Extending OpenMP Metadirective Semantics for Runtime Adaptation

Yonghong Yan[1], Anjia Wang[1], Chunhua Liao[2],
Thomas R.W. Scogland[2], and Bronis R. de Supinski[2]

[1] University of South Carolina
Columbia, SC 29208, USA
yanyh@cse.sc.edu,anjia@email.sc.edu
[2] Lawrence Livermore National Laboratory
Livermore, CA 94550, USA
{liao6,scogland1,bronis}@llnl.gov

Abstract. OpenMP 5.0 introduces the metadirective to support selection from a set of directive variants based on the OpenMP context, which is composed of traits from active OpenMP constructs, devices, implementations or user-defined conditions. 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 has its limitation, and we explore extension of user-defined contexts to support variant selection at runtime. We use 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. Given a large range of input sizes, our experiments demonstrate that one of the adaptive versions of Smith-Waterman always chooses the parallelism and device that delivers the best performance, with improvements between 20% and 200% compared to non-adaptive versions that use the other approaches.

Keywords: OpenMP 5.0 · Metadirective · Dynamic context.

1 Introduction

OpenMP 5.0 [5] introduces the concept of OpenMP contexts and defines traits to describe them by specifying the active construct, execution devices, and functionality of an implementation. OpenMP 5.0 further introduces the metadirective and declare variant to support directive selection based on the enclosing OpenMP context as well as user-defined conditions. This feature enables programmers to use a single directive to support multiple variants tailored for different contexts derived from the hardware configuration, software configuration or user defined conditions. With context traits that are available to the compiler when performing OpenMP transformations, a user can much more easily optimize their application for specific architectures, possibly resolving to multiple different directives in the same compilation in different call chains or different contexts.

OpenMP 5.0 restricts context traits to be fully resolvable at compile time. Thus, the ability to optimize OpenMP applications based on their inputs and runtime behavior is severely constrained, even with user-defined conditions. The semantics of context selection are naturally applicable to support both compile time and runtime directive selection. Given a low overhead runtime selection mechanism, the extension for enabling runtime adaptation would improve performance of an application based on system architecture and input characteristics. 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 math kernel libraries parallelize and optimize matrix multiplication based on input matrix sizes.

In this paper, we extend the semantics of user-defined contexts to support runtime directive selection. We use the Smith-Waterman algorithm as an example to demonstrate that the extensions enable runtime adaptive selection of target devices, depending on the size of the input. We develop a prototype compiler implementation in the ROSE compiler and evaluate the performance benefits of this extension. Our experiments demonstrate that one of the adaptive versions of Smith-Waterman always chooses the parallelism and device that delivers the best performance for a large range of input sizes, with improvements between 20% and 200% over the non-adaptive versions.

The remainder of this paper is organized as follows. Section 2 presents the current syntax and semantics of OpenMP context and metadiretive in the latest standard. A motivating example is given in Section 3 to demonstrate the need to support dynamic selection of directives at runtime based on user defined conditions. Section 4 introduces our extension to allow dynamic user-defined context. We discuss a prototype compiler implementation for the dynamic extension in Section 5. Section 6 evaluates performance of our prototype that automates adaptation of the Smith-Waterman algorithm. Finally, we mention related work in Section 7 and conclude our paper in Section 8.

2 Variant Directives and Metadirective in OpenMP 5.0

Variant directives is one of the major features introduced in OpenMP 5.0 to facilitate programmers to improve performance portability by adapting OpenMP pragmas and user code at compile time. The standard specifies the traits that describe active OpenMP constructs, execution devices, and functionality provided by an implementation, context selectors based on the traits and user-defined conditions, and the metadirective and declare directive directives for users to program the same code region with variant directives. A metadirective 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 the metadirective but selects a function variant at the call-site based on context or user-defined conditions. The mechanism provided by the two directives for selecting variants is more convenient to use than the C/C++ preprocessing since it directly supports variant selection in OpenMP and allows an OpenMP compiler to analyze and determine the final directive from variants and context.

In this paper, we use metadirective to explore the runtime adaptation feature since it applies to structured user code region (instead of a function call as in declare variant), which poses more adaptation needs based on the program inputs. The metadirective syntax for C and C++ is:

#pragma omp metadirective [clause[[,]clause]...]new-line

The clause in a metadirective can be either when(context-selector-specification:[directive-variant]) or default (directive-variant).

