Xilinx SDx tool to compile some SYCL programs to run on a CPU host connected to some FPGA PCIe cards, by using OpenCL and SPIR standards from Khronos.

While SYCL provides functional portability, we made a few FPGA-friendly extensions to express some optimization to the SDx back-end in a pure C++ way.

We present some interesting preliminary results with simple benchmarks showing how to express pipeline, dataflow and array-partitioning and we compare with the implementation written using other languages available for Xilinx FPGA: HLS C++ and OpenCL C.

CCS Concepts • **Computing methodologies** → **Parallel programming languages**; *Massively parallel algorithms*; • **Software and its engineering** → **Parallel programming languages**; **Domain specific languages**; *Object oriented frameworks*; • **Computer systems organization** → *Heterogeneous (hybrid) systems*;

Keywords SYCL, C++17, DSeL, FPGA, reconfigurable computing, OpenCL, SPIR, Clang, LLVM, triSYCL

ACM Reference Format:
Ronan Keryell and Lin-Ya Yu. 2018. Early experiments using SYCL single-source modern C++ on Xilinx FPGA: Extended Abstract of Technical Presentation. In *Proceedings of IWOCL '18.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Unpublished working draft. Not for distribution.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
IWOCL '18, May 14-16, 2018, Oxford, UK
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Computing architectures nowadays are huge hybrid multiprocessor system-on-chips with different kinds of processors, GPU, configurable specific accelerators (video CODEC...), reconfigurable programmable logic (FPGA), various hierarchies of memory and memory interfaces, configurable IO and network support, security support, and power control etc. High-performance applications may use a hierarchy of such systems scaling up towards utilizing a full-scale data-center.

So the programmer is facing a fractal architecture, demanding also more and more control for power efficiency. This tends to require a dense fractal set of skills and tools.

SYCL [8, 9] is a new open standard from Khronos Group aiming at solving some of the programming issues related to heterogeneous computing. This pure C++14 (for SYCL 1.2.1) or C++17 (for SYCL 2.2) domain-specific embedded language

Unpublished working draft. Not for distribution.

allows the programmer to write single-source C++ host code with accelerated code expressed as functors. The data accesses are described with accessor objects that implicitly define a task graph that can be asynchronously scheduled on a distributed-memory system including several CPU and accelerators.

This programming model is quite generic but provides also an interoperability mode with the OpenCL realm, another standard from Khronos Group aimed at heterogeneous computing with a C host API and separate language for the kernels (C, C++, SPIR and SPIR-V). This allows a SYCL C++ application to recycle existing OpenCL kernels into a higher level C++ programming model, relieving the programmer from explicitly defining the memory transfers.

In this article we present in Section 2 the SYCL standard, then in Section 3 how to express pipelined and dataflow execution in SYCL and how dealing with partitioned arrays in Section 4.

2 SYCL

SYCL [8, 9] (pronounced "sickle") is a royalty-free, cross-platform abstraction C++ programming model for OpenCL [5, 6]. SYCL builds on the underlying concepts, portability and efficiency of OpenCL while adding much of the ease of use and flexibility of single-source C++.

Developers using SYCL are able to write standard C++14/C++17 code, with many of the techniques they are accustomed to, such as inheritance and templating. At the same time developers have access to the full range of capabilities of OpenCL both through the features of the SYCL libraries and, where necessary, through interoperation with code written directly to the OpenCL APIs [6].

SYCL implements a single-source multiple compiler-passes design which offers the power of source integration while allowing tool-chains to remain flexible. This design supports embedding of code intended to be compiled for an OpenCL device, for example a GPU or an FPGA, inline with host code.

SYCL is a pure single-source C++ DSeL (Domain-Specific Embedded Language) providing simpler abstractions for heterogeneous computing. In Figure 1 is presented a small application using SYCL concepts to create a graph of 3 asynchronous tasks to initialize 2 matrices and addition them before checking for the final result.

