Document number: P0236R0
Date: 2016-02-12
Project: Programming Language C++, SG14, SG1, EWG
Authors: Michael Wong, Andrew Richards, Maria Rovatsou, Ruyman Reyes
Reply to: fraggamuffin@gmail.com

Khronos's OpenCL SYCL to support Heterogeneous Devices for C++

1. Introduction
2. Compare SYCL to other programming models
3. Design Considerations of Massive Parallelism languages
    3.1 Language Style
        3.1.1 C++ Embedded DSL
        3.1.2 C++ Kernel languages
        3.1.3 C++ single-source
    3.2 How Parallelism is Exposed
        3.2.1 Directive-based Parallelism
        3.2.2 Thread-based Parallelism
        3.3.3 Explicit Parallelism
    3.4 Memory Model
        3.4.1 Cache coherent single virtual address space
        3.4.2 Non cache coherent single-address space
        3.4.3 Multiple-address spaces
4. SYCL and OpenCL Overview
    4.1 Data Management in SYCL
    4.2 Parallelism in SYCL
    4.3 Shared Source in SYCL
5. Impact on the C++ Standard
6. Using SYCL to implement C++17 features
7. Goals and Conclusion
8. Acknowledgement
9. References

1. Introduction

This paper shows the experience gained from a heterogeneous device programming model that is designed to work with C++ templates, that has been in production compilers for a consumer domain of a wide variety of devices.

SYCL™[1][6] was designed as an enhancement to Khronos's OpenCL™[2][9] specification specifically to focus on enabling C++ developers to dispatch work to massively parallel cores, and expose the performance capability of OpenCL devices such as GPUs, FPGA, DSPs and other accelerators. This paper describes some of the main design considerations and how we came to the decisions in the final spec.

SYCL is a specification which defines a single source C++ programming layer that is built on top of OpenCL. This allows developers to leverage C++ features on the range of heterogeneous devices supported by OpenCL, providing a foundation for creating efficient, portable and reusable middleware libraries and applications. The SYCL specification consists of two tightly coupled components: a C++ runtime library and a SYCL device compiler. The runtime library contains an abstraction layer built on top of the core OpenCL runtime APIs and interoperable data structures for sharing data between host and device code.

It uses modern C++14 and allows us to separate the "what" from the "how", similar to current thinking in the C++ community. Specifically, it allows us to separate what the user wants to do, such as science, computer vision, or AI from how it is to be run, which could include one or more OpenCL devices.

The design aim of SYCL was to enable a complete C++ ecosystem for OpenCL, which was previously based on top of the C99 specification. It enables C++ libraries to work with OpenCL. As such, it must run on many devices such as GPUs, FPGAs, DSPs, but also CPUs. SYCL also requires code to work with C++ template libraries to best support compile-time polymorphism, and embedded devices on which dynamic runtime compilation is not possible. One of the purposes of defining an open standard is to enable an ecosystem with full support for the entire development toolchain such as compilers, debuggers and IDE's. In the long-term, SYCL aims to continue to support the features of OpenCL as it develops.

When designing SYCL, we decided we would not change, or extend, the C++ language. , There already exists established ways to map C++ to parallel processors (for example, using the existing Parallelism TS proposal). Therefore, SYCL followed established approaches as much as possible. We also worked to enable building the C++ Parallel STL [3] on top of SYCL by adding no more than what was needed.

[1] SYCL and the SYCL logo are trademarks of the Khronos Group Inc
[2] OpenCL and the OpenCL logo are trademarks of Apple Inc.

One of the major benefits that SYCL provides is to enable developers to write OpenCL kernels using a subset of C++ features. As SYCL 1.2 [5] is built on top of OpenCL 1.2, the subset of C++ features supported by SYCL is defined by the set of features supported by OpenCL 1.x devices. SYCL can be built on top of SPIR™[3][7], however the specification does not mandate it. (**Standard Portable Intermediate Representation** (SPIR™) is an intermediate language for parallel compute and graphics by the Khronos Group, originally developed for use with OpenCL.) Any binary format supported by an OpenCL implementation can be targeted by a SYCL implementation.

Examples of C++ features supported in the current SYCL 1.2 specification are: templates, classes, operator overloading, static polymorphism and lambdas.

Examples of features that cannot be supported (on device) in the current SYCL 1.2 specification are: function pointers, dynamic memory allocation, dynamic polymorphism, pointer struct members, runtime type information, exception handling and static variables.

It is royalty-free to implement, and its design and implementation serves as a basis for building support for massive parallelism in C++ in several key domains. However, future considerations should not be constrained by its design.

2. Compare SYCL to other programming models

SYCL is designed to align with the future direction of the C++ Standard, but also aims to ensure future OpenCL devices can be supported. By adding no language extensions to C++ we enable a CPU-only implementation of SYCL to be implementable on any C++ compiler. Apart from some very minor special cases, there are no macros or extensions that would break language compatibility. It does not add 'restrict', which means all code can compile also for the host as well as on the device. It provides the full OpenCL feature set in C++ which means anything you can do with OpenCL can now be done through C++ . Figure 1 shows how SYCL fits within the OpenCL environment. SYCL has a C++ compiler that can output SPIR as an Intermediate Representation for accessing OpenCL devices, as well as an OpenCL runtime. This means the same C++ Template library on top of SYCL can be compiled both for the host CPU, or a custom processor, or SYCL to access target devices through a single source.