The expressiveness of a metadirective to enable conditional selection of a directive variant at compile time is due to the flexibility of its context selector specification. The context selector defines an OpenMP context, which includes a set of traits related to active constructs, execution devices, functionality of an implementation and user defined conditions. Implementations can also define further traits in the device and implementation sets.

```
context_selector_spec : trait_set_selector
                      | context_selector_spec trait_set_selector;
trait_set_selector : trait_set_name '=' '{' trait_selector_list '}';
trait_set_name : CONSTRUCT | DEVICE | IMPLEMENTATION | USER;
trait_selector_list : trait_selector
                    | trait_selector_list trait_selector;
trait_selector : construct_selector
               | device_selector
               | implementation_selector
               | condition_selector;
condition_selector : CONDITION '(' trait_score const_expression ')';
device_selector : context_kind | context_isa | context_arch;
context_kind : KIND '(' trait_score context_kind_name ')';
context_kind_name : HOST | NOHOST | ANY  | CPU | GPU | FPGA;
context_isa : ISA '(' trait_score const_expression ');
context_arch : ARCH '(' trait_score const_expression ')';
implementation_selector : VENDOR '(' trait_score context_vendor_name ')'
                        | EXTENSION '(' trait_score const_expression ')'
                        | const_expression '(' trait_score ')';
                        | const_expression;
context_vendor_name : AMD  | ARM | BSC | CRAY | FUJITSU | GNU | IBM |
                      INTEL | LLVM | PGI | TI | UNKNOWN;
construct_selector : parallel_selector;
parallel_selector : PARALLEL | PARALLEL '(' parallel_parameter ')';
parallel_parameter : trait_score parallel_clause_optseq;
trait_score : | SCORE '(' const_expression ')' ':';
const_expression : EXPR_STRING;
```

Fig. 1: Context Selector Grammar

Figure 1 shows the grammar for context selectors in Backus-Naur Form. A context selector contains one or more trait set selectors. Each trait set selector may contain one or more trait selectors. Each trait selector may contain one or more trait properties. All traits must be resolved to constant values at compile time, as indicated by condition_const_expression at line 12. The upper case tokens throughout the grammar are enum names that the lexer returns.

Figure 2(b) shows an example that uses a metadirective to specify a variant to use for NVIDIA PTX devices, and a variant that is applied in all other cases by default. Figure 2(a) shows the code using C/C++ macro to achieve the same goal. In 2(b), a trait selector named arch from the device trait set specifies the context selector. If the trait's property is resolved to be nvptx at compile-time then the directive variant that has one thread team and the loop construct is applied. Otherwise, a target parallel loop directive is applied. Using metadirective has two major benefits. One is that compiler could be aware of more context information. In 2(a), the preprocessor will prune one of the conditional statement before passing the source code to compiler. However, in 2(b), compiler has all the information of branches. The other advantage is that the redundant code is optimized. The two lines of for loop only appear once while using metadirective.

```
int v1[N], v2[N], v3[N];
#if defined(nvptx)
 #pragma omp target teams distribute
      parallel loop map(to:v1,v2)
     map(from:v3)
  for (int i= 0; i< N; i++)
     v3[i] = v1[i] * v2[i];
#else
 #pragma omp target parallel loop
     map(to:v1,v2) map(from:v3)
  for (int i= 0; i< N; i++)
     v3[i] = v1[i] * v2[i];
#endif
```

(a) Original code

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

(b) Using metadirective

Fig. 2: An Example using metadirective

3 A Motivating Example

While the metadirective can be used to specify multiple variants in a program, it requires the corresponding traits to be resolved at compile time, which limits customization of the user code at runtime. In this section, we use the Smith-

Waterman algorithm to demonstrate the need for customization and dynamic adaptation.

