P00069r0 : HCC: A C++ Compiler For Heterogeneous Computing

Author: Ben Sander

Contact: ben.sander@amd.com

Author: Greg Stoner

Contact: greg.stoner@amd.com

Author: Siu-chi Chan

Contact: siuchi.chan@amd.com

Author: Wen-Heng (Jack) Chung

Contact: jack@multicorewareinc.com

Author: Robin Maffeo

Contact: robin.maffeo@amd.com

Audience: Concurrency Study Group

1 Introduction

Recent generations of GPU hardware have steadily improved the programmability, and are now able to support a rich set of C++ functionality. Additionally, the memory system for GPU systems is becoming more tightly integrated with the CPU. On many systems, the CPU and GPU are physically located on the same piece of silicon, share the same page tables, and have high-performance access to the same physical memory. Other systems have physically separate CPU and GPU, but use high-bandwidth, low-latency bus connections. These innovations in the memory architecture allow the GPU and CPU to share the same memory system and efficiently share pointers. More importantly, they eliminate the need for programmers to explicitly manage a separate pool of "accelerator" memory, and further allow GPUs to act as high-performance, high-bandwidth parallel acceleration engines using many of the same programming techniques that can target multi-core CPUs with wide vector units.

The C++ committee is already moving in a direction to embrace this opportunity, and proposals currently under discussion provide some of the functionality required for the vision of a single, portable compiler that can target all parallel devices. This paper describes the authors' experience developing a C++ compiler which uses several of the proposed C++ features (and some additional ones) to target a heterogeneous computing platform consisting of a host CPU and at least one GPU accelerator.

2 HCC (Heterogeneous C++ Compiler)

HCC is an open-source compiler based on the Clang/LLVM compiler infrastructure. A key feature of the compiler is that HCC generates code for both the host processor and the accelerator. This is in contrast to other approaches that use pre-processors to generate the accelerator code. Currently the compiler generates x86 host code and accelerator code for AMD GPU architectures via the HSAIL intermediate language [HSA]. Other micro-architectures could also be targeted using the same language syntax.

HCC supports multiple iterations of the C++ standard (i.e. C++11, C++14, and some C++17 proposed features). Additionally, HCC supports two dialects:

- C++AMP : Implements C++ AMP specification [C++AMP]
- HC : (*-hc*, "heterogeneous computing"). HC mode borrows many ideas from C++AMP, and adds new features to provide more control over asynchronous execution, memory management, and removes the need for the "restrict" keyword. The remainder of this document describes the capabilities of HC mode in more detail.

2.1 Marking Parallel Regions

N4354 (Technical Specification for C++ Extensions for Parallelism) defines a parallel interface to the Standard Template Library and represented a significant addition to the C++ standard. Benefits include:

- The proposal defines a standard syntax for programmers to mark parallel computations. The markers are used as an assertion that the marked section of code can execute in parallel (i.e. it does not contain cross-loop dependencies or memory aliasing). Also, they are an indication that the marked code may benefit from parallel execution (i.e. the computation is sufficiently large enough to overcome the overhead of initiating a parallel task).
- The STL specifies the requested behavior at the algorithm level, rather than at the loop level. This provides significant opportunities to optimize and utilize the hardware features available on the platform. As a specific example, many GPUs contain hardware features for efficient communication between groups of threads that can be used to optimize reduction, sort, and scan operations. GPUs may also prefer different memory accessing patterns when compared to a CPU architecture. The Parallel STL interface allows these architecture-specific optimizations to be embedded inside a standard parallel interface.
- Execution policies provide a flexible means to extend the developer's control over the parallel execution. The default execution policies allow programmers to specify serial execution or parallel execution. Implementations may extend the execution policy, and we expect HCC will define a custom executor which specifies the accelerator where the algorithm will execute.

In addition to Parallel STL, HCC also defines two other mechanisms to mark parallel regions for execution:

- hc::parallel_for_each(). The HC interface is similar to the parallel_for_each defined in the C++AMP specification. The important features of this interface include:
    - Provides an optional parameter that allows the user to specify a command queue. This can be useful to optimize performance in systems where more than one accelerator is available. (See later section on Asynchronous Execution Control)

- Returns a “completion_future”, which is derived from the std::future type. HC’s version adds additional features to provide more control and insight into asynchronous command execution (discussed later in this paper). HC also can record timestamp information (recording the start and stop times) in the completion future. This is a generally useful capability for performance analysis of asynchronous tasks, and is difficult to collect accurately without low-level runtime or language support.
- Provides control over hierarchical parallelism, and “group” dimensions. As discussed later, this remains an important optimization knob for GPU accelerators. This parameter is optional, and the runtime will select a sensible default if it is not explicitly specified.

