Extending the LLVM/Clang Framework for OpenMP Metadirective Support

Alok Mishra
Stony Brook University
Stony Brook, NY, USA
alok.mishra@stonybrook.edu

Abid M. Malik
Brookhaven National Laboratory
Brookhaven, USA
amalik@bnl.gov

Barbara Chapman
Stony Brook University
Stony Brook, NY, USA
barbara.chapman@stonybrook.edu
Brookhaven National Laboratory
Brookhaven, USA

Abstract—OpenMP 5.0 introduces many new directives to meet the demand of emerging high performance computing systems. Among these new directives, the metadirective and declare variant directives are important to control the execution behavior of a given application by compile-time adaptation based on the OpenMP context. The metadirective directive allows the selection of OpenMP directives based on the enclosing OpenMP context as well as on user-defined conditions. The declare variant directive declares a specialized variant of a base function and specifies the context in which that specialized variant is used. Support for these directives are available in few compilers with some limitations.

Support for metadirective is not available in Clang. This paper presents our implementation of the metadirective directive in Clang. In this paper, we present an implementation which supports the OpenMP 5.0 metadirective specification. However, in addition, this work also implements a dynamic extension to the user-specified conditions. A dynamic evaluation of user-defined conditions provides programmers more freedom to express a range of adaptive algorithms that improve overall performance of a given application. For example, a program can estimate the cost of execution, with respect to time taken or energy consumption, of a kernel based on some dynamic or static variables and decide whether or not to offload the kernel to GPU using the metadirective. Since there is a significant lack of knowledge about the usage and performance analysis of metadirective, the work also studies its impact on application characteristics.

To achieve this, we have modified several benchmark codes in the Rodinia benchmark suite. The Rodinia benchmark includes applications and kernels which target multi-core CPU and GPU platforms which helps programmers study the emerging computing platforms. Our modification to the Rodinia benchmarks enables the application developer to study the behavior of metadirective. Our analysis reveal that the main advantage of the dynamic implementation of metadirective is that it adds minimal to no overhead to the user application, in addition to allowing flexibility to the programmers to introduce portability and adaptability to their code. Our modification of the Rodinia benchmark suite provides several guidelines for programmers to achieve better performance with metadirective.

Index Terms—OpenMP 5.0, Metadirective, LLC, Clang, Dynamic context.

I. INTRODUCTION

To meet the demand of productive parallel computing on existing and emerging high performance computing systems, the OpenMP standard has evolved significantly in recent years [1]. Since the creation of the standard in 1997, that specified a handful of directives, substantial amount of new constructs have been introduced and most existing APIs have been enhanced in each revision. The latest version of OpenMP 5.0, released in 2018, has more than 60 directives [2]. Variant directives is one of the major features introduced in OpenMP 5.0 specification to facilitate programmers to improve performance portability. These directives can enable adaptation of OpenMP pragmas and user code at compile and runtime. The OpenMP specification defines traits to describe active OpenMP constructs, execution devices, functionality provided by an implementation, and context selectors based on the traits and user-defined conditions. It also defines variant directives like metadirective and declare variant which uses context selectors to choose various directives or functions.

- The metadirective directive is an executable directive that conditionally resolves to another directive at compile time by selecting from multiple directive variants based on traits that define an OpenMP condition or context.
- The declare variant directive has similar functionality as metadirective but selects a function variant at the call-site based on context or user-defined conditions.

The mechanism provided by these directives for selecting variants is more convenient to use than the C/C++ pre-processing since it directly supports variant selection in OpenMP and allows an OpenMP compiler to analyze and determine the final directive from variants and context as shown in Figure 1.

However, OpenMP 5.0 restricts context traits to be completely resolvable at compile time. This constrains the potential to optimize an OpenMP application based on runtime behavior or input data. An extension to allow runtime adaptation, based on properties like system architecture or input characteristics, may increase the efficiency of an application. Applications that would benefit from this feature include those that use traits based on problem size, loop count, and the number of threads. For example, most libraries parallelize and optimize matrix multiplications depending on the sizes of the input matrix. In this work we explore the possibility of extending metadirective user-condition to dynamically resolve the context selector based on runtime attributes.

