Supporting Function Variants in OpenMP

S. John Pennycook[1(✉)], Jason D. Sewall[1], and Alejandro Duran[2]

[1] Intel Corporation, Santa Clara, USA
john.pennycook@intel.com
[2] Intel Corporation Iberia, Madrid, Spain

Abstract. Although the OpenMP API is supported across a wide and diverse set of architectures, different models of programming – and in extreme cases, different programs altogether – may be required to achieve high levels of performance on different platforms. We reduce the complexity of maintaining multiple implementations through a proposed extension to the OpenMP API that enables developers to specify that different code paths should be executed under certain compile-time conditions, including properties of: active OpenMP constructs; the targeted device; and available OpenMP runtime extensions. Our proposal directly addresses the complexities of modern applications, allowing for OpenMP contextual information to be passed across function call boundaries, translation units and library interfaces. This can greatly simplify the task of developing and maintaining a code with specializations that address performance for distinct platforms and environments.

1 Introduction

As the OpenMP* programming model has evolved to keep pace with evolving architectures, it has introduced many new features (*e.g.* task-based parallelism, offloading to accelerators and explicit SIMD programming). These features are both sufficient to ensure that OpenMP is portable to a wide range of devices, and also expressive enough that developers are able to write codes that are capable of extracting a high level of performance from their targeted device.

However, writing a code that is able to run well on *all* of the devices that OpenMP supports – a code that exhibits high levels of "performance portability" [12,13] – remains a challenge; different devices and/or OpenMP implementations may prefer different ways of expressing parallelism, and some may prefer different algorithms altogether. Several frameworks have been developed on top of OpenMP in an attempt to simplify development when targeting multiple architectures [4,7]; our interpretation of this is that many find the current tools in OpenMP lacking in expressibility.

The proposed `concurrent` directive from OpenMP TR6 [11] addresses part of this problem, effectively providing a mechanism for developers to request that the decision of which OpenMP construct(s) should be used on a given loop nest be made by the implementation, based on analysis of the loop nest and the implementation's knowledge of the targeted device. Such a *descriptive* approach

B. R. de Supinski et al. (Eds.): IWOMP 2018, LNCS 11128, pp. 128–142, 2018.
https://doi.org/10.1007/978-3-319-98521-3_9

covers simple cases well, but we believe it is insufficient for the needs of expert programmers: the implementation's decisions cannot be overridden when the developer has additional information (or simply believes that they know better), and a general-purpose OpenMP implementation will likely not be able to identify and replace algorithms (although this may be possible for common idioms). In this paper, we focus on providing a *prescriptive* complement to the functionality of `concurrent`, enabling users to assert direct control over which code is executed and under which conditions.

It should be noted that our proposal is primarily focused on furnishing expressibility rather than providing new functionality to developers; our intent is to make existing functionality more accessible, and to present a simple mechanism with common syntax that can be employed across all base languages supported by OpenMP. There are already myriad options for maintaining different code paths for different devices and compilation contexts: preprocessors and `#ifdef` guards; template functions (in C++); and the aforementioned "performance portability" frameworks to name but a few. In our own attempts to use such approaches, we have found them wanting: standardized preprocessors are not available for all languages, and handling multiple conditions through nested `#ifdef` clauses can quickly lead to unreadable code; templates may be too complex for average users to reason about, and are not available in C or Fortran; the use of non-standard interfaces may lead to interoperability or composability challenges; and developing bespoke solutions to this problem for all codes (or even all domains) is not productive.

2 Related Work

Since OpenMP 4.0 [10], developers have been able to request that a compiler create alternative versions of a function specialized for execution in SIMD or on accelerator devices via the `declare simd` and `declare target` directives respectively. Developers are able to influence the code generation inside such functions using clauses to these directives (*e.g.* specifying `uniform` or `linear` will lead to different optimizations for `simd` functions), but are unable to exert any direct control using standard OpenMP functionality. Furthermore, since both directives only alter the way in which a single implementation of a function is compiled, the optimizations that can be employed are limited to those that compilers can identify and (safely) implement automatically after static analysis of the function, with no further input from the developer.

