# P0687R0: Data Movement in C++

**Date: 2017-05-30**
**Project: Programming Language C++, SG14, SG1**
**Authors: Ruyman Reyes, Gordon Brown, Michael Wong, Hartmut Kaiser**
**Emails:** ruyman@codeplay.com, michael@codeplay.com, gordon@codeplay.com, hartmut.kaiser@gmail.com
**Reply to:** ruyman@codeplay.com

## Background and History

As of an SG14 evening session in Jacksonville demonstrating SYCL and HPX [1] [2], there was interest in supporting a mandate to drives towards a future C++ that supports distributed and heterogeneous (hereafter DH) computing. At a later meeting papers were submitted providing feedback on a variety of concurrency and languages features from the perspective of heterogeneous devices [3] [4]. Since then it was decided that the best path towards that direction was through completing the unified executors proposal [5] as this will provide a way forward that enables defining where, when and how work is executed. All of the current C++ standard execution functions such as std::invoke, std::async, parallel algorithms, etc, assume the usage of std::thread on only CPUs. While progress is being made in a unified proposal for executors, and while it enables DH computing, executors deliberately makes no attempt to support DH computing as it is simply too wide a scope.

Since then, there has been several papers which tries to address DH computing, (though none has identified in a complete manner all the issues that are faced by this domain,) that require a solution in the C++ standard. More specifically, a paper on the managed pointer [6] identifies a possible solution towards synchronizing data between different nodes. A paper on asynchronous algorithms [7] adds support for asynchronous algorithms vs the current synchronous versions.

The goal of this paper is to identify one set of the problems, that of data movement, faced by DH computing to non-DH programmers, which may still be a majority of the people on the C++ Standard Committee. It aims to identify those problems which may be as large as Memory Model once was before we adapted a memory model, then elicit a discussion on the best way to move forward towards a solution. While we have a solution in mind, using Channels, it is only one possible solution. A reasonable design of Channels can serve both homogeneous and DH computing.

## Motivation

When developers implement applications on DH systems, one of the key aspects to consider for performance is the placement of data. We identify DH systems to have similar issues, and therefore it would be ideal for them to have similar solutions. DH systems can feature complex memory hierarchies of different regions of memory with different properties (e.g bandwidth, latency, capacity). Developers must understand and use these memory regions appropriately on each DH system to leverage the maximum performance and reduce power consumption. Traditionally, the C++ standard has offered some limited capabilities to interact with complex memory hierarchies. For example allocators can be provided to containers, offering a mechanism to customize the allocation of the data. However, the ability to customize the placement of the data itself is not possible, except for creation of custom allocators.

In this paper, we describe various memory architectures for DH systems, describe the problematics of data movement in these architectures, and demonstrate the limitations of allocators. We hope for this paper to serve as a motivation for the C++ community to tackle the challenge of improving the native capabilities of the language to deal with these limitation.

## Current Status of the C++ Specification

The C++ standard as of C++17 offers allocators to customize data allocation, which can be used to allocate memory on a DH device.

There is ongoing work being done in defining a unified executors interface for C++, which includes execution on DH systems, however this work as it is does not yet attempt to created a unified interface for data movement.

As the act of dispatching work to a remote device to be executed was once only a problem for third party APIs and so too was the act of moving data to those devices for computations to be performed on. However, now that C++ is progressing towards a unified interface for execution the act of moving data is now a problem for C++ to solve as well.

Although some vendors offer complete system-level solutions where a pointer can be shared across multiple devices, this is not the common case. C++ developers will encounter situations where they want to plug in Execution units from different devices that do not share the same memory address - or they may choose to use a different mechanism to communicate the components for other reasons.

# Heterogeneous and Distributed Systems

## Heterogeneous Systems

An heterogeneous system features multiple processing units with potentially different ISAs and memory storage units. A heterogeneous system can execute the same program in all units, or different programs in different units that communicate among each other when required. The communication across different processing units can be via memory, via a bus or via network. In either case, communication may happen asynchronously and there is a potential overhead on communication.

An important aspect of heterogeneous systems is that different processing units may have different execution capabilities: Not all processing units will be able to execute fully featured C++ threads, atomics or transactional memory. This limits any interface from using advanced synchronization mechanisms.

## Distributed Systems

Distributed systems consist of a widely varying number of separate machines interconnected with a network allowing for the machines to communicate. Even for tightly coupled systems (beowulf clusters) each of those machines exposes its own address space, the OSes running on those machines are not ‘aware’ of the whole system. Applications running on distributed systems usually explicitly manage data placement and data-transfer where each of the machines manages its own address space. Some environments expose a global address space spanning the whole application (X10, UPC, Chapel, etc. use PGAS - partitioned global address space, HPX uses a more modern solution AGAS - active global address space). The solutions using a global address space simplify the data management, access, and transfer across machine boundaries. They however have to rely on either compile-time or runtime software support. Future systems may expose hardware support for global address spaces.

Loosely coupled distributed systems (cloud based, sensor networks, etc.) rely even more on ad-hoc data transfer solutions tailored to the used interconnect.

In all of the mentioned cases, the latencies introduced by accessing data across a network are several magnitudes higher than local memory access.

## CPU+GPU

A common DH system is the combination of a CPU and a Graphics Processing Unit (GPU). GPUs have gained popularity in the last 10 years as a co-processor capable of assisting the main CPU in highly intensive computational tasks. In this systems, a CPU normally orchestrates

the main program, and portions of the computation are offloaded to the GPU to speed up computation.