The SPIR IR is in binary format and allows the user to put other languages on top of OpenCL. It is an OpenCL extension for OpenCL v1.2, to be superseded by SPIR-V[8], a new IR, which is part of the core OpenCL 2.1 specification. SPIR prior to the 2015 SPIR-V release was based on the LLVM Intermediate Representation. SPIR-V is a high-level intermediate language, exchanged in binary form. Functions are represented by a control flow graph of basic blocks, using static single assignment(SSA) form. Data structures retain high-level hierarchical representation. It is not lossy

[3] *SPIR and the SPIR logo are trademarks of the Khronos Group Inc*

like previous byte-code or virtual machine-like intermediate representations that is used for graphical shaders. This allows higher performance lowering to target devices

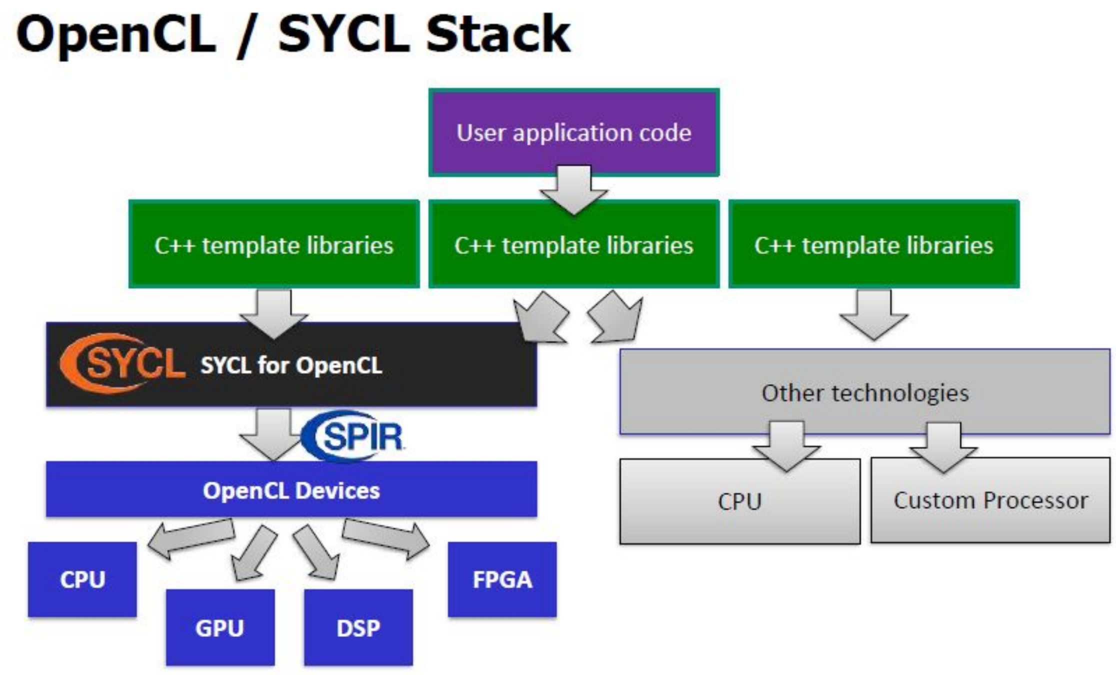


Figure 1: how SYCL fits within the OpenCL environment

3. Design Considerations of Massive Parallelism languages

To design language support for massive parallelism, there is a spectrum of possible language choices. We can contrast between :

1. Language style which selects between
    a. Embedded DSLs,
    b. Kernel Languages,
    c. Single-source
2. Parallelism can be exposed to compilers using
    a. Directive-based approach
    b. Thread-based approach
    c. Explicit parallelism approach
3. Different platforms and devices have different memory models that vary between:
    a. Cache-coherent single-address space
    b. Non-coherent single address space
    c. Multiple-address spaces

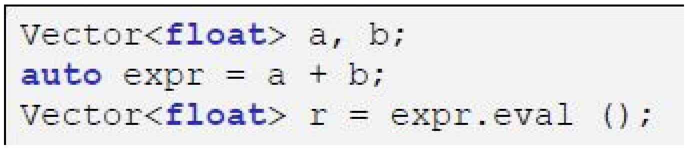
By examining a history of such designs, for which there is now a large sample of previous attempts, we can formulate a basis for how massive parallelism in C++ can best take advantage from their learning.

3.1 Language Style

This describes the various styles of adding massive parallelism to C++ through Domain-specific language usually as a library, a separately-compiled Kernel language, or a Single-source that combines both CPU and device language. There have been experiments both commercially and in research to support such efforts. We list many of their benefits and disadvantages.

3.1.1 C++ Embedded DSL

A **domain-specific language** (**DSL**) is a computer language specialized to a particular application domain. This is in contrast to a general-purpose language (GPL), which is broadly applicable across domains, and lacks specialized features for a particular domain. An **embedded** (or **internal**) **domain-specific language** is implemented as libraries which exploit the syntax of their host general purpose language or a subset thereof, while adding domain-specific language elements (data types, routines, methods, macros etc.). (e.g. RapidMind, Halide, Embedded SQL, LINQ).



