Document No.: N4008
Date: 2014-05-22
Reply to: Arch D. Robison <arch.robison@intel.com>
Pablo Halpern <pablo.g.halpern@intel.com>
Robert Geva <robert.geva@intel.com>

SIMD Polymorphism

1. Introduction

N3831 [1], presented at the February 2014 Issaquah meeting, proposed (among other things) adding SIMD-enabled functions to C++. Although there was significant interest in this concept, members of the committee felt that the description in N3831 would be improved by better integrating SIMD-enabled functions with the C++ type system. This paper explores such an integration. We show that though these functions exhibit a form of polymorphism, it is best kept distinct from the existing features for polymorphism in C++. In particular, SIMD polymorphism is best *not* treated as C++ template instantiation or overload resolution. Instead, we propose extending the C++ type system.

Our proposed direction gives SIMD-polymorphic functions types distinct from ordinary functions. The type of a SIMD-polymorphic function indicates *which* forms it can take on, but leave the choice of which form is called to the execution system. Our proposal supports strongly-typed pointers to SIMD-polymorphic functions.

Our proposal is not tied to a specific syntax for SIMD loops. Calls to SIMD-polymorphic functions can happen in a variety of other contexts too, such as auto-vectorized loops, parallel algorithms [2], or even “Superword Level Parallelism” [3] within a basic block. Some existing auto-vectorizing compilers have their own ad-hoc scheme for SIMD-polymorphic functions, limited to a predefined set of standard math library functions such as `sin` and `exp`. Our proposal enables programmers to define their own SIMD-polymorphic functions.

Section 1 explains the problem. Sections 2-4 explore rejected alternatives. Section 5 describes our preferred direction.

2. Background

This section summarizes the desirable features for SIMD-polymorphic functions.

Given a SIMD loop that calls a function *f*, we would like to be able to create a variant of *f* that evaluates multiple iterations at once. Better yet, we would like to have multiple variants, because different call sites may benefit from different kinds of variants, based on whether particular parameters are:

- *uniform* (same for all iterations),
- *linear* (arithmetic sequence), or
- *varying* (arbitrary values).

For linear parameters, there may be different variants for different strides. Furthermore, we would like to have different variants for call sites inside branches (i.e. the variant employs masking), different chunk lengths, or different alignment guarantees.[1] A non-SIMD variant where all parameters are scalar is required for non-SIMD contexts.For example, consider the following code:

```
extern void foo( float x, float* y, float z );
extern void d[], e[];
for simd( int i=0; i<n; ++i ) {
    foo(d[0], d+i+1, e[i]);
}
```

A compiler can recognize that the ideal variant of `foo` would be optimized for uniform *x*, linear y, and varying z.

If the definition of `foo` were visible to the translation unit, the compiler could generate that ideal variant. However, with traditional separate compilation, it becomes desirable to declare `foo` in a shared header, but define it in a different compilation unit. In that case we must specify which variants to generate when compiling `foo`, and for type safety, the compiler must know which variants are available when compiling a call site of `foo`. This paper does not recommend a specific syntax for specifying those variants, and instead focuses on functionality issues, assuming that syntax exists to specify variants, such as proposed in N3831 [1].

2.1. Pointers to SIMD-polymorphic Functions

Pointers to SIMD-polymorphic functions seem useful, such as to take the address of `foo`, pass it around, and deference it later. Here's an example, where ***spec*** is a placeholder for extra information, which might be quite elaborate, to indicate which variants exist.

```
void bar( void(*pf)(float,float*,float) spec ){
    extern float d[], e[];
    for simd( int i=0; i<n; ++i )
        (*pf)(d[0], d+i+1, e[i]);
}
```

```
extern void foo( float x, float* y, float z ) spec;
bar(&foo);
```

[1] OpenMP 4.0 supports all of the variants mentioned in this section.

```
void foo(float x, float* y, float z) spec {
    *y = (x+z)/2;
}
```

Note that the three portions could be in separate translation units.

When the address of a SIMD-polymorphic function is taken, it essentially creates a pointer to a sheaf of all variants, and a member of the sheaf is chosen when the pointer is dereferenced.