It is important to note that the requirements for memory from a CPU differ greatly from the requirements of a GPU. CPUs require high performance for random access patterns to memory, since the program that is executing may contain branches. In CPU computing, throughput is imperative, unlike a GPU. Also CPUs typically run an operating system, which triggers interruptions and I/O operations, which complicates the reuse of data memory. On the other hand, GPUs excel in running a single program, highly computational intense with little branching. GPU memories need to be optimized for multiple hardware microprocessors accessing simultaneously to the same or contiguous pieces of memory, delivering the requested data with low latency. Also, GPUs feature different levels of memory that must be programmed manually, and are accessible only by the GPU itself. This memory must be exploited in order to achieve performance.

Although in some systems the GPU and the CPU share a common physical memory, they are typically different physical memories that require some update on each side to maintain coherence. This update may be manual (via, for example, memory copy operations via a PCI-Express bus) or automatic (using system DMA or Cache Coherency protocols if there is hardware support).

In systems where more than one GPU exists, communication to other GPUs is orchestrated typically by the host, although some vendors offer capabilities to communicate directly to other GPUs typically using some DMA mechanism under the hood.

## System on Chip (SoC)

An SoC integrates on the same chip a number of processing units, such as one or multiple CPU, analog system processors, GPUs or Network processors. This highly complex design requires from all the different processing units to be able to communicate efficiently on demand. SoCs are heavily used on Mobile Phones, where they must run the operating system alongside the different user applications. SoCs usually need to ensure low-power consumption due to its high-density, hence, additional processing units are used only on demand, depending on the requirements of the different applications that are executing.

SoCs typically feature a common system memory that is visible by all units, but each unit can have additional storage to reduce latency. Due to the high number of co-processors on an SoC and the limited bandwidth available, efficient usage of the local storage and the global memory is critical to obtain performance.

## CPU + GPU/FPGA clusters

Datacenters and High Performance Computing use large clusters composed of multiple devices to attain performance on large scale applications. These clusters are typically composed by

multiple nodes that have one or more CPUs , and possibly one or more GPUs. FPGAs are increasingly being used more in this environments due to its adaptable programmability in the latest years.

These large clusters may run one or multiple applications simultaneously. An application running on these clusters must take into account at least three different memories: The individual memory of each node, the individual memory of each accelerator device, and the overall view of the distributed cluster memory.

Different vendors offer different networking/interconnect capabilities depending on the hardware users. Traditional Ethernet network connection is the most widespread, but alternative high-speed/low-latency networks, such as Infiniband, also exist.

Communication is typically synchronous across each pair of node or across multiple nodes if using optimized collective communications. Some distributed clusters offer the possibility of remotely accessing a portion of the memory of a remote node in an asynchronous mode.

Some vendors even offer the possibility for each individual GPU to communicate with another GPU in the cluster directly using this mechanism without interaction of the Host.

Each one of the aforementioned communication and memory mechanisms is programmed by a different library, programming model or API.

## CPU + FPGA systems / Programmable SoC

Some vendors offer systems where a fully-featured CPU is put together with an FPGA and a number of coprocessors. The FPGA shares memory with the CPU, but, as any FPGA, can be programmed to function as other co-processors. The FPGA can also be configured to offer features, such as advanced compression or ciphering functionality. The programmer must configure the FPGA to interact with the host CPU in the most efficient way possible.

FPGAs are partitioned into sub-devices which can communicate among each other using different mechanisms.

## Systems with HBM

Recent developments in memory technology have enabled the creation of High Bandwidth Memory (HBM), or 3d-stacked DRAM. HBM offers a very large width bus (eg. 4096 bits vs 32/64bit in typical GPU memories) with a slightly higher latency. HBM can (and is) used alongside other kinds of memory, and developers must evaluate carefully which data structures will be placed on this memory.

Different libraries exist from different vendors to program HBM memories, some of them implement HBM as a specific malloc function that returns a pointer which points to HBM memory.

## Definitions

Some definitions are required to clarify the following sections:

- Memory Accessibility: We define a memory M as accessible from processing unit A if and only if processing unit A can dereference a pointer to said memory M without requiring additional memory synchronization (e.g, barriers/fences) from the developer. For example, on a system with a CPU and a discrete GPU, RAM is accessible from a CPU, but (typically) not from a GPU.
- Memory Space: Memory residing in one component of a DH system. May or may not be accessible by other components.
- System Memory: Memory, typically off-chip, that is visible and accessible by all components of the DH system. Not all systems have this memory. Each processing unit of the DH systems may have different latency to access this memory. For example, a distributed cluster will probably not have a “system memory”.
- “Communicator” : The C++ interface object that would enable the data movement across processing units of a DH system.

## Design Considerations for a Data Movement Interface

When developers write applications in C++, they assume the machine model described by the C++ standard. However, this machine model does not take into account the different Memory Spaces of DH Systems, and the performance implications that leveraging locality offers. In some cases such as locality and affinity, even Homogeneous systems can take advantage of such features. The C++ machine model assumes pointers can be used to manipulate sequences of objects, hence a pointer to the start and a pointer to the end would suffice to describe an object. Note that, as we have defined previously, pointer address space is not contiguous in most DH systems, and even when it is, it is very important for users to precisely know where in the memory data is available.

Altering the machine model of C++ can be a challenging and difficult proposal and is out of scope for this document. Although we encourage and would collaborate on a complete holistic proposal in that regards , it will be unnecessary for the most common cases where programmers simply code a routine for a specific processing unit of a DH system.
We believe it is important that only some avid developers that need to be aware with the overall DH system should use the specifics of the interface. The rest of the developers should be able to reason beyond the normal C++ machine model.

For this document, we will focus on the interface for those developers who want to leverage the memory architecture of DH systems.

A communication interface for a DH system must enable developers to “connect” or to “move data” across Memory Spaces that are not accessible.

