P1261R0: Supporting Pipelines in C++

Date: 2018-10-08

Project: ISO JTC1/SC22/WG21: Programming Language C++

Audience SG14, SG1

Authors: Michael Wong (Codeplay), Daniel Garcia (UC3M), Ronan Keryell (Xilinx)

Contributors

Emails: michael@codeplay.com
josedaniel.garcia@uc3m.es
rkeryell@xilinx.com

Reply to: michael@codeplay.com

Introduction

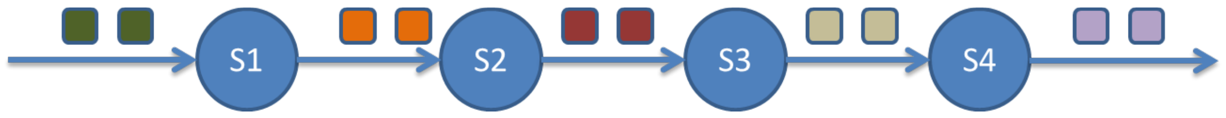
A pipeline pattern allows processing a data stream where the computation may be divided in multiple stages. Each stage processes the data item generated in the previous stage and passes the produced result to the next stage.

Motivation

In media processing, network protocol, or an online algorithm, processing starts before all the input is available, and the output starts to be written before all processing is complete. In other words, computing and I/O take place concurrently. Often the input is coming from real-time sources such as keyboards, pointing devices, and sensors. Even when all the inputs are available, it is often on a file system, and overlapping execution and the algorithm with input/output can yield significant improvements in performance. A pipeline is one way to achieve this overlap, not only for I/O but also for computations that are mostly parallel but require small sections of code that must be serial. Without pipelines, you would have to rely on the external memory architecture. In embedded systems and heterogeneous architectures, pipelining can increase the performance and decrease the power consumption.

Pipeline Matters

A pipeline is a linear sequence of stages. Data flows through the pipeline, from the first stage to the last stage. Each stage performs a transform on the data. The data is partitioned into pieces that we call items. A stage's transformation of items may be one-to-one or may be more complicated. A serial stage processes one item at a time, though different stages can run in parallel. There are several types of pipelines.



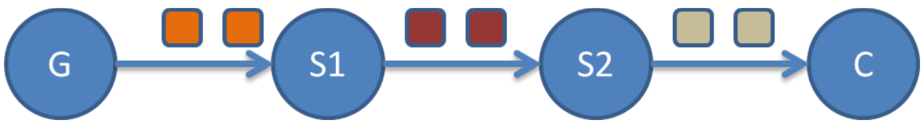
Pipelines are beneficial for a number of reasons:

- Composition is simple: the output of a pipeline feeds directly into the input of another pipeline;
- Mapping serial pipeline directly to serial I/O devices: even for random access devices (disks), serial access is still fast;
- Overlapping computation and I/O: by having separate stages for computation and I/O;

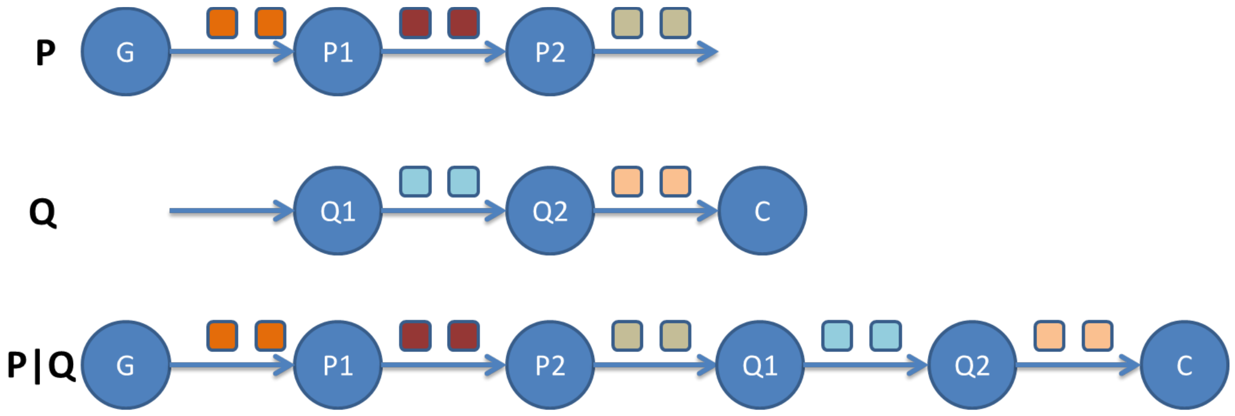
- Reduce the latency and support soft real-time and online applications: early items can flow all the way through a pipeline before later items are available. This imposes a weaker synchronization than map where all items must be available at the start, and no output data is ready until the map operation completes;
- Analysis: each stage can be analyzed and debugged separately;
- Resource limits: the number of items in flight can be controlled to match those limits by processing a large amount of data using a fixed amount of memory;
- Reasoning about deadlock freedom: mostly linear structure (even though they may still have cycles) makes it easy to reason about deadlocks, but does not mean it will be deadlock-free .