The main interesting features that SYCL brings are asynchronous task graphs, hierarchical parallelism, buffers defining location-independent storage, accessors to express usage for buffers and other objects, implicit dependency graph, automatic data motion, automatic overlapping kernels and communications, single-source, host fallback for debugging or when no accelerator is available and cross-platform (buffers used by different devices from different vendors), OpenCL interoperability mode.

SYCL retains the execution model, runtime feature set and device capabilities of the underlying OpenCL standard. This is why SYCL 1.2.1 [9] targets devices with OpenCL 1.2 [4] capabilities, while SYCL 2.2 [8] targets devices with OpenCL 2.2 [6] capabilities, adding for example the pipes and the shared virtual memory between the host and devices.

The OpenCL C specification imposes some limitations on the full range of C++ features that SYCL is able to support. This ensures portability of device code across as wide a range of devices as possible.

As a result, while the code can be written in standard C++ syntax with interoperability with standard C++ programs, the entire set of C++ features is not available in SYCL device code. In particular, SYCL device code, as defined by this specification, does not support virtual function calls, function pointers in general, exceptions, runtime type information or the full set of C++ libraries that may depend on these features or on features of a particular host compiler.

Actually these features are anyway often used in C++ high-performance code in the hot-path because of performance issues. Fortunately, the use of C++ features such as templates and inheritance on top of the OpenCL execution model opens a wide scope for innovation in software design for heterogeneous systems, giving workarounds for some of the unsupported features.

Clean integration of device and host code within a single C++ type system enables the development of modern, templated libraries that build simple, yet efficient, interfaces to offer more developers access to OpenCL capabilities and devices. SYCL is intended to serve as a foundation for innovation in programming models for heterogeneous systems, that builds on an open and widely implemented standard foundation in the form of OpenCL.

This is why the OpenCL version of TensorFlow [13], the C++ machine learning framework from Google, is actually using SYCL instead of plain OpenCL.

SYCL is one of the candidates giving inputs on parallelism and heterogeneous computing to the C++ ISO/IEC JTC1/SC22/WG21 standardization committee [7, 14–16].

3 Fine grain parallel execution

High-level programs on FPGA are typically compiled with each operation translated in some hardware implementation. Compared to the compilation for architectures with fixed hardware functions such as CPU, DSP or GPU, that means that the hardware implementations of the operations do coexist in space inside the FPGA, which means that they can be used in parallel to increase the throughput of the system.

In a "normal" computer this is not a parameter to be taken in consideration, but for FPGA this is something useful to express somehow, when compiling a C/C++ program for FPGA. Each FPGA vendor have some extensions to express

Unpublished working draft. Not for distribution.

```cpp
// Demonstrate the use of an asynchronous task graph of kernels to initialize and add 2 matrices.
#include <CL/sycl.hpp>
#include <iostream>
using namespace cl::sycl;
// Size of the matrices
constexpr size_t N = 2000;
constexpr size_t M = 3000;

int main() {
  // Create a queue to work on
  queue q;
  // Create some 2D buffers of N*M floats for our matrices
  buffer<float, 2> a { { N, M } };
  buffer<float, 2> b { { N, M } };
  buffer<float, 2> c { { N, M } };
  // Launch a first asynchronous kernel to initialize a
  q.submit([&] (handler &cgh) {
      // The kernel writes a, so get a write accessor on it
      auto A = a.get_access<access::mode::write>(cgh);

      // Enqueue a parallel kernel iterating on a N*M 2D iteration space
      cgh.parallel_for<class init_a>({ N, M },
                                     [=] (id<2> index) {
                                       A[index] = index[0]*2 + index[1];
                                     });
    });
  // Launch an asynchronous kernel to initialize b
  q.submit([&] (handler &cgh) {
      // The kernel writes b, so get a write accessor on it
      auto B = b.get_access<access::mode::write>(cgh);
      /* From the access pattern above, the SYCL runtime detects this command group is independent from the first one
          and can be scheduled independently.
         Enqueue a parallel kernel iterating on a N*M 2D iteration space */
      cgh.parallel_for<class init_b>({ N, M },
                                     [=] (id<2> index) {
                                       B[index] = index[0]*2014 + index[1]*42;
                                     });
    });
  // Launch an asynchronous kernel to compute matrix addition c = a + b
    q.submit([&] (handler &cgh) {
        // In the kernel a and b are read, but c is written
        auto A = a.get_access<access::mode::read>(cgh);
        auto B = b.get_access<access::mode::read>(cgh);
        auto C = c.get_access<access::mode::write>(cgh);
        // From these accessors, the SYCL runtime will ensure that when this kernel is run, the kernels computing
        // a and b completed. Then Enqueue a parallel kernel iterating on a N*M 2D iteration space
        cgh.parallel_for<class matrix_add>({ N, M },
                                           [=] (id<2> index) {
                                             C[index] = A[index] + B[index];
                                           });
      });
    /* Request an accessor to read c from the host-side. The SYCL runtime ensures that c is ready when the accessor is
        returned */
    auto C = c.get_access<access::mode::read>();
    std::cout << std::endl << "Result:" << std::endl;
    for (size_t i = 0; i < N; i++)
      for (size_t j = 0; j < M; j++)
        // Compare the result to the analytic value
        if (C[i][j] != i*(2 + 2014) + j*(1 + 42)) {
          std::cout << "Wrong value " << C[i][j] << " on element "
                    << i << ' ' << j << std::endl;
          exit(-1);
        }
  std::cout << "Accurate computation!" << std::endl;
  return 0;
}
```

