Proceedings of the Federated Conference on Computer Science and Information Systems pp. 697–700 DOI: 10.15439/2017F217 ISSN 2300-5963 ACSIS, Vol. 11

Towards Programmable Address Spaces

Andrew Gozillon[†] and Paul Keir[*]
University of the West of Scotland
High St., Paisley PA1 2BE, Scotland, United Kingdom
Email: [*]andrew.gozillon@uws.ac.uk, [†]paul.keir@uws.ac.uk

***Abstract*—High-performance computing increasingly makes use of heterogeneous many-core parallelism. Individual processor cores within such systems are radically simpler than their predecessors; and tasks previously the responsibility of hardware, are delegated to software. Rather than use a cache, fast on-chip memory, is exposed through a handful of address space annotations; associating pointers with discrete sections of memory, within trivially distinct programming languages. Our work aims to improve the programmability of address spaces by exposing new functionality within the LLVM compiler, and then the existing template metaprogramming system of C++. This is achieved firstly via a new LLVM attribute, `ext_address_space` which facilitates integration with the non-type template parameters of C++. We also present a type traits API which encapsulates the address space annotations, to allow execution on *both* conventional and extended C++ compilers; and illustrate its applicability to OpenCL 2.x.**

I. Introduction

THE MAJORITY of today's architectures are heterogeneous. This means they contain at least two different types of processors or different local memory units. A familiar example of this is the modern personal computer (PC) which contains both a graphics processing unit (GPU) and a central processing unit (CPU). These architectures have immense potential as the extra processors tend to be specialized for particular tasks. For example, GPUs are specialized for rendering graphics. However, the parallel structure of the GPU lends it incredibly well to single instruction multiple data (SIMD) tasks on large data sets. This is commonly referred to as general-purpose computing on graphics processing units (GPGPU). Due to this several programming models centered on taking advantage of this aspect have been created. The two most iconic are OpenCL [1] and CUDA [2]. GPUs are exceptional at performing SIMD tasks, so much so that several of the supercomputers in the TOP500 list [3] use them.

However, power often comes at a cost. In the case of heterogeneous architectures, the complexity of the program increases for software aiming to take full advantage of the power available. One of the more common added complexities is memory management. Memory management is an important aspect of several heterogeneous architectures, as auxiliary processors of these architectures can have separate memory from their primary processor. One such architecture are PCs containing GPUs. Each GPU has dedicated memory and data must explicitly be transferred across from main memory by the CPU. Current GPGPU programming models also segment GPU memory into several distinct address spaces with different properties that help increase throughput.

Figure 1 C++ Address Space API

```
add_as_t<int,42> i = 12345;
static_assert(get_as_v<decltype(i)>==42);
assert(i == 12345)
```

In some cases, *named address space* qualifiers have been introduced to help associate variables with certain address spaces and thus certain properties. Named address space began in the Embedded C Extension [4], a set of optional extensions to the C programming language for use with embedded processors and have since spread. In fact, several GPGPU programming models make use of them for example CUDA, OpenCL and Metal [5]. An example from OpenCL is the `__private` qualifier which restricts the scope of a variable to a thread. Other programming models such as OpenACC [6] aim to promote a higher level view of GPGPU programming and abstract away address spaces from the programmer.

Both C and C++ are commonly used or extended in GPGPU programming. In fact all of the above mentioned programming models and languages with address spaces use or extend C/C++ in some way to achieve their goal. However, despite the number of models that make use of both C/C++ and address spaces, there is no standard compiler or library support for address spaces within C/C++. We believe that a C++ library-based approach could assist greatly in bringing address space support to C++. An API that new programming models could explore and integrate with would help improve portability and programmabillity. Having an API like this available also removes the requirement to extend the compiler for named address space support. As such we have created an C++ template API that takes inspiration from the C++ standard libraries type traits, an example can be found in Figure 1.

To aid in showcasing our API's viability we have decided to use the Clang [7] compiler's address space implementation. The Clang compiler's address space implementation is different from named address spaces. It takes the generic approach of accepting an integral parameter, provided by an end-user, to specify the value's address space, instead of having a fixed name set while building the compiler itself. This lends itself well to our API, which proposes integral parameters to index address spaces. Modifications to the LLVM compiler were made to add support for C++ *non-type* template parameters; since submitted as a patch. Note however that the API works with or without our LLVM extension, having two separate implementations hidden behind one interface.

