Document number: P0363R0
Date: 2016-05-30
Project: Programming Language C++, SG14, EWG
Authors: Michael Wong, Andrew Richards, Maria Rovatsou, Ruyman Reyes, Lee Howes, Gordon Brown
Emails: michael,andrew,maria,ruyman,gordon@codeplay.com, lwh@fb.com
Reply to: michael@codeplay.com

Towards support for Heterogeneous Devices in C++ (Language aspects)

1. Introduction
2. SYCL features to support heterogeneous computing
3. Terminology
4. Lambdas and Reflection
5. Conclusion
6. Acknowledgement
7. References

1. Introduction

This is a paper to continue the specific features and needs towards supporting Heterogeneous Devices which was discussed in an evening session at Jacksonville 2015 [p0236r0]. In that evening session, Michael Wong presented the motivation to support Heterogeneous devices and how it has been done in OpenMP, followed by a plan to have this support in C++ by 2020. This was followed by showing two C++-specific designs that supported this direction. Hartmut Kaiser presented the HPX design which caters more to a high-performance computing viewpoint. Lee Howes presented the Khronos SYCL/OpenCL design which caters more to a consumer device viewpoint. The discussion that followed, indicated enthusiastic support to move C++ towards full support for Heterogeneous computing by 2020, likely through an initial TS.

C++ is in fact not that far away from it. The enablement of SYCL and HPX demonstrates how this can be done with only a few medium-sized additions. But it does open the door for C++ to become the dominant language that can support all the MIPS in your system, rather than forcing

you to drop out to some low-level or proprietary language to gain access to the highest MIPS. This is traditionally what we had to do to access SIMD or GPU devices.

C++ is also well positioned to fully integrate the HPC and consumer device market with a single programming model, rather than the separate and diverse programming models in use today, where OpenMP/MPI dominates HPC and OpenCL and other models dominate consumer devices.

At the evening session, there was feedback guidance to focus on consumer devices such as mobile devices. There was comment that some DOE labs would favor switching completely to C++-only code instead of any OpenMP/OpenACC code at all in the future. There was also concern that FPGAs may be too-far reaching a target market. Overall, it was felt that if we have a focused approach with a goal, C++ would only be about 20% away as it is already nearly 80% on target to support massive dispatch on heterogeneous devices.

This paper aims to discuss what is needed for language support in the form of lambdas and reflections.. A separate paper [P0362R0] will comment on current directions in concurrency aspects with regards to executors, simd, accessors, and address space, from the point of view of the SYCL experience in implementing support for heterogeneous devices. A separate paper [P0361R0] will discuss the HPX guidance and experience. There will be many surprising parallels between the two groups' approaches from which SG14 plans to abstract a common design.

2. SYCL features to support heterogeneous computing

SYCL grew out of demand to better support C++ in OpenCL (which has been traditionally based on C99). OpenCL uses a programming model with separate host and device code. Recent efforts to add C++ support directly to OpenCL still maintain this separation. But the weak link between host and device code is error prone, due to a weak connection between variables and lack of inter-device compiler inference and error checking. These problems can be reduced via generation of type-safe callee stubs and code generation scripts, but that is unnecessarily complex, and probably contrary to C++'s design as a single-source programming model. There is definite desire to improve the interface by making it strongly-typed, and that resulted in SYCL which is much closer to C++ without adding any new language support. C++ is also already moving towards fundamental support for parallelism and concurrency, though mostly from a homogeneous non-numa architecture point of view.

As such SYCL was designed as a high-level C++ abstraction that still gives full access to the lower-level OpenCL operations, without changing standard C++ with any language extensions.

The resulting design enables C++ source to contain functions that can be called on both host and device and allows support for template algorithms. It leverages the well-defined C++ memory model, and the emerging parallelism and concurrency technical specifications. The result is that SYCL provides a far better match for programmers transforming from Standard C++ to heterogeneous programming.

Although SYCL has been designed as a high-level programming language from the point of view of OpenCL/device developers, it is actually a perfect fit for composable template libraries. The basic SYCL objects are templated by type, and all objects are reference counted - avoiding excessive overhead with copy semantics, whilst not affecting move semantic behaviour.