- [[hc_grid_launch]]. hc_grid_launch is an attribute that indicates the functions will launch with an implicit “for_each” wrapper. An example:

```
[[ hc_grid_launch ]]
hc::completion_future
vector_copy(const hc::grid_launch_parm &lp, float *A,
float *B, int N)
{
    int gid = lp.gridIdx[0];
    if (gid<N) A[gid] = B[gid];
};


{
    float *A, *B; // initialization not shown.
    int N;

    hc::grid_launch_parm lp;
    lp.gridDim[0]=10000;
    lp.groupDim[0]=256;
    vector_copy(lp, A, B, 10000);
}
```

A “grid_launch_parm” structure passed as the first argument specifies the dimensions of the for-each wrapper. The HCC grid_launch_parm also specifies additional launch-time information including the target accelerator, group dimensions, and group memory allocations. Like hc::parallel_for_each, hc_grid_launch functions return an hc::completion_future that can be used to track completion and create dependencies.

The function call syntax provided by hc_grid_launch has proven popular with programmers who are perhaps less familiar with functors and C++11-style lambdas. Also, hc_grid_launch provides a syntax for the C11 dialect supported by HCC. hc_grid_launch functions are conveniently identified with the name of the function and can be manipulated as function pointers. This

capability can be useful for supporting function query interfaces (ie to query the registers or other resources used by the compiled form of the kernel, which can be important for GPU optimization).

The function syntax provides a clean, well-defined interface to the accelerator code region, allowing compiler tools to generate isolated blocks of accelerator code that can be saved and later executed on the correct accelerator.

2.2 Code Generation

One of the challenges that HCC must overcome is to determine which code to compile and optimize for the accelerator. A simple approach is to compile all functions for all architectures in the system. However, this can significantly increase compilation time, in particular when targeting heterogeneous computing architectures where one component specializes for serial execution and one for parallel execution.

The C++AMP specification defined a new "restrict" keyword which restricted the language features allowed in the code region and also identified the code to compile for the accelerator. However, in hindsight the "restrict" keyword has several downsides. Composing functions becomes tricky as the "restrict" keyword tends to ripple through the call hierarchy. Additionally, re-using existing code is non-trivial, even in cases where the functions are short and available in header files (e.g. std::min).

HCC employs a more sophisticated mechanism so that compiler effort focuses on the parallel loops, and only generates and optimizes accelerator code where required. As described in the previous section, HCC provides several mechanisms for programmers to mark parallel regions - specifically hc::parallel_for_each and [[hc_grid_launch]]. These markers indicate that the marked functions will execute on the accelerator. Likewise, any functions called by these marked functions execute on the accelerator. Thus the compiler can identify which functions will execute on the accelerator and, in cases where the functions are defined in header files, can automatically generate correct accelerator code without the need for additional function attributes.

HCC does define an "[[hc]]" function attribute which can be used to explicitly mark kernels which are only visible at link-time (for example, a function defined in another compilation unit which is referenced by an accelerated function). An alternative implementation could choose to generate accelerator code at link-time, when the compiler has more information on the call chains used by the accelerator.

2.3 Language Restrictions

As mentioned in the introduction, modern GPUs are now capable of running a nearly complete set of advanced C++ features, including:

- Indirect function calls, allowing virtual functions and function pointers
- Byte-addressable memory
- Support for basic data types (float, double, float16, 64-bit integers)
- Misaligned memory access
- Stacks, enabling function calls and recursion
- Goto, for the cases where you absolutely need it

The HCC compiler design employs the same Clang front-end for both host and accelerator targets and thus all "static" C++ features are fully and consistently supported on both targets. However, some functionality gaps may still exist between the C++ code that can run on the host and the C++ code that can run on the accelerator, due to either hardware or software limitations. A good example is try/catch block – while these can be supported by the hardware, the HCC compiler does not currently implement these. The HCC compiler will currently generate a compile-time error if an accelerator code region uses a feature that is not supported on the GPU. At this point we see only a narrow set of remaining features required for the GPU to support full C++ capability, and expect those to further shrink in the near future, and do not anticipate a need to define a restricted subset of the C++ language.