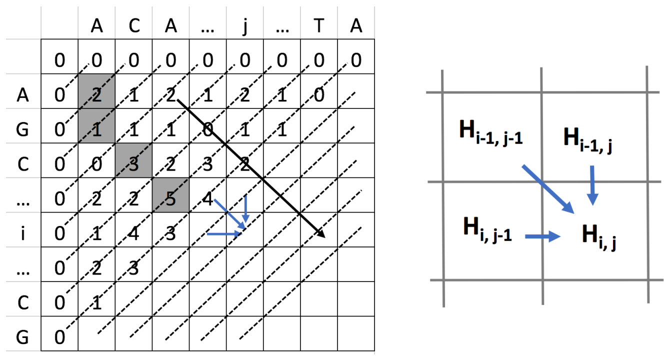

Fig. 3: Wavefront Computation Pattern of the Smith-Waterman Algorithm

The Smith-Waterman algorithm performs local sequence alignment [8] to find the optimal occurrence of a sub-sequence within a DNA or RNA sequence. The algorithm compares segments of all possible lengths and optimizes the similarity measure. Similarity is represented by a score matrix H. The update of the score is derived from one-to-one comparisons between all components in two sequences from which the optimal alignment result is recorded. Figure 3 shows the scoring step of the algorithm. Arrows in the figure denote data dependency between points of the computation. The scoring process is a wavefront computation pattern. Figure 4 shows a typical OpenMP implementation of the scoring wavefront pattern by parallelizing the computation that iterates on a wavefront line. The implementation of the algorithm has O(M*N) time complexity in which M and N are the lengths of the two sequences that are being aligned. The space complexity is also O(M*N) since the program must store two string sequences and two matrices, one for scoring and the other for backtracking.

```
long long int nDiag = M + N - 3;
for (i = 1; i <= nDiag; ++i) {
  long long int nEle, si, sj;
  nEle = nElement(i); calcFirstDiagElement(i, &si, &sj);
#pramga omp parallel for shared (nEle, si, sj, H, P, maxPos) private(j)
  for (j = 0; j < nEle; ++j)
    similarityScore(si-j, sj+j, H, P, &maxPos);
}
```

Fig. 4: An OpenMP Implementation of the Smith-Waterman Algorithm

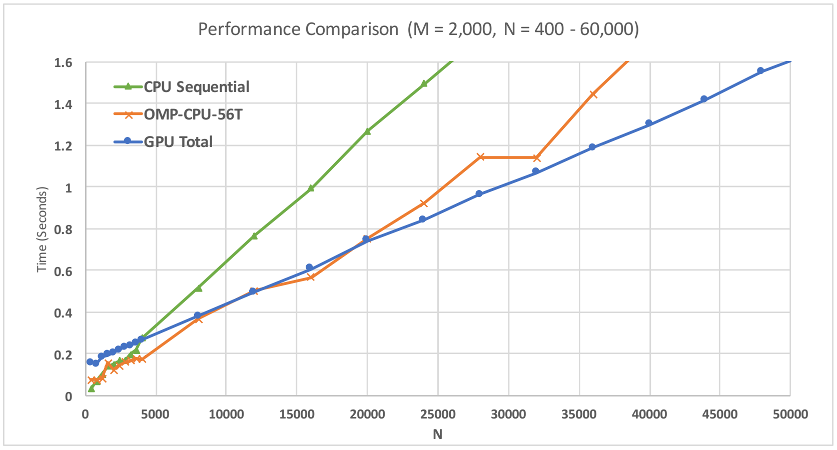

Fig. 6: Smith-Waterman Execution Times (Fixed M, Varying N)

```
long long int nDiag = M + N - 3;
#pragma omp target enter data map(to:a[0:m],...) map(to:H[0:asz],...)
for (i = 1; i <= nDiag; ++i) {
  long long int nEle, si, sj;
  nEle = nElement(i); calcFirstDiagElement(i, &si, &sj);
  #pragma omp target teams distribute parallel for map (...)
  for (j = 0; j < nEle; ++j)
    similarityScore(si-j, sj+j, H, P, &maxPos);
}
#pragma omp target exit data map(from:H[0:asz],...)
```

Fig. 5: An OpenMP implementation using offloading on GPUs

One can add OpenMP device constructs to create a version for GPUs, shown in Figure 5. In our early evaluation, we compare the performance of three baseline versions of the algorithm: CPU sequential, OpenMP parallel with 56 threads, and OpenMP offloading on a NVIDIA V100 GPU. Figure 6 shows that the performance of three versions varies dramatically with regards to the length of one sequence (N), indicated by the cross points of the three plotted lines. Thus an algorithm that adapts between the three versions based on the lengths of the input sequences would perform best overall.