2.2. Typedef

Pointer to SIMD-polymorphic functions ought to be declarable in typedefs. For example, function bar from above could instead be declared the following way:

```
void(*pftype)(float,float*,float) spec;
void bar(pftype pf);
```

2.3. Custom Definition of a Variant

An oft-requested addition is to allow a variant to be defined with a custom body, because sometimes there are multiple ways to implement a function, and different ways are more efficient for different variants. Consider computing the area of a triangle with corner coordinates (ax,ay), (bx,by), and (cx,cy):

```
float area(float ax, float ay, float bx, float by, float cx, float cy) spec_1 {
    return 0.5*((ax*by-ay*bx) + (ay-by)*cx + (bx-ax)*cy);
}
```

Given two corners with uniform coordinates and one with varying coordinates, the operation count is smallest if (ax,ay) and (bx,by) are the uniform corners. To deal with the other two cases, we might like to provide alternative definitions. For example, here is an alternative definition optimized for uniform (bx,by) and (cx,cy):

```
float area(float ax, float ay, float bx, float by, float cx, float cy)
spec for custom variant with uniform bx, by, cx, cy (and suppress non-SIMD variant)
{
    return 0.5*((bx*cy-by*cx) + (by-cy)*ax + (cx-bx)*ay);
}
```

The syntax for ***spec***$_1$ should indicate which custom variants will be defined elsewhere.

2.4. Address of a Variant

Another requested feature is the ability to take the address of a specific variant. We believe this is a fundamentally flawed feature, because the programmer generally does not know if that variant is appropriate to call at a particular call site, as explained in Section 3.3.

3. An Attempt at SIMD-morphism Using the Current C++ Type System

The next section describes our attempt to implement SIMD-polymorphic functions using only existing features of the C++ type system. Because the experiment taught us what *not* to do, we provide only a sketch. Readers interested only in the conclusions can skip to Sections 3.2 and 3.3, which argue the conclusions from first principles.

3.1. A Syntax using Templates and Overloading

At a high level, our prototype vaguely resembled `valarray`. The idea was to create special template types that specified *linear* or *varying* parameters. The *uniform* case was indicated by lack of the special types. We also tried a variation where *uniform* had its own special template type. Template instantiation was used to create variants, and overload resolution was used to select the right variant.

Below is a sample of our prototype, showing the creation of a SIMD-polymorphic function `bonzai` with two variants.

```
template<typename Y, typename X>
void bonzai( Y y, float a, X x ) {
    *y = 1 + *y + a*x;
}

// Export two variants
template void bonzai(float*,float,float);
template void bonzai(linear<1,float*,8>,float,varying<float,8>);
```

The first explicit instantiation creates a non-SIMD version. The second explicit instantiation creates a variant for the case of (*linear*, *uniform*, *varying*) with a chunk size of 8. Operators `+` and `*` were overloaded on the special template types to behave in an intuitive way.

Consumption of a SIMD-polymorphic function could look like this[2]:

```
template<typename Y, typename X>
void bonzai( Y y, float a, X x );

// Import two variants
extern template void bonzai(float*,float,float);
extern template void bonzai(linear<1,float*,8>,float,varying<float,8>);

void example( int n, float a, float x[], float y[] ) {
    for simd(i=0; i<n; ++i)
        bonzai(y+i, a, x[i]);
```

[2] Our prototype actually used a template function `simd_for` and polymorphic lambdas instead of the "for simd" syntax, with numerous distracting work-arounds for issues incurred by trying to do everything as a library.

```
    }
```

For simple examples, we got respectable vectorized code out of this approach, since each overload could be implemented as a vector operation. However, two problems became apparent for more complex examples:

- There was no support for pointers to SIMD-polymorphic functions.
- The best variant is chosen too soon in some cases.

The next two sections show that these problems are fundamental to any scheme that tries to use templates and overloading to implement SIMD-polymorphic functions.

3.2. Why Template Instantiation is Not Quite the Right Model

It is tempting to treat variants as instantiations of a template, where the instantiations differ by whether parameters are uniform, linear, or varying, because SIMD-polymorphism has aspects of parametric polymorphism. This model is attractive because custom definition of a variant acts like an explicit specialization.

Alas pointers to SIMD-polymorphic functions break the analogy with templates, because the variants must travel together as a sheaf when the address of a SIMD-polymorphic function is taken and dereferenced later. The address would have to be the address of a table of all instantiations in the sheaf. Indeed, we propose essentially that in Section 4. In principle, the table could be a library-defined object with a variadic constructor taking pointers to each instantiated variant. But we think it better to let the compiler do the bookkeeping for creating and maintaining such tables.

3.3. Why Using C++ Overload Resolution Is a Poor Model

Implementations of similar functionality (e.g. OpenMP 4.0) resolve which variant to call at compilation time. Thus it is tempting to treat variants as overloads of a function, and handle variant resolution as part of C++ overload resolution. The problem with this approach is *when* resolution happens in the compilation process. Typically, a compiler performs C++ overload resolution during parsing and template instantiation. But the ideal variant to call depends on dataflow analysis (possibly inter-procedural or even between translation units), which happens well *after* overload resolution.