The high performance computing community has been exploring novel platforms to push performance forward [3].
Field Programmable Logic Arrays (FPGAs) and Graphics Processing Units (GPUs) have been widely used as accelerators for computational intensive applications. The heterogeneous cluster is one of the most promising platforms as it combines characteristics of multiple processing elements to meet the requirements of various applications. In this context, a self-adaptive framework for heterogeneous clusters would be an ideal solution [4]. OpenMP context based directives like metadirective can provide an excellent solution to build such frameworks.

Current compilers do provide implementation of OpenMP 5.0 with some limitations, but as far as we know only the ROSE [5] source-to-source compiler infrastructure has partial implementation of metadirective [6]. Other compilers which claim active development of OpenMP 5.0 specifications are:

- **Cray** – The latest CCE 10.0 compiler has partial support for OpenMP 5.0 [7]. However, the list of enhancements claimed by the compiler does not include the metadirective.
- **GNU** – GCC 10 [8] adds a number of newly implemented OpenMP 5.0 features on top of the GCC 9 release such as conditional lastprivate clause, scan and loop directives, order(concurrent) and use_device_addr clauses, if clause on simd construct or partial support for the declare variant directive, and they are getting closer to the complete OpenMP 5.0 standard. However, metadirective is not implemented yet.
- **Intel** – The Intel compiler provides most of the OpenMP Version Technical Report 4, version 5.0 features [9]. Their implementation of OpenMP 5.0 specification is still incomplete and metadirective is not included yet.

The Clang project provides a language front-end and tooling infrastructure for languages in the C language family (C, C++, Objective C/C++, OpenCL, CUDA, OpenMP, and Render-Script) for the LLVM project [10]. Clang is evolving rapidly under the Exascale Computing Project’s (ECP) SOLLVE project [12] and now supports many OpenMP 5.0 features that are essential for ECP applications. These features do not include metadirective yet and to the best of our knowledge, our work is the first one which successfully add OpenMP 5.0 specified metadirective to the Clang framework. In addition to this, we also implement a dynamic extension to the user-specified conditions (detailed discussion in Section III-E). The main contributions of our work are:

- Enhancing Clang capability by implementing OpenMP’s metadirective directive in the LLVM framework.
- The implementation supports the compile time selection of hardware support and compiler implementations for code portability across different architectures.
- The work also presents a novel implementation of the semantics for user-defined contexts to enable runtime directive selection using LLVM.
- We also modified the Rodinia benchmarks to study the impact of the directives. These benchmarks can provide a guide line to end users to help them apply these features in real applications.

The remainder of this paper is organized as follows. Section II presents the motivation for using metadirective and hence the need for it to be supported in compilers, especially in Clang. Section III gives details of the implementation in Clang. Section IV introduces our experimental setup and Section V provides, and analyzes, the results of our experiments. Section VI gives related work. Finally, Section VII presents our conclusion and future directions.

### II. Motivation

OpenMP is an integral part of many important HPC codes that rely on hybrid programming models (e.g. MPI + OpenMP). Therefore, tuning an OpenMP code to get better per-node performance for a given resources is an important
research problem. In this section, we explain how the metadirective is relevant for tuning resource-constrained OpenMP applications. Whether the OpenMP metadirective is beneficial depends on the response to the following questions:

- Does the best configuration for a given OpenMP region remain the same across different resource levels and workloads?
- Does the performance gain due to the best configuration persist across all configurations?

To answer these questions, we did some experiments with the BT benchmark [13]. We took an OpenMP region from the BT benchmark application and ran it with different power levels or power caps using different numbers of threads, scheduling policies, and chunk sizes (150 different configurations). The region studied belongs to the xsolve function, and has coarse grain parallelism, i.e., the outermost loop is parallelized. Figure 2 compares the execution time using the best identified configuration with the default configuration at different power levels. The default configuration uses the maximum number of available threads, static scheduling, and chunk sizes calculated dynamically by dividing the total number of loop iterations by the number of threads. The figure clearly shows that the best configuration is different from the default configuration at all power levels.

It also shows that the best configuration improves the execution time of the parallel region by up to 20% compared to the default configuration at the same power level. Also, we can see that the best configuration at a lower power level gives better execution time performance than the default configuration with maximum power level prescribed by the manufacturer or Thermal Design Power (TDP). For example, the best configuration at a 70W power cap improves execution time by almost 9% in comparison to the default configuration at TDP (115W in our case).