Pipeline variations

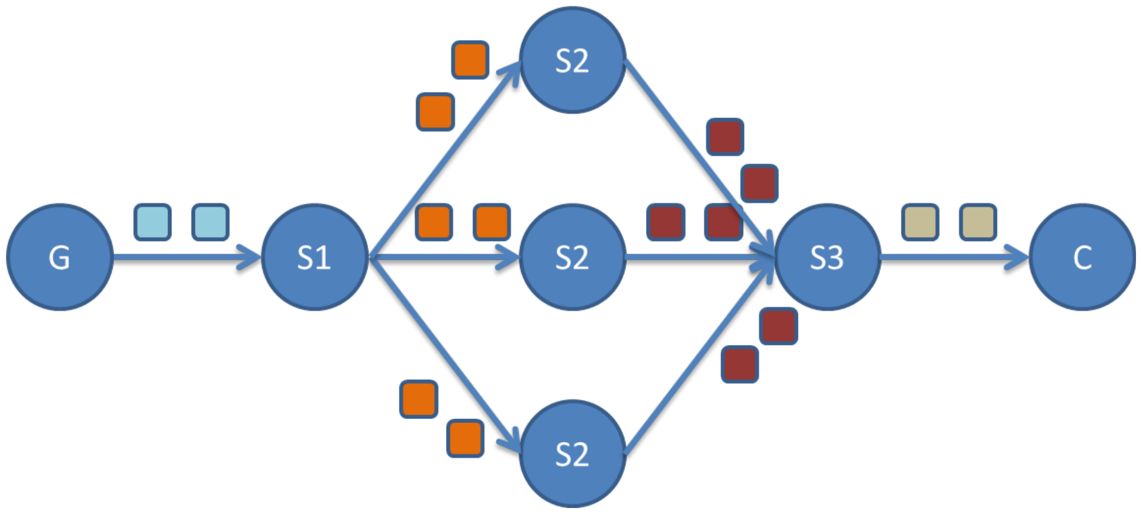
The simplest pipeline is a standalone or top-level pipeline. Invoking such a pipeline translates into its execution. Such pipeline has a first stage that acts as a generator of items and a last stage acting as a consumer of items. This is what happens when the generator stage obtains data from an external source (e.g. network) and the last stage sends data to another external source (e.g. disk).



An interesting and desirable property of a pipeline is composability. Given two pipelines P and Q. Its composition connects the last stage of P with the first stage of Q. In such a case, pipeline P does not have a consumer and pipeline Q does not have a generator.

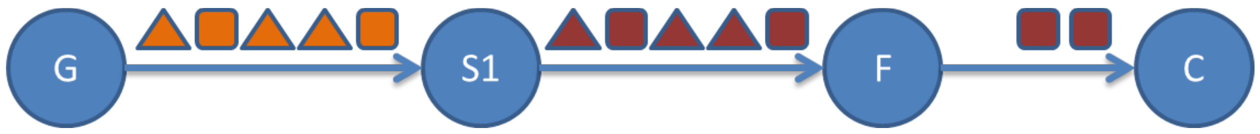


By default, every stage in a pipeline is a serial stage processing elements from the input and sending results to the output. However, when a stage is stateless, it can be replicated multiple times in different execution units to improve throughput.