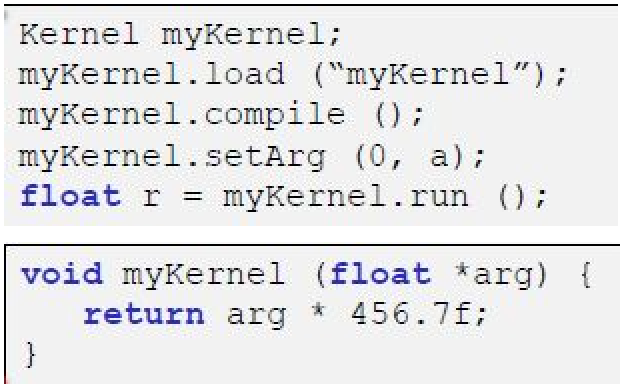
```
Vector<float> a, b;
auto expr = a + b;
Vector<float> r = expr.eval ();
```

Figure 2: A simplified example of what a C++ DSL might look like

This is convenient and fast to implement and can work with existing C++ compilers, but it is hard to express control flow, or be composable. One Example is Sh/RapidMind, which is exposed as a set of C++ libraries, that provide types and operations used to express parallel computations. The programming model is primarily data parallel, although it is sufficiently generic to express task-parallel operations. The platform targeted multi-core x86 processors, GPUs (via OpenCL), and the Cell processor. Another example is Halide, a computer programming language designed for writing image processing code that takes advantage of memory locality, vectorized computation and multi-core CPUs and GPUs. The problem with such compile-time approaches is that it is limited in expressibility as you have no way to build up control-flow-trees in C++ using overloading and each data type you define needs to be output in the runtime-generated code. In Figure 2, a C++ template library would use overloading to build up an expression tree to compile at runtime. Such code can still be implemented and compiled on CPU, or on a full single-source C++ platform like SYCL, without the control-flow or user-defined-type complications of dynamic code generating implementations of a DSL.

3.1.2 C++ Kernel languages

A C++ Kernel language uses separate source for host code and device code. The host (CPU) code loads and compiles kernels for specific devices, while setting arguments and dispatching the execution. One Example is GLSL, a high-level shading language based on the syntax of the C programming language. Shader languages are very widely used in graphics, where the separate source nature enable a separation between the graphics *engine* and the specific shading, or lighting, of individual triangles being drawn on screen. Another kernel language is the OpenCL C and C++ kernel languages. This is the normal OpenCL approach, where kernels are loaded and compiled separately from the host CPU source code.



```
Kernel myKernel;
myKernel.load ("myKernel");
myKernel.compile ();
myKernel.setArg (0, a);
float r = myKernel.run ();
```

```
void myKernel (float *arg) {
   return arg * 456.7f;
}
```

Figure 3: OpenCL Kernel Language (top part is CPU code, and bottom part is device Kernel)

The advantage of this approach is that it is very explicit what is running where. Also, there is a clear independence between source code, compiler, and runtimes for each device and the host CPU. Also, this approach enables code to be generated at runtime. The main problem is that it is still hard to compose across multiple devices and hard to move code around and define where the interface is. For example, it is not possible to define the C++ Parallel STL in a kernel language environment, as Parallel STL assumes a single source file with shared data types between host and device.

3.1.3 C++ single-source

Many of the most widely-used C++ programming models for accelerators (outside the graphics domain) are single-source. One example is C++ AMP, which provides an easy way to write programs that compile and execute on data-parallel hardware, such as graphics cards (GPUs). C++ AMP is a library implemented on DirectX 11 and an open specification from Microsoft for implementing data parallelism directly in C++. CUDA is also a single-source C++ programming model created by NVIDIA. The Thrust C++ library provides a modern single-source C++ style of programming using CUDA. The OpenMP open-standard is also single-source, which uses pragmas to support many form of accelerators with an HPC focus. OpenACC is similar to OpenMP and was developed from a group of OpenMP members to bring to market an accelerator programming standard earlier than OpenMP.

Such languages are easy to use, are composable and can be type-checked as everything is in one source file. They enable offline compilation, so that code is shipped in binary format and checked at compile time. This is the design chosen by SYCL, where a single source file can be compiled for host CPU, with the kernels also being extracted from the source and compiled for one or more OpenCL devices.

Single source is likely to be the future direction for the C++ Standard support for massive parallelism, as it is consistent with the current design direction in C++, such as the Parallel STL.

Fire 4 below is a simplified example of what a single-source C++ parallelism model might look like:

```
Vector<float> a, b, r;
parallel_for (a.range (), [&](int id)
{
    r [id] = a [id] + b [id];
});
```

Figure 4. Example of SYCL single source

3.2 How Parallelism is Exposed

One of the most important areas in this design space is how massive parallelism is exposed in code. This is important not just from an esthetic point of view, but more importantly what is acceptable as elegant for the C++ community, and is in line with the future direction of the upcoming Technical Specifications.