We distinguish the following actors that will interact with this interface:

- Vendors (V): Hardware vendors that offer interfaces to hardware capabilities
- System Developers (SD): Developers creating software interfaces or libraries for DH systems without having access to programming the hardware itself
- Library Developer (LD): Developers writing software libraries that will work on DH systems but without caring for an specific configuration of such system (e.g, a generic Machine Learning library for DH systems vs a machine learning library for a specific SoC)
- End user (EU): Any C++ developer that wishes to use an DH system

An interface to connect the DH system must expose the basic operations required to update different memory spaces in the system. These operations can be either synchronous or asynchronous, but the actual actions required to perform the update are unspecified. The only guarantee from the interface is that, when an operation is finished, the memory space is updated or, in case of an error, an exception is thrown.

Vendors and System Developers must be able to implement these basic operations for different combinations of Memory Spaces, by either offering pre-built facades or partially customized for particular systems.

Library Developers can leverage the communication interface to provide higher level abstractions that offer user friendly interfaces, such as managed pointers, containers, allocators, etc.

An “Heterogeneous Aware” library would simply expect to receive a concrete communicator implementation for a platform, but can be still written in a generic way.

Finally, End Users will simply instantiate the relevant communicator for their platform and use it in the DH-enabled libraries. In some cases, EU may not even need to care about this interface if it is encapsulated on higher levels of abstractions, such as Execution Policies.

# Motivational Examples

## #1: Data Movement Across a Heterogeneous System

In this section we are going to define a synthetic motivational example that will justify the necessity of an interface for dealing with data movement alongside execution on DH systems. Note that, although we present a simple example, this is based in our experience dealing with customers and other application developers writing Machine Learning, High Performance Computing, Vision and other applications in a large number of systems.

In these domains, it is common to read some input data from an off-system source (e.g, an image from a file, or data from the network) and run various computational kernels on the data to either generate new data or re-create a new output file (e.g, an inverted image).

In our example, we are going to use an application that reads a picture from a camera and detects whether if there is a face on it. If there is a face, the picture will be matched to an existing stored image, and if there is a match, the system will write an entry to a log file. The system is constantly running, matching faces and adding entries to the log file.

These operations are typically represented as stages on a pipeline, or as a data flow graph:


Each stage can be described as follows:

1) The input from the camera must be retrieved. Depending on the characteristics of the system, this can be IO intensive.
2) Image processing must be applied to the image: In order to simplify later stages, image contrast/bright/sharpness must be tuned. This is typically highly parallel and memory bound.
3) The Face Detection algorithm must process the image and extract the face from it, if exists, or simply bail out if no face has been found. These algorithms are highly parallel but can be more computationally bound
4) The system must go through all existing stored images trying to find a match with the existing one. There are multiple approaches to this problem, and vary from the size of the matching set it may ever require access to external storage.
5) Writing the log file is an I/O bound operation that requires access to off-chip memory.

Note that depending on the size of the problem at hand, the approaches for each stage will vary (e.g, matching the face of someone using the laptop for security purposes vs matching all faces going through airport security against a list of most wanted criminals). However the overall stages would be the same.

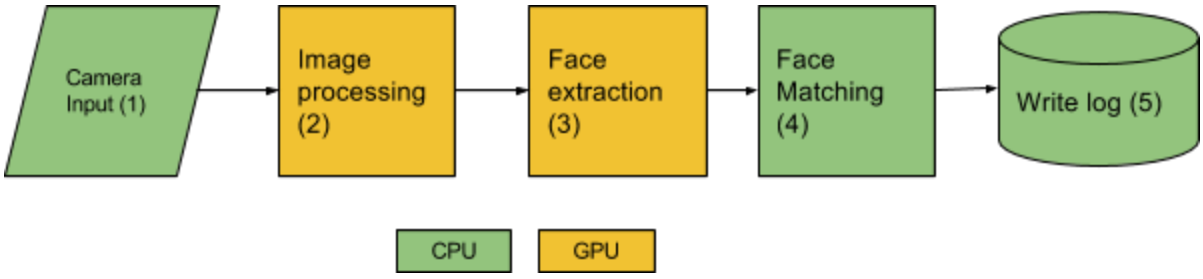
When a developer writes this application flow on a traditional homogeneous system, the only design consideration is how to optimally use the CPU threads to leverage the maximum performance, and possibly how to hide the latency to access IO. The focus is then on the *execution model* of the system, since, for example, threads can be used to parallelise computation or to enqueue a number of CPU threads.

However, when the same developer wants to port its application to a DH system, reasoning only on the *execution model* is not enough. IO cannot be performed from any processing unit. Special memory paths may exist between components (e.g, between the camera and a GPU) that will speed up computation. Hence, the *data model* becomes part of the application design.

However, when writing C++ applications, leveraging the *data model* requires in most cases dropping out of the language and entering into the realm of vendor specific libraries or non-standard code.

Let's map the flow of our application to different DH systems.

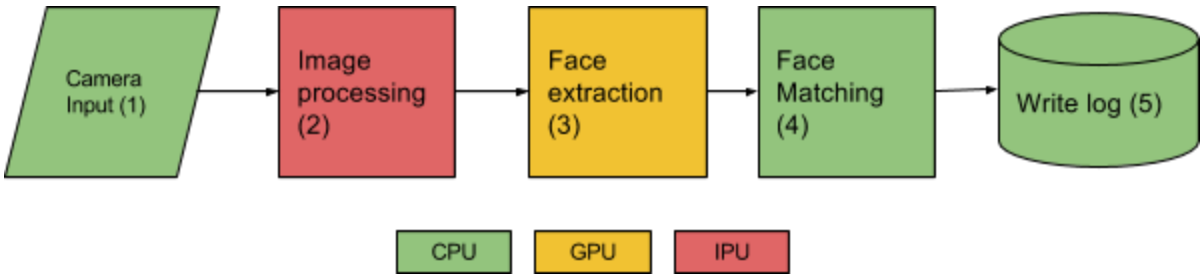
The first case is a system with CPUs and a single GPU. The CPU is the only one that can access I/O ports, hence all I/O operations must be executed on it. However, once data is on memory, the image processing and face extraction stages can be performed on the GPU to maximize parallel efficiency.