In a typical SYCL program, host code and device code are in the same source (i.e., single-source approach). This is a novelty w.r.t other heterogeneous programming languages such as OpenCL (where the device program is loaded from a runtime string).
This is common also in directive-based approaches (OpenMP >=4 when heterogeneous directives were introduced, OpenACC), where the code is shared between host and accelerators.

Data structures are created in the host code. Programming constructs (directives in OpenMP/OpenACC or plain C++ objects in SYCL) define the boundary between the host and device interfaces. Type-checking is supported at the boundary interface between the host-created buffer, and the usage by the device kernel.

In SYCL, The build process enables both a cpu and a device compiler. The cpu compiler generates the cpu object code, while the device compiler can generate SPIR or binary format which can both be combined with the cpu object code at link-time to create an executable object. The SYCL runtime chooses the best binary for the device at runtime.

As everything is still Standard C++, the host compiler path can act as an efficient fallback path, when there is no device on the system. The triSYCL implementation is one such example which supports only the library side without the device compiler.

The host code defaults to simplify object construction such as creating a default queue to enqueue work on a default device/context, then wrap the data in a type-safe buffer. Developers can provide more detailed information, such as defining classes for constructing a device, building a custom-selector type to pick specific device features.

In the device code, it is designed to match the Parallelism and Concurrency TS in C++. There is a well-defined scope and the ordering of parallel operations is traced through standard C++ memory model, to enable RAII well-defined lifetime for buffers and access.

SYCL introduces the idea of Accessors to acquire access to a host buffer and the access falls out of scope at the end of the queue entry. One can define for the duration of the access to be

read, write or both. The access also encodes asynchronous task dependencies based on read and write access of a particular buffer at a particular time. By the time the host code finishes that block, it has constructed all its tasks and dependencies and placed it on the underlying OpenCL queue with all the event dependencies constructed effectively transparently.

The execution is in parallel through an SPMD (Single Program Multiple Data) execution model using a parallel_for operation. This operation also prerequisites an iteration space which the number of processing elements in a space of dimensionality from one to three. The parallel function also includes the style of access, and passes a runnable function object or lambda that represents the function to be executed at each point. The execution could also be SIMD execution where the iteration space is defined as unary but SIMD vector operations could be executed.

The use of lambdas which do not have well-defined names in C++ within source that is parsed by different compilers (host and device) forces SYCL to require a name assigned to the lambda by the user. That name is used for matching the lambda between the host and device compiler. While this issue is common in both SYCL and HPX, and will be discussed in detail below with some idea of how to resolve it in C++.

In summary, SYCL enables generic heterogeneous computing model with

- Command group functor which encapsulates a collection of memory and kernel execution commands and is scheduled atomically
- queues class that waits for all commands to complete in its destructor and express where computation occurs
- parallel_for, single_task, parallel_for_workgroup that launch computations operating on one or multiple processing elements depending on whether we have a SIMD or SPMD execution model. We could also have a hierarchical execution (the various hierarchical parallelism concepts defined in the SYCL standard)
- accessor that defines the way and where we access data, which with the command groups it forms a task dependency model
- buffer/image for storing data
- allocators for defining how host data is allocated.

The remainder of this paper will describe the motivation for some of these features, and how SYCL has tried to solve them, with some suggestion and guidance on how they can be similarly solved in C++.

3. Terminology

There has already been significant experience with many forms of compilation when device and host compiler code needs to be mixed together. This section outlines some of these differences

based on our discussion in Section 3 of P0236R0 Khronos's OpenCL SYCL to support Heterogeneous Devices for C++.

Host compiler / device compiler

Heterogeneous support today often requires a separate host and device compiler compiling over either one combined source (single-source) or separate source.

Separate Source

This 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 example 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 1: 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.

C++ is a single source compilation model. That is all code targeted to either host, or whatever special device (DSP, hardware transactional memory, accelerators) are combined in a single source. It is unlikely that it will become separate source as that would require some significant changes to C++ design. However, it is instructive to see what are the interesting problems that can be presented when compiling single source.

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.

Figure 2 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 2. Example of SYCL single source

Single-compiler / multiple-compiler approach

Even when you have a single-source model for heterogeneous dispatch, that source is still compiled separately for the CPU and and the heterogeneous devices. This is done either via a single-compiler approach where a single compiler compiles both the CPU code and the heterogeneous device code or a multiple-compiler approach where separate compilers are used to compile the CPU code and the heterogeneous device code.