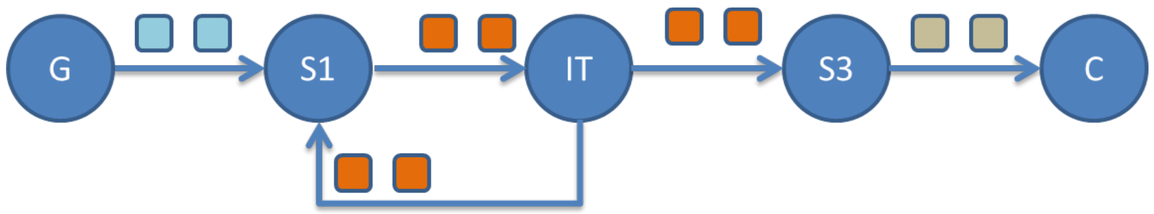


This is usually referred as a farm pattern and the replicated stage (S2 in this case) is said to be a farmed stage. As different replicas in the farm might produce items at different rates a key aspect here is ordering. Two options are possible here: ordered pipelines and unordered pipelines. An ordered pipeline causes more contention as out-of-order items need to wait until it is their turn to make progress. On the other hand, an unordered pipeline exhibits faster progress at the prices of losing order preservation. This is typically a decision that is at the will of application programmers and is highly dependent on the application domain.

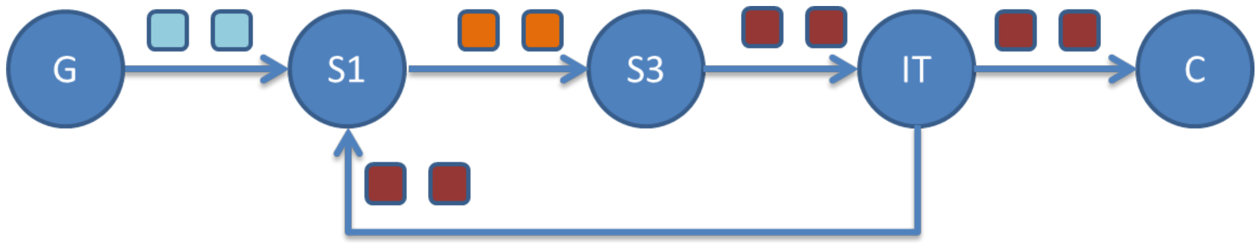
In some applications it might be interesting to discard items that do not satisfy a given predicate. A filter is just that. A special stage that either copies its input items to the output depending on some specific predicate. F is a filter.



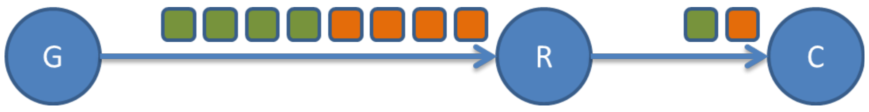
While a pipeline is usually unidirectional (from the generator to the consumer), sometimes it is interesting to allow cycles in the form of iterative pipelines. That is, for every item a predicate is evaluated and, when not satisfied, the output item is sent again the previous stage. IT is an iteration stage.



Thanks to composition, the body of the pipeline does not need to be restricted to a single stage, as the iteration body may be a pipeline itself.



In some online processing applications a version of reduction may be used. Obviously, a total reduction is nonsense as data items enter in the pipeline over time. However, a windowed reduction may be useful. In such reductions all elements in a window are reduced through some combination operation (satisfying the same requirements than in a regular reduction). Windows may be defined either as count-based (every n data items) or as time-based (for a given time window). R is a reduction stage.



Background Research: State of the Art

Pipelining means different things, based on the abstraction level, and there are various approaches for execution pipelines such as `tbb::pipeline` [TBB], [GrPPI], [FastFlow], range view with the | operator [Range-v3], etc.

A parallel stage processes more than one item at a time and can make pipelines scalable, but it does introduce an ordering issue. In a pipeline with only serial stages, each stage receives items in the same order. But when a parallel stage intervenes between two serial stages, the later serial stage can receive items in a different order from the earlier stage. Some applications require consistency in the order of items flowing through the serial stages, and usually the requirement is that the final output order be consistent with the initial input order [Structure2012].