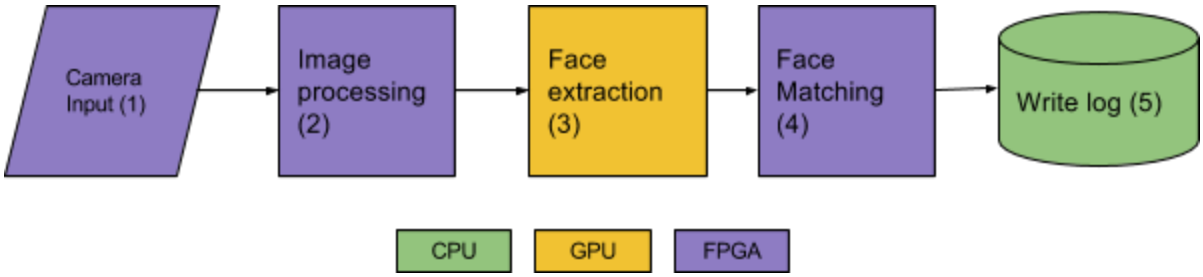

Note that there are multiple mechanism to transfer data from the CPU to a GPU that vary across architectures and vendors, but data must be moved in and out in all cases.

Now the application developer aims to port his application to an embedded system, where specific processing units can be used to optimize different stages. In this case, an Image Processor (IPU) can perform fast image processing, and has a direct data path to the camera. This processor is programmed using a set of library calls, but a pointer to the data is required.

The action of querying the picture must be performed on the host CPU but the data will not travel to the system memory, it will reside on an intermediate memory accessible by the IPU. Once the image processing operations are performed, the face extraction can be performed on the GPU. Depending on the actual embedded system configuration, the GPU may or may not share memory with the IPU, hence the data transfer may involve host or not. Finally, the CPU performs the final face matching and writing to log file.




Another possible DH system configuration that our application developer may target is a Programmable SoC, which contains an embedded CPU core and programmable logic (FPGA). In this case, the FPGA can directly access the camera and can be configured for high performance image processing operations. The GPU can still be used for face extraction, and depending on the system the GPU may have a direct path to the FPGA or not (hence going through system memory or not). The Face matching can also be performed on the FPGA itself, since it can have direct access to off-chip memory. The final step of writing the log file is kept in the CPU to save area on the FPGA.




In all the different examples, the flow of the application is the same, and although individual algorithms may be interchanged for performance reasons, the flow of the data through the system is the same. For example, each stage of this pipeline ma**y use a different Execution Policy from Parallel STL to define the different algorithms.**

However, Our application developer is faced with a challenge in this situation: Every time she ports his application to a new system, she has to re-program all the different stages to take into account the data movement across the system. Note that the initial assumptions the application developer made about threads and pointers are no longer valid, and she has to create new abstractions to tackle with this heterogeneity.

**There is no “Data Movement Policy” on Parallelism or Concurrency TS to help in this matter.**

An easy solution for this developer would be to use SYCL, the Khronos standard for C++ programming on DH systems. SYCL implementations are validated per DH system, hence, when running a SYCL program on a certified DH system, the data movement will be performed using the optimal datapath. This is possible due to the fact that the implementor of SYCL targets a specific DH system, and the abstractions of SYCL clearly distinguish between data storage and data access, enabling runtime dataflow generation.

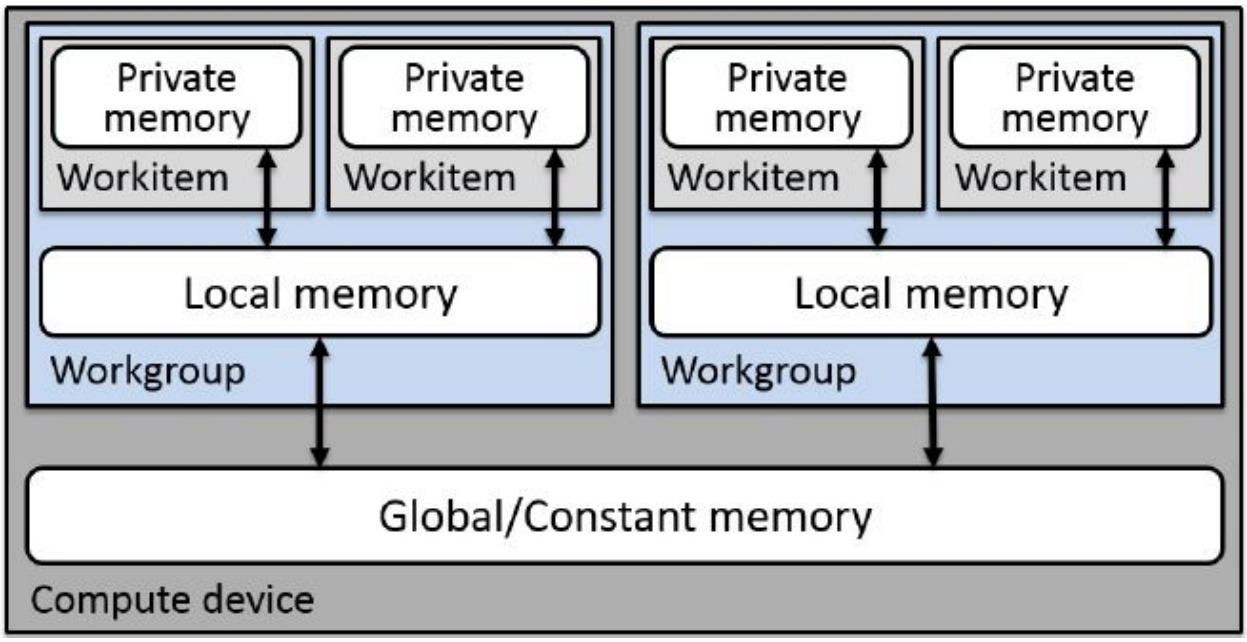
In general C++ programming, however, this assumption cannot be made. On DH systems, the C++ compiler typically does not target the whole system. Different C and C++ compilers and libraries are used to build libraries for different components that are then linked together into a single binary (or dynamically loaded at runtime). It is also important to note that, with some notable exceptions, **no single vendor controls all the components of an DH system**: Multiple vendors produce different components (compiler, libraries, interfaces, etc) for each piece. This prevents the developer of a piece of this systems to know how the whole system is plugged in together. In some cases, the team or company responsible for a piece of the whole does not even know the entire description of the whole system.