3.2.1 Directive-based Parallelism

This is the easiest way to expose parallelism to the compiler without disturbing the base language and is the approach taken by OpenMP and OpenACC. This is achieved in C/C++ by annotating code with pragmas (or in the case of Fortran by hijacking the Comment) to state where parallelism exists. Its attractiveness is that it can add to the base language without disturbing it, and gives a uniform appearance to all the base languages (in the case of OpenMP or OpenACC which works on C, C++, and Fortran). This is ideal in HPC where code is often formed from multiple legacy kernels (written in Fortran, C, and C++ for example) in different languages. It also enables easy incremental parallelism by adding parallelism to hot regions in steps, as opposed to planning parallelism from the ground up. It is simple to understand.

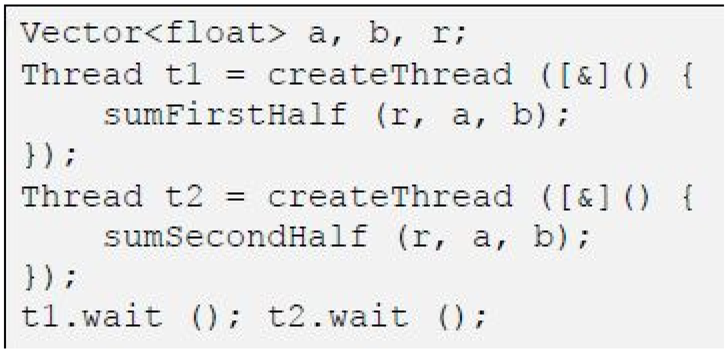
```
Vector<float> a, b, r;
for (int i=0; i< a.size(); i++)
{
#pragma parallel_for
    r [i] = a [i] + b [i];
}
```

Figure 5: OpenMP C++ code using pragmas.

Its drawback is that it fits poorly with C++, especially with templates, is difficult to compose, and the outlining that occurs for parallel regions leads to execution that is out of order with the source code. An erroneous annotation can make code behave incorrectly or unpredictably and compilers have a hard time associating error regions back to the pragma leading to weak error messages.The major drawback for C++ is that pragmas are considered inelegant.

3.2.2 Thread-based Parallelism

Explicit thread-based parallelism is required for situations where an application must respond to asynchronous operations, such as a web server, but can also be used for acceleration parallelism. Examples of libraries that provide this approach are the C++11 threads, pthreads, boost.thread, or Threaded Building Blocks. They create explicit threads to break up tasks into parallel sections. However, they assume an architecture which supports threads with shared memory and independent forward progress, which is not a capability of many highly parallel accelerators.



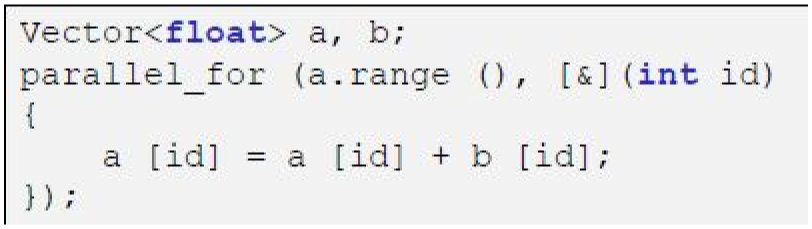
```
Vector<float> a, b, r;
Thread t1 = createThread ([&]() {
    sumFirstHalf (r, a, b);
});
Thread t2 = createThread ([&]() {
    sumSecondHalf (r, a, b);
});
t1.wait (); t2.wait ();
```

Figure 6: Example of Thread-based parallelism.

Threads can form the basic building blocks for more advanced parallel idioms, because they are well understood, and can work with a variety of algorithms. However, requiring full thread support for an accelerator model would make major restrictions on the types of accelerator that can be supported. For example, vertical vectorization, of the type used in most GPGPU programming models like OpenCL, blocks the independent forward progress guarantees required in threading systems.

3.3.3 Explicit Parallelism

Parallelism can be expressed explicitly in a program in a way that is a natural fit with the C++ language. This is the basis on which the C++ Parallelism TS is formed, and is also the approach SYCL and C++ AMP follow. A highly simplified form of C++ explicit parallelism is shown below:



```
Vector<float> a, b;
parallel_for (a.range (), [&](int id)
{
    a [id] = a [id] + b [id];
});
```

Figure 7: A SYCL parallel_for

The user does need to know where the parallelism is and what form it takes. The benefit is that it enables working with a wide variety of architectures and so is ideal for heterogeneous devices. We also believe that this approach is highly composable and easy to debug. In explicit parallelism, the parallelism in the source code is the same as that when executed.

Different devices have different performance characteristics and parallel execution models. OpenCL exposes this via queues, work groups, optional sub-groups and work items. Explicit parallelism enables users to express the full complexity of their parallelism at a reasonably high level and then allow that to be mapped to different devices. In particular, it allows developers to express their own parallelism and then map it to devices, such as that shown by Parallel STL from Parallelism TS.

We chose Explicit Parallelism for SYCL because it is inline with the Parallelism TS and believe it alleviates the compiler of the burden while giving the code clarity and the elegance of expression of C++ while working with templates and future expressions of the Concepts TS. It also enables developers to write algorithms that take full advantage of the explicitly parallel execution model of OpenCL.

