Extending OpenMP to Support Automated Function Specialization Across Translation Units

Giorgis Georgakoudis(✉), Thomas R. W. Scogland, Chunhua Liao, and Bronis R. de Supinski

Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
{georgakoudis1,scogland1,liao6,desupinski1}@llnl.gov

Abstract. OpenMP's variant directives support specialization at compilation time using OpenMP context for portability or performance. This specialization is confined either to variants explicitly written by the code author, which can cross translation units, or to implicit context passed by the compiler. The implicit context allows a `metadirective` directive to choose a directive variant based on context or the compiler to optimize out runtime interactions. However, that implicit context only exists in a single translation unit (TU), either in a single compilation or with link-time optimization linking the set of TUs. In order to enable more optimization opportunities, we propose the `metavariant` directive, a new variant directive to define possible function specializations over a set of specified OpenMP contexts that are available across different translation units. The compiler lowers the definition of a `metavariant`-annotated function to different instances for each specified context. Calls of the function in different translation units use local OpenMP context for specialization, relying on the fact that the compiler will have generated appropriately mangled symbol names for those instances. Using a prototype source-to-source tool and a set of use cases, we evaluate our approach to observe a speedup of up to 30× with inter-procedural specialization versus no specialization, while simplifying and enhancing modular adaptation with modest user effort.

Keywords: OpenMP · Function Specialization · Translation Units

1 Introduction

Recent versions of OpenMP have added several directives that enable the application programmer to specify context-specific optimization hints and specializations as well as application-specific requirements. However, these mechanisms currently provide limited support across translation units and often include restrictions that the programmer must ensure that they are used consistently across them. Mechanisms to extend optimization opportunities across translation units and to enforce requirements across them would enable greater modularity in application source code while also preserving and, perhaps, simplifying implementation-specific optimizations.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
M. Klemm et al. (Eds.): IWOMP 2022, LNCS 13527, pp. 159–173, 2022.
https://doi.org/10.1007/978-3-031-15922-0_11

For example, the metadirectives and declare variant directives allow the application programmer to specify context-specific specialization. The use of OpenMP contexts allows these mechanisms to incorporate aspects of the OpenMP (and, through the `user` selector, application-specific) context. These contexts may arise in a different translation unit. However, metadirectives do not provide any mechanism for the application programmer to specify that the context-specific specialization is available to the calling context. Alternatively, declare variant directives allow specialized functions to be called, possibly from a different translation unit, based on the calling context. However, the programmer must fully specify the specialized function and the OpenMP implementation is not expected to exploit the information about the calling context that the directive provides.

Application programmers would benefit from OpenMP extensions that better support optimization and specialization across translation units. In this paper, we define the `metavariant` directive, a mechanism that enables the programmer to specify that the implementation should generate function variants that can be safely called from a specific context. When combined with the use of metadirectives and declare variant directives, the programmer can ensure that the function conforms to any requirements of the calling context and that it provides information about assumptions that may hold in it.

We evaluate the `metavariant` directive using a prototype source-to-source translation tool. We explore potential use cases of the directive that illustrate the benefits that it can provide in terms of programmability and in making optimization hints more useful across translation units. Our preliminary experiments demonstrate that it can yield substantial performance benefits.

2 Background

Variant directives, including `metadirectives` and `declare variant` directives, are major new features introduced in OpenMP 5.0 [3] to improve performance portability by enabling adaptation of OpenMP pragmas and user code at compile time. The basic idea of variant directives is to allow programmers to suggest a suitable code variant for a given OpenMP context, which includes traits that describe active OpenMP constructs, execution devices, functionality of an implementation, and user-defined conditions. While OpenMP 5.0 supported only matching compile-time conditions on traits, OpenMP 5.1 [4] extended that to include user-defined conditions resolved at runtime. A metadirective is a declarative directive that conditionally resolves to another executable or declarative directive by selecting from multiple directive variants based on traits that define an OpenMP context. The declare variant directive has similar functionality as the metadirective but selects a function variant at the call site based on context or user-defined conditions.