C++ currently imposes a singler-compiler compilation model, however looking forward towards heterogeneous dispatch there is a great deal of motivation that suggests it should move towards a multiple-compiler compilation model. Here we present the motivations behind a multiple-compiler compilation model in C++ and in later section we look at some of the issues that would arise as a result and how they could be overcome.

- A single-compiler compilation model for heterogeneous dispatch is only possible in the cases where either in the rare situation where the entire ecosystem is controlled such as the case with CUDA or all devices within the ecosystem can be managed via a common standard such as SYCL.
- Hardware vendors will continue to independently develop new devices that will want to have adopted into existing systems and tool chains at a later stage. By supporting a multiple-compiler compilation model, those hardware vendors can simply provide a new compiler for their new architecture rather than having to either integrate their compiler into an existing compiler and tool chain or add the addition existing CPU options, optimizations or builtins from tool chains they are seeking to be adopted by.
- Systems vendors will want to compose systems that leverage these new architectures. By supporting a multiple-compiler compilation model, those system manufacturers would be able to compose systems with devices from different vendors and be able to compose a tool chain with the individual compilers for each device rather than having to implement an entirely new compiler and tool chain in order support their system.
- Library and applications developers will want to be able to continue using their existing tool chains and compilers when adapting to addition of new architectures. By supporting a multiple-compiler compilation model, those developers will be able to continue using the existing tool chain with only the addition of the additional compiler rather than potentially having to adapt to an entirely new tool chain.
- As CPU architectures and heterogeneous architectures have such different requirements in terms of execution and memory models they will have very different back-ends, optimization passes and potentially even front-end semantics depending on the C++ std library implementation details. This means that in the majority of cases the source code that is compiled by a single compiler is actually still passed through two completely different compiler invocations, and is in fact only giving the illusion of a single-compiler compilation model by hiding the compiler inter-op in the compiler driver.

In the more common case, where you have separate compilers for host and device, it is more amenable to innovation and usage of Open Source contributions such as clang, and gcc, which can supply either the host or the device part.

It is also worth mentioning that, when dealing with heterogeneous platforms, it is not always the same vendor that controls the entire toolchain or environment. It is common to have multiple vendors collaborating with their toolchains in the system, and they are often delivered independently and have to be composed together later in the system integration phase.
This makes unfeasible the possibility of sharing a common compiler driver written in the same toolchain (e.g, a common clang compiler driver) since the system integrator does not control the different components.

4. Lambdas and Reflection

These four sections are transported verbatim from P0362R0 for EWG consideration, but is left in tact here for completeness.

Background

Lambda functions have become an important component of parallel programming approaches, since they serve as a mechanism for both anonymous functors, and convenient mechanism for kernel parallel dispatch.

For single-source programming models it has become the defacto standard to represent device functions that are to be offloaded to a heterogeneous device as a lambda or functor object, generally having the capture variables or fields respectively be the the arguments to the device function.

In the case where the source is shared across two independent C++ compilers; this introduces the requirement for lambdas to have a common representation across different C++ compilers in order to support heterogeneous dispatch.

These requirements do not extend to the case where the source is compiled twice by the same C++ compiler (such as CUDA or OpenMP) as most of the issues encountered can be resolved internally within the compiler.

The heterogeneous dispatch in the case of multiple-compiler solutions also involves a step where the lambda will be dispatched by a heterogeneous runtime library on a device. In order to do that it will also need a unique identifier for the lambda.

Naming

The first issue is that the current standard does not require lambdas to have a common name mangling, which means that two C++ compilers are free to mangle the lambda differently. This can cause a problem for heterogeneous dispatch because there is no way for two compilers to make the link between a lambda which the host compiler and the device compiler sees, for it to be able to be dispatched on device side.

The current solution for this problem in SYCL is to have a template parameter attached to all heterogeneous dispatch apis provided which specifies a common name for the lambda among the different C++ compilers that are compiling it.

```
cgh.parallel_for<class device_func>(ndRange, [=](nd_item<2>
itemID){
 /* device function */
});
```

There are a few potential solutions to this:

1. One option would be to force a common name mangling of lambdas in the ABI, however this is a drastic change to the standard.
2. Another option is to specify the lambda name using a generalized attribute, however, this still requires the user to specify the name of the lambda.
3. Another option is to extend static reflection (P0194R0) to (according to the current proposal) require the the *Meta-Class* object in the case of lambdas to always derive from *Meta-Named* and have a standard naming that can be returned from *get_name()*.