3.4 Memory Model

This section describes the evolving technology of memory architecture and hierarchy, and how it affects the design of massive parallelism languages. It is one of the most critical areas of design for Accelerators to ensure high performance. The access to data has a huge performance impact in modern systems and becomes a greater issue as the level of parallelism scales up, or power consumption is scaled down. It is one of the central question that should be answered for C++ to ensure it works for all domains. We describe three general memory models representing existing and future memory hierarchy.

3.4.1 Cache coherent single virtual address space

In multi-core CPUs, HSA, OpenCL 2.0 System Sharing mode, this fully-coherent memory model enables data to be shared just by passing pointers around, which is therefore very low latency and is easy to express.

```
float *a = new float [size];
processCodeOnDevice (a, size);
```

Figure 8: Pointer passing in a cache coherent single virtual address space

In this mode, all memory accesses go through the virtual memory system and caches communicate ownership across all cores. This makes communication and offloading very low-cost with very little impact on the programming model. It is however, bandwidth limited, and requires special OS support and thus costs power. For SYCL, we wanted to target a wide range of devices and operating systems, as well as achieving very high performance and so requiring a single address space or cache coherency was not a design decision we could require.

For Standard C++ to target highly parallel and power efficient architectures, it needs to consider that a cache-coherent single-address space with virtual memory has a cost that is not acceptable in a wide range of domains. We believe that alternative memory models should be supported in standard C++ as well.

What a fully cache-coherent, single virtual address space allows is separation between references to data (i.e. a pointer) and ownership of data (e.g. a mutex). This is well-understood as a problem, but not easy to solve. Often, combining references to the data with ownership of that data is a better solution as it allows abstraction of the complications of data ownership as well as performance. This is the approach we take in SYCL.

Security concern is another issue with a single virtual address space. This is because an operating system must give any accelerator access to any data within a process's address space on demand. This requires ensuring that code running on an accelerator be secure in its data access according the source of that code.

Since this memory model is an optional feature of OpenCL 2.0, it is likely that a future SYCL version will support this memory model on hardware with this capability.

There are strong benefits to this approach, as well as genuine concerns and costs. But it does seem this is the future memory model. This will likely remain a continued area of significant debate within SG14 and SG1 of the C++ Standard.

3.4.2 Non cache coherent single-address space

There are a variety of trade-offs between totally separate memory and totally coherent memory. One useful tradeoff is to have a single address space, but with some level of user management of sharing. In this case, all data is still referred to via shared pointers, but the user must manage the memory ownership between different cores by passing ownership to device. Race-free software requires control and transfer of ownership , so this is not necessarily a cost to the user.

```
float *a = NewShared<float> (size);
a.passOwnershipToDevice (size);
processCodeOnDevice (a, size);
```

Figure 9: A simplified example of what this type of memory model might look like with ownership transfer.

This is supported to varying levels by OpenCL 2.x core and HSA Coarse Grained. C++ already enables users to manage ownership and it benefits from not requiring much OS or hardware support as it assumes non cache-coherent memory.

The advantage of this memory model is that users can still pass around pointers and separately control access to the data. This model actually has a range of different capabilities and tradeoffs that impact programmability, performance and power consumption. Examples tradeoffs include: do atomics work between different devices on pointers? What memory can be shared: all host memory, or only specially-allocated memory? And, are there restrictions on when ownership can be transferred, such as can ownership be transferred during the execution of a kernel or only after a kernel has completed?

Because this is a required feature of OpenCL 2.x, a future version of SYCL is expected to support this memory model. But there are still a variety of capabilities that vary by device and platform that need to be handled. Specifying this is an issue.

3.4.3 Multiple-address spaces

In this model, which has been historically common with discrete GPUs, data cannot be universally referenced by a single pointer but must instead be encapsulated using a combination of a normal C style pointer and a separate address space which is part of the type. As such, data needs to be encapsulated in new datatypes that are able to manage both physical location as well as ownership between host CPU and different devices.

```
Shared<float> a (size);
processCodeOnDevice (a);
```

Figure 10: A very simplified view of what this addressing model looks like with passing explicitly to device (note no pointers in this code as we use classes instead to abstract references to data):

This is how C++ AMP, OpenCL 1.x, OpenAcc, OpenMP 4.x and SYCL 1.2 can offer high performance and very efficient memory access with wide device support, often by just installing the right driver. It has a negative impact on the programming model in the form of how pointers are specially treated.

The embedded C standard shows how cv-qualifiers can be extended to support address spaces as part of a pointer type. This is also a C TR 18037, a proposed TR for the next C Standard. This is the approach in OpenCL C, but in SYCL we decided this was a language extension and so did not follow this approach. Instead, in SYCL, we define pointer classes (such as `global_ptr<`*T*`>`) which contain an address space within the type and a pointer value in the class data. Pointer classes of different address spaces may be different sizes and are not convertible.

One problem in this model is the handling of code which requires pointers. This is common in C++, such as the ‘`this`’ pointer for member functions. In SYCL, we support a defined form of type inference which enables member functions to be defined and called on objects in different address spaces.