To address these issues, there have been several previous efforts to extend OpenMP to enable developers to provide drop-in replacements for the functions generated by `declare simd` and `declare target`. OmpSs supports an extension to `target` – the `implements` clause [3] – which specifies that one function is an alternative implementation of another, specialized for particular devices; this idea has been adopted, with the same syntax and semantics, in OpenMP TR6 [11]. The Intel® C/C++ compiler provides similar functionality for functions specialized for particular SIMD widths and instruction sets – so-called

"vector variants" [8] – via function attributes; RIKEN and Arm explored the same concept via an OpenMP extension – `alias simd` [9] – with different syntax but similar semantics. None of these proposals address situations in which a developer wishes to specialize a function for SIMD *and* a particular device simultaneously, nor considers the utility of extending function implementation/-variant/alias support to other situations. We proposed yet another version of this functionality – `declare version` – for memory allocators [14] in previous work. In this paper, we attempt to unify all of the directives discussed above into a single directive, which permits the specialization of functions based on multiple criteria and which is designed to be extensible to future OpenMP constructs.

It should also be noted that similar functionality has already been adopted outside of OpenMP. Thrust [2] and the parallel extensions to the C++ Standard Template Library (STL) [6] allow an execution policy (encapsulating the programming model, device, etc) to be passed to a function call, enabling different implementations to be selected by the library; the Kokkos [4] framework provides a similar facility through the use of tags passed to functors. The PetaBricks [1] language takes a different approach, providing a mechanism to declare a function in terms of multiple candidate algorithms from which the compiler and an autotuning framework can construct an optimized application.

3 Specialization

Specialization is a valuable concept to deploy in software development, optimization, and maintenance. At its core, it is simply a programming construct – a function, a type declaration or even a snippet of code – paired with a mechanism for expressing when that construct should be used. The most common forms of specialization in real code are highly manual in nature: the programmer decides that a particular function (for example) needs to be used in a particular context and makes it happen by forking the codebase, or by employing a conspicuous branch in the code to perform the discrimination.

Consider a parallel histogram computation, where many inputs are divided among threads and reduced into a relatively small array. A developer has a common implementation in a portable language that runs on many of their platforms of interest, and which uses atomic additions found in the language; this implementation provides correct results in all cases. The developer may discover that the atomics primitives on some platforms are very slow, and that giving each thread its own copy of the accumulation array (*i.e. privatizing* the array) runs with much greater efficiency on those platforms. This could be described as specialization for performance's sake, but it is also easy to imagine a platform with no support for atomics at all, in which case the privatized implementation would be necessary for compatibility.

That specialization is useful and even necessary in real code bases is evident; the challenge – which our proposal aims to address – is to make the *mechanism* for deploying the appropriate specialization as easy-to-use and expressive as possible.

4 Enabling Specialization in OpenMP

We propose an extension that uses functions and function calls as the language-level granularity of specialization; developers are able to use a new directive, `declare variant`, to indicate that a given *function variant* is intended to be a specialization of another *base function* with a compatible type signature. The specialization mechanism is then used to decide which function calls to the base function are replaced with function calls to specialized variants (if any). Our proposal allows the user to annotate function variants with *selector* information that guides the specialization mechanism; these selectors generally interact with the *context* around function calls to select meaningful specializations.

Specialization of function calls enables composability (*e.g.* across translation units and library interfaces) via a mechanism that is simple to understand, familiar due to its similarity to other approaches (*e.g.* template specialization) and in keeping with good software engineering practices (*i.e.* modular design). A simple example of a function *variant family* is given in Fig. 1.

```
// Default behavior (i.e. the base function)
float my_rsqrt(float x)
{
  return 1.0f / sqrt(x);
}

// Variant that uses an approximation
float my_rsqrt_approx(float x)
{
  // e.g. result after several Newton-Raphson iterations
}

// Variant vectorized with AVX-512
__m512 my_rsqrt_avx512(__m512 x)
{
  return _mm512_rsqrt_ps(x);
}

// Variant which forwards to library implementation (where it exists)
float my_rqsrt_native(float x)
{
  return rsqrt(x);
}
```

Fig. 1. An example of a function variant family for computing reciprocal square roots.