Figure 1. Example of a SYCL C++ program producing and adding 2 matrices, coming from `https://github.com/triSYCL/triSYCL/blob/master/tests/examples/demo_parallel_matrix_add.cpp`.

Unpublished working draft. Not for distribution.

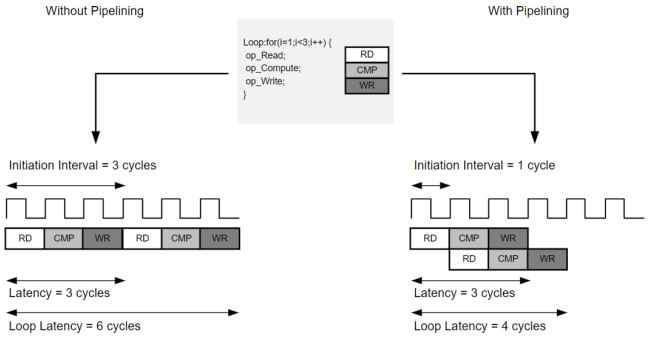

Figure 2. Execution of operations of a loop in normal and pipelined way.

```
template<typename T, typename U>
void compute(T *buffer_in, U *buffer_out) {
  for(int i = 0; i < NUM_ROWS; ++i) {
    for (int j = 0; j < ELE_PER_ROW; ++j) {
      xilinx::pipeline([&] {
        int inTmp = buffer_in[ELE_PER_ROW*i+j];
        int outTmp = inTmp * ALPHA;
        buffer_out[ELE_PER_ROW*i+j] = outTmp;
      });
    }
  }
}
```

Figure 3. Example of a typical loop with pipelined execution. What is added is shown with a blue background.

these optimizations, such as [3, 17, 18, 20, 21], based on #pragma, attributes or even configuration files or compiler options.

The issues with extensions such as #pragma or attributes is that they require a specific compiler to parse and understand them. This prevents some emulation on CPU for example, with some compile-time and run-time checking.

In our SYCL implementation, we prefer to have some pure C++ function-based and class-based extensions, compatible with CPU emulation and more aligned with SYCL DSeL philosophy.

3.1 Pipelined execution

In a loop, all the operations of an iteration will be executed sequentially, as shown on Figure 2. The operations are used in sequence and if there is no intra-loop dependency in the program preventing this, the operations from different iterations can actually be executed in parallel, to increase the throughput using the pipelined execution shown also on Figure 2, by reducing the *initiation interval*, to use the FPGA jargon.