The problem of address spaces in accessing data is combined with the problem of specifying ownership of data, along with the problem of defining a race-free schedule, along with the problem of efficient movement of data, and combined into a single solution: the ‘buffer’ and ‘accessor’ abstraction. This enables us to make just one requirement on the programmer: encapsulate data that will be accessed in parallel and then define the access to that data. The SYCL system can then find an efficient race-free schedule, provide high performance access to data and compile for a wide range of devices.

4. SYCL and OpenCL Overview

A requirement of SYCL is that it must be easy to write high-performance OpenCL code in C++. This means SYCL code in C++ must use memory and execute kernels efficiently while providing developers with all the optimization options from OpenCL.

SYCL also enables all OpenCL features in C++. These include support for wide range of OpenCL devices, high performance in data movement and access, as well as efficient data placement on host and device. We must handle OpenCL parallelism through the support of OpenCL ND ranges which are broken up into work-groups, and work-items with other entities such as barriers, queues, and events.

Example SYCL source code:

```
#include <CL/sycl.hpp>

void func (float *array_a, float *array_b,
           float *array_c, float *array_r, size_t count)
{
    buffer<float, 1> buf_a(array_a, range<1>(count));
    buffer<float, 1> buf_b(array_b, range<1>(count));
    buffer<float, 1> buf_c(array_c, range<1>(count));
    buffer<float, 1> buf_r(array_r, range<1>(count));
```

```
        queue myQueue (gpu_selector);

        myQueue.submit([&](handler& cgh)
        {
            auto a = buf_a.get_access<access::read>(cgh);
            auto b = buf_b.get_access<access::read>(cgh);
            auto c = buf_c.get_access<access::read>(cgh);
            auto r = buf_r.get_access<access::write>(cgh);
            cgh.parallel_for<class three_way_add>(count, [=](id<1> i)
            {
                r[i] = a[i] + b[i] + c[i];
            });
        });
    }
```

4.1 Data Management in SYCL

In order to access data in OpenCL1.2, one must encapsulate data in a buffer or image object that is bound to an OpenCL ‘context’, which in turn is bound to one or more OpenCL devices. Access to that data in OpenCL C requires creating a kernel with pointer parameters that are global or constant, or via image ‘samplers’.

In SYCL, we have C++ `buffer` and `image` classes, which abstract away the OpenCL buffer or image objects underneath. There are no methods on buffers or images that give direct access to the underlying data of the buffer or image classes, allowing the runtime to create one or more efficient data storage objects to hold the data. The only way to access data on buffers or images is via *accessors*, which are templated by the mode of access. This approach has several benefits:

- It allows the runtime to determine a race-free and efficient schedule for both the execution of the kernels and also any data movement that may be required.
- Data movement and kernel execution can be scheduled to run in parallel where possible. The task-graph built up using accessors can determine where data needs to be moved or copied from and to in advance of a kernel being executed.
- It allows users to specify the kind of access to data that matches the best performance on their device for a specific kernel.
- It allows fully asynchronous operation, so that kernels are executing independently of the host until a user specifically requests access to data on the host.
- Different kernels and devices can access the same data in ways that provide the best performance and features for the specific kernel and device.

The lifetimes of buffers, images and accessors are defined via normal C++ RAII semantics. This makes the programming model natural and easy to use for C++ developers.

4.2 Parallelism in SYCL

In OpenCL, there are a series of levels of parallelism:

1. The host CPU executes code in serial (but can be multi-threaded) and can create OpenCL objects and enqueue work to devices.
2. There are multiple platforms, devices, from which contexts can be created.
3. Data can be moved, copied or mapped asynchronously or on-demand between host and individual devices
4. Work can be enqueued to queues.
5. Queues contain kernels which are executed over an *nD-range*
6. An nD-range is divided into a number of work-groups, which can be executed in parallel or serial
7. Each work-group can be optionally divided into sub-groups (this is an OpenCL 2.x feature only)
8. Work-groups or sub-groups are divided into work-items, which are defined to execute in parallel. Synchronization within sub-groups and work-groups can occur via barriers.

This example SYCL code demonstrates the different levels of parallelism in SYCL v1.2:

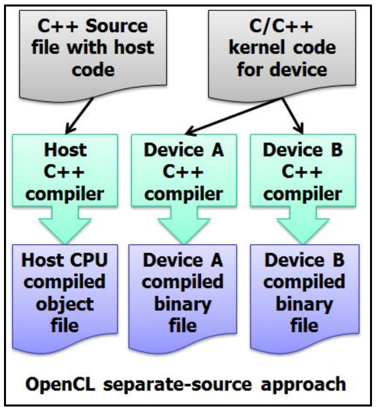
```
buffer<int> my_buffer(data, range<1>(10));

q.submit([&](handler& cgh)
{
    auto in_access = my_buffer.get_access<cl::sycl::access:read>(cgh);
    auto out_access = my_buffer.get_access<cl::sycl::access:write>(cgh);
    cgh.parallel_for_workgroup<class hierarchical>
       (range<1>(group_size), range<1> (local_size), [=](group<1> grp)
    {
        parallel_for_workitem(grp, [=](item<1> tile)
        {
            out_access[tile] = in_access[tile] * 2;
        });
    }));
});
```