4.1 `declare Variant` Syntax

The C/C++ syntax[1] of our proposed `declare variant` directive is as follows:

```
#pragma omp declare variant(base-function) [match(context-selector)]
<specialized-function-definition-or-declaration>
```

where `base-function` is the name of the function that the programmer wishes to specialize. The function the directive is applied to is a specialized variant that is

[1] Analogous syntax is proposed for Fortran but we omit it for brevity.

defined as a suitable replacement of `base-function`. The scope of this directive is the translation unit where it appears; therefore, we expect the directive to be applied to the specialized function declaration in headers and also to the specialized function definition itself.

If no `match` clause is provided, then all calls to the `base-function` will become calls to the specialized function. Figure 2 shows how the `declare variant` directive is used to provide an OpenMP specialization, `my_rsqrt_omp_approx`, of the function `my_rsqrt`. All calls to `my_rsqrt` will be replaced by calls to `my_rsqrt_omp_approx`. As the `variant` directive is only recognized by OpenMP-enabled compilers, this provides a mechanism for users to have different code used when compiling for OpenMP (which could also be achieved by means of `#ifdef _OPENMP` with preprocessors).

```
#pragma omp declare variant(my_rsqrt)
float my_rsqrt_omp_approx(float x) { ... }

void foo (float x)
{
   ... = my_sqrt(x); // will call my_rsqrt_omp_approx
}
```

Fig. 2. Example of the `declare variant` directive.

The `match` clause allows the developer to specify a *context-selector* that specifies the context in which calls to *base-function* should be substituted with calls to the specialized function. The syntax of a *context-selector* is as follows:

```
match(trait-class-name={trait[(trait-properties)][,...][,...]})
```

We propose a number of traits for specialization, organized into four trait classes that can specified in the `match` clause: OpenMP construct traits, for specialization based on OpenMP constructs; device traits, for specialization based on the target device; implementation traits, for specialization based on characteristics of the underlying OpenMP implementation; and user-specified traits. Section 4.2 details the traits of each class and their properties (if any).

Multiple specializations of the same base function can be specified using a `declare variant` directive on each specialization. Figure 3 shows an example where two variants have been defined: `my_rsqrt_omp_approx`, to be called when the base function appears in the context of a `simd` directive; and `rsqrt` (possibly provided by a library), to be called when the base function appears in the context of a `target` directive.

In the previous example, it was unambiguous which specialization should be used in each call to `my_rsqrt`, but that is not always the case. For example, if the call were to happen inside a `target parallel for simd` directive, it would be unclear which specialization should be called. We handle such cases by assigning different priorities to each variant, and selecting the variant with the highest priority at a callsite; this algorithm is described in detail in Sect. 4.3.

```
#pragma omp declare variant(my_rsqrt) match(construct={simd})
float my_rsqrt_omp_approx(float x) { ... }

#pragma omp declare variant(my_rsqrt) match(construct={target})
float rsqrt(float x); // library provided

void foo ( float x )
{
  #pragma omp simd
  for ( ... ) { ... = my_rsqrt(x); } // will call my_rqsrt_omp_approx

  #pragma omp target
  {
     ... = my_rsqrt(x); // will call rsqrt
  }
}
```

Fig. 3. Example of `declare variant` directives with match clauses.

4.2 Context Selection Traits

We have identified several concepts that give rise to a need for specialization, and for the purposes of their use and description, we have organized them into *classes* of *traits*. This taxonomy aids the user in clearly expressing their intents for when a particular variant takes precedent over another for a given context. This proposal identifies four distinct classes of traits that help distinguish the conditions for specialization.