**When we foresee C++ used in DH systems, we do not picture a SYCL-style solution.** On the other hand: **we expect the C++ standard to be able to provide enough pieces for library and application developers to build their own high-level solutions, either via generic programming models like SYCL or domain-specific solutions**. We would like that all solutions are capable of interacting with each other so that the “glue code” required among different components is the minimum possible.

## #2: Memory Hierarchies inside Processing Units

The existence of multiple memory spaces on DH systems can also affect the programming inside individual processing units.

In this example, we focus on the OpenCL Memory model show in the picture below, but different programming models for DH systems will face similar challenges.




A Compute Device in OpenCL contains multiple Compute Units, each of them containing individual processing elements. When a program is mapped to OpenCL hardware, it is called an execution kernel, and the iteration space is mapped on the hardware by defining a number of workgroups which in turn contains a number of work items. Work items map to individual processing elements of the system (e.g, physical threads), whereas workgroup map to computation units. Note, however, that the hardware may be able to execute multiple workgroups simultaneously on the same computation units.

The OpenCL memory model describes three different memories on the system:

- Global/Constant memory: Global memory accessible by all processing elements of the compute device. This memory is visible by all compute units and processing elements but accessing it requires typically off-chip access which adds latency.
- Local memory: This memory is located inside the compute units and can be accessed by all processing elements of the workgroup, however, multiple concurrent access may only be performant on certain patterns due to bank conflicts.
- Private memory: This memory usually represents physical registers on each individual processing elements. However, if a kernel uses too many register, the compiler is allowed to “spill-off” into local memory (hence, reducing performance)

The three different memory spaces use different pointer addressing. In OpenCL the concept of “address space” is used to differentiate between the three of them. OpenCL device compilers must deal with “address space conversions”, which results in operations that will transfer memory across the three levels of the memory hierarchy.

A trivial example using operation that uses the three memories is shown below:

```
__kernel void test(__global float *x) {
 __local xcopy[GROUP_SIZE];
1:  int globalid = get_global_id(0);
2:  int localid = get_local_id(0);

3:  xcopy[localId] = x[globalId]
4:  barrier(CLK_LOCAL_MEM_FENCE)
  … rest of the code ...
}
```

*Example A: Accessing global memory using direct element access.*

Note that this is expressed in OpenCL C kernel language, where the different address spaces are expressed as pointers with attributes (__global or __local). Private memory does not need qualification as it is the default.

The movement of data is expressed as assignments , for example, on line 3, there is a copy from global memory (x) to local memory (xcopy). Since the address spaces are different, the compiler will generate the instructions required to perform the actual data movement internally.

Note that, depending on the memory architecture and capabilities, the time spent at the barrier will vary. In heterogeneous systems, the variation of time spent in the barrier can be noticeable. Compilers will try to identify invariants to the off-chip data access to execute before the barrier to hide latency. However, in some architectures with slow access off-chip to the global memory, this may not be enough.

To enable further latency hiding OpenCL C defines the async_workgroup_copy operation. This is a builtin function that enables all work-items on a workgroup to asynchronously copy information from global memory into local memory and vice-versa. The function returns a device event, that the user can manually wait at any point. The user can then use this mechanism to manually hide latency access to / from global memory.

Some low-power devices use this builtin function to perform the copy using a DMA device, freeing the processing elements to perform additional computations.

However, the kernel code will be different in this case. The listing below illustrate a basic usage of the work group copy function.

```
__kernel void test(__global float *x) {
 __local xcopy[GROUP_SIZE];
  int globalid = get_global_id(0);
  int localid = get_local_id(0);
  event_t e = async_work_group_copy(xcopy, x+globalid-localid, GROUP_SIZE, 0);
  wait_group_events(1, &e);
  ...  rest of the code ...
```

```
}
```

*Example B: Accessing global memory using asynchronous work group copy functions.*

Note that In some platforms, Example A or B will perform better, depending on the hardware architecture. Note also that the overall implications in the code are meaningful: Example A uses a barrier that waits for all the work items to perform their operations to local memory, whereas wait group events only waits for the actual local memory pointer being updated.

In order to program this kind of architectures in C++, an interface for both approaches must be present. If heterogeneous architectures are not taken into account, all approaches to memory accesses will follow the pattern on Example A, and may forget those situations where Example B is better.