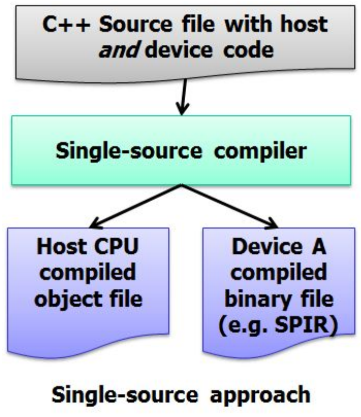
At the outer level, the example submits work to a queue, which will be attached to a specific device. The 'work' is called a 'command group' in SYCL and is defined via the `handler` object passed to the command-group lambda. The command-group is added atomically to the queue once the C++ lambda returns. Within the command-group, the accessors are created which defines the data access of the command-group for determining a correct and efficient schedule. Then, there is a `parallel_for_workgroup`, which is a parallel iteration over all of the work-groups in the *n*D-range. Inside the workgroup is a `parallel_for_workitem`, which is a parallel iteration over all the work-items in the work group. In OpenCL terms, all code at the workgroup level is compiled to execute once per work group and all code at the `workitem` level is compiled to execute once per OpenCL work item. This execution model can be achieved via looping, or vectorization, or some other parallelization technique. For OpenCL, this is expected to use a compiler transformation which adds OpenCL barriers between different `parallel_for_workitem` and adds predication to all code at the `parallel_for_workgroup` scope to execute once per workgroup.

The advantage of this hierarchical parallelism approach is it matches the underlying hardware capabilities, while enabling users to provide some level of performance portability.

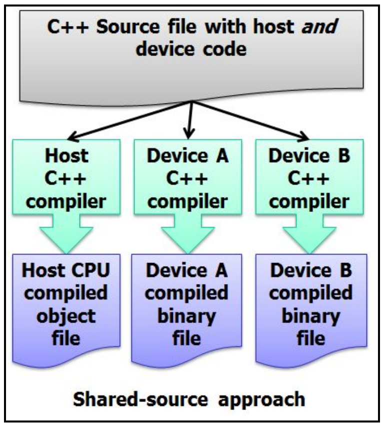
4.3 Shared Source in SYCL


C++ Source file with host code
C/C++ kernel code for device
Host C++ compiler
Device A C++ compiler
Device B C++ compiler
Host CPU compiled object file
Device A compiled binary file
Device B compiled binary file
OpenCL separate-source approach

OpenCL targets a wide range of processors, each of which will have different compilers. OpenCL v1.2 core deals with this by having separate source code for the kernels and runtime compilation of the kernel source code for the device the kernel will run on as shown in the diagram on the right. This separate-source approach does not work for the single-source C++ programming model we wanted for SYCL.


C++ Source file with host *and* device code
Single-source compiler
Host CPU compiled object file
Device A compiled binary file (e.g. SPIR)
Single-source approach

Another possible solution to single-source is the (optional) OpenCL 1.2 feature: SPIR, and the new improved SPIR-V for OpenCL v2.1 (which is a required feature of OpenCL 2.1). This would enable a single compiler to compile the host (CPU) code, as well as output SPIR (or SPIR-V) binary code for devices (see diagram to the left). However, this SPIR has a 32-bit and a 64-bit form, so a single-source compiler might have to output both 32-bit and 64-bit SPIR code as well as host CPU code. And the SPIR solution does not work for devices that do not support SPIR, although this is likely to become less of an issue as vendors move to OpenCL 2.1.


C++ Source file with host *and* device code
Host C++ compiler
Device A C++ compiler
Device B C++ compiler
Host CPU compiled object file
Device A compiled binary file
Device B compiled binary file
Shared-source approach

For the SYCL spec, we enable all the single-source solutions above, but we also enable another potential approach we call “shared source”. In the shared source approach (see diagram to the right), the same source code is compiled by multiple compilers. A CPU host compiler outputs the CPU code which will call the host runtime for SYCL.One or more SYCL device compilers will extract the kernel code and compile for different devices. This approach has a range of benefits:

- Separate compilers can be optimized for different architectures. For example, the host code can be compiled with OpenMP and parallelized.

- Users can use templates or macros to customize the same source code for different devices.
- Multiple device binaries (such as 32-bit and 64-bit SPIR) can be compiled for the same source code.

However, one problem we face is that if the kernel is a C++11 lambda, then there is no way to link between the host and device compiled code. For functors, this isn't a problem, only for anonymous lambda functions. For C++11 lambda kernels, we require a typename to name the kernel. This is only for linking between different compilers for host and device(s).

5. Impact on the C++ Standard

Here are some features we would like to see in a future C++ standard that could benefit acceleration. The following are very high-level ideas.