However, a major limitation of the current variant directives is that programmers need to specify both context and variant information (i.e., the mapping of context and variants) within the same translation unit. Currently, OpenMP does not include any direct language support to propagate context-variant mapping

information easily across multiple translation units for either specialization or facilitating compiler optimizations. Existing context-aware directives, such as `declare simd` and `declare target` have a narrow definition of context and provide limited specialization across translation units. The `declare variant` directive supports OpenMP traits for expanding the possible context specification but requires explicit user specification of the correspondence of variant function symbols to the matching context that specializes the base function symbol. Further, this correspondence is visible to callers of `declare variant` functions, to specialize their call sites, while function variants themselves are not required to include a notion of their corresponding context, thus lacking in using and propagating context in their definitions.

The following code snippet shows an example use of `metadirective`. An OpenMP compiler can readily generate two specializations (or variants) of the for loop based on the context-variant mapping information explicitly specified by the `metadirective`.

```
void foo (int* v1, int* v2, int*v3, size_t N)
{
  #pragma omp metadirective \
      when(target_device={arch(nvptx)}: \
        target teams distribute parallel for \
       map(to:v1[0:N],v2[0:N]) map(from:v3[0:N]) ) \
      otherwise(parallel for)
  for (int i= 0; i< N; i++)
    v3[i] = v1[i] * v2[i];
}
```

The function `foo()` may be defined in one source file while many of its call sites are located in other source files. A compiler may better optimize the function (such as generating more or less specializations) if it knows all possible contexts within which the function will be called, when compiling the source file with the function definition. Similarly, a compiler may invoke the right function variant at each call site, if it knows all available variants of the function definition when compiling a source file containing the call site.

For example, if a compiler knows that all call sites of `foo()` will be within a parallel region, it may specialize the function definition to implement only `omp for` and to avoid generating code for nested parallelism. A CPU implementation involves relatively little difference between these choices other than the fork and join overhead. However, the difference on a GPU though can be between running the entire region directly on the native parallel threads, and being forced to use a heavy-weight concurrent state machine to implement varying levels of parallelism as the kernel progresses. The difference in performance between the "lightweight" native runtime and the state-machine can be orders of magnitude in some cases, even for otherwise identical code.

While in some cases, the compiler can perform LTO (Link-Time-Optimization) to propagate the OpenMP context and code variant information across

translation units, LTO for most heterogeneous platforms is currently prohibitively expensive. Enabling LTO may entirely serialize the process of compilation of large-scale code bases. This cost is too high in practice as many large scientific applications already take hours to build while being compiled in parallel. Another limitation of compiler-based solutions is that statically computing the caller-callee relationship (such as the one used by Interprocedural Optimization or IPO) is still a challenging problem when facing complex uses of function pointers and dynamic dispatch. Yet another approach is to detect activated context at runtime and to trigger runtime code specialization. However, this approach requires the implementation to generate code variants at runtime, with potentially significant runtime overhead.

Therefore, OpenMP needs to allow programmers to communicate the context-variant mapping information across multiple translation units explicitly. An implementation could then automatically exploit such semantics to optimize code generation at compile time, without relying on sophisticated program analyses or incurring runtime overhead. While we could introduce such functionality through significant re-definition of the semantics and specification of `declare variant`, we choose to introduce a new directive, named `metavariant`, both for exposition and because it is cleaner and more concise than retrofitting it to `declare variant`.

3 The `metavariant` Directive

We introduce a new directive, the `metavariant` directive, to enable compile-time function specializations across translation units, by matching the OpenMP context that propagates at call sites of those functions. For the translation unit that contains the definition of a metavariant-annotated function, compilers generate OpenMP context specializations of the function by specializing `metadirective` directives in the function's body, assuming a specific OpenMP context is matched. Correspondingly, at the translation units containing metavariant function callers, OpenMP compilation tracks the OpenMP context at call sites of the metavariant function to call the function specialization that matches the context, if any, otherwise falling back to the function specialization that does not assume a specific context. Since the metavariant function and its caller functions may be in different translation units, we propose *semantic function symbol naming* to encode the different function specializations, so that OpenMP compilation infers the function specialization symbol to call at call sites by using the OpenMP context and the original function symbol.