For example, consider:

```
void oomwa(){
    extern float a[];
    extern void g(float,float,float) spec;
    for simd( int i=0; i<n; ++i ) {
        float a,b,c;
```

```
                if( i==n-1 ) {
                    a = 0;
                    b = i;
                    c = i;
               } else {
                    a = 0;
                    b = i;
                    c = 0;
               }
               g(a,b,c);
           }
       }
```
This example requires dataflow analysis[3] to determine that `a` is uniform, `b` is linear, and `c` is varying. Furthermore, if the compiler peels off the last iteration into a separate loop, then `c` becomes uniform too. The point is that C++ overload resolution occurs much too early in the compilation process to be making these decisions.

4. A Strongly-Typed Late-Resolution Model

This section proposes a strongly-typed late-resolution model for SIMD polymorphism. The type system ensures correctness by tracking which variants are available at a call site. The choice of which of those variants is used is delegated to the implementation. In practice, most compilers will likely make the choice after running data-flow analysis, though in principle the decision (or part of it) could be deferred to run time.

4.1. Typing

In the strongly-typed model, a pointer to a SIMD-polymorphic function is conceptually a pointer to a sheaf of all available variants, and a correct variant (possibly more than one) is chosen when the pointer is dereferenced at a call site. The type of a SIMD-polymorphic function is distinct from the type of an ordinary function. The type indicates which kinds of variants are available in the underlying sheaf, enabling the implementation to choose one that is appropriate. The sheaf always includes the non-SIMD variant, which provides an ultimate fallback that will work in any context. This extension of the C++ type system enables strong type safety for pointers to SIMD-polymorphic functions, detailed later sections.

Each variant has a *kind*, which describes the variant's SIMD-related properties. The kind includes information such as:

- The SIMD chunk size.
- Can the variant can be called from inside a SIMD branch (i.e. supports masking)?

[3] Or a much more advanced type system with "typestate".

and per-parameter information such as:

- Is the parameter *uniform*, *linear*, or *varying*?
- For pointer and reference parameters, is the related address aligned beyond the default? If so, by how much?

The qualities listed above are taken from the OpenMP 4.0 `declare simd` construct.

In our type system the entire sheaf has a type and the individual variants do not have a type separate from their sheaf, with the exception of the scalar variant. For example, overload resolution never chooses among variants, and template argument deduction can deduce a sheaf type, but not a variant therein. Except for the non-SIMD variant, the only way code can semantically distinguish a variant as a separate entity is by providing a custom definition *and* making a SIMD-polymorphic call that resolves to that variant.

4.2. Pointer Conversions

It seems essential to allow implicit conversion from a pointer-to-SIMD-polymorphic-function to a pointer-to-ordinary-function, so that a SIMD-polymorphic function can be used in any context where an ordinary function can be used. The conversion results in a pointer to the non-SIMD variant.

Furthermore, for dealing with legacy code, it seems useful to allow explicit conversion (casting) from a pointer-to-scalar-variant to a pointer-to-SIMD-polymorphic function. For example:

```
extern (*wa_handler)(float,float*,float);
void bar() {
    extern float d[], e[];
    typedef void (*psf)(float,float*,float) spec;
    for simd( int i=0; i<n; ++i )
        // Cast wa_handler to pointer-to-SIMD-polymorphic function and invoke it.
        *(psf)(wa_handler)(d[0], d+i+1, e[i]);
}
```

```
void init() {
    set_wa_handler(foo); // foo implicitly cast to pointer-non-SIMD variant
}
```

The conversion must be explicit, and the programmer must be certain the function really is SIMD-polymorphic with the expected variants. The risks of uncertainty are essentially identical to those for downcasting pointers to class objects.

4.3. One Definition Rule

A SIMD-polymorphic function must have a single definition with respect to the number of variants, their kinds, and which variants have custom definitions. Each variant must have one definition. The syntax for declaring a SIMD-polymorphic function should declare the kinds of all variants in the sheaf, and which variants have custom definitions elsewhere.

4.4. Pointer Equality and Identity

It seems desirable that pointer identity be preserved by the implicit casting and explicit casting described above. For example, the following assertions in the following code should hold:

```
typedef void (*psf)(float,float*,float) spec; // SIMD-polymorphic
typedef void (*pf)(float,float*,float);       // Ordinary
void bar(psf p) {
    pf q = p;
    assert(q==p);
    pfs r = (psf)q;
    assert(r==q);
}
```

These "obvious" identities should be kept in mind when considering solutions to the problem posed in the next section.