We consider two adaptive versions. First, we optimize the program such that it automatically selects one of the three versions, i.e. CPU sequential, or CPU parallel or GPU based on the lengths of the sequences, which can be represented by the outer loop count `nDiag`. A typical use case of this approach could be that a user wants to align a large number of sequences of varying lengths (N) with a sequence of fixed length (M). From Figure 6, the best choice among the three

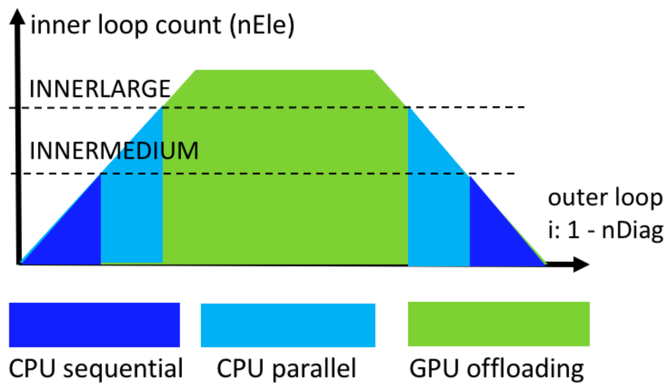

Fig. 7: The relationship between the outerloop index and inner loop count

versions clearly depends on the evaluation of the length of N against a threshold. Since all the three versions exhibit good weak scaling, the two thresholds are the value of N at which the performance crossover occurs. These two thresholds separate the three versions according to the problem size.

For the second version, we observe that when computing the scoring matrix, the inner loop count varies between outer loop iterations. One can optimize the inner loop such that it uses one of the three versions based on the inner loop count. Figure 7 shows the relationship between the outer loop index, the inner loop count and the two hypothetical thresholds for inner loop count (INNERMEDIUM and INNERLARGE) for determining which parallelism approach to use for the inner loop.

In either approach, the dynamic nature of the outer loop count (nDiag), inner loop count (nEle) and their impact on performance would benefit from metadirective's support of runtime selection of different code variants.

4 Extension of the Metadirective Semantics and its Application to Smith-Waterman

We present an initial exploration of extending metadirective by relaxing its restriction of compile-time only selection. We allow runtime evaluation of user-defined conditions. Our future work includes exploration of semantic extensions of other selectors to allow for combined compile-time and the runtime selection of variants. We anticipate that those extensions may require new clauses to facilitate low overhead selection.

4.1 Adaptation Based on the Outer Loop Count

Figure 8 shows the first version which uses the metadirective to adapt the algorithm based on the outer loop count (nDiag) to control the switch between

```
long long int nDiag = M + N - 3;
//Copy the data if GPU will be used
#pragma omp metadirective \
 when(user={condition(nDiag >= OUTERLARGE)}: \
 target enter data map(to:a[0:m],...) map(to:H[0:asz],...))
for (i = 1; i <= nDiag; ++i) {
  long long int nEle, si, sj;
  nEle = nElement(i);  calcFirstDiagElement(i, &si, &sj);

  #pragma omp metadirective \
   when (user={condition(nDiag < OUTERMEDIUM)}: ) /*serial*/ \
   when (user={condition(nDiag < OUTERLARGE)} : \
     parallel for private(j) shared (nEle, ...)) /*CPU parallel*/ \
   /*nDiag>=OUTERLARGE, GPU offloading*/ \
   default (target teams distribute parallel for ...)
  for (j = 0; j < nEle; ++j)
    similarityScore(si-j, sj+j, H, P, &maxPos);
}
//Copy data back to CPU if GPU is used
#pragma omp metadirective \
 when (user={condition(nDiag >= OUTERLARGE)}: \
 target exit data map(from:H[0:asz],...)
```

Fig. 8: Selection via metadirective Based on the Outer Loop Count(nDiag)

the three versions. OpenMP 5.0 provides a scoring mechanism for the directive variants to guide the compiler's selection among them. In our prototype, the variants and their conditions are evaluated in the order that they appear in the metadirective construct. The first variant for which its condition is true is chosen and the following variants are ignored by the runtime. These semantics are familiar to programmers since standard programming languages use them to evaluate the conditions of if-else and switch-case statements.

To identify the two thresholds (OUTERMEDIUM and OUTERLARGE) in this version, we can profile each of the three versions, using a small data set. Since they all have good weak scaling, as demonstrated in Figure 6, we can easily extrapolate the performance to find the crossover points of the three versions, which represent the two thresholds.