OpenMP Construct Traits. The traits in the `construct` class are related to existing OpenMP constructs that might impact a developer's choices for specialization. Table 1 describes the traits in the `construct` class. Each trait specified for this class restricts the associated variant to calls to the base function that appear in the context of the directive of the same name.

Table 1. Traits in the `construct` trait class

Trait name	Example uses
`target`	Code paths that track host/target allocations and perform transfers
`parallel`	Code paths that choose between serial & parallel algorithms; code paths that discriminate based on memory model (*e.g.* atomics, critical, etc)
`teams`	Code paths that choose algorithms or implementations based on synchronization behavior; code paths that perform differently when synchronization is expected to be fast within a team
`simd`	Code paths that deploy *horizontal* vector operations (*e.g.* conflict detection); code paths that override default auto-vectorization behavior

For the `simd` trait, we also propose to allow different *trait properties* that represent clauses available in the `declare simd` directive. These properties further restrict the context in which a variant can be selected. In Fig. 4 two variants

are defined to be used in the context of a `simd` construct. The first variant can be used in any `simd` context but the second one can only be used when the `simd` context also determines that the argument of the function is `linear`. Consequently, the first invocation of *foo* in Fig. 4 will be substituted with the first variant as the compiler cannot determine that the argument is `linear`, whereas the second call to *foo* will be substituted with the second variant as the compiler can determine that i is `linear`.

```
float foo(float *x);

#pragma omp declare variant(foo) match(construct={simd})
__mm512 foo_simd_gather(__m512 *x); // needs to use gather instructions

#pragma omp declare variant(foo) match(construct={simd(linear(x))}
__mm512 foo_simd_linear(__mm512 *x); // can avoid gather instructions

#pragma omp parallel for simd linear(i)
for ( i = 0; i < N; i++ ) {
    ... = foo(x[rand()]); // will call foo_simd_gather
    ... = foo(x[i]);      // will call foo_simd_linear
}
```

Fig. 4. Example of `simd` trait properties.

Device Traits. The traits in the `device` class are based on properties of the hardware that the code is being compiled for. Therefore, they restrict the contexts where a variant can be selected to only those where the specified device traits are true. Table 2 describes the traits in the `device` class. Multiple `isa` traits can be specified for a single variant: all of them must be supported by the target device for a variant to be selected. We propose that implementations not be required to be able to compile the function body for variants with `device` traits that are not supported (*e.g.* an unknown ISA), thereby simplifying the use of device-specific intrinsics by programmers. However, we still require that extensions used in the context of a variant function should at least allow other implementations to properly parse (and ignore) the function (*i.e.* by allowing to find the closing bracket of the variant function).

Table 2. Traits in the `device` trait class.

Trait name	Example uses
`uarch`(*uarch-name*)	Code paths that use different optimizations for different microarchitectures; code paths that care about particular implementations of instruction sets or compute capabilities
`isa`(*isa-name*)	Code paths that use specific instruction sets

OpenMP Implementation Traits. Traits in this class are concerned with properties of the particular OpenMP implementation that will be used to run

the generated code. Only implementations that support the traits specified in a selector can select that variant as a replacement of the base function. Table 3 describes the traits in the `implementation` class.

Table 3. Traits in the `implementation` trait class.

Trait name	Example uses
`unified_shared_memory` `unified_address`	Code paths that require runtime support for unified shared memory/address spaces across devices. The behavior of these traits is documented in OpenMP TR6 [11]
`vendor`(*vendor-name* [,*extensions*])	Code paths that require vendor-specific and/or prototype concepts