## #3: Parallel STL

The example below shows a trivial example of the current Parallel STL usage in C++17, extracted from the getting started guide of Intel Thread Building Blocks [8].

```
#include <vector>
#include "pstl/execution"
#include "pstl/algorithm"

int main()
{
    std::vector<int> data(10000000);
    // Fill the vector with -1
    std::fill_n(std::execution::par_unseq, data.begin(), data.size(), -1);
    return 0;
}
```

This example fills an std::vector from the start until the end with values of -1 using a parallel execution policy. Since this is executing on the CPU, the individual threads of execution executing under the parallel execution policy can access the vector directly, and access concurrently, using a mutex or atomic operations to avoid race conditions. In this particular algorithm, the operation is embarrassingly parallel, thus, no synchronization is required in principle. Due to the usage of the par_unseq policy, the compiler will also try to use vectorization to optimize the operations, meaning more than one element of the std::vector will be accessed from the same thread using SIMD instructions.

With the proposed unified interface for execution, this algorithm can be instructed to execute on a particular execution context (a collection of execution resources targetable by an executor). The example belows shows how the previous example would look if we to target a GPU using an executor.

```
#include <vector>
#include "pstl/execution"
#include "pstl/algorithm"
#include "experimental/execution"

using namespace std::experimental::execution;

int main()
{
    std::vector<int> data(10000000);
    // Fill the vector with -1
    std::fill_n(std::execution::par_unseq.on(vendor_x::gpu_executor()), data.begin(),
data.size(), -1);
    return 0;
}
```

In the example above we create what's referred to as an inline executor (as it's constructed inline) that describes how work is to be executed on a particular GPU. The proposed interface for executors provides a range of executors for executing work in a variety of ways. These executors can then be used by control structures such as std::async, std::invoke or in this case the parallel algorithms.

An implementor of std::fill_n for a DH system has limited information on the input data. The categories of the iterators define how data can be accessed, but there is no information on where the data resides (or even if the data has been created all together, or is created whenever the iterator operators are used). Depending on the DH system being targeted, this can be critical to performance. When the execution of the parallel algorithm is dispatched to a processing unit of the DH system, the data must be moved as well. If the processing units reside on different memory spaces, the whole iteration range must be transferred alongside the dispatch. An implementation of Parallel STL is forced to first instantiate the entire iteration range on the caller, storing each element into temporary storage, then send the data to the processing unit for the processing to complete. Since the algorithm modifies the elements accessed, data then must be sent back to the caller of the algorithm, collected in temporary storage, and then applied again to the iterator range specified.
Different implementations of Parallel STL for different architectures will feature different ways of performing this operations! For example, in some platforms where sending elements individually is not expensive but the cost of intermediate copies is, the implementation may directly send each value of the iterator to the processing unit for the computation.

Note that this assumes there are no side effects on the increment/decrement of the iterators, thus, only being possible on Mutable Forward Iterators. InputIterator does not guarantee multiple passes, required for the heterogeneous dispatch. Indeed, a recent defect that was fixed before Parallelism TS was adapted into c++17 was to to change from InputIterators to ForwardIterators so as not to suffer from the invalidation multipass problem [p0467]

One other possible solution is to only allow Contiguous Iterators for those policies that target DH systems. When data is contiguous, it can be directly sent to the processing unit and written back without extra intermediate storage. However, an implementation can no longer detect the usage of Contiguous Iterators, since the contiguous_iterator_tag has been removed from the C++ specification.

The SYCL Parallel STL implementation [9] avoids this problem by relying on SYCL buffer objects to store the data. The example below, extracted from the github repository [10], shows the usage of various algorithms with the SYCL execution policy.

```
  std::vector<int> v = {3, 1, 5, 6};
  sycl::sycl_execution_policy<> sycl_policy;

  {
    cl::sycl::buffer<int> b(v.data(), cl::sycl::range<1>(v.size()));

    sort(sycl_policy, begin(b), end(b));

    cl::sycl::default_selector h;

    {
      cl::sycl::queue q(h);
      sycl::sycl_execution_policy<class transform1> sepn1(q);
      transform(sepn1, begin(b), end(b), begin(b),
                [](int num) { return num + 1; });

      sycl::sycl_execution_policy<class transform2> sepn2(q);

      long numberone = 2;
      transform(sepn2, begin(b), end(b), begin(b),
                [=](int num) { return num * numberone; });

      transform(sycl_policy, begin(b), end(b), begin(b),
                multiply_by_factor(2));

      sycl::sycl_execution_policy<std::negate<int> > sepn4(q);
      transform(sepn4, begin(b), end(b), begin(b), std::negate<int>());
    }  // All kernels will finish at this point
  }    // The buffer destructor guarantees host synchronization
std::sort(v.begin(), v.end());
```

When using the SYCL execution policy, the algorithms are executed on the SYCL device selected by the implementation. The std::vector object is mapped on the SYCL buffer. The SYCL implementation now can use this pointer in an implementation-specific way that is optimal for the platform memory architecture. The different policies operate on iterators to the SYCL buffer, instead of iterators to the original std::vector.

Following SYCL rules, data remains under the control of the SYCL runtime until the end of the buffer scope. After the buffer is destroyed, the data will be guaranteed to be back on the original std::vector.

When using iterators to SYCL buffers, information is passed to the SYCL implementation as to where the data is. The data flow execution rules guarantee data is available on the different processing units.