As shown on Figure 3, we chose to mark a zone to be pipelined with a kind of decorator *à la* Python [10] `xilinx::pipeline (...)` taking a functor, typically expressed in modern C++ as a lambda expression.

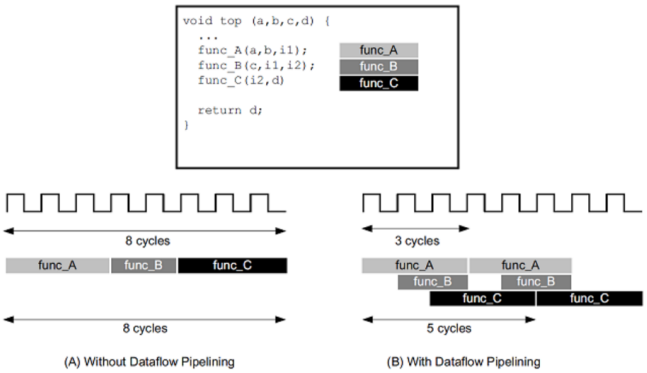

Figure 4. Execution of functions without or with dataflow mode.

3.2 Dataflow execution

While pipeline execution is very fine-grain parallelism, a coarser-grain parallelism can be used when having different functions with data flowing from one function to another: dataflow execution.

Since on the FPGA the functions are instantiated in hardware, they can execute in parallel and if data dependencies allow it, data can move from one function to another either using dual-buffering ping-pong buffers (preventing the time-consuming and power-hungry external memory) or even a FIFO (preventing the use of more complex power-hungry memory access), as shown on Figure 4.

To mark an area in the program to apply such an optimization, we use the same trick as for pipelining (§ 3.1) with a `xilinx::dataflow (...)` as shown on Figure 5. Currently we have a limitation in our device compiler and runtime that requires changing the lambda captures with `drt::` accessor like adding in the example some lines such as

```
d_a = drt::accessor<decltype(a_a)> { a_a },
```

but this limitation should be removed soon.

3.3 Experiments with pipelined and dataflow execution

We have run various experiments on a benchmark transferring 2D arrays with and without these optimizations in SYCL (code from Figure 5), but also with non-single source HLS C++ or OpenCL C kernels, coming from [19]. The SYCL examples have been translated for this study from the HLS C++ and OpenCL examples, leading to quite shorter examples, typically less than 100 lines compared to the original 250+ line examples in HLS C++ or OpenCL C.

When possible, we compared 2 different versions of the tools (Xilinx SDx 2017.2 and 2017.4) that can also be configured to use either a Clang/LLVM 3.1 or 3.9 front-end. The performance results are shown on Figure 6 for a default kernel execution clock of 200 MHz with an ADM-PCIE-7V3

Unpublished working draft. Not for distribution.

```
template<typename T, typename U>
void readInput(T *buffer_in, const U &d_b) {
  for(int i = 0; i < NUM_ROWS; ++i)
    for (int j = 0; j < ELE_PER_ROW; ++j)
      xilinx::pipeline([&] {
        buffer_in[ELE_PER_ROW*i+j] = d_b[ELE_PER_ROW*i+j
            ];
      });
}

template<typename T, typename U>
void compute(T *buffer_in, U *buffer_out) {
  for(int i = 0; i < NUM_ROWS; ++i)
    for (int j = 0; j < ELE_PER_ROW; ++j)
      xilinx::pipeline([&] {
        int inTmp = buffer_in[ELE_PER_ROW*i+j];
        int outTmp = inTmp * ALPHA;
        buffer_out[ELE_PER_ROW*i+j] = outTmp;
      });
}

template<typename T, typename U>
void writeOutput(T *buffer_out, const U &d_a) {
  for(int i = 0; i < NUM_ROWS; ++i)
    for (int j = 0; j < ELE_PER_ROW; ++j)
      xilinx::pipeline([&] {
        d_a[ELE_PER_ROW*i+j] = buffer_out[ELE_PER_ROW*i+j
            ];
      });
}

cgh.single_task<class add>
           ([=,
             d_a = drt::accessor<decltype(a_a)> { a_a },
             d_b = drt::accessor<decltype(a_b)> { a_b }
             ] {
                 int buffer_in[BLOCK_SIZE];
                 int buffer_out[BLOCK_SIZE];
                 xilinx::dataflow([&] {
                     readInput(buffer_in, d_b);
                     compute(buffer_in, buffer_out);
                     writeOutput(buffer_out, d_a);
                 });
             });
```