We also experimented with OpenMP regions from other NAS Parallel benchmark applications [13] using different runtime configurations. We observed that a significant number of the OpenMP regions showed similar behavior. We observed these OpenMP regions to have poor load balancing and cache behavior with the default configuration. We also saw that these poor behaviors persist across different power levels and workloads for these kernels with the default configuration. As a result, irrespective of power level or workload size an optimal configuration always shows consistent performance improvement compared to the default configuration for these kernels. However, we observed that the optimal configurations for these kernels change across different power levels and workloads.

In the future HPC facility, the application workload may change dynamically. If the facility is operating under a resource constraint, the resource manager may add or remove nodes and adjust their power level/frequency dynamically. To get the best per-node performance for a given resource constraint, the runtime configurations need to be changed dynamically. Our proposed OpenMP metadirective extension of dynamically selecting user-condition gives the user the ability to achieve this efficiently with minimal code change.

The other strong motivation for metadirective is an efficient portability across different hardware architectures. The key application kernels to be ported on a given device need to be coded specific to the device's architecture for the best performance gain. The metadirective gives the user ability to write multiple kernels, each specific for different devices, and the compiler then selects the required kernel during the compilation phase based on the available back-end device. Figure 3 explains this more clearly. The implementation selector set is specified in the when clause to distinguish between AMD and NVIDIA platforms. Additionally, specific architectures are specified with the device selector set. In the

```c
for (idev=0; idev<omp_get_num_devices(); idev++)
#pragma omp target device(idev)
#pragma omp metadirective
    when( implementation={vendor(nvidia)}, device={arch("kepler")}):
      teams num_teams(512) thread_limit(32) :
    when( implementation={vendor(amd)}, device={arch("fiji")}):
      teams num_teams(512) thread_limit(64) :
    default (teams)
#pragma omp distribute parallel for
for (i=0; i<N; i++) work_on_chunk(idev,i);
```

![Fig. 2](image1.png) Execution time comparison for the x solve region of BT using different OpenMP runtime configurations at different power levels. Smaller value is better. The function was run on Intel Sandy Bridge [4].

![Fig. 3](image2.png) Compile time selection of hardware support.
code, different teams constructs are employed as determined by the metadirective directive. The number of teams is restricted by a num_teams clause and a thread limit is also set by a thread_limit clause for vendor AMD and NVIDIA platforms and specific architecture traits. Otherwise, just the teams construct is used without any clauses, as prescribed by the default clause.

III. METADIRECTIVE IMPLEMENTATION IN CLANG

A. Metadirective

The syntax of metadirective is shown in Figure 4, where clause is one of the when clause or the default clause as defined. In the when clause, context-selector-specification specifies a context selector. In the when and default clauses, directive-variant has the form shown Figure 4. It specifies a directive variant that specifies an OpenMP directive with clauses that apply to it.

A metadirective, by definition [2], is a directive that acts as though it is either ignored or substituted by the directive variant specified in one of the when or default clauses appearing on the metadirective. Whether a particular directive is selected or not depends upon the context selector as specified in the when clauses.

B. Context Selector

The syntax of a context selector specification is shown in Figure 5. A trait-selector can be one of the following:

1) construct
2) device
3) implementation
4) user

The construct selector set defines the construct traits that should be active in the OpenMP context. We can define the following selectors in the construct set: target, teams, parallel, for and simd. Each selector's properties are the same properties that are specified for the respective trait.

The device set includes traits that define the characteristics of the device being targeted by the compiler at that point in the program. A device set can have one of the following trait properties:

1) kind: This property specifies the general kind of the device - host, nohost, cpu, gpu, fpga.
2) isa: This property specifies the Instruction Set Architectures supported by the device which is implementation defined.
3) arch: This property specifies the architectures supported by the device.

The implementation set includes traits that describe the functionality supported by the OpenMP implementation at that point in the program. An implementation set can have one of the following trait property:

1) vendor: This property specifies the vendor identifiers of the implementation.
2) extension: This property specifies vendor specific extensions to the OpenMP specification.

fig:metadirective

fig:context-selector
3) A trait with a name that is identical to the name of any clause that can be supplied to the required directive.

The user selector set defines the condition selector that provides additional user-defined conditions.