II. The Programmable Address Space

The named address space implementation of address space qualifiers does not lend itself easily to user programmability. The main reason for this is that they have fixed names that differ per architecture, this makes creating portable libraries dealing with address spaces challenging. This type of qualifier can be found in Embedded C, OpenCL and CUDA.

The Clang compiler has chosen another more generic direction with its address space qualifier choosing an attribute as its basis, an example can be found in Figure 2. In this variation, a single qualifier is developed for applying address spaces to a variable; rather than separate names for each address space in an architecture or programming model. The attribute accepts a constant integer as an argument. This argument can be hard coded as shown in the example in Figure 2 or be a constant integer provided that it is not from a function or template argument. Each unique integer value corresponds to a unique address space.

Whilst Clang's current address space qualifier is a lot more generic and portable than named address spaces, it is not standard C++; rather being a non-standard extension of LLVM. It is also not as programmable as we might like, as it *cannot be used with template parameters*. As such we have built on Clang's address space qualifier and created a variation named `ext_address_space`. This variation allows non-type template integer parameters to be used as arguments.

Figure 2 An address space attribute in the Clang compiler

```
__attribute__((address_space(1))) int*as;
```

III. C++ Trait API

The implemented address space has increased programmability and allows for a variety of C++ template API's to be put in place; which in turn can make using address spaces easier and safer. In our case, we created an API that borrows from the C++ standard library's *type traits*. Type traits are used for gathering compile time type information and manipulating types. This C++ API lends itself to being overloaded, allowing other types of address spaces to be encapsulated inside. This allows it to act as an interface for different types of C++ programming models. This is exemplified in this section, as our API does not require our Clang address space extension to function; instead it acts as an interface for it when present and falls back on another implementation when it's absent.

A. The Traits API with `ext_address_space`

Our address space API uses three main class templates. The first is `get_as` (Figure 5), which allows the retrieval of the address space value from a type. The other two are `remove_as` (Figure 4) and `add_as` (Figure 3) which allow the programmer to both remove and add the address space on a type respectively. This allows explicit and easy manipulation of the address space as a qualifier similar to `const` and `volatile`. In fact, `remove_as` and `add_as` are parallels to `remove_const`, `remove_volatile`, `add_volatile` and `add_const` from the standard library.

Figure 3 The add_as class template and its type alias

```
template <typename T, unsigned Nv>
struct add_as {
using type = T __attribute__((
    ext_address_space(Nv)));
};

template <typename T, unsigned Nv>
using add_as_t = typename add_as<T,Nv>::
    type;
```

The `add_as` class template accepts a typename parameter `T` and unsigned integer `Nv`. The parameter `Nv` denotes the address space which we qualify the passed in type `T` with. The new type can then be accessed with the classes type alias type, this keeps with common template metaprogramming practice. Another common practice is the use of type aliases like `add_as_t` to simplify template class calls. We stick to this general pattern throughout our API, however future aliases will be elided for brevity. The `add_as` template works similarly to `add_const` in that it only adds the qualifier to the top most level of a type.

The `remove_as` class template requires specialization. The `remove_as` class template similarly to `add_as` accepts in a type and an address space. However, in this case the address space is ignored, instead it will be deduced from the type passed in. Deducing the value in this way allows the template to generically remove all available address spaces. Without this an explicit specialization for each template would have to be generated. The return value of `remove_as` is the passed in type with the address space qualifier removed. Both `const` and `volatile` qualifiers should remain untouched if present. The base template of `remove_as` specializes for types with no qualifier and returns the base type. Other specializations specialize for different combinations of qualifiers on the type and then return the type with the address space removed. The most specialized example of this specializes for `const`, `volatile` and the address space qualifier. It returns a type with the address space removed and other qualifiers intact. We showcase this specialization but elide the rest for brevity.

The class template `get_as` accepts the same parameters as the other class templates. However, like `remove_as` the address space parameter will be deduced. The output of this template class is a value that corresponds to the address space of the passed in type. Sticking with template metaprogramming practice it's named `value`. The base template and specialization is again similar to `remove_as`. The base template with no address space qualifier returns 0. As 0 is the default address space. Its specialization again fits all address space values and deduces the address space which is then returned.