4.5. Subtyping: Semantically Clean, But Costly to Implement

Subtyping considerations drive the implementation of pointer-to-SIMD-polymorphic functions. Let *p* and *q* have types *P* and *Q* respectively, where both are pointers to SIMD-polymorphic functions where the types of the non-SIMD variants are equivalent. Suppose the variant kinds in *P* are a subset of the variant kinds in Q. Should the assignment *p=q* be allowed? From a type-safety perspective, the assignment is okay, and indeed seems useful in practice, as it would be frustrating to not be able to use *q* in contexts requiring a value of type P. Alas, enabling this form of subtyping incurs some costs.

For example, one possible implementation of pointer-to-SIMD-polymorphic function is to lay out a table of pointers to SIMD variants just before the entry point for the non-SIMD variant. This way, the pointer conversions discussed in previous sections require nothing. More generally, *p=q* requires no special effort if additional entries needed in the table for *q* comprise a suffix of the table needed for *p*. If not, we need to build a new table (with the right lifetime) and a stub function (built like a "trampoline") for the non-SIMD variant to put after the table. Alas that approach loses the identity of *q*, in the sense that there's no way to easily ascertain that *p==q*.

Having a table slot always reserved to point to the non-SIMD variant would work around that problem. Then those special slots could be used to implement *p==q*.

But that still leaves the problem of how to manage the lifetime of the table. Perhaps the best solution is to leave the lifetime issue to the programmer. The on-the-fly tables could become explicit objects managed by the programmer. For example, let P and Q denote the types of *p* and *q* in our example. The explicit object might look like this:

```
template<typename P>
class simdmorphic_adaptor {
public:
    //  Construct from  ptr-to-func type Q, which must have at least the variants in P.
    template<typename Q>
    simdmorphic_adaptor( Q pf );

    // Get ptr-to-func with type P, which is valid only until *this is destroyed.
    P get() const;
};
```

An alternative that avoids the lifetime problem, in exchange for more run-time cost, is to *not* subset the table at each conversion, but instead do associative lookup. The table would contain key-value pairs, where the key indicates the variant's form and the value points to the code for the variant. For example, the key could be encoded with 2 bits per parameter, denoting one of four cases:

0. uniform
1. linear with unit stride
2. linear with non-unit stride, with the stride value in a secondary table.
3. varying

Now the compile-time type indicates some subset of the table, but not where that subset's entries are within the table. Lookup would now resemble a hash-table lookup, where the key is the kind of variant of interest, which would cost some cycles. But remember that the call context is inside a SIMD loop, so the lookup could be hoisted out of the loop. The type system guarantees that the table lookup will succeed.

Associative lookup would make code more robust against ABI breakage. If a new variant is added to the definition of a SIMD-polymorphic function, consumers of the function would still work without recompilation, even though the One Definition Rule was being broken. In the non-associative schemes, the programmer must be careful to add a new variant in a way that puts it in the first slot of the extended table.

There is no free lunch on the subtyping issue. A zero-cycle solution requires limiting *p=q* to cases where the table for *p* is a prefix of the table for *q.* The explicit adaptor approach has a pay-as-you-go appeal. The associative approach is easiest to use and most robust against ABI changes, but incurs a lookup cost.

5. Extension to ISA-polymorphism

A heterogeneous machine has execution engines with different instruction set architectures (ISA). For these machines there is a need to write a function once and have the compiler generate multiple translations, each targeting a different ISA. Furthermore, there is often a desire to write some of the variants differently at the source code level. For example, `std::sort` might be best implemented with bitonic sort on one ISA, and merge sort on another. Dispatch often has to be resolved late: at load time, based on whether the hardware is present, or at run-time, based on whether the hardware is idle.

Our strongly-typed late-resolution approach might extend to “ISA-polymorphism” to simplify programming heterogeneous machines. Though given the additional complexities and subtle differences with SIMD polymorphism, we are not proposing it at this time.

6. Acknowledgement

Artur Laksberg provided valuable comments on an earlier draft.

7. References

[1] N3831, Language Extensions for Vector level parallelism.

[2] N3960, Working Draft, Technical Specification for C++ Extensions for Parallelism, Revision 1.

[3] Larsen and Amarasinghe, “Exploiting superword level parallelism with multimedia instruction sets”, PLDI ’00.