User Traits. In addition to the above trait classes associated with OpenMP contexts, hardware, and runtime capabilities, there is a `user` class that accepts logical expressions as traits. These logical expressions, expressed in the base language, must be able to be evaluated at compile-time, and they can be used to add arbitrary user-specified conditions that can inform variant selection. Figure 5 shows an example of a `user` trait in use; the logical test for the value of the static constant variable `layout` may enable or disable each of the variants in the example.

```
void foo(float *x);

typedef enum {AoS=0, SoA} layout_t;

#pragma omp declare variant(foo) match(user={condition(layout==AoS)})
void foo_AoS(float *x);

#pragma omp declare variant(foo) match(user={condition(layout==SoA)})
void foo_SoA(float *x);

...
static const layout_t layout = AoS;
...
foo(z); // will call foo_AoS
```

Fig. 5. Example of a `user` trait; the value of the compile-time variable `layout` determines the logical value of the `user` selector in each variant.

These classes and their traits are those that we have chosen as the most important and tractable in this initial proposal; they are all static notions that allow contextual information to be tracked or inferred during compile-time. Many other concepts – for example, the values of certain variables or certain arguments at runtime, system conditions, and other language constructs – would be interesting to explore in future work.

4.3 Caller Context and Variant Selection

Our proposed selection mechanism is not explicit; instead, the choice of which variant to call is performed through the interaction of the various selectors and the *context* around the function call. This is necessary to correctly handle cases in which the arguments to a function may be generated or modified by the compiler (*e.g.* during auto-vectorization); to leverage contextual information not exposed to the developer (*e.g.* during auto-parallelization); and to enable selection to be employed transparently for functions maintained by other developers (*e.g.* library functions). Such an implicit mechanism does not remove any control from the user; they remain free to call specific variants explicitly by name.

In the remainder of this section, we discuss how such contextual information can be established and tracked by way of: lexical scope; compiler configuration for a given translation unit; and a function variant's selector information.

Lexical Scope. The OpenMP construct trait class described in Sect. 4.2 contains traits related to a number of OpenMP directives that establish lexical blocks with specific behavior. Conceptually, as each such directive is encountered in lexical order, the corresponding trait is added to the context. As the block for each directive closes, the corresponding trait is eliminated from the context. See Fig. 6 for an example of how contexts vary with the presence of OpenMP directives.

```
void main()
{
  // construct context = {} (i.e. empty)
  #pragma omp target
  {
    // construct context = {target}
    #pragma omp parallel
    {
      // construct context = {target,parallel}
      #pragma omp simd
      {
        // construct context = {target,parallel,simd}
      }
      // construct context = {target,parallel}
    }
    // construct context = {target}
  }
  // construct context = {} (i.e. empty)
}
```

Fig. 6. Example of how the construct context is changed upon entering and exiting lexically scoped OpenMP regions.

Traits in the OpenMP construct context may also be set implicitly, as the result of certain optimizations: regions that are automatically parallelized and/or vectorized without use of the corresponding OpenMP directives may add `parallel` and/or `simd` to the context; similarly, implementations may alter the context to reflect decisions taken as a result of certain descriptive constructs

(*e.g.* `concurrent`). Such transformations still imply a lexical scope, albeit one that is usually not exposed to the developer.

Translation Units. Many traits are not established by explicit directives that annotate lexical structures; the traits found in the device and OpenMP runtime trait classes (see Sect. 4.2) are generally established for a translation unit implicitly, by the compiler itself. The exact behavior of these traits will vary from compiler to compiler, but tracking them as part of the context is necessary for matching variants effectively.

For example, a user may specify options to a compiler instructing it to generate code for a specific instruction set or to optimize for the characteristics of a particular microarchitecture; in the presence of such flags, the `isa` and `uarch` traits should be defined appropriately in the context. Specific examples of how such compiler options can impact the device context are given in Fig. 7.

```
icpc -xMIC-AVX512:
device context = { uarch(knl), isa(avx512f, avx512er, avx512cd, ...) }

icpc -xCORE-AVX512:
device context = { uarch(skx), isa(avx512f, avx512cd, ...) }

gcc -msse2 -msse3:
device context = { isa(sse2, sse3) }

clang++ --cuda-gpu-arch=sm_70:
device context = { isa(sm_70) }
```

Fig. 7. Example of how the context is changed by compiler flags. Flags for enabling and configuring OpenMP are omitted.