For each when clause that appears in a metadirective, the specified directive variant, if present, is a candidate to replace the metadirective, if the corresponding context selector is compatible with the OpenMP context. A given context selector is compatible with a given OpenMP context if the following conditions are satisfied:

1) All selectors in the user set of the context selector are true. (See Section III-E for our extension to this rule).
2) All selectors in the construct, device, and implementation sets of the context selector appear in the corresponding trait set of the OpenMP context.
3) For each selector in the context selector, its properties are a subset of the properties of the corresponding trait of the OpenMP context.
4) Selectors in the construct set of the context selector appear in the same relative order as their corresponding traits in the construct trait set of the OpenMP context.

C. Clang/LLVM

For our implementation\(^1\), we used the current development version of llvm monorepo (version 12.0.0) from their git repository - https://github.com/llvm/llvm-project. LLVM follows the popular three phase design (Figure 6) for a traditional static compiler, whose major components are the front end, the optimizer and the back end.

![Fig. 6: Common 3-Phase Compiler Design](image)

Clang is the frontend for C language family compilers for LLVM. Clang parses the source code, checks it for errors, and builds an Abstract Syntax Tree (AST) to represent the input code. This AST is then translated into an intermediate LLVM IR, upon which all LLVM optimizations are performed and finally the backend translates this to generate the machine code.

A core design principle for clang is the use of a library-based architecture, where different parts of the frontend are separated cleanly into several libraries, which can then be combined for different needs and usages. This approach encourages good interfaces and makes it easier for new developers to get involved. To implement a new directive, like metadirective, we need to modify several of these libraries.

We primarily updated the following clang libraries:

1) libast - This library provides classes to represent the AST node and various helpers for analyzing and manipulating the AST.
2) libparse - The parsing library invokes coarse-grained ‘Actions’ provided by the libsema.
3) libsema - Semantic analysis provides a set of parser ‘Actions’ to build a standardized AST for programs.
4) libccodegen - This library lowers the AST to LLVM IR for optimization and code generation.

D. Design

The major design challenge lies in the parsing and evaluating the context selector. We do this in libparse, where we parse the context selector and save all the information in a data structure (OMPTraitInfo) which represents all the traits. The outer level of this data structure is an ordered collection of selector sets, each with an associated kind and an ordered collection of selectors. A selector has a kind, an optional score/condition, and an ordered collection of properties. We save this information to be used by libsema to make the decision on which directive variant need to be selected. In libsema, we check the kind of the selector and its properties. If the selector is of the kind user_condition, we read its condition and try to resolve it into an integer constant expression. If the expression is true, the user condition passes successfully. Otherwise, it is not selected. For all other selector kind, we compare the properties associated with the selector with the compiler parameters. For instance, if the context selector in a when clause is specified as device={arch("nvptx")}, then its selector kind will be device_arch and property will be "nvptx". In this case we will resolve the selector based upon whether the architecture of the device for which the code is being built is "nvptx" or not.

Based on the resolution of the context selector, we decide at compile time which directive variant need to be selected. If only one compatible context selector specified by a when clause has the highest score and it specifies a directive variant, the directive variant will replace the metadirective. If more than one when clause specifies a compatible context selector that has the highest computed score and at least one specifies a directive variant, the first directive variant specified in the lexical order of those when clauses will replace the metadirective. If no context selector from any when clause is compatible with the OpenMP context and a default clause is present, the directive variant specified in the default clause will replace the metadirective. We parse the statement associated with metadirective in the context of the selected directive variant. If no directive variant is selected to replace a metadirective according to the above rules, metadirective has no effect on the execution of the program.

---

\(^1\) Github link - https://github.com/almishra/metadirective.git. A patch is submitted to the LLVM mono-repo and is currently under review.
E. Dynamic User Condition

Yan et. al. explored a strategy for extending the metadirective [6] by relaxing its restriction to compile-time only selection and allowing runtime evaluation of user defined conditions. They implemented this in the ROSE compiler [5]. We have implemented the same feature in LLVM to establish that dynamic selection of user defined conditions provides greater flexibility to users. By definition, all the context selectors in metadirective need to be resolved at compile time. Based upon the compatibility of the context selector, as defined above in Section III-B, the compiler will replace metadirective with the directive variant specified by the user. In this extension, we extend metadirective to resolve a user condition at runtime. So instead of substituting metadirective with the given directive variant, we create an if-then-else statement whose condition will be evaluated and resolved during execution. For this, we parse all the when and default clauses in the metadirective and create statements for each of the directive variants specified in the clause. An example of this resolution is shown in Figure 7.