Figure 4 The remove_as class template

```
template <typename T, unsigned Nv = 0>
struct remove_as {
using type = T;
};

template <typename T, unsigned Nv>
struct remove_as<T __attribute__((
    ext_address_space(Nv)))> {
using type = T;
};

template <typename T, unsigned Nv>
struct remove_as<T const volatile
    __attribute__((ext_address_space(Nv)))
    > {
using type = T const volatile;
};
```

B. The Traits API with `as_val`

Each API function has a fall-back version for compilers that do not support the `ext_address_space` extension. This means that the API will not cause compilation errors or undefined behaviour, ensuring portability. Outwardly the API calls do not change, nor do the required includes. Only the implementation of the functions change significantly. This is handled by macros that check if the `ext_address_space` attribute is implemented, then includes the relevant header files.

Figure 5 The get_as class template

```
template<typename T, unsigned Nv = 0>
struct get_as {
static const unsigned value = Nv;
};

template <typename T, unsigned Nv>
struct get_as<T __attribute__((
    ext_address_space(Nv)))> {
static const unsigned value = Nv;
};
```

A template class facilitates the API implementation's mimicking of the address space qualifier. The `as_val` class template (Figure 6) accepts a template type parameter and two non-type template parameters. The type parameter represents the type the address space qualifier is to be applied to. The two non-type template parameters represent the address space of the top most pointer (the `Nv` parameter) and the address space of the pointee (the `Np` parameter). For example, `as_val` would only support address space qualifiers on the first two pointers of a pointer to a pointer to an integer type. The integer itself would not be qualifiable. Of the parameters, only the type is stored, the two address space values are stored at a type level and can be deduced. Through C++ implicit conversion the various overloaded functions allow the user to use assignment operators and dereference operators as if they were using the base type. This means there should be no discernible difference between using a normal pointer type and `as_val`.

Whilst the implementations of the templates are different using the `as_val` class template. The change is not drastic. The functions all take in another non-type template parameter for the pointee address space (`Np`). However, this is hidden from the programmer using type aliases for each template.

The `get_as` implementation changes very little. Instead of deducing the value from the `ext_address_space` attribute currently tied to the type. It deduces the address space from the `Nv` parameter of the `as_val` class template.

Figure 6 The as_val class template

```
template <typename T, unsigned Np = 0,
    unsigned Nv = 0>
struct as_val {
as_val( ) {}
as_val(T x) : x(x) {}
operator T() { return x; }
T x;
};
```

The implementation of `remove_as` is simplified. Instead of having multiple specializations for every qualifier combination. We can simply specialize for `as_val`. There is however a base case and specialization as we cannot simply return the type with the `as_val` (address space) removed. As there is also a pointee address space tied to `as_val`. The base case checks for an `as_val` with an 0 address space and pointee address space and returns `as_val`'s stored type. The specialization checks for values greater than 0 in the `Np` parameter through deduction; then returns an `as_val` type with an `Nv` parameter set to 0 whilst keeping the same type and `Np` parameter.

For `add_as` we require a base template for non-`as_val` types and a specialization for types with `as_val`'s. The base template wraps the given type with an `as_val` and sets the `as_val` types address space parameter to the given address space. Whereas the specialization simply replaces the current address space parameter of the `as_val` type with the newly given address space.

C. The `add_pointee_as` *and* `remove_pointee_as` *Traits*

From the description of these templates it's possible to notice that there is no way to set the pointee address space of an `as_val` template class. As such there are another two template classes that allow manipulation of the pointee. They are `add_pointee_as` and `remove_pointee_as`.

For the `ext_address_space` attribute extension the `add_pointee_as` template class requires base classes and specializations similar to the `remove_as` template. They also provide a similar use, to peel off const and volatile qualifiers from the passed in type and then reapply them to the return type. The template parameters and type alias in this case are also identical to add_as. The next step is to peel off the top level pointer to get access to its pointee (if it has one) and then apply an address space to it. This is achieved by a helper template which breaks down the type using the C++ standard libraries pointer_traits template class and then rebuilding it. It does this in three steps, first it uses pointer_traits element_type parameter to get the pointees type. Secondly it applies the address space to this type. It finally uses pointer_traits rebind to bind the new type to the old type. The final result will be a type with a new address space on the pointee of the original pointer. For the non-extension implementation this is again much simpler. It is identical to add_as, except the pointee address space parameter is set instead.