Option 3 looks like the best solution as firstly it is the least invasive as it is solely a library feature allowing the language details to be implementation defined, without any ABI changes. It removes the requirement for a user specified attribute or name with an extension to an existing proposal.

Moreover, options 1 and 2 both provide the name of the lambda as information to the compiler, either via the ABI or are part of the lambdas type. This information, however, is not enough as it only addresses the issue between the mentioned compilers. There is still the problem of dispatching the lambda via some implementation specific run-time api, which still needs a form of static reflection in order to retrieve the name of the lambda.

Data Member Representation

The second issue is that the current standard does not specify any guarantees for the ordering of the data member of lambdas. More specifically variables and references captured within a lambda closure can be declared in the resulting functor object in any order and can be of any access modifier. This means that the type of a lambda closure is not required to be a standard layout type.

This can cause a problem for heterogeneous dispatch because there is no guarantee that two distinct C++ compilers will have the same ordering of data members within a lambda, making it impossible to reply on a mapping of the lambda data members as the host CPU compiler see them and as the compiler of the heterogeneous device see them.

A solution to this would be to standardise the way in which variables and references captured within a lambda closure are declared in the resulting functor object. One way in which this could be done is by adding the following rules to lambda captures:

- All variables and references captured within a lambda closure are declared in the functor object in the order in which they are captured, with explicitly captures first then implicit captures.
- All variables and references captured within a lambda closure are declared in the functor object with the same access modifiers, whether that be private, protected or public.
- The functor object that is declared as a result of the lambda closure must be a standard layout type.

Iterating Over Data Members

The third issue is that as a lambda that is being used to represent a device function being dispatched to a heterogeneous device can have any arbitrary number of captures, it is necessary to identify the arguments of the device function in a generic way.

The ability to do this could be facilitated by way of static reflection (P0194R0), which is already making good progress towards this. There are a couple of points of the current proposal that are very useful for run-times using lambdas for heterogeneous dispatch.

The static reflection defined in the current proposal could be very useful for implementing such a run-time when iterating over the data members of a lambda in order to marshal data from the host to a device.

The current proposal includes the ability to retrieve private data members as well as a public data members using the *get_all_data_members()* function of the *Meta-Class*, this is important as lambda captures are declared as private data members.

The proposal also specifies that the order of the *Meta-ObjectSequence* returned by *get_all_data_members()* will follow the same ordering as how the data members are laid out in the translation unit, this is also important, however this is affected by the issue raised in the previous section regarding ordering of captured variables within a lambda closure.

In heterogeneous architectures it is very common to have not only host and device compiler, but multiple device compilers matching the different devices on a system. Requiring only one compiler per system is limiting significantly the number of different devices in the system, which is not reflecting what the current systems require. It is common practise to require the host code, which is the CPU code, by a different device's compiler, however, that removes the ability to be able to compile the CPU code with the best CPU compiler for that architecture.

It is interest that HPX also sees the same issues when it tries to dispatch lambdas distributed computing. They also solve it using a user-supplied named lambda and has commented that introspection of lambdas with captures in the static reflection proposal is extremely important.

5. Conclusion

This paper aims to begin the conversation of the specific features we need to support for massive parallel dispatch for heterogeneous devices, integrated with the host. This paper describes a single-source, multiple compiler model but suggests adaptation for lambdas, while a separate paper aimed at SG14/SG1 discusses executors, task blocks, simd, and possible address spacing considerations.

6. Acknowledgement

We thank Khronos, OpenMP, OpenACC, MPI, HPX, HCC, and C++ AMP where many of this experience for delivering C++ on heterogeneous devices was learned.

7. References

[1] *Lee W. Howes, Anton Lokhmotov, Alastair F. Donaldson and Paul H.J. Kelly.* Proceedings of the 4th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC, AR: 28%) Paphos, Cyprus. *January, 2009.*

[P0362R0] Towards support for Heterogeneous Devices in C++ (Concurrency aspects)

[P0361R0] Invoking Algorithms Asynchronously