The major design challenge lies in the implementation of the dynamic user selection code. In the static case, we only need to parse and evaluate the context selector. Based upon the outcome, we generate just one directive statement. In the dynamic case, we need to parse all the when and default clauses and generate a directive statement for each of the valid clauses, and build a corresponding if-then-else statement for it. After careful consideration of the above mentioned libraries, we designed our implementation as follows:

1) In libast, we define the classes corresponding to the AST nodes for metadirective directive and when & default clauses.

2) In libparse, we parse the AST node and collect information related to all the when and default clauses and the statement associated with metadirective. For each when clause we parse and store the context selector. For each when and default clause we parse the directive variant and create a statement corresponding to this directive variant and store it. If no directive variant is provided, we store the statement associated with metadirective as it is.

3) In libsema, we gather all the stored directive variants and create the if-then-else statement based upon the condition parsed in the context selector. We store this statement in the AST node of metadirective.

4) In libcodegen, we read the AST node for metadirective and generate LLVM IR code corresponding to the stored if-then-else statement.

F. Example of Dynamic Metadirective

Let us consider a well known scenario where a programmer is writing a library function to multiply two matrices, where the size of the matrix will only be known at runtime. If the size of the matrices are small (< 10), the programmer might want to run the multiplication in serial to avoid parallel overhead. If the size of the matrices are fairly large (> = 10 & & < 100), programmers might want to run the computation in parallel while collapsing the two loops on the CPU. But if the size of the matrices are larger (> 100), and the code can be built for offloading to an NVIDIA GPU, then programmers might want to offload the computation onto the GPU, else continue with the parallel execution on CPU. The resulting code for such a scenario will look like shown in Figure 8a.

Since the size of the matrix can only be resolved at runtime, there is no way the programmer can use only preprocessing
if(N<10) {
  // Block 1
  for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
      for(int k=0; k<N; k++)
        C[i][j] += A[i][k] * B[k][j];
} else if (N<100) {
  // Block 2
  #pragma omp parallel for collapse(2)
  for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
      for(int k=0; k<N; k++)
        C[i][j] += A[i][k] * B[k][j];
} else {
  // Block 3
  #ifdef NVPTX
    #pragma omp target teams distribute
    when(device=
      arch("nvptx64")
    )
  else
    #pragma omp parallel for collapse(2)
  #endif
  for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
      for(int k=0; k<N; k++)
        C[i][j] += A[i][k] * B[k][j];
}

a: Without metadirective

if(N<10) {
  // Block 1
  for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
      for(int k=0; k<N; k++)
        C[i][j] += A[i][k] * B[k][j];
} else if (N<100) {
  // Block 2
  #pragma omp parallel for collapse(2)
  for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
      for(int k=0; k<N; k++)
        C[i][j] += A[i][k] * B[k][j];
} else {
  // Block 3
  #pragma omp metadirective
    when(device=
      arch("nvptx64"), user=condition(N>=100):
      target teams distribute \ parallel for) \ default()
  else
    #pragma omp parallel for collapse(2)
  for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
      for(int k=0; k<N; k++)
        C[i][j] += A[i][k] * B[k][j];
}

b: With static metadirective

if(N<10) {
  // Block 1
  for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
      for(int k=0; k<N; k++)
        C[i][j] += A[i][k] * B[k][j];
} else if (N<100) {
  // Block 2
  #pragma omp parallel for collapse(2)
  for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
      for(int k=0; k<N; k++)
        C[i][j] += A[i][k] * B[k][j];
} else {
  // Block 3
  #pragma omp metadirective \ when(device=
    arch("nvptx64")):
    target teams distribute \ parallel for) \ default(parallel for collapse(2))
  for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
      for(int k=0; k<N; k++)
        C[i][j] += A[i][k] * B[k][j];
}

c: With dynamic metadirective

Fig. 8: Different implementation of matrix multiplication

directives to take the decision of which block to compile. They need to write the same block of code multiple times, with different directives. Even with static metadirective code, unfortunately the programmer will only be able to use metadirective in the last else statement (Block 3 in the Figure 8b). But with dynamic metadirective programmers can write a much cleaner code (Figure 8c) without replicating the blocks multiple times. The code in Figure 8c will be resolved and substituted by the compiler to act like the code in Figure 8a.

In our implementation of dynamic metadirective we have made sure that all properties of static metadirective as mentioned in the OpenMP 5.0 specification are maintained.