In more detail, the syntax of the `metavariant` directive is:

```
#pragma omp metavariant [clause,[[,] clause]...] new-line
  function definition or declaration
```

where *clause* is the following:

```
#pragma omp metavariant \
    match(construct={parallel})
void foo(int* v1, int *v2, int* v3, size_t N)
{
  #pragma omp metadirective \
    when(construct={parallel}: taskloop) \
    otherwise(parallel for)
  for (int i= 0; i< N; i++)
    v3[i] = v1[i] * v2[i];
}
```

Fig. 1. An example translation unit with the definition of a metavariant function

```
match(context-selector-specification)
```

Semantically, a `metavariant`-annotated base function has one specialization associated with each `match` clause and this specialization assumes the context specified in the clause is in effect. Thus, the specified context of this specialization will forward to `metadirective` directives in the function definition or calls to other `metavariant` functions, which will be specialized by this propagating context. Also, the base function without specialization remains available without assuming any specific OpenMP context. The symbol name of each function specialization is determined from base language rules extended by a string determined by the effective context selector of its associated `match` clause. The symbol name of the function without specialization is determined from base language rules without any string extension.

The function specialization variant is determined at the call site depending on its OpenMP context. If the OpenMP context at the call site matches the context of a specialization, the call is replaced with a call to the function variant of this specialization. Otherwise the call is not replaced, thus resolving to the original function, which is the no-specialization variant.

To make use of a `metavariant` function across translation units, a function prototype with its corresponding `metavariant` directive can be put into a header file, which is then included by other translation units. Symbol names of specializations provide the ABI contract that supports this functionality.

In summary, on a metavariant function definition, metavariant-enabled OpenMP compilation generates different functions for the different context specification in each `match` clause, specializing any metadirectives, calls declare variant functions, or to other metavariant functions in the function body by forwarding the matching context. Each different function variant follows an ABI convention that encodes context-awareness across translation units. On metavariant function declarations, metavariant-enabled OpenMP compilation generates the variant function declarations following the same ABI convention and transforms call sites of the metavariant function to use the function variant that matches the context at the call site.

```
// Variant matching execution in a parallel context.
void foo_construct_parallel(int* v1, int *v2, int* v3, size_t N)
{
#pragma omp taskloop
  for (int i= 0; i< N; i++)
    v3[i] = v1[i] * v2[i];
}
// Variant matching execution in sequential context.
void foo(int* v1, int *v2, int* v3, size_t N)
{
#pragma omp parallel for
  for (int i= 0; i< N; i++)
    v3[i] = v1[i] * v2[i];
}
```

Fig. 2. Lowering of the translation unit with the metavariant function definition

Figure 1 illustrates the definition of a `metavariant` function named `foo`, performing a simple vector addition that specializes the execution of its loop depending on whether it is called within a parallel or a sequential context. Specifically, `foo` uses a metadirective to use a `taskloop` construct for parallel execution of the loop when it is called from a parallel context, whereas the metadirective specifies `parallel for` for parallel work-sharing execution when it is called from a sequential context. Figure 2 presents a source-to-source lowering of the metavariant function, using symbol naming that uniquely identifies different specializations corresponding to different metavariant contexts, including flattening of metadirectives in the function's body.

Figure 3 also shows how the metavariant function is declared and called in a different translation unit. The source code must include the prototype of the metavariant-annotated function declaration to declare its possible specializations. The caller function `bar` calls the metavariant function using its declaration symbol (`foo`). OpenMP compilation replaces this symbol at each call site with the symbol name that corresponds to the metavariant function specialization that matches the enclosing OpenMP context of the call site. Figure 4 shows a source-to-source lowering of the translation of a caller to a metavariant function. The metavariant function declaration resolves to two function declarations, one matches the parallel context specialization assuming the same function symbol in the callee's translation unit in Fig. 2, the other matches the non-parallel context specialization with the function symbol unchanged.

3.1 Discussion

In this section we discuss several aspects of metavariant-enabled compilation and its relation to other compilation and specialization approaches.