Intel TBB [TBB] deals with the ordering issue by defining three kinds of stages:

- parallel: Processes incoming items in parallel
- serial out of order: Processes items one at a time, in arbitrary order
- serial in order: Processes items one at a time, in the same order as the other serial_in_order stages in the pipeline

The throughput of the two serial pipelines is still limited by the throughput of the slowest stage. The advantage of the serial_out_of_order kind of stage is that by relaxing the order of items, it can improve locality and reduce latency in some scenarios by allowing an item to flow through that would otherwise have to wait for its predecessor.

Pipelines with an arbitrary number of stages are not expressible in frameworks such as Cilk Plus. However, clever use of a reducer enables expressing the common case of a serial–parallel–serial pipeline. The general approach is:
1. Invoke the first stage inside a serial loop.
2. Spawn the second stage for each item produced by the first stage and feed its output to a consumer reducer.
3. Invoke the third stage from inside the consumer reducer, which enforces the requisite serialization.

There has also been a few C++ Std proposals including [N3534], [P0367R0], [P0374R0].
[N3534] uses a pipe operator which may be ok for simple cases, but may not be sufficient for a loop. However, we would be interested in collaborating with the authors. [P0367R0] is one of the co-authors of this paper and its design is covered in the Future Directions section. [P0374R0] is also one of the co-authors of this paper and their design is in [GRPPI].

A C++ standard proposal discussion on channels is tangentially related but we view pipeline and channels being slightly different [Proposal].

Aaron Robinson has a channel design which requires you to create threads and is mostly a synchronization primitive and is more low level.
https://bitbucket.org/linuxuser27/c-channel

Similarly, Boost.Fiber is also more a channel supporting both bidirectional MPMC concurrent unbounded and bounded queues. A talk by John Bandala in CPPCON 2016 also summarizes channels nicely though it intermixes with promises and futures [Bandela2016].

Meta-Direction Discussions and FAQ (no design yet)

Which thread owns which end?

Same thread owns both producing and consumer end.
Different thread owns producing and consumer end.
Some libraries use one thread per stage in the pipeline. This is one approach.
Intel TBB uses a task-based approach where every stage is a task and scheduler allocates task to threads which may be the same thread or different threads. In general we can choose thread based or task based and we think this should be just an implementation detail. We are not sure this will really affect the ABI a lot. TBB just say maximum number of tokens for the task based

approach. A thread-based approach would not usually add any more to the API. Ideally API should allow for both.

How does it differ from queues?

Queues [P0260R2] are an implementation mechanism that is usually needed to implement pipelines. Application does not need to see the implementation detail. Queues add parameters that should be passed to executors such as lock-free or locking properties of queues. Another dynamic parameter is queue size. There are a few parameters from queues that we may want to expose, but not all of them.

Eg. If there are more elements in flight, then we can make the generator stop generating new items to control the back pressure.

Are pipelines different from channels?

Yes, though the industry term is often mixed together with queues and pipelines. In this proposal, channels are SPSC concurrent bounded queues with capacity 1. Most similar to this are Golang Channels (not the buffered channels which can have any capacity), in that reads and writes are guaranteed to block, and the producer and consumer ends are logically separated. We view channels as important, but reserved for a follow-on proposal.

Is this for CPU and or GPU?

This proposal conforms to the abstract machine supported by existing C++ 17, but we would like the proposal to not block future efforts towards supporting pipeline between heterogeneous devices.

Fibers vs pipelines?

Fibers are just lightweight threads. Pipelines are much higher level.
Pipelines can be multiple or single unit of execution. Single unit execution can use fibers or coroutines. Multiple units of execution may only need threads. Specifically, Boost.Fiber supports both bidirectional MPMC concurrent unbounded and bounded queues..

Block or non-blocking?

We feel it should be a property of the pipeline.

SPSC vs SPMC vs MPMC?

When different threads can call push and/or pop without data races then, we have a multi-producer multi-consumer queue (MPMC) [Proposal]. If we will never have multiple people calling push() simultaneously (but still might be multiple people calling pop()), then its single-producer multi-consumer (SPMC). Or alternatively MPSC, or the simplest case, SPSC.

Those queues can be used to communicate pipeline stages. In a simple pipeline where a stage just receives data from the previous stage an SPSC queue may suffice. However, in a more complex case (e.g. a farm pattern) queues with either multiple consumers or multiple producers might be needed.