Functions. Generally speaking, contexts are established within a function body without accounting for any surrounding contexts that hypothetical callers may establish, and the user should not assume that contextual information is passed across function boundaries. Our proposal makes two exceptions: a function's initial context may be modified by its variant selector; and compilers are free to broaden contexts when inlining. An example with both behaviors is shown in Fig. 8.

Variant Selectors. The context of a variant is defined to contain *at least* the traits specified in the variant's selector; additional traits may be present (defined for the translation unit) but only if they are compatible with the selector. This behavior allows for traits derived from lexical scope to be passed explicitly across translation unit boundaries.

```
void bar();

#pragma omp declare variant(bar) match(construct={teams})
void baz()
{
  // without inlining: construct context = {teams}
  //    with inlining: construct context = {parallel, teams}
  ...
}

void foo()
{
  // without inlining: construct context = {}
  //    with inlining: construct context = {parallel}
  #pragma omp teams
  {
    // without inlining: construct context = {teams}
    //    with inlining: construct context = {parallel, teams}
    baz();
  }
}

void main()
{
  #pragma omp parallel
  {
    foo();
  }
}
```

Fig. 8. Example of context propagation via variant selectors and/or inlining.

Inlining Behavior. It is common for a compiler to inline function bodies to satisfy user requests and to enable many optimizations. In such cases, the compiler may be able to supply additional context to the inlined function body based on where it has been inlined. While we would like to have consistent behavior of how OpenMP constructs should behave with respect to inlining, the OpenMP specification is not clear on this point and implementations vary in their interpretation. As such, developers should not *depend* on particular inlining behavior – since it is compiler-specific – but it does not introduce problems for our selection mechanism.

Selecting a Variant Based on Context. Given a calling context C and a variant family V, the variant selection algorithm proceeds as follows:

1. Eliminate all variants from V with selectors that are incompatible with C.
2. Compute a *specificity* score associated with each remaining selector.
3. If the most specific (highest scoring) selector is unique, return its variant.

A selector's specificity score is computed by assigning a value of 2^i to each context trait, where i reflects the trait's position in the context, and summing these values. Traits are ordered according to their class – `construct`, `device`, `implementation`, `user` – and level of nesting in lexical scope (if appropriate). If the most specific score is not unique, but the selectors can be ordered by

a strict subset/superset relationship of their properties, the selector with the largest superset should be chosen; otherwise, the choice between the most specific selectors is unspecified.

Figure 9 shows this algorithm applied to an example calling context and a family of six variants. First, the variants are compared with the calling context to assess their compatibility: the first two variants are eliminated in this step, as neither `target` nor `teams` is present in the calling context. Second, the traits