JIT/LTO Context Propagation and Specialization. Using LTO for context propagation across translation units is an interesting proposition assuming some

```
#pragma omp metavariant \
    match(construct={parallel})
void foo(int* v1, int *v2, int* v3, size_t N);

void bar(int* v1, int *v2, int* v3, size_t N)
{
  // Calling foo() in a parallel context.
  #pragma omp parallel
  {
    // do other parallel work.
    // Call foo() by main thread.
    if (omp_get_thread_num() == 0)
        foo(v1, v2, v3, N);
  }

  // Calling foo() in a sequential context.
  foo(v1, v2, v3, N);
}
```

Fig. 3. An example translation unit with a call site of a metavariant function

mechanism conveys context information from existing context-aware directives, such as `declare variant`, to both callers and callees. This alternative requires that LTO completely re-constructs the call graph to couple it with context information. However, this alternative can be problematic since LTO compilation is time consuming and also necessitates shared libraries to be amenable to it. Context-aware JIT compilation is also a possibility, however the cost of runtime compilation and tracking context may be prohibitive. The metavariant approach avoids those issues by providing an elegant way to specify and to propagate context at compile time, to generate specializations through automatic code generation, and proposing a context-dependent ABI convention for cross-translation unit specialization. It does so without relying on LTO or JIT compilation, which may be unavailable or undesirable due to their limitations.

Consumers of Context for Specialization. In this formulation of the metavariant directive possible consumers of context for specialization include metadirectives and calls to declare variant or other metavariant functions, propagating context, in the metavariant function's definition. While this approach gives explicit control to the user, a specification-based rule set for automatic transformations of OpenMP directives can avoid pathological use-cases, such as nesting parallel regions shown in the example of Fig. 3. Such rule sets can also be used for error checking to emit warning/errors when incompatibilities or performance degradation occurs between the caller and the assumed callee context of a metavariant function. We leave that as interesting future work.

Differences with `declare variant` Specialization. The `declare variant` directive explicitly specifies the symbol of function variant specializations of a

```
void foo_construct_parallel(int* v1, int *v2, int* v3, size_t N);
void foo(int* v1, int *v2, int* v3, size_t N);

void bar(int* v1, int *v2, int* v3, size_t N)
{
  // Calling foo() in a parallel context.
  #pragma omp parallel
  {
    // do other parallel work.
    // Call the parallel context specialization of foo().
    if (omp_get_thread_num() == 0)
        foo_construct_parallel(v1, v2, v3, N);
  }
  // Call the non-parallel specialization of foo().
  foo(v1, v2, v3, N);
}
```

Fig. 4. Lowering of the translation unit with a call site of a metavariant function

base function that correspond to different matching contexts. Calls to the base function symbol are specialized to the function variant that matches the caller's context. By the specification, context cannot be not assumed to propagate to the function variant. Thus, the user must explicitly implement any specialization in the definition of the function variant symbol without assuming any context is propagated during compilation for compiler-based specialization, e.g., through metadirectives. By contrast, metavariant function variants are context-aware, auto-generated during compilation using compiler-based code generation for specialization by propagating context to existing variant directives (metadirectives, declare variant) or to other metavariant functions called by the variant. The base function definition of a metavariant function is a fallback that does not assume any OpenMP context. We purposefully avoid providing user-visible naming of function variants in the metavariant directive, relying instead on an ABI convention. The functionality of calling a function variant without requiring explicitly naming the function symbol is possible through a metadirective, using a `when` clause with the matching context. Also, exposing function variants to users is error prone, since users could call them within incompatible caller contexts.

Tracking Context. The example specialization of Fig. 1 uses `taskloop` for parallelizing loop execution when called within a parallel context instead of a work-sharing construct. This choice is necessary to avoid possibly incompatible nesting of work-sharing constructs. Implementors of metavariant functions should be aware of such limitations to implement compatible specializations or context specification should be enhanced to include extended traits, such as work-sharing execution. The metavariant directive proposal opens up this discussion, which we leave as future work.

4 Evaluation

4.1 Experimentation Setup

We experimented on a computing node of the Lassen cluster at Lawrence Livermore National Laboratory. The node is equipped with IBM Power9 processors (2 sockets $\times$ 20 user-level usable cores), 256 GB of main memory, and four NVIDIA Tesla V100 GPUs with 16 GB of device memory each. We build a source-to-source transformation tool in Python that parses `metavariant` annotations and lowers metavariant function definitions, declarations and call sites, tracking the OpenMP context, similarly to the way presented in the examples of the previous section. The compiler used for generating executables from the lowered sources is Clang/LLVM version 13.0.1.