The SYCL Parallel STL implementation only uses iterators as offsets from the starting position of the buffer. The buffer itself is retrieved back from the iterator to perform the actual operations. The listing below shows the implementation of the fill algorithm on Parallel STL [11].

```
/* fill.
 * Implementation of the command group that submits a fill kernel.
 * The kernel is implemented as a lambda.
 */
template <typename ExecutionPolicy, typename ForwardIt, typename T>
void fill(ExecutionPolicy &sep, ForwardIt b, ForwardIt e, const T &value) {
  cl::sycl::queue q { sep.get_queue() };
  auto device = q.get_device();
  size_t localRange =
    device.get_info<cl::sycl::info::device::max_work_group_size>();
  auto bufI = helpers::make_buffer( b, e );
  // copy value into a local variable, as we cannot capture it by reference
  T val = value;
  auto vectorSize = bufI.get_count();
  size_t globalRange = sep.calculateGlobalSize(vectorSize, localRange);
  auto f = [vectorSize, localRange, globalRange, &bufI, val](
      cl::sycl::handler &h) mutable {
    cl::sycl::nd_range<1> r{
        cl::sycl::range<1>{std::max(globalRange, localRange)},
        cl::sycl::range<1>{localRange}};
    auto aI = bufI.template get_access<cl::sycl::access::mode::read_write>(h);
    h.parallel_for<typename ExecutionPolicy::kernelName>(
        r, [aI, val, vectorSize](cl::sycl::nd_item<1> id) {
          if (id.get_global(0) < vectorSize) {
            aI[id.get_global(0)] = val;
          }
        });
  };
  q.submit(f);
}
```

The helper function “make_buffer” detects when the iterators are SYCL Buffer iterators and retrieves the original buffer used as input.

When the iterators are not SYCL buffer iterators, the buffer must be constructed and the iterator range copied. The implementation details can be found in the sycl_buffers.hpp header [12].

Since no assumptions can be made from the input range data, temporary storage and copies must be used. This causes a noticeable performance drawback.

In general, making copies of the argument for Parallelism TS was disallowed but has now also been enabled for C++17, through a Swiss National Body comment [CH11, P0518] which now

allows parallel algorithm arguments to function objects to be copied to a separate space for non-sequenced policies.

Additionally as of C++17 the C++ standard now defines a set of forward progress guarantee definitions; concurrent forward progress, parallel forward progress and weakly parallel forward progress. Due to the nature of many processing units such as GPUs there are many DH devices which can only guarantee weakly parallel forward progress. However C++17 currently requires the parallel algorithms to guarantee concurrent or parallel forward progress guarantees. These forward progress guarantee definitions will need to be extended if C++ is to natively support execution on DH systems.

# Open Questions for C++ Capabilities

## Moving vs Using

It is important to decouple the concept of moving data across the DH system with how the data is used. Moving data implies updating memory spaces on separate processing units, but once the data is moved (or, better put, updated), data can be accessed in various ways.

The executors proposal explains how to expose parallel and asynchronous execution of functions in different execution agents.

Moving data is an important part of the synchronization, since tasks that depend on data that has yet to be moved cannot start until such data is available. For this reason, a data movement interface for C++ must offer different mechanism that facilitate other higher level mechanism to check the status of the data.

How can the user express the usage of data in different processing units of the DH system?

The managed pointer proposal provides an interface that enables synchronisation and accessing data in an DH system at a high level, but avoids defining how this data is moved across the system at the lower level.

## Iterators

C++ uses Iterator concepts to define the generic interface of the STL algorithms. Iterators are used (and abused) in multiple libraries to express multiple different ways of accessing and even generating data for algorithms on the fly. However, an iterator only specifies the point of access to a single element on a container (or a single element that has been generated on the fly). The operator overload for the increment and decrement of the pointer are only guaranteed to return a valid element access (or the end of an iteration space, i.e std::end(Iterator)). There is no guarantee on the layout of the data being accessed. This poses various limitations on DH systems.

## Using std::copy

A potential solution for DH data movement is to use a std::copy where origin and destination can simply be pointers or iterators on different parts of the DH system.

Although this is not obvious on a normal homogeneous system, a copy operation mandates a specific order of execution. A copy states an origin, a destination, and an order of execution. A copy is performed by the caller thread, and simply operates on the memory that is visible by it. The other thread needs not to be aware of when this copy is performed, and if it needs, other execution synchronization mechanism, such as mutex or atomics, can be used to keep the threads in sync.

In DH systems, data movement can be performed by one side (e.g, a host puts data on a GPU), by both sides (e.g, a node sends data while other receives data), or by a third party (e.g a Vision Processor offloads data via a DMA controller to an off-chip HBM memory block). In any of the three cases, the destination must know when/if the data is ready to be read, but it should also not be blocked by waiting for the data to be available.

In HPX we use the copy() algorithm for data transfer from a to an HD. The implementation of copy() relies on a set of traits which enable various optimizations and specializations depending on the used iterators as data allocated on the device is represented by special iterator types. For instance, this allows for mapping the data transfer initiated by copy() either onto DMA or other highly optimized operations specific to the used device.

Also, the asynchronous algorithms implemented by HPX enable a transparent overlap between the data transfer operation to or from the device with other, possibly unrelated work on the CPU. The asynchronous copy() algorithm in HPX returns a future object which becomes ready once the data transfer operation has finished. This enables seamless synchronization with other operations running on the CPU.

## Using dedicated containers

A possible approach to tackle the above problem is to use allocators and containers to represent data on DH systems. This approach has been used by various libraries. For example, the example below is extracted from the Thrust documentation [13].