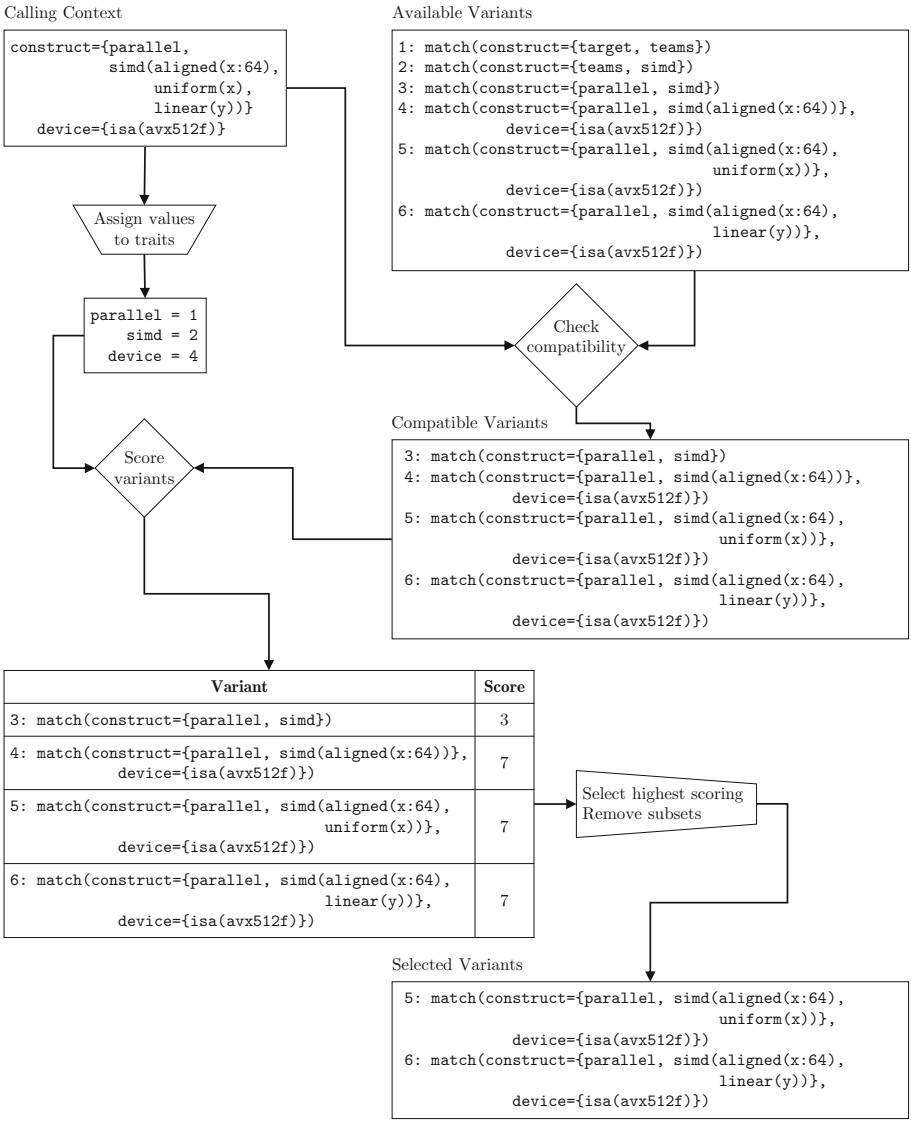

Fig. 9. Example of the variant selection algorithm.

are assigned a value according to their position in the calling context, and these values are used to assign specificity scores to the variants: the variants receive scores of 3, 7, 7 and 7 based on the values assigned to the `parallel`, `simd` and `device` traits. The last step selects the variants with the highest score and, from those, removes variants that are a subset of other variants: the variant with just `simd(aligned(x:64))` is eliminated, since it is included in the other two variants. The algorithm selects two variants: the `simd` properties for the selected variants contain `aligned(x:64)` as a common subset, but since `uniform(x)` and `linear(y)` cannot be ordered an implementation is free to choose either.

4.4 Relation to Existing Directives

The behavior of `declare variant` as defined in our proposal is orthogonal to the behavior of the `declare simd` and `declare target` directives: it provides a mechanism for associating user-provided variants to base functions, but does not provide a mechanism for requesting compiler-generated variants of base functions. At the time of writing, it is unclear whether or not consolidating these functionalities into a single directive is desirable. By design, extending `declare variant` to support more contextual information is easier than extending the existing directives, but deprecating existing functionality may break existing user code. Should it be decided that deprecating `declare simd` and `declare target` *is* desirable, then modifying our proposed syntax to support this could be as straightforward as making the base-function part of an optional clause (as shown in Fig. 10), or introducing a separate `create variant` directive (as shown in Fig. 11).

```
// Request compiler-generated variant of foo, specialized for simd context
// Base-function omitted; equivalent to "declare simd"
#pragma omp declare variant match(construct={simd})
void foo();

// Associate user-provided variant of foo, specialized for simd context
// Uses "implements" clause; equivalent to current syntax
#pragma omp declare variant match(construct={simd}) implements(foo)
void bar();
```

Fig. 10. Example of using a modified `declare variant` directive to replace `declare simd` and `declare target`.

```
// Request compiler-generated variant of foo, specialized for simd context
// Equivalent to "declare simd", but user provides name for generated function
#pragma omp create variant(foo_simd) match(construct={simd})
void foo();
```

Fig. 11. Example of using a new `create variant` directive to replace `declare simd` and `declare target`.