IV. EXPERIMENTATION

To test and evaluate our implementation, we used the Summit Supercomputing Cluster at the Oak Ridge Leadership Computing Facility [26] and the SeaWulf computational cluster at Stony Brook University [27]. Summit has a hybrid architecture, where each node contains 2 IBM POWER9 CPUs and 6 NVIDIA Volta (V100) GPUs all connected together with NVIDIA’s high-speed NVLink. We use GCC version 6.4.0 and CUDA version 10.1.105 to build our implementation of LLVM. A Seawulf node, on the other hand, contains 2 Intel Xeon E5-2683v3 CPUs and 4 Nvidia Tesla (K80) GPUs all interconnected via a high-speed InfiniBand® network by Mellanox® Technologies. We use GCC version 6.5.0 and CUDA version 9.1.185 to build our implementation of LLVM.

We compiled the three codes as shown in Figure 8 on Summit to be executed using 1 NVIDIA V100 GPU. The size of the executable without metadirective was 204448 kB, with static metadirective was 204456 kB and with dynamic metadirective was 204712 kB. We can observe that the overhead on the size of executable for dynamic metadirective is minimal. When we ran these executables for varying input sizes, we found there was no overhead added due to the use of static or dynamic metadirective as can be seen in Figure 9.
We modified benchmark applications\(^2\) from the Rodinia Benchmark Suite [28] to use our dynamic metadirective implementation. We also implemented four microbenchmarks to represent several common methods: Matrix Multiplication, Gaussian Seidel Method, Laplace Equation and SAXPY. All the benchmarks used in our experiments are listed in Table I. The Rodinia Benchmark Suite was chosen primarily because of the diversity of the domains in which each of its applications falls. They are also intended to help the application developer learn how to use dynamic metadirective in real applications from different domains. We built these application for compute capability 7.0 on Summit (for V olta V100) and 3.5 on SeaWulf (for Tesla K80). An example of one such implementation can be seen in Figure 8a and Figure 8c, which shows a comparison of matrix multiplication implementations using preprocessing directives and the metadirective, respectively. All benchmarks were modified to replace preprocessing directives with dynamic metadirective.

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BvTree (b+tree) [14]</td>
<td>Search domain. This is a data structure used as an index to facilitate fast access to the elements of a larger body of data.</td>
</tr>
<tr>
<td>Back Propagation (backprop) [15]</td>
<td>Pattern Recognition domain. This application tracks the movement of a mouse heart over a sequence of 104 660x590 ultrasound images to record response to the stimulus.</td>
</tr>
<tr>
<td>Breadth First Search (bfs) [16]</td>
<td>This benchmark provides the GPU implementations of BFS algorithm which traverses all the connected components in a graph.</td>
</tr>
<tr>
<td>Heart Wall (heartwall) [17]</td>
<td>Medical Imaging domain. This application tracks the movement of a mouse heart over a sequence of 104 660x590 ultrasound images to record response to the stimulus.</td>
</tr>
<tr>
<td>Hotspot [18]</td>
<td>Physics Simulation domain. We re-implemented the transient differential equation solver from HotSpot using target offloading directives for GPU.</td>
</tr>
<tr>
<td>K-means (kmeans) [19]</td>
<td>Data Mining domain. K-means is a clustering algorithm used extensively in data-mining and elsewhere, important primarily for its simplicity. Many data-mining algorithms show a high degree of data parallelism.</td>
</tr>
<tr>
<td>k-Nearest Neighbor (knn) [20]</td>
<td>Data Mining domain. In the implementation of this application, it finds the k-nearest neighbors from an unstructured data set.</td>
</tr>
<tr>
<td>Lava MD (lavamd) [21]</td>
<td>Molecular Dynamics domain. The code calculates particle potential and relocation due to mutual forces between particles within a large 3D space.</td>
</tr>
<tr>
<td>LU Decomposition (lud)</td>
<td>Linear Algebra domain. This benchmark is a good example where multiple kernels care called from within a loop and some data shared by these kernels are also used on the host.</td>
</tr>
<tr>
<td>Needleman Wunsch (nw) [22]</td>
<td>Bioinformatics domain. Needleman-Wunsch is a nonlinear global optimization method for DNA sequence alignments.</td>
</tr>
<tr>
<td>Particle Filter (p-filter) [23]</td>
<td>Medical Imaging domain. This particular implementation is optimized for tracking cells, particularly leukocytes and myocardial cells.</td>
</tr>
<tr>
<td>Path Finder (p-finder)</td>
<td>Grid Traversal domain. FPathFinder uses dynamic programming to find a path on a 2-D grid from the bottom row to the top row with the smallest accumulated weights, where each step of the path moves straight ahead or diagonally ahead.</td>
</tr>
<tr>
<td>Speckle Reducing Anisotropic Diffusion (srad) [24]</td>
<td>Image Processing domain. This is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations (PDEs).</td>
</tr>
<tr>
<td>Stream Cluster (stream) [25]</td>
<td>Data Mining domain. For a stream of input points, it finds a predetermined number of medians so that each point is assigned to its nearest center. The quality of the clustering is measured by the sum of squared distances (SSQ) metric.</td>
</tr>
<tr>
<td>Three Matrix Multiplication (3mm)</td>
<td>This is the most basic implementation of multiplying three large matrices. This is a benchmark where two kernels are reusing same data. The experiment used matrices of size 5000 × 5000 each.</td>
</tr>
<tr>
<td>Gaussian Seidel Method (gauss)</td>
<td>The method for solving linear equations is an iterative method, in which the values for the given variables keep changing until a certain threshold of variance is reached. The experiment used a matrix of size 2^{13} × 2^{13}.</td>
</tr>
<tr>
<td>Single-Precision A·X Plus Y (saxpy)</td>
<td>SAXPY is a function in the standard Basic Linear Algebra Subroutines (BLAS)library. In its simplest form this is a benchmark where two kernels are reusing same data. The experiment used two vectors of size 2^{27} each.</td>
</tr>
</tbody>
</table>