Our experimentation includes three use cases: (1) a use case that avoids nested parallelism; (2) a use case that specializes a metavariant function for concurrent parallel host execution and offloading execution; and (3) a use case that avoids nesting target regions with unspecified behavior.

The example programs with which we experiment invoke a reasonably optimized blocked GEMM kernel in single precision, using square matrix inputs. The kernel is implemented as a metavariant function, specialized depending on the propagated caller's OpenMP context. For each experiment we perform 10 trials of each configuration to present the mean execution time. Any confidence intervals shown correspond to 95% confidence level. We also ensure that experiments run on an exclusively allocated node, using threads pinned to the physical cores of the machine to reduce variability.

4.2 Using Metavariant to Avoid Nested Parallelism

Figure 5 presents this use case in pseudo-code. The `main` function in the driver translation unit (Fig. 5a) is the caller of the `gemm` function, which implements the matrix multiplication in another translation unit. The driver emulates calling `gemm` within a parallel context, masked to execute by the main thread. There are three versions of the `gemm` function: (1) `gemm` is a `metavariant` function (Fig. 5b) that specializes at compile time to `taskloop` execution, when called in a parallel context, or to a `parallel for` work-sharing construct when called in a sequential context; (2) `gemm` is specialized at runtime (Fig. 5c) to use either `taskloop` or `parallel for` depending on the result of the runtime call `omp_in_parallel()` which dynamically detects a parallel context; (3) `gemm` is not specialized (Fig. 5d), nesting instead a parallel region within masked execution that results in sequential execution within the main thread. We omit the metavariant function declaration in the driver (when applicable) since the function prototype is the same as in the metavariant function definition. The runtime mode replicates the functionality of the metavariant mode using a dynamic context selector in a metadirective by checking the result of `omp_in_parallel`. However, this specialization is limited to the context information available through runtime calls, in contrast to the much more widely available context specification available through OpenMP

```
int main() {
// Init
#pragma omp parallel
  {
   #pragma omp masked
    gemm(···)
  }
}
```

(a) Caller

```
#pragma omp metavariant \
 match(construct={parallel})
void gemm(···) {
#pragma omp metadirective \
  when(construct={parallel} : \
    taskloop collapse(2)) \
  otherwise (parallel for collapse(2))
  for(···) {}
}
```

(b) Metavariant

```
void gemm(···) {
  if (omp_in_parallel())
    #pragma omp taskloop \
       collapse(2)
    for(···) {}
  else
    #pragma omp parallel for \
      collapse(2)
    for(···) {}
}
```

(c) Runtime specialization

```
void gemm(···) {
  #pragma omp parallel for \
    collapse(2)
  for(···) {}
}
```

(d) Nested parallel region (no specialization)

Fig. 5. A use case that avoids nested parallelism

traits. Further, dynamic context selectors require extra conditionals generated at compile time to select the specialization variant, which add to the overhead of the runtime call. Metavariant avoids those overheads while also providing a more powerful specialization mechanism through context forwarding that propagates context to both metadirectives and calls to other metavariant functions.

Figure 6 shows the execution time of the GEMM kernel for different matrix dimensions in the different modes of execution. The metavariant mode, denoted as *meta*, and the runtime mode, denoted as *runtime*, both avoid the nested parallel region. The serialized, nested parallel region execution mode, denoted as *nometa*, is the slowest. The performance of *meta* and *runtime* is comparable, with *meta* being slightly faster by avoiding the runtime call to `omp_in_parallel()`, around 5% on average.