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>

#include <iostream>

int main(void)
{
    // H has storage for 4 integers
```

```
    thrust::host_vector<int> H(4);

    // initialize individual elements
    H[0] = 14;
    H[1] = 20;
    H[2] = 38;
    H[3] = 46;

    // H.size() returns the size of vector H
    std::cout << "H has size " << H.size() << std::endl;

    // print contents of H
    for(int i = 0; i < H.size(); i++)
        std::cout << "H[" << i << "] = " << H[i] << std::endl;

    // resize H
    H.resize(2);

    std::cout << "H now has size " << H.size() << std::endl;

    // Copy host_vector H to device_vector D
    thrust::device_vector<int> D = H;

    // elements of D can be modified
    D[0] = 99;
    D[1] = 88;

    // print contents of D
    for(int i = 0; i < D.size(); i++)
        std::cout << "D[" << i << "] = " << D[i] << std::endl;

    // H and D are automatically deleted when the function returns
    return 0;
}
```

In this case, the library exposes different host and device containers. Data is moved from a host CPU to a GPU using copy or assignment operations. Data on the device can be directly accessed using subscript operators, but this comes with a great performance penalty.

Thrust algorithms are overloaded to use this container types in a similar way that SYCL Parallel STL does.

Note that this is only a host-centric approach: data is allocated on the host and offloaded to the device. We should consider all the other possible approaches.

The concept of a container per processing unit forces developers to manipulate manually the data copy operations, or accidentally fall on a performance pitfall. The implementor of a library needs to know the properties of the container it is accessing in order to implement the algorithm efficiently. For example, if the container is in a device memory but the execution is on the host, individual element-wide access must be avoided.

Similarly, this limitation makes it impossible for a container to span across device and host memory. In addition, the problem of the address spaces is not clearly resolved. Inside of the processing unit, a container for the different memory spaces is required. In the case of OpenCL, a container for local and global memories would be created. Developers must query the properties of these containers to ensure performant access. The different execution units (using nomenclature from the Executors proposal) would have to collaborate to ensure efficient access to the container is performed. In addition, the whole range of containers will need to be ported for the different devices.

## Allocators and Containers

Using standard STL containers with custom allocators for the different memory spaces could potentially offer a solution for the different memory spaces. An allocator for each memory space can be provided by the implementor, and then generic STL containers used to prove access to the data.

Note however that, although the allocator is specific to the memory space, the container is not. An implementor of a library that uses containers would need to query for the properties of the allocator to identify in which memory space the data resides, and map the correct processing unit of the DH system to perform the operation.

A side problem of explicit copy operations on vectors and containers is the maintainability of the code. In the following example, the user creates some data on the host and uses a Custom Device Allocator to allocate data on a processing unit of the DH system. The data is initialized on the host, and then copied on the device.

The user calls some third party library code that operates on device data. This third party code is outside of the application developers control.

Then, the application developer performs afor_each operation on the device, and copies data back to the host.

```
{
  std::vector<int> hostData;
  std::vector<int, CustomDeviceAllocator> deviceData;

  std::fill(std::begin(hostData), std::end(hostData), 1);
  std::copy(hostData, deviceData);
  // Call some existing library that uses data on the device
  some::third::party::library::code(heterPolicy, deviceData);
  // Now execute a for each on a processing unit
  for_each(heterPolicy, std::begin(deviceData), std::end(deviceData), [&](int& elem){
elem++} );
  std::copy(deviceData, hostData);
}
```

Note that this has created an explicit ordering of the code:

- What if the library changes now decides to leave data on a different memory space?
- What if we change the policy of the for_each to execute on the host?

The maintenance of the code is now more complex, even in this simple example, since **each statement has a dependency on where the data is, and not just whether the variable that represents the storage is visible or not**.

Note that compilers can do little to nothing to solve this problem: all this operations are expressed at a much higher level that what the compiler can optimize in terms of data movement.

Should this data movement implications be solved at C++ level? Or is it a problem for library vendors?

# Final Remarks

The paper has presented a number of examples and open questions that arise when trying to use C++ on DH systems. The current approach by users and industry is plagued with vendor-specific solutions that do not interoperate among each other. Meanwhile, application and library developers are faced with a myriad of interfaces and porting issues across different architectures. Given the current trends towards more DH hardware, this will only aggravate the programmability issues of C++.

# References

[1] P0234r0 Towards Massive Parallelism (aka Heterogeneous Devices/Accelerator/GPGPU) support in C++ with HPX
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0234r0.pdf

[2] P0236r0 Khronos's OpenCL SYCL to support Heterogeneous Devices for C++
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0236r0.pdf

[3] P0362r0 Towards support for Heterogeneous Devices in C++ (Concurrency aspects)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0362r0.pdf

[4] P0363r0 Towards support for Heterogeneous Devices in C++ (Language aspects)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0363r0.pdf

[5] P0443r1 A Unified Executors Proposal for C++
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0443r1.html

[6] P0567r1 Asynchronous Managed Pointer for Heterogeneous and Distributed Computing

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0567r1.html

[7] Invoking Algorithms asynchronously
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0361r1.pdf

[8] Intel Threaded Building Blocks: Getting started with ParallelSTL
https://software.intel.com/en-us/get-started-with-pstl

[9] SYCL ParallelSTL implementation
https://github.com/KhronosGroup/SyclParallelSTL/

[10] SYCL ParallelSTL sample code
https://github.com/KhronosGroup/SyclParallelSTL/blob/master/examples/sycl_sample_01.cpp

[11] SYCL ParallelSTL fill algorithm implementation
https://github.com/KhronosGroup/SyclParallelSTL/blob/master/include/sycl/algorithm/fill.hpp

[12] SYCL ParallelSTL make_buffer implementation
https://github.com/KhronosGroup/SyclParallelSTL/blob/master/include/sycl/helpers/sycl_buffers.hpp

[13] CUDA Thrust
http://docs.nvidia.com/cuda/thrust/index.html#axzz4kBASUDD3

[14] Working Draft, Standard for Programming Language C++
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4618.pdf

[15] Abstraction and the C++ Machine Model
http://www.stroustrup.com/abstraction-and-machine.pdf

[16] The Portals 4.1 Network Programming Interface
http://www.cs.sandia.gov/Portals/portals41.pdf

[17] Altera OpenCL SDK
https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf

[18] OpenMP Technical Report 5: Memory Management Support for OpenMP 5.0
http://www.openmp.org/wp-content/uploads/openmp-TR5-final.pdf