As soon as queues are used as a communication mechanism among stages a number or properties from the queues become configuration parameters from the pipeline:

- **Blocking versus non-blocking**: Queues may use some synchronization mechanism leading to locked queues. On the other hand lock-free queues make use of atomic memory accesses to avoid the expensive use of locks. However, there is no definitive answer on which approach is better as this is highly application dependent.
- **Bounded versus unbounded queues**: Many applications may use bounded queues as a mechanism to control maximum memory footprint and as back-pressure mechanism. Still some applications may decide to use unbounded queues and use the memory they effectively need. In the case of bounded queues an additional configuration parameter is the **queues sizes**.
- **Ordering**; The simplest approach is not to preserve relative ordering of items flowing through the pipeline. Such non-ordered behavior may arise, for example, from the use of farms which make such a stage not to be FIFO. Non ordered pipelines tend to exhibit a better throughput. On the other hand, it is also possible to use an ordered pipelines where relative order is preserved, which may be needed in some applications.

How is a Pipeline vs a DAG?

Pipeline implementation should be simpler then a DAG. Pipeline is SPSC not in a DAG.

What are some example applications (especially suited for pipelines)?

Signal processing, image processing, video processing, radar, medical imaging, 5G Software Defined Radio, Machine Learning, online processing, media, network protocol...

Bounded or unbounded?

Unbounded require dynamic allocation. If you have a generator faster then consumer (or some stage faster), you could run out of memory. Most pipeline application prefer bounded queues. We are interested in cases where unbounded queues is needed. Does unbounded need to be infinite, memory? The size would need to be set for unbounded queue in that case.

Bidirectional or unidirectional?

Pipelines are Unidirectional.

Customization point?

Since the concept of pipelining is broad and we want to keep it as a 0-cost abstraction, some kind of adaptability is required according to the application.
An obvious connection is with the executor proposals [P0443] about dispatching the pipeline stages.
Another customization point needs to address how the information actually flows across the pipeline stages. For example in the embedded world it would be interesting to use either co-routines and shared memory or on the opposite, some more hardware-level features such as OpenCL or SYCL pipes or even AXI4 stream busses on FPGA.

Independent forward progress?

The underlying implementation and hypothesis about what can be used inside the pipeline node will be different according to the warranty about Independent forward progress (IFP). Otherwise some deadlock might happen.
It would be interesting to have an implementation running on a free-standing implementation, like envisioned in [FreeStanding].