4.3 Using Metavariant for Concurrent CPU or GPU Execution

Figure 7 shows the pseudo-code of the second case where the program performs a batch of matrix multiplications of different sizes. The main program, in Fig. 7a, processes the batch within a parallel work-sharing loop. Based on a threshold of the matrix sizes, the loop issues a matrix multiplication to execute on the

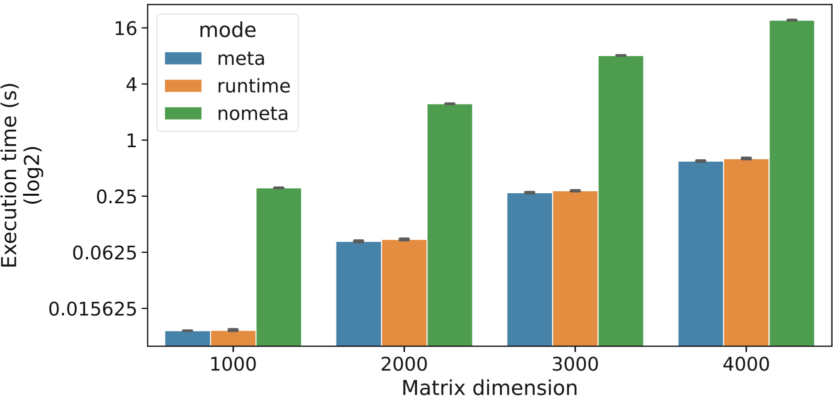

Fig. 6. Execution time for GEMM over various matrix dimensions using the metavariant or runtime to avoid serialized nested parallel regions

CPU or to the GPU, through the metavariant function `gemm` specialization for different contexts (shown in Fig. 7b) including a parallel context, a target context, or defaulting to a parallel work-sharing construct (does not apply in this use case). Adding a `nowait` clause to the target region of Fig. 7a could enable more concurrency through asynchronous execution. However, we did not observe a noticeable performance difference when we used it.

Besides concurrent execution on both CPU and GPU, we experiment with executing on the GPU only, setting the threshold to 0, or the CPU only (by setting the threshold above the largest dimension in the batch, i.e., 4000. Figure 8 shows the results. CPU-only execution is the slowest, as expected, since for larger matrices the speedup from CPU parallelization reaches its limits. GPU-only execution is much faster, about 5×. Concurrent CPU and GPU execution performs even better, by about 3%, compared to GPU-only execution by utilizing both processing elements.

4.4 Using Metavariant to Avoid Nested Target Regions

Figure 9 presents a use case in which the metavariant is used to avoid nesting target regions. Nesting a target construct within a target region, without providing the `ancestor` modifier for reverse offloading to the host, results in unspecified behavior [4]. The metavariant definition of `gemm()` avoids this issue, by specializing to use the `distribute` construct, within a target region, otherwise defaulting to using a target construct for offloading.

We experiment by running the GEMM kernel for various matrix dimensions, using either the target context specialization by executing within a target region, or by using the default execution through a target construct. Figure 10 shows the resulting execution times. Execution through the target context specialization is denoted as *target*, while execution outside a target region using a target construct is denoted as *default*. Execution times are not significantly different.

```
int main() {
 int size[BATCH_SIZE] = {
   100, 200, 400, 800,
   1000, 2000, 3000, 4000 };
 #pragma omp parallel for
 for(i=0; i<BATCH_SIZE; ++i) {
   int N = size[i];
   if (N < /*THRESHOLD=*/1000)
      // CPU taskloop variant
      gemm(batch[i])
    else
    #pragma omp target teams \
      map(to:A[i][0:N*N]) \
      map(to:B[i][0:N*N) \
      map(tofrom:C[i][0:N*N])
      // GPU distribute variant
      gemm(batch[i]);
  }
}
```

(a) Caller

```
#pragma omp metavariant \
  match(construct={parallel}) \
  match(construct={target})
void gemm(···) {
  #pragma omp metadirective \
    when(construct={parallel}:\
      taskloop collapse(2)) \
    when(construct={target}: \
      distribute collapse(2)) \
    otherwise (parallel for \
      collapse(2))
  for(···) {}
}
```

(b) Metavariant

Fig. 7. A use case of concurrent CPU and GPU execution

However, the combined construct of *default* shows higher performance as the matrix dimension grows, ranging between 10% to 18%, compared to the nested `distribute` construct within the target context. Observing the generated LLVM IR, we notice more aggressive optimization in the combined construct case. We plan to investigate further compilation differences due to specialization and especially how to integrate context information in the metavariant specification for additional inter-procedural optimization during compilation.