Figure 5. Example of a typical loop with functional dataflow execution with also pipelined execution of each function. What is added for dataflow execution has an orange background and for a pipelined execution has a blue background.

PCIe FPGA accelerator card [12] plugged into an Ubuntu 17.10 Linux system with an Intel core i7-6700 CPU and 64 GB of RAM. The triSYCL device compiler is using Clang/LLVM 3.9 and we used the Xilinx SDx OpenCL runtime 2017.2.

The performance of triSYCL are quite reasonable compared to some other programming models and tools, and sometimes even gives the best result. This shows that higher-level programming model does not mean always trading programmability with execution efficiency, for example because the single-source approach provides to the compiler a global interprocedural view between host code and device code, allowing for example some variable that appears to be a constant on the host-side to be directly evaluated on the device code without requiring some explicit host-device PCIe data transfer as for OpenCL or HLS C++.

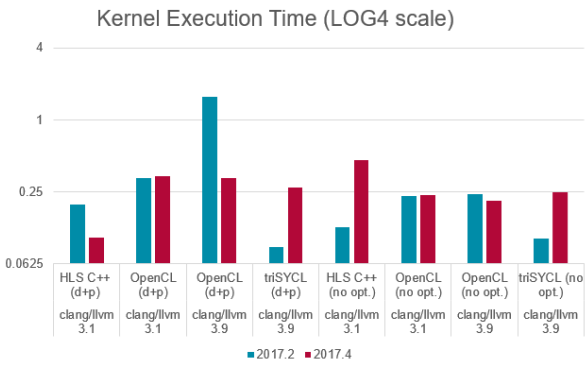

Figure 6. Comparison of execution times of 2D array transfers with various tools and languages with a Xilinx FPGA acceleration card.

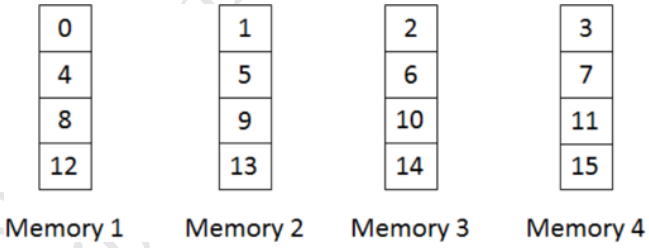

Figure 7. Array of 16 elements distributed in a cyclic way on 4 memory banks. The numbers represent the indices of the array elements stored on the memory banks.

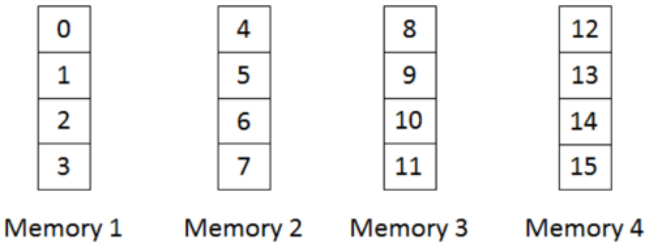

Figure 8. Array of 16 elements distributed by blocks on 4 memory banks. The numbers represent the indices of the array elements stored on the memory banks.

4 Array partitioning

On FPGA, even the memories are configurable and can be implemented with different levels of parallelism, providing some trade-off between hardware complexity (cost), bandwidth and latency.

For example, an array in a program on an FPGA can be distributed on different memory banks in a cyclic way (Figure 7), partitioned by block (Figure 8) or fully partitioned (Figure 9) for a maximum bandwidth but at the cost of a very complex routing.