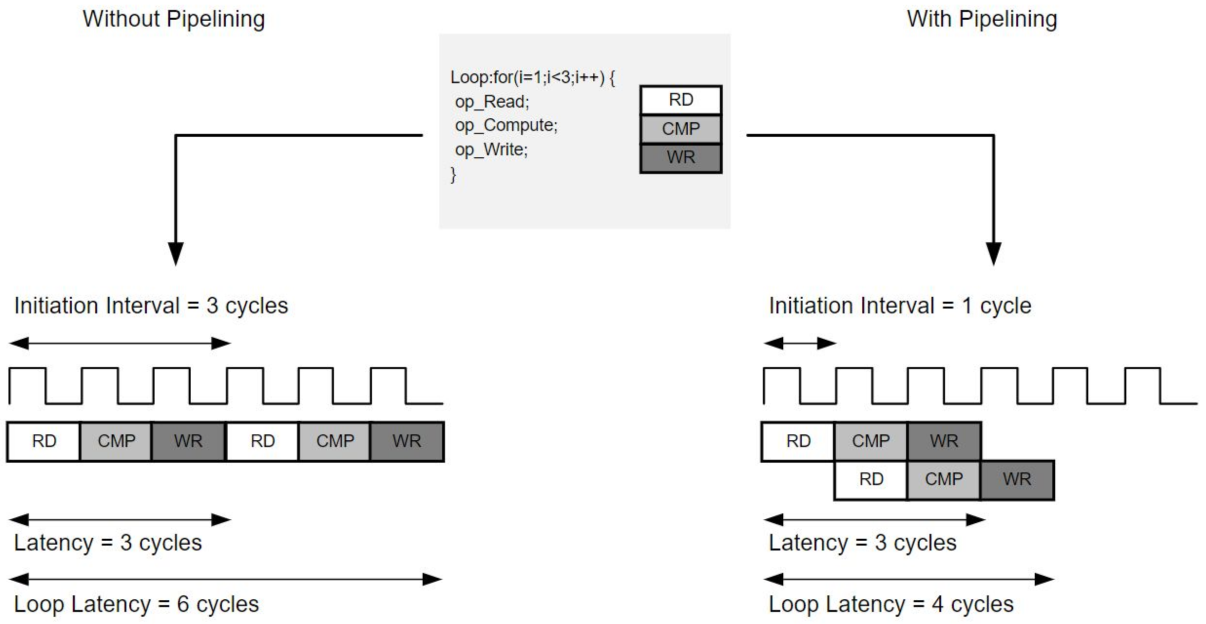
Future Directions

Compile-time pipeline specification

In some embedded system, setting up the pipeline resources might be too resource-hungry or time-consuming [FreeStanding]. But often the full structure of the pipeline is known at compile-time so it would be possible to devise a constexpr description of the graph such as [MetaGraph] and weave the pipeline plumbing using metaprogramming libraries such as [Boost.Hana], allowing some optimizations at compile-time, finding some schedules without preemptions, etc.

Pipelining at the operation level

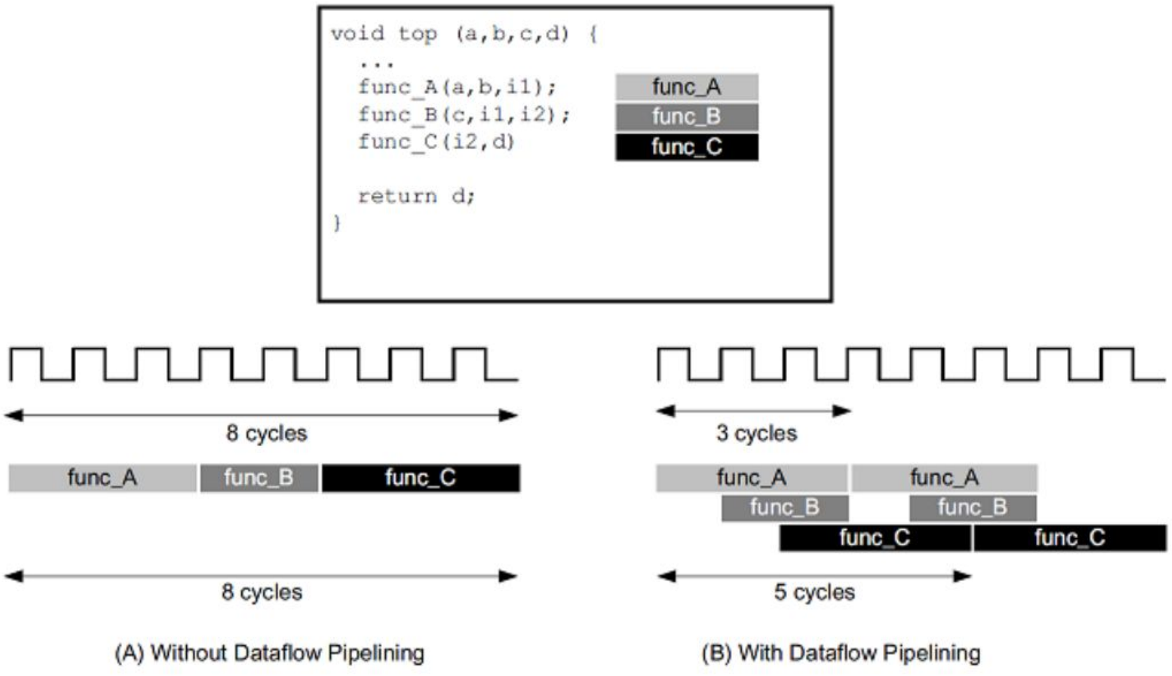
C++ is more and more used to program embedded systems and some architectures very far from the usual processors. For example in the case of FPGA, the C++ code is actually translated into a state machine driving a datapath with some operators, with the same semantics as the original program, but without an actual CPU. To increase the throughput, the datapath itself can be pipelined at a very fine level, quite finer than what is mainly exposed in this proposal. To pipeline or not to pipeline some parts of the code, that is the design question with different space/time/power trade-offs and thus the programmer should have control on this level of details, if required.