1. The main problem with the SYCL shared-source approach is the requirement of naming lambdas to provide kernel names. Using attributes would help on a device compiler that generates SPIR binaries, but there is no way for the host compiler to understand which lambda corresponds to which device binary. The current SYCL specification uses a type declaration to name the lambda in both the device and the host compiler, enabling the SYCL runtime to match the user-lambda with the device binary. However this is not ideal, as it forces users to name every single lambda, making problematic the usage of SYCL on deeply templated libraries or in generators. Although some workarounds could be implemented, e.g using the __*COUNTER*__ macro, they are not desirable. Some potential solutions from the point of view of the C++ standard:
    a. Standardize the naming of lambdas
    b. Provide a standardized type reflection for lambdas that can enable the same lambda compiled with different compilers to have the same reflection information.
    c. Use an attribute for the kernel naming, but having reflection support from the C++ standard to read that attribute at runtime
    d. Having a way of generate types that is common across multiple compilers
    e. Having a mechanism to count the number of times a template has been instantiated (not guaranteed to be reliable)
2. Executors can facilitate writing SYCL code assuming they allow a hierarchical execution model that is composable: Some algorithms would like to use work-group/work-item, others just work-item, others will need as well sub-groups. It is not feasible to assume that executors will spawn threads that can do all sort of operations (like deleting pointers, using thread local storage or doing system calls). Executors should be capable of spawning threads that have some limited functionality similar to that proposed in Light-Weight Execution Agents[1] and Integrating Executors with Parallel Algorithm Execution [2].
3. Futures, promises and continuation can allow multiple device-tasks (command-groups in SYCL) to synchronize and be executed in an orderly fashion. However, note that current

SYCL specification relies on “queues” to submit command groups (device-task) but also uses memory dependency checking to ensure data running in different devices/queues has the correct visibility of data. This is not a big problem when there is a unified view of the memory and a single device, but when having multiple devices from multiple vendors (e.g, a CPU, a GPU and a FPGA), synchronization of data needs to be done to ensure all components view the same information. Futures/continuation allows users to express the order of execution they want but the runtime must be aware of data dependencies in tasks to ensure the information is provided in order.

4. The cost of data movement can often be higher than computation in a highly-parallel system. In SYCL, data movement is abstracted via the buffer/accessor abstraction, which exposes some of this requirement to the user, aiding in optimization and understanding of performance. SYCL tries to minimize the effort required by users, but not totally hide essential performance information. In the case of futures, promises and continuations, this data-movement-cost should be considered, along with the different ways of minimizing the data-movement-cost.
5. Also future/promises allows the synchronization of tasks that start and end. However, some devices (e.g. FPGAs) will spawn tasks that will not end until the program finish, and will communicate with the main program receiving and sending data. This cannot, by definition, be expressed by continuation semantics (.then), and requires some concurrent communication mechanism (e.g, a pipeline).

6. Using SYCL to implement C++17 features

It is possible to use SYCL today to implement features from the upcoming C++17 standard. The SYCL working group has been working on an implementation of the Parallelism TS [3] that allows STL algorithms to be executed on a wide range of heterogeneous platforms just by using SYCL.

The development is public on the Khronos Group github [4].

The project implements a “sycl_execution_policy” that can be passed to the Parallel STL algorithms and calls sycl-implementations of those. The implementations are straightforward to read, customize or improve since they are pretty much C++ code using SYCL concepts. Vendors can produce their own optimized implementations of some algorithms by using either compile-time detection (via an specific vendor policy derived from the sycl one) or runtime detection (by checking the available OpenCL platforms and using an specific implementation of an algorithm).
This experimental project demonstrate how SYCL can be integrated on a C++ library without major modifications to the library itself, and by following a modern C++ coding style.

7. Goals and Conclusion

This paper shows the experience gained from a heterogeneous device programming model that is designed to work with C++ templates, that has been in production compilers for a consumer domain of a wide variety of devices.
It shows using the design considerations of a sample of programming languages how it is important to consider style of programming, how parallelism is exposed, and the all important data movement in the memory model.

It is our conclusion that C++ should aim to be single-sourced with explicit parallelism but that the memory model needs to be adaptable to all devices, as future heterogeneous devices may not necessarily converge to any one single memory model. This will remain a matter of debate within SG14/SG1.

We hope this will form the basis of a massively parallel model for C++ in SG14/SG1 that works across many domains and devices, making C++ the dominant language to express such parallelism.

8. Acknowledgement

We thank the Khronos group and specifically the SYCL working group that has worked to develop this model. We also like to thank the implementation offered by Codeplay and its staff.

9. References

[1] P0072R0 Light-Weight Execution Agents, T. Riegel ,
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0072r0.pdf
[2] N4406 Integrating Executors with Parallel Algorithm Execution, J. Hoberock et al,
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4406.pdf
[3] N4507 Technical Specification for C++ Extensions for Parallelism, (ed) J. Hoberock,
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4507.pdf
[4] Khronos SYCL Development github, https://github.com/KhronosGroup/SyclParallelSTL
[5] Khronos SYCL 1.2 Specification, https://www.khronos.org/registry/sycl/specs/sycl-1.2.pdf
[6] Khronos SYCL: https://www.khronos.org/sycl
[7] Khronos SPIR: https://www.khronos.org/spir
[8] Khronos SPIR-V: https://www.khronos.org/registry/spir-v/specs/1.0/SPIRV.pdf
[9] Khronos OpenCL: https://www.khronos.org/opencl/