Unpublished working draft. Not for distribution.

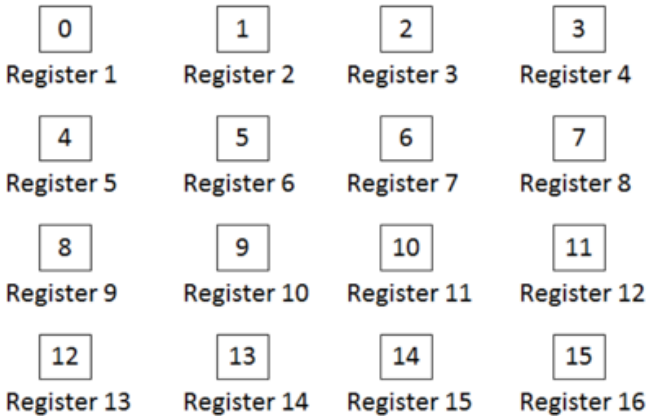

Figure 9. Complete partition of an array of 16 elements, with 1 element by register.

4.1 Extending SYCL to express array partitioning

Since in C++ there is a std :: `array`, we introduced a new cl :: sycl :: `xilinx::partition_array` similar to std :: `array` but taking a new templated class parameter to model the partition.

For now we support only 1-dimension arrays similar to std :: `array` because we are waiting for the ISO C++ standardization committee to define multidimensional arrays [2] so we can align on the same syntax.

The example on Figure 10 shows how partitioned arrays are used to speed-up the computation, in combination with pipelining, by reducing the bank conflicts even if the pipelining increase the bandwidth of the computation.

In a similar way, a complete partitioning could be used as shown on Figure 11.

4.2 Experiments using partitioned arrays

We made some experiments in the same way as in § 3.3 with the SYCL code shown on Figure 10 and also with non-single source HLS C++ or OpenCL C kernels, coming from [19].

The performance results are shown on Figure 12 in the same context as § 3.3.

The performance of triSYCL are here also quite reasonable compared to some other programming models and tools, while never reaching the best performance but for the unoptimized case.

5 Conclusion

C++ provides direct mapping to hardware and zero-overhead abstraction, as summarized by Bjarne Stroustrup in 2 lines. With modern C++, there are less reasons to choose between "slow and convenient" and "fast and hard-to-use" programming models and we are seeing more an evolution from "fast and hard-to-use" towards a "fast and easy-to-use" paradigm.

At the same time, heterogeneous computing in embedded and high-performance computing is here to stay because of physical constraints. This puts the pressure on the programmers to integrate a full system across the various accelerators. The SYCL standard C++ DSeL allows a single-source approach for both host and accelerators parts in type-safe way to simplify the process while interoperable with the ubiquitous C/C++ world. The SYCL runtime provides an implicit task graph managing asynchronicity and data transfers across the various memory spaces.

SYCL is one of the candidates inside the ISO C++ committee to push more heterogeneous computing as a first-class citizen into the language and is an experimental sand-box to show-case new non-functional concepts such as the pipelining/dataflow execution models or the way some arrays are implemented. Our experiments with triSYCL show that it is competitive with other more conventional programming tools and languages.

With even more modern meta-programming features comping into C++ such as introspection and generative programming [1, 11], this brings directly at the C++ level some system-wide design capabilities with some software-hardware trade-offs, without requiring some external generative tools