2.4 Asynchronous Execution Control

GPUs are designed to deliver massive throughput, requiring an efficient command queueing and scheduling system. Most GPUs support multiple memory-based queues of commands which are monitored and scheduled automatically by the GPU hardware. In some implementations, the queues are directly mapped into user-mode memory, allowing applications to enqueuer work directly to the GPU without transitioning through the system drivers. Commands include parallel computations, data movement commands (executed on dedicated data movement engines), dependency waiting and signaling, or configuration commands. The host CPU (or potentially another accelerator on some implementations) places commands into one of the queues monitored by the hardware, and the hardware executes the commands in the queues. Several commands can be queued well in advance of execution to minimize the communication latency between the accelerator and the control thread. The GPU hardware is capable of monitoring dependencies and scheduling tasks when dependencies have been resolved, and can also perform memory copy operations (i.e. between memory physically close to the host CPU and memory physically close to the accelerator). HCC extends existing C++ concepts to provide programmers access to these important architectural features:

Command Queues:

HCC provides runtime APIs to create and manage communication queues with GPU accelerators. Unlike traditional queues, these often have a fixed size. An implementation could use the fixed-sized queue until full, and then spill over to a larger, dynamically allocated memory to provide a virtual queue with size bounded by available memory. (Note HCC currently does not provide this spill-over support, and returns an error if the queue size is exceeded.)

Continuations:

HCC uses an "hc::completion_future" that is derived from the std::future type. hc::completion_future has been extended so that the C++ programmer can leverage the GPU's support for asynchronous commands with hardware-resolved dependencies. Specifically, completion_future adds a "then" member function which provides a mechanism to attach a continuation to the future. The continuation object is an asynchronous task that will execute when the continuation is ready. [N4501] proposes the addition of a "then" syntax to add standard support for continuations. HCC can use this proposed API unchanged to support both "host" continuations and "accelerator" continuations, defined as follows:

- Host continuations are CPU-side functions. HCC saves the function pointers on a list associated with the future, and then executes all the host continuations when the future becomes ready.
- Accelerator continuations refer to commands that can run directly on the accelerator hardware. As described above, these commands can include parallel execution functions, or data movement commands, or dependencies. Accelerator continuations are placed into an appropriate command queue when the "then" function is invoked, and will be automatically scheduled and executed by the GPU hardware when the dependent future becomes ready without further involvement from the host. Thus the accelerator continuations use a different flow than host continuations, but this occurs in the details of the implementation and requires no changes to the API proposed in the N4501 document.

2.5 Accelerator memory architecture

Accelerator memory systems are rapidly becoming more tightly integrated with the CPU host memory. Legacy accelerators provided separate address spaces and physically separate memories. The separate address space was particularly burdensome for programmers because pointers could not be shared across address spaces, and programmers had to explicitly copy between address spaces.

Already we are seeing systems where the CPU and accelerators share the same page tables (or have different page tables but use clever software to ensure they appear to support the same virtual address space). At the physical level, accelerators for many markets already tightly integrate the CPU and accelerator, and provide high-bandwidth access from all accelerators to a single pool of shared memory. Other markets, such as HPC, continue to value discrete accelerator devices, but these are also becoming more tightly integrated via high-speed interconnects. This is an active development area but we anticipate two significant architectural directions:

- "Shared Virtual Memory" – the CPU and all accelerators will use the same address space, and pointers can be freely shared between accelerators and CPUs.
- Accelerator memory is a cache – many accelerators are equipped with high-speed memory. In the future, this accelerator memory will be automatically managed as a cache of the single large memory heap. Hardware or software will automatically move frequently accessed blocks to the accelerator memory, and programmers will not have to explicitly manage data transfers to benefit from the high-bandwidth memory.

There remains some debate over the cost and benefits of CPU-style coherency on accelerators. Some accelerator systems already support coherency, and can run C++ code targeted for CPUs with little or no special modifications.

2.6 HCC Memory Management Interface

One of the challenges HCC must address is to provide a consistent memory interface as the hardware migrates to include the capabilities described in the previous section. HCC provides two mechanisms to manage memory:

2.6.1 Structured access:

hc::array and hc::array_view provide a combination of array indexing, data movement, and command synchronization functionality:

- These types define a contiguous array of memory, and provide convenient multi-dimensional array types.
- The runtime can easily detect data references on the accelerator or host CPU, and the runtime can automatically copy data where appropriate.
- Additionally, data references through these classes provide automatic synchronization – for example, ensuring that commands executing on the accelerator finish writing their outputs before the data is read on the CPU.

These APIs have proven to be very approachable for developers who can benefit from acceleration without managing the details of data movement and synchronization.