IV. Example Use Case

Figure 7 OpenCL Reverse Array Example

```
__kernel void
reverseArray(add_pointee_as_t<float *, 1>
    d, int size){
  add_as_t<float[64], 2> s;
  int t = get_local_id(0);
  int tr = size-t-1;
  s[t] = d[t]
  barrier(CLK_LOCAL_MEM_FENCE);
  d[t] = s[tr]
}
```

The example we chose to showcase the C++ API can be found in Figure 7. The example is an OpenCL kernel function that will reverse one dimensional array data passed into it. The kernel function is based off a CUDA shared memory example found on NVIDIA's developer blog [8]. The idea is that each thread within a block will run an instance of this kernel and swap the relevant value based on its thread id. In this example the OpenCL named address spaces have been traded out for API calls which represent them as integer values. In the context of this example global is represented by the value one, local by the value two and private by the lack of a qualifier.

In the example the kernel accepts a pointer to some floating point data d. Alongside an integer size that represents the number of elements contained within d. The float pointer type is wrapped in an add_pointee_as_t class template from our API with an integer representing the global address space. This applies the address space to the target of the float pointer. Which makes the type equivalent to __global float*. Further down we create a statically allocated array of floats s which we can store values from d in for swapping later. We apply add_as_t to its type alongside an integer representing the local address space. Which makes the type of s equivalent to __local float[64]. The function then creates two index values t and tr which represent the values we wish to swap within the current thread. After which we use the t index to copy a value per thread from d in global memory to s in local memory. However, before we can swap the data we must place a local barrier to prevent data races. After the barrier we can proceed to reverse the array.

A feature of our API is that if the attribute extension for ext_address_space is not found in the compiler it will still compile. It will fall back on the implementations that make use of the as_val template class which stores the address space value and variable within itself. Despite the variable now being wrapped within as_val it can still be used as if it was the raw type. This works through overloading certain operators in the class so that implicit conversion allows access to the underlying data. This fall-back functionality is provided by including all the required API functions through a single include file. This include file can then add different API implementations based on the presence of the ext_address_space attribute. This functionality has been tested with the GCC and Clang C++ compilers. Despite choosing OpenCL as the language for the example, the C++ API should be usable in the same way for C++ and C++ based programming models.

V. Conclusion

In conclusion, we have presented a C++ template API that we believe improves the programmability of address spaces. The API borrows concepts from C++'s type traits. We believe this API will help facilitate bringing address space qualifiers further into the C++ type system allowing further template metaprogramming and type safety opportunities. To help showcase the ability of our API to integrate existing address space implementations, we integrated it with Clang's address space attribute. However we also presented an API implementation that would also allow it work as a standalone implementation of address spaces that requires no compiler extensions.

References

[1] A. Munshi, "The opencl specification," in *Hot Chips 21 Symposium (HCS), 2009 IEEE*. IEEE, 2009. doi: 10.1109/HOTCHIPS.2009.7478342 pp. 1–314. [Online]. Available: http://dx.doi.org/10.1109/HOTCHIPS.2009.7478342

[2] C. Nvidia, "Compute unified device architecture programming guide," 2007.

[3] T. S. Sites, "Top500 lists," 1993. [Online]. Available: https://www.top500.org/

[4] JTC1/SC22/WG14, "Programming languages - c - extensions to support embedded processors," 2006. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg14/

[5] Apple, "Metal," 2014. [Online]. Available: https://developer.apple.com/metal/

[6] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openacc - first experiences with real-world applications," in *European Conference on Parallel Processing*. Springer, 2012. doi: 10.1007/978-3-642-32820-6_85 pp. 859–870. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32820-6_85

[7] "clang: a c language family frontend for llvm." [Online]. Available: http://clang.llvm.org/

[8] M. Harris, "Using shared memory in cuda c/c++," 2013. [Online]. Available: https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/