Unpublished working draft. Not for distribution.

```
cgh.single_task<class add>([=,
                            d_out = drt::accessor<decltype(a_out)> { a_out },
                            d_in1 = drt::accessor<decltype(a_in1)> { a_in1 },
                            d_in2 = drt::accessor<decltype(a_in2)> { a_in2 }
] {
      // Cyclic Partition for A as matrix multiplication needs row-wise
      // parallel access
      xilinx::partition_array<Type, BLOCK_SIZE,
                              xilinx::partition::cyclic<MAX_DIM>> A;
      // Block Partition for B as matrix multiplication needs column-wise
      // parallel access
      xilinx::partition_array<Type, BLOCK_SIZE,
                              xilinx::partition::block<MAX_DIM>> B;
      xilinx::partition_array<Type, BLOCK_SIZE> C;

      // As A and B Matrix are partitioned with the factor of MAX_DIM, so
      // to get parallel row/column access, input square matrix[DIMXDIM]
      // should be written into local Array in MATRIX[MAX_DIM * MAX_DIM]
      // format

      // Burst read for matrix A
      for (int itr = 0, i = 0, j = 0; itr < DIM * DIM; itr++, j++) {
        xilinx::pipeline([&] {
          if (j == DIM) { j = 0; i++; }
          A[i*MAX_DIM + j] = d_in1[itr];
        });
      }

      // Burst read for matrix B
      for (int itr = 0, i = 0, j = 0; itr < DIM * DIM; itr++, j++) {
        xilinx::pipeline([&] {
          if (j == DIM) { j = 0; i++; }
          B[i * MAX_DIM + j] = d_in2[itr];
        });
      }

      for (int i = 0; i < DIM; i++) {
        // As A and B are partition correctly so loop pipelining is
        // applied at 2nd level loop and which will eventually unroll
        // the lower loop
        for (int j = 0; j < DIM ; j++) {
          xilinx::pipeline([&] {
            int result = 0;
            for (int k = 0; k < MAX_DIM; k++) {
              result += A[i * MAX_DIM +  k] * B[k * MAX_DIM + j];
            }
            C[i*MAX_DIM + j] = result;
          });
        }
      }

      // Burst write from output matrices to global memory
      // Burst write from matrix C
      for (int itr = 0, i = 0, j = 0; itr < DIM * DIM; itr++, j++) {
        xilinx::pipeline([&] {
          if (j == DIM) { j = 0; i++; }
          d_out[itr] = C[i * MAX_DIM + j];
        });
      }

  });
```

Unpublished working draft. Not for distribution.

Figure 10. Example of a matrix multiplication using partitioned arrays (orange background) and pipelining (blue background).

References

[1] Matúš Chochlík, Axel Naumann, and David Sankel. 2018. Static reflection. (Feb. 2018). Issue P0194R5. http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p0707r3.pdf

[2] H. Carter Edwards, Daniel Sunderland, David Hollman, Christian Trott, Mauro Bianco, Ben Sander, Athanasios Iliopoulos, John Michopoulos, and Daniel Sunderland. 2018. Polymorphic Multidimensional Array Reference. (Feb. 2018). Issue P0009R5. https://wg21.link/p0009r5

[3] Jeff Fifield, Ronan Keryell, Hervé Ratigner, Henry Styles, and Jim Wu. 2016. Optimizing OpenCL Applications on Xilinx FPGA. In *Proceedings of the 4th International Workshop on OpenCL (IWOCL '16)*. ACM, New York, NY, USA, Article 5, 2 pages.

```
xilinx::partition_array<float, 16,
                        xilinx::partition::complete<>> h;
```

Figure 11. Complete partition of an array of 16 elements, with 1 element by register.

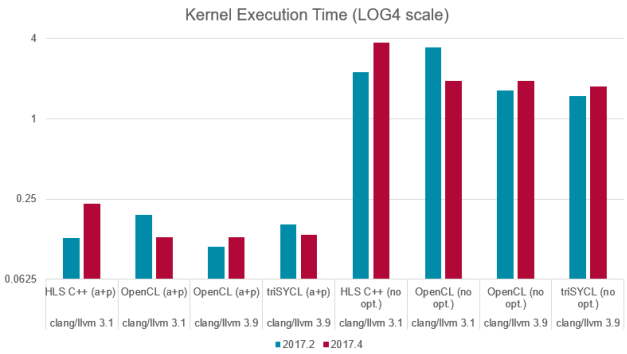

Figure 12. Comparison of execution times of matrix multiplication various tools and languages with a Xilinx FPGA acceleration card.

https://doi.org/10.1145/2909437.2909447 http://ronan.keryell.fr/Talks/2016/2016-04-21-IWOCL-optimizing_OpenCL_Xilinx_FPGA/2016-04-21-IWOCL-optimizing_OpenCL_Xilinx_FPGA.pdf http://ronan.keryell.fr/Talks/2016/2016-04-21-IWOCL-optimizing_OpenCL_Xilinx_FPGA/2016-04-21-IWOCL_Xilinx_OpenCL-presentation.pdf.