It has been experimented on Xilinx FPGA by using the SYCL C++ DSeL, an open standard from Khronos Group for heterogeneous computing, and adding some extensions to [triSYCL] open-source implementation. While this is a good candidate for using C++ generalized attribute such as `[[pipeline::instruction]]` on a block of statements since it does not change the semantics of the program, it would require some compiler support to handle it. So we prefered using some decorator functor `pipeline()` taking a lambda of the code to pipeline.
In the following code the body the body of the innerloop is asked to be executed in a pipelined way at the operation level.

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

Since on FPGA implementation the code can be synthesized into real hardware, all the functions of a program can co-exist and thus being executed in parallel mode in pipeline with compiler automatically inserting some pipelined hardware structures instead of using memory accesses, if the semantics of a sequentially-consistent order is preserved.
The following figure shows an example of code execution with low-level dataflow pipeline used.


```
void top (a,b,c,d) {
  ...
  func_A(a,b,i1);
  func_B(c,i1,i2);
  func_C(i2,d)

  return d;
}
```

It has also been implemented in [triSYCL] with a syntax relying on a decorator function `dataflow()` as shown in the following code sample.

```
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];
                             vendor::xilinx::dataflow([&] {
                                 readInput(buffer_in, d_b);
                                 compute(buffer_in, buffer_out);
                                 writeOutput(buffer_out, d_a);
                               });
                           });
```

More details can be found in [triSYCL-FPGA].

References

[Bandela2016] Channels in C++. John Bandela. CPPCON 2016 video.
[Boost.Hana] Your standard library for metaprogramming. Louis Dionne
https://github.com/boostorg/hana
[FastFlow] FastFlow: high-level and efficient streaming on multi-core, M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, In Programming Multi-core and Many-core Computing Systems, S. Pllana and F. Xhafa, Ed., Wiley, 2014.
[GRPPI] A Generic Parallel Pattern Interface for Stream and Data Processing. David del Río, Manuel F. Dolz, Javier Fernández, J. Daniel García. Concurrency and Computation: Practice and Experience. 29(24):e4175. 12/2017.
[FreeStanding] D1105: Leaving no room for a lower-level language: A C++ Subset. Ben Craig, Ben Saks.
https://htmlpreview.github.io/?https://github.com/ben-craig/freestanding_proposal/blob/master/core/leaving_no_room.html
[MetaGraph] Metagraph, Gordon Woodhull. https://github.com/gordonwoodhull/metagraph
[N3534] C++ Pipelines, Adam Berkan, Alasdair Mackintosh,
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3534.html
[P0260R2] C++ Concurrent Queues, Lawrence Crowl, Chris Mysen
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0260r2.html
[P0367R0] Accessors—a C++ standard library class to qualify data accesses. Ronan Keryell, Joël Falcou (05/2016) www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0367r0.pdf
[P0374R0] Stream parallelism patterns. J.D. Garcia, David del Rio, Manuel F. Dolz, Javier Garcia-Blas, Luis M. Sanchez, Marco Danelutto, Massimo Torquati. 05/2016.
www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0374r0.pdf
[P0443] A Unified Executors Proposal for C++ http://wg21.link/P0443
[Proposal]
https://groups.google.com/a/isocpp.org/forum/#!msg/std-proposals/PFsRPzrUrJ8/Xt7S8b0_BwAJ;context-place=msg/std-proposals/TrvJIkdAr7U/UggEpGfuAwAJ
[Range-v3] Range v3 — Experimental range library for C++11/14/17
https://github.com/ericniebler/range-v3

[Structure2012] Structured Parallel Programming, McCool, Robinson, Reinders, 2012
[SYCL] C++ Single-source Heterogeneous Programming for OpenCL
https://www.khronos.org/sycl/
[TBB] Threading Building Blocks (Intel® TBB) https://www.threadingbuildingblocks.org/
[triSYCL] https://github.com/triSYCL/triSYCL
[triSYCL-FPGA] Early Experiments Using SYCL Single-Source Modern C++ on Xilinx FPGA. Ronan Keryell & Lin-Ya Yu. IWOCL/DHPCC++ 2018, May 2018.
http://www.iwocl.org/wp-content/uploads/DHPCC_trySYCL.pdf