5 Summary

Application developers hoping to achieve high levels of performance on multiple platforms require a mechanism for selecting and executing different code paths based on properties of the current execution context. This paper proposes a set of extensions to the OpenMP API that provide such a mechanism, introducing the ability to perform function dispatch based on contextual information known to OpenMP at compile-time. The specific contributions of this work are as follows:

1. We review the complex interaction between modern OpenMP constructs and representative OpenMP devices, thus motivating the introduction of powerful and expressive developer tools for specializing code for different execution environments.
2. We propose a new directive, `declare variant`, for declaring variants of functions that should be preferentially selected under certain conditions. Our proposal unifies several previous proposals, and is designed to be easily extended to cover future additional functionality.

We have designed `declare variant` to ensure that the contextual information it supports can be extended as the OpenMP API evolves, and there are many exciting future directions to explore. Incorporating an ability for dynamic (run-time) dispatch is the most obvious: many performance-impacting variables in OpenMP can be chosen dynamically (*e.g.* number of threads, scheduling policies); users may wish to select different devices or algorithms based on properties of program input; and just-in-time (JIT) compilation for problem size has been demonstrated to significantly improve performance in some cases [5]. When considering this extension, it will be important to consider the cost of run-time selection and dispatch.

Acknowledgements. Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.
* Other names and brands may be claimed as the property of others.

References

1. Ansel, J., et al.: PetaBricks: a language and compiler for algorithmic choice. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, pp. 38–49. ACM, New York (2009)
2. Bell, N., Hoberock, J.: Thrust: a productivity-oriented library for CUDA. In: GPU Computing Gems Jade Edition, pp. 359–371. Elsevier (2011)
3. Duran, A.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process. Lett. **21**(02), 173–193 (2011)
4. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. **74**(12), 3202–3216 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing

5. Heinecke, A., Henry, G., Hutchinson, M., Pabst, H.: LIBXSMM: accelerating small matrix multiplications by runtime code generation. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, pp. 84:1–84:11. IEEE Press, Piscataway (2016)
6. Hoberock, J.: Technical specification for C++ extensions for parallelism. Technical report ISO/IEC TS 19570:2015, ISO/IEC JTC 1/SC 22 (2015)
7. Hornung, R.D., Keasler, J.A.: The RAJA portability layer: overview and status. Technical report LLNL-TR-661403, Lawrence Livermore National Laboratory (2014)
8. Intel Corporation: `vector_variant`. https://software.intel.com/en-us/node/523350
9. Lee, J., Petrogalli, F., Hunter, G., Sato, M.: Extending OpenMP SIMD support for target specific code and application to ARM SVE. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 62–74. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_5
10. OpenMP Architecture Review Board: OpenMP Application Programming Interface Version 4.0 (2013)
11. OpenMP Architecture Review Board: OpenMP Technical Report 6: Version 5.0 Preview 2 (2017)
12. Pennycook, S., Sewall, J., Lee, V.: A metric for performance portability. In: Proceedings of the 7th International Workshop in Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (2016)
13. Pennycook, S., Sewall, J., Lee, V.: Implications of a metric for performance portability. Future Gen. Comput. Syst. (2017). https://doi.org/10.1016/j.future.2017.08.007
14. Sewall, J.D., Pennycook, S.J., Duran, A., Tian, X., Narayanaswamy, R.: A modern memory management system for OpenMP. In: Proceedings of the Third International Workshop on Accelerator Programming Using Directives, WACCPD 2016, pp. 25–35. IEEE Press, Piscataway (2016)