[4] Khronos OpenCL Working Group. 2012. The OpenCL Specification, Version: 1.2, Document Revision: 19. (Nov. 2012). https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

[5] Khronos OpenCL Working Group. 2016. The OpenCL C++ Specification, Version: 1.0, Document Revision: 22. (April 2016). https://www.khronos.org/registry/OpenCL/specs/opencl-2.2-cplusplus.pdf

[6] Khronos OpenCL Working Group. 2016. The OpenCL Specification, Version: 2.2, Document Revision: 06. (March 2016). https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf

[7] Ronan Keryell and Joël Falcou. 2016. Accessors — a C++ standard library class to qualify data accesses. (May 2016). www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0367r0.pdf

[8] SYCL subgroup Khronos OpenCL Working Group. 2016. SYCL Provisional Specification, SYCL integrates OpenCL devices with modern C++ using a single source design. (Feb. 2016). https://www.khronos.org/registry/SYCL/specs/sycl-2.2.pdf

[9] SYCL subgroup Khronos OpenCL Working Group. 2017. SYCL Specification, SYCL integrates OpenCL devices with modern C++. (Dec. 2017). https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf

[10] Kevin D. Smith, Jim J. Jewett, Skip Montanaro, and Anthony Baxter. 2003. PEP 318 — Decorators for Functions and Methods. (June 2003). https://www.python.org/dev/peps/pep-0318/

[11] Herb Sutter. 2018. Metaclasses: Generative C++. (Feb. 2018). Issue P0707R3. http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p0707r3.pdf

[12] Alpha Data Parallel Systems. 2017. ADM-PCIE-7V3. (Feb. 2017). https://www.alpha-data.com/dcp/products.php?product=adm-pcie-7v3

[13] TensorFlow 2017. TensorFlow 1.0: An open-source software library for Machine Intelligence. (Feb. 2017). https://www.tensorflow.org/

[14] Michael Wong, Andrew Richards, Maria Rovatsou, and Ruyman Reyes. 2016. Khronos's OpenCL SYCL to support Heterogeneous Devices for C++. (Feb. 2016). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0236r0.pdf

[15] Michael Wong, Andrew Richards, Maria Rovatsou, Ruyman Reyes, Lee Howes, and Gordon Brown. 2016. Towards support for Heterogeneous Devices in C++ (Concurrency aspects). (May 2016). www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0362r0.pdf

[16] Michael Wong, Andrew Richards, Maria Rovatsou, Ruyman Reyes, Lee Howes, and Gordon Brown. 2016. Towards support for Heterogeneous Devices in C++ (Language aspects). (May 2016). www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0363r0.pdf

[17] Xilinx. 2017. Vivado HLS Optimization Methodology Guide. (Dec. 2017). Issue UG1270 (v2017.4). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1270-vivado-hls-opt-methodology-guide.pdf

[18] Xilinx. 2018. SDAccel Environment Profiling and Optimization Guide. (Jan. 2018). Issue UG1207 (v2017.4). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1207-sdaccel-optimization-guide.pdf

[19] Xilinx. 2018. SDAccel Example Repository. (Feb. 2018). https://github.com/Xilinx/SDAccel_Examples

[20] Xilinx. 2018. SDSoC Environment Profiling and Optimization Guide. (Jan. 2018). Issue UG1235 (v2017.4). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1235-sdsoc-optimization-guide.pdf

[21] Xilinx. 2018. SDx Pragma Reference Guide. (Jan. 2018). Issue UG1253 (v2017.4). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1253-sdx-pragma-reference.pdf

Unpublished working draft. Not for distribution.