2.6.2 Pointer Access:

The architectural trends towards shared-virtual-memory enables programmers to use complex pointer-containing data structures, and programmers expect to be able to freely use pointers rather than funneling all memory accesses through array* classes. Thus, HCC also supports a pointer-based memory management scheme – similar to malloc/free or new/delete. The pointers exist in the shared-virtual-memory space, and programmers can use these pointers automatically. HCC also provides a data management API to optimize data placement – including prefetching, setting replacement priorities, and discarding data. This mode separates the synchronization controls from the data references – providing more control and power, but also placing more responsibility on the programmer to ensure that asynchronous tasks are correctly synchronized. The recent C++ standards are headed in this same directions – providing more language features for asynchronous tasks and synchronization. Programmers are becoming more comfortable with these requirements even for CPU programming, and we believe asynchronous accelerator tasks can utilize a similar model and programming interface as asynchronous CPU tasks.

2.7 Multi-dimensional Array Indexing and Layout Control

Many HPC applications use multi-dim arrays, and benefit from high-performance and flat memory layouts (without indirection through "arrays-of-pointers-to"). A programmer-friendly syntax for multi-dim array access that supports these objectives is an important contribution, and was one of the useful features in the array and array_view structures.

The "Polymorphic Multidimensional Array View" proposal [P0009] defines a syntax for efficient multi-dimensional array access, including flexible controls for layout, padding, and stride. The extensible layouts and associated tiling support appear to be a powerful optimization for programmers to optimize for the large accelerator caches described previously. We anticipate a future version HCC will support this proposal.

2.8 Hierarchical Parallelism

GPUs have historically provided hardware to accelerate communication between a "group" of hardware threads. This hardware goes by a variety of names including "shared", "local", and "group". Here we use the term "group". The hardware includes:

- Barrier-level synchronization for the threads in the group.
- High-performance cache memory which is shared by all threads in the group, and can be used for efficient data communication.

- Relaxed atomic operations.

These features continue to be an important performance feature for GPUs. The bandwidth of the group cache is typically 2X or more compared to other caches in the system, and the group-scope atomics can provide 10X speedup over equivalent global-scope atomics. Also, modern accelerators can have thousands of threads and thus the ability to synchronize hierarchically (first with a small group of threads, then synchronizing across the groups) can provide a significant performance uplift. We believe this hardware will continue to be useful for GPUs and have provided facilities in HCC so that programmers can access it. HCC's support includes:

- Additional parameter to the hc::parallel_for_each or hc_grid_launch calls to specify the size of the group, and the amount of group memory required by the function.
- An API call to enforce synchronization across all threads in the group. This functionality could likely use the "barrier" function described in N4501.
- Scoped atomic operations, which allow atomics at the global, accelerator, or group scope.

One of the benefits of group memories is that they allow programmers to identify frequently accessed memory and lock it into the cache hierarchy. Many CPU architectures also provide similar a cache-locking feature, which suggest that a feature to mark these frequently accessed data structures might be useful across a broader set of architectures. This feature could possibly use the existing attribute syntax added in C++11. Also, it might improve the usability of group memories by eliminating the need to remap data structures into a fixed index. Further investigation is required.

3 Summary

This paper described our experience in implementing a heterogeneous C++ compiler targeting high-performance GPU hardware. The HCC compiler benefits from several of the features already proposed to the C++ standard – including C++11 attributes, Parallel STL (N4354), futures and continuations (N4501), polymorphic multidimensional array view (P009), and barriers (also N4501). We expect GPU hardware and runtime will solve some historical challenges with GPU computing, so that GPUs can run a full set of C++ language features, and GPUs memory will appear to be a single coherent heap. HCC provides software controls to optimize data placement, but these can be implemented as a standalone library outside of the C++ standard. We do see opportunities to improve C++ via "scoped" atomic operations (possibly building on the proposed barrier implementations) and support for "asynchronous" flavors of the Parallel STL libraries which return std::future objects. Recording timestamp information for asynchronous task execution is another concept that has proven useful in GPU computing and may apply to other architectures as well. We may present these ideas as more formal proposals to the C++ group in the future.

4 Reference:

[C++AMP] C++ AMP 1.2 specification. http://download.microsoft.com/download/2/2/9/22972859-15C2-4D96-97AE-93344241D56C/CppAMPOpenSpecificationV12.pdf

[HSA] Heterogeneous System Architecture http://www.hsafoundation.com/standards/

[N4354] "Programming Languages – Technical Specification for C++ Extensions for Parallelism", http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4354.pdf

[N4501] – “Working Draft, Technical Specification for C++ Extensions for Concurrency”, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4501.html

[P0009] Polymorphic Multidimensional Array View