5 Related Work

Many related research efforts extend OpenMP for better productivity, portability, and performance, especially in the context of programming heterogeneous architectures. We only name a few examples in this section.

A popular approach to achieving portable performance of OpenMP is through autotuning. One early study [1] leveraged source code outlining to extract OpenMP loops from large-scale scientific applications and subsequently enable the tuning of different OpenMP execution parameters. Similarly, Sreenivasan et al. [7] proposed a lightweight autotuner of OpenMP pragmas to optimize OpenMP execution parameters such as scheduling policies, chunk sizes, and thread counts.

To study the benefits of specialization mechanisms introduced in OpenMP 5.0, Pennycook et al. [6] used the miniMD benchmark from the Mantevo suite to investigate how `metadirective` and `declare variant` may impact real-life

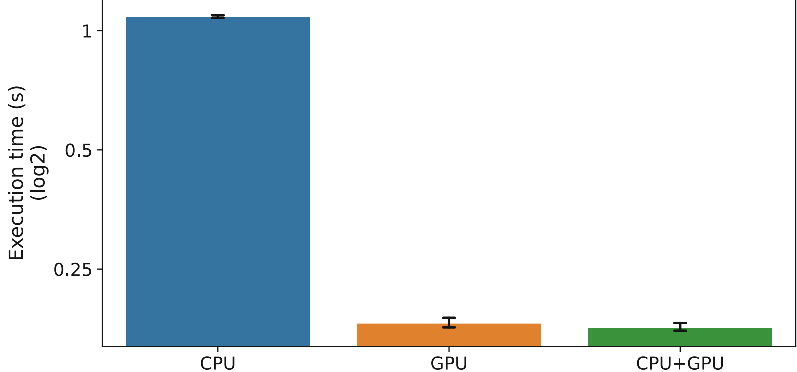

Fig. 8. Execution time of the matrix multiplication batch.

```
int main() {
 #pragma omp target teams \
   map(to: A[0:N*N]) \
   map(to: B[0:N*N) \
   map(tofrom: C[0:N*N])
   gemm(···);
}
```

(a) Calling metavariant function within a target region

```
#pragma omp metavariant \
  match(construct={target})
void gemm(···) {
#pragma omp metadirective \
 when(construct={target} : \
 distribute collapse(2)) \
 otherwise (target teams distribute \
 collapse(2) map(tofrom: C[0:M*N])) \
 map(to: A[0:M*K], B[0:K*N])
  for(···) {}
}
```

(b) Metavariant

Fig. 9. A use case for avoiding nested `target` regions

codes. They reported that these features allowed a more compact source code form to express code variants, resulting in a performance portability of 59.35% across CPU and GPU hardware. While `metadirective` allows user-guided runtime adaptation as proposed by Yan et al. [8], having users define portable conditions across different hardware platforms is impractical. To address this limitation, Liao et al. [2] proposed a `declare adaptation` directive that enables automatic model-driven runtime adaptation, by integrating machine learning techniques into OpenMP compiler and runtime systems. The directive allows programmers to express semantics related to the desired type of machine learning model, the input parameters to the model and the ranges of the parameters.

In order to avoid writing different directives for different devices, Ozen and Wolfe [5] proposed a descriptive model with a new OpenMP loop directive to demonstrate how a compiler implementation could automatically decide target-specific parallelization for multiple devices based on a single directive. In their work, they exploited the parallelism semantics associated with the loop construct

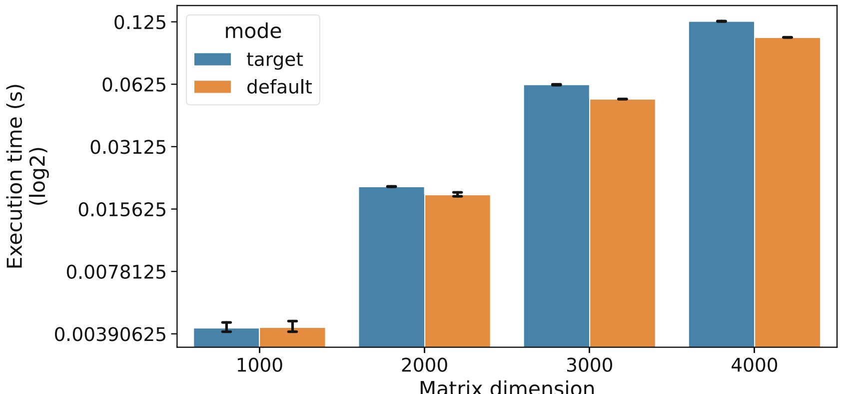

Fig. 10. Executing on GPU directly or using target context specialization

and also used dependence analysis to discover more parallelism. Their evaluation showed that 60% fewer directives were required for the SPEC ACCEL benchmark suite while yielding competitive performance compared to other compilers.