4.2 Adaptation Based on the Inner Loop Count

Figure 9 shows the version of using metadirective and INNERMEDIUM and INNERLARGE thresholds shown in Figure 7 to control switching the execution between CPU and GPU. Since the inner loop is offloaded across consecutive outer loop iterations, we optimize data movement with target enter data and target exit data directives such that it is copied only once when the INNERLARGE threshold is met.

```
bool GPUDataCopied = false;
for (i = 1; i <= nDiag; ++i) {
  long long int nEle, si, sj;
  nEle = nElement(i);  calcFirstDiagElement(i, &si, &sj);

  //Copy the data for the first time GPU will be used
  if (nEle >= INNERLARGE && !GPUDataCopied) {
    #pragma omp target enter data map(to:a[0:m],...) map(to:H[0:asz],...)
    GPUDataCopied = true;
  }
  //Copy data back to CPU after the last time GPU is used
  if (GPUDataCopied && nEle < INNERLARGE ) {
    GPUDataCopied = false;
    #pragma omp target exit data map(from:H[0:asz],...)
  }
  #pragma omp metadirective \
   when (user={condition(nEle < INNERMEDIUM)}:  ) /*serial*/ \
   when (user={condition(nEle < INNERLARGE)} : \
     parallel for private(j) shared (nEle, ...)) /*CPU parallel*/ \
   default (target teams distribute parallel for \
       map (to:a[0:m], b[0:n], ...) map(tofrom: H[0:asz], ...) \
       shared (nEle, ...)) //GPU offloading
  for (j = 0; j < nEle; ++j)
    similarityScore(si-j, sj+j, H, P, &maxPos);
}
```

Fig. 9: Selection via metadirective Based on the Inner Loop Count (nEle)

For both of the adaptive versions, an OpenMP compiler must generate three versions of the inner loop for the three base versions. The runtime uses the condition checks in the when clause of the directive to determine which version to invoke.

5 Prototype Implementation

We use ROSE to prototype our metadirective implementation and extension. Developed at LLNL, ROSE [7] is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for Fortran and C/C++ applications. ROSE supports OpenMP 3.0 [1] and part of 4.0 [2]. It parses OpenMP directives and generates an Abstract Syntax Tree (AST) representation of OpenMP constructs. The OpenMP AST is then lowered and unparsed into multithreaded CPU or CUDA code. A backend compiler, such as GCC or NVCC, compiles the CPU or CUDA code and links the generated object files with a runtime to generate the final executable. Our prototype implementation includes the following components:

```
  ...

  if (nEle < INNERMEDUIM) {    //serial
   for (j = 0; j < nEle; ++j) similarityScore(si-j, sj+j, H, P, &maxPos);
  } else if (nEle < INNERLARGE) {    //CPU parallel
   #pragma omp parallel for private(j) shared (nEle, ...) )
   for (j = 0; j < nEle; ++j)
     similarityScore(si-j, sj+j, H, P, &maxPos);
  } else {    //GPU offloading
   #pragma omp target teams distribute parallel for \
       map (to:a[0:m], b[0:n], ... ) \
       map(tofrom: H[0:asz], ...) shared (nEle, ... ))
   for (j = 0; j < nEle; ++j)
     similarityScore(si-j, sj+j, H, P, &maxPos);
  }
  ...
}
```

Fig. 10: Lowering metadirective with Dynamic Conditions to an if Statement

- A new OpenMP parser for metadirective, which is treated as nested directives;
- An extension of the internal ROSE AST to represent metadirective;
- A new phase of OpenMP lowering as the first step to translate the AST of metadirective into the OpenMP 4.0 AST using if-else statement as Figure 10 shows for the input code in Figure 9;
- Existing OpenMP lowering phase that generates CUDA code and connections to a thin layer of the XOMP runtime [1]; and
- Generated CUDA code compilation with NVCC and linking with XOMP.

6 Experimental Results

Our experimental platform has 2 CPUs, each with 28 cores, and one NVIDIA Telsa V100 GPU with 16 GB of HBM. The system has 192 GB of main memory and runs Ubuntu 18.04 LTS, GCC 8.2.0 and NVIDIA CUDA SDK 10.1.105.

6.1 Evaluation of Adaptation Based on Outer Loop Count

To evaluate performance of the version that Figure 8 shows, we performed the following experiment. First, we measured individual performance of CPU sequential, CPU parallel and GPU versions. As in Figure 6, we identified the crossover points for the OUTERMEDIUM and OUTERLARGE thresholds as 3200 and 22000. Figures 11 shows the performance results. The adaptive version always chooses the parallelism and device that delivers the best performance for a large range of input sizes, with improvements between 20% and 200% over the non-adaptive versions.