Table I: Updated benchmarks from the Rodinia Benchmark Suite

\(^2\)Github - https://github.com/almishra/rodinia_3.1_meta.git

**Fig. 9:** Execution time (in sec) of matrix multiplication for 3 different implementations
of all the benchmark applications listed in Table I using Clang on both Summit and SeaWulf. Summit has IBM Power9 CPUs and NVIDIA Volta V100 GPUs. For GPU codes LLVM, used CUDA version 10.1.105 in the backend. Also the applications are built for compute capability of 7.0. Seawulf has Intel Xeon E5-2683v3 CPUs and NVIDIA Tesla K80 GPUs. For the GPU codes, LLVM used CUDA version 9.1.85 in the backend on this platform. Also the applications are built for compute capability of 3.5. Due to the difference in architecture and backend, the executables for the original benchmark code already vary in size, as can be seen in Table II.

As expected there is some increase in the size of the executable for our dynamic metadirective implementation, depending upon the size of the kernel which is generated multiple times. But by how much? This question can be answered by looking at Figure 10 which shows the percentage increase in the size of executable of the original and dynamic metadirective codes. The size of all the executables on Summit are in the range of 70 kB to 980 kB, while on Seawulf they are in the range of 12 kB to 622 kB. For some applications like bfs, hotspot, k-nn, gauss and jacobi this change is less than 0.01%, hence shown as 0% in the figure. On Summit the maximum difference is 4.94%, while on SeaWulf it is 10.08%. The maximum difference on both platforms is in the Kmeans application, which has fairly larger kernels compared to the other benchmark applications analyzed. Overall, the change in the size of the executable will be directly related to the size of the kernels for the application and the number of when clauses used with user defined context selector.

But does using dynamic metadirective effect the actual runtime of the application? To answer this question we executed both the original code and our modified code 10 times and collected the average runtime for each application. The result can be seen in Figure 11 which shows the percentage
change in the runtime of the modified code when using the dynamic metadirective from the original code. The difference in runtime is almost negligible for all the application on both the platforms, which could easily be caused due to multiple hardware or environmental reasons. On Summit the maximum difference of the time is -1.76% in the SAXPY application, whose runtime changed from 1.705s to 1.675s. Whereas on Seawulf, the maximum change is -2.07% for the same application, where the runtime changed from 24.443s to 23.937s. Looking at this result we can conveniently conclude that there is very minimal to no runtime overhead added to the application due to dynamic metadirective.