While it is not a significant point in that paper, the solution proposed is heavily influenced by OpenACC, which provides the `acc routine` directive to provide context information on parallelism across translation units. That feature is highly related to ours, but differs in that the `acc routine` directive states the levels of parallelism that a function intends to consume where metavariant states the set of possible contexts from which a function may be called and, thus, for which it should be specialized. The context specification of metavariant is a superset of the `acc routine` parallelism-only context specification. Also, metavariant supports specialization through metadirectives or calls to other metavariant functions in the metavariant function's body by propagating context, whereas `acc routine` designates functions for device compilation using parallelism-level information for error checking. Context matching is explicit in metavariant, by contrast, `acc routine` defines an implicit rule set to determine whether parallelism levels are composable. Interesting future work for the metavariant specification includes exploring specification-based implicit rule sets both for automated transformations and error checking.

6 Conclusion

In this paper, we propose to extend OpenMP to support automated function specialization across translation units by providing the `metavariant` directive. The `metavariant` directive explicitly communicates information about calling context and specializations between call sites and function definitions residing in different source files. Using a source-to-source prototype tool and a set of use cases, we have shown the feasibility and benefits of this extension.

In the future, we plan to extend other directives (such as `assume` and `requires` directives) to support optimization across translation units further. We will explore fully automated generation of function specialization without relying on `metadirective`. We will also expand the evaluation by using a production quality implementation based on LLVM and comparing it to alternative approaches such as LTO and runtime specialization, combined with more use cases running on more platforms. Further, we will propose this extension for upcoming versions of the OpenMP specification by drafting a more rigorous functional specification and expanding the use cases using our robust LLVM implementation.

Acknowledgments. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-837685), partially supported by the LLNL-LDRD Program under Project No. 21-ERD-018 and by the U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Program (ASCR SC-21).

References

1. Liao, C., Quinlan, D.J., Vuduc, R., Panas, T.: Effective source-to-source outlining to support whole program empirical optimization. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 308–322. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13374-9_21
2. Liao, C., et al.: Extending OpenMP for machine learning-driven adaptation. In: Bhalachandra, S., Daley, C., Melesse Vergara, V. (eds.) WACCPD 2021. LNPSE, vol. 13194, pp. 49–69. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-97759-7_3
3. OpenMP Architecture Review Board: OpenMP Application Programming Interface 5.0, November 2018. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
4. OpenMP Architecture Review Board: OpenMP Application Programming Interface 5.1, November 2020. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf
5. Ozen, G., Wolfe, M.: Performant portable OpenMP. In: Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction, pp. 156–168 (2022)
6. Pennycook, S.J., Sewall, J.D., Hammond, J.R.: Evaluating the impact of proposed OpenMP 5.0 features on performance, portability and productivity. In: 2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 37–46, November 2018. https://doi.org/10.1109/P3HPC.2018.00007
7. Sreenivasan, V., Javali, R., Hall, M., Balaprakash, P., Scogland, T.R.W., de Supinski, B.R.: A framework for enabling OpenMP autotuning. In: Fan, X., de Supinski, B.R., Sinnen, O., Giacaman, N. (eds.) IWOMP 2019. LNCS, vol. 11718, pp. 50–60. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-28596-8_4
8. Yan, Y., Wang, A., Liao, C., Scogland, T.R.W., de Supinski, B.R.: Extending OpenMP Metadirective semantics for runtime adaptation. In: Fan, X., de Supinski, B.R., Sinnen, O., Giacaman, N. (eds.) IWOMP 2019. LNCS, vol. 11718, pp. 201–214. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-28596-8_14