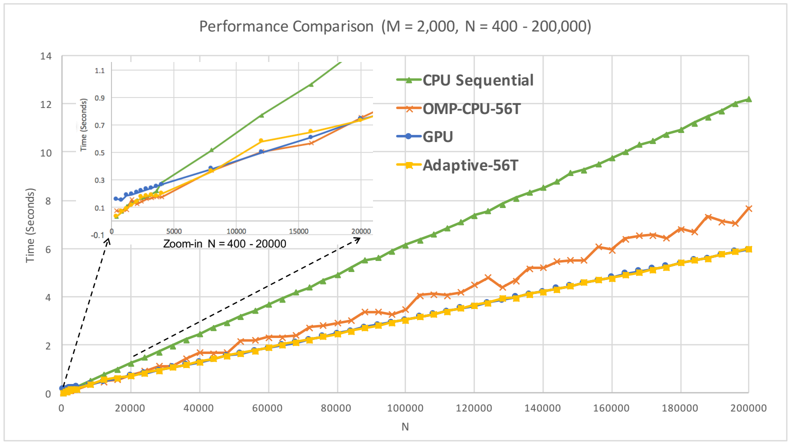

Fig. 11: Adaptive Smith-Waterman Performance using Outer Loop Count

6.2 Evaluation of Adaptation Based on Inner Loop Count

As Figure 7 shows, this version tries to adaptively divide the inner loop iterations among CPU sequential, CPU parallel and GPU such that it could perform better than any individual version alone. In the experiments, we decide to use just the INNERLARGER to switch the computation between CPU parallel and GPU since the impact of CPU sequential is minimal. We evaluated the performance using five different M-N configurations: 45,000-45,000, 2,000-200,000, 200,000-2,000, 20,000-40,000, 40,000-20,000. In each configuration, we experiment with different INNERLARGE threshold values to control the switch between CPU parallel and GPU.

Our experiment shows that the benefits of using the adaptive version of the Smith-Waterman (SW) algorithm can be observed for M=20,000 and N=200,000, shown in Figure 12. However, the performance advantage (when the inner loop count threshold is at 200, 1300, 1600, etc) is very small compared to the best non-adaptive GPU version.

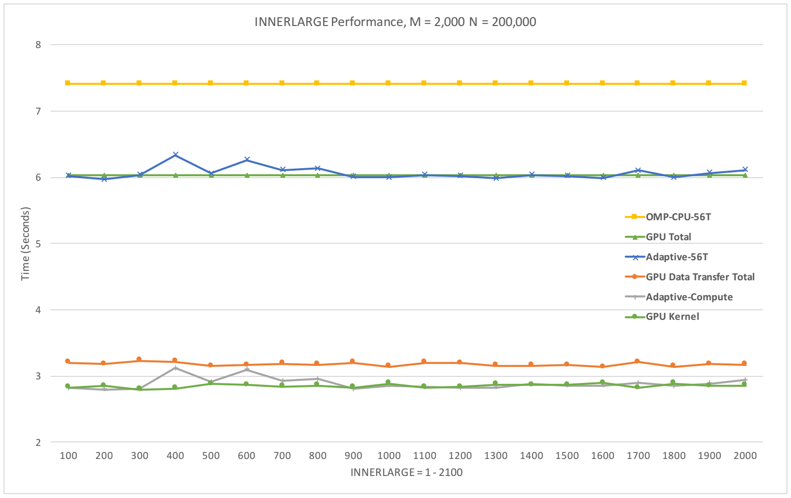

Fig. 12: Adaptive SW's Performance using Inner Loop Count (M !=N)

For all other configurations, the adaptive version is not able to improve the overall performance over the best non-adaptive baseline version. Figure 13 shows one example for M=45,000 and N=45,000.

To understand our results, we profiled the execution to break down the GPU time into GPU kernel time and GPU data transfer time (shown in both Figure 12 and 13). For M=N=45,000, the profiling results show that the GPU data transfer overhead dominates the GPU offloading time, about 80%. Instead of only transferring the wavefront that needed for calculation, it always transfers all the data unnecessarily. Also, as we increase the inner loop count's threshold value, the compute time for the adaptive version also slightly increase, making it difficult to outperform its non-adaptive baseline version. Further investigation is still needed to make this adaptive version more effective.

6.3 Overhead Discussion

Since the transformation of metadirective simply uses the if-else statement and the overhead is expected to be negligible. However, the multi-version fat-binary code generated by the compiler may have a large code footprint in both disk and instruction memory when being executed.

From Fig. 11, we observe that the execution time of adaptive version is not significantly different from the the individual version. With configurations that M=2,000