VI. RELATED WORK

The OpenMP 4.5 version has been relatively well supported in the LLVM/Clang framework. OpenMP 5.0 has introduced many new features especially related to accelerated devices to take care of heterogeneity in the future HPC machines. These new features are being actively added in LLVM/Clang under the ECP SOLLVE project [12]. The site, https://clang.llvm.org/docs/OpenMPSupport.html, gives up-to-date feature support in Clang. The SOLLVE team has also developed an OpenMP verification and validation suite to support and test the OpenMP 5.0 features [29]. The new OpenMP features are heavily towards accelerated devices.

Many new GPU features such as unified memory feature provided by Kepler and later GPUs are supported in LLVM/Clang [30], [31]. However, these new features are not effectively used in the LLVM backend optimizations for OpenMP. It is important for the compiler to be aware and to leverage these new features for better performance. Currently, OpenMP dialects using the MLIR framework [32] are trying to fill this performance gap.

The work [6] uses the Smith-Waterman algorithm as an example to show the need for adaptive selection of parallelism and devices at runtime, and present a prototype implemented in the ROSE compiler using OpenMP metadirective. In [33], the authors explored the benefits of using two OpenMP 5.0 features, including metadirective and declare variant, for the miniMD benchmark from the Mantevo suite. The authors concluded that these features enabled their code to be expressed in a more compact form while maintaining competitive performance portability across several architectures. However, their work only explored compile-time constant variables to express conditions. Many researchers have studied using GPUs to speedup Smith-Waterman algorithm, beginning as far back as Liu et al. in 2006 [34]. Our implementation resembles some of these early attempts in terms of data motion and synchronization behavior, mainly as a simple case study. Later work uses a variety of techniques to reduce the data motion and memory requirement by doing backtracing on the GPU [35] and even exploring repeating work to accomplish the backtrace in linear space [36]. These techniques would likely make the inner-loop optimization we discussed more attractive.
by removing the high cost of moving the complete cost matrix to and from the device, and may be worth exploring in the future.

VIII. Conclusion and Future Work

OpenMP 5.0 introduces the metadirective directive that conditionally resolves to another directive at compile time by selecting from multiple directive variants based on traits that define an OpenMP condition or context. We have implemented this feature in LLVM for general use.

OpenMP 5.0 restricts the selection to be determined at compile time, which requires that all traits must be compile-time constants. Our analysis of real applications indicates that this restriction limits the usefulness of metadirective, so we explored an extension of user-defined contexts to allow directive variant selection at runtime. We successfully modified the open source LLVM compiler to extend the user-defined condition of context selector, as used in metadirective, to be resolved at runtime. A dynamic evaluation of user-defined conditions can provide more flexibility for programmers to express a variety of adaptive algorithms that boost overall performance.

To study its impact we have modified 13 benchmarks application in the Rodinia benchmark suite and 4 common micro benchmark applications. Rodinia includes applications and kernels which target multi-core CPU and GPU platforms across a variety of domains in computer science. Our modifications to the Rodinia benchmark suite enabled us to explore the impact of dynamic metadirective in an OpenMP context. It also provides a guideline to the end users to help them apply these features in real applications. The results of our evaluation reveal that dynamic user condition can easily be used by programmers to achieve better portability and adaptability in their code.

The main advantage of the dynamic metadirective is that it adds minimal or no overhead to the user application. In addition to allowing flexibility to the programmers to introduce portability and adaptability to their code, it can enable researchers aimed at automatic code generation and computation on heterogeneous devices, such as Mishra et al.’s [37] work on automatic GPU-offloading, or Mendonça et al.’s [38] work on automatic parallizer or Poesia et al.’s [39] work on static placement of computation on heterogeneous devices. All these efforts take decisions based on compile-time analysis and can be significantly assisted by a dynamic metadirective, which would further reduce the barriers to use of GPUs for scientific computing. Potential future extensions of this work include the following:

- Provide optimization of selected code and help better data handling between heterogeneous devices. This can be an extension to Mishra et al.’s [40] work on data reuse analysis.
- Explore more complex user-defined conditions, where a function can be called from a user condition.
- Adding template definition to metadirective for type manipulation.

ACKNOWLEDGEMENT

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. The authors would like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the SeaWulf computing system, which was made possible by a $1.4M National Science Foundation grant (#1531492).

REFERENCES

[8] GNU, “Gnu compiler suite.” [Online]. Available: https://gcc.gnu.org/gcc/protect discretionary{ char \ hyphenchar\[font\] } \{ } \{ } \{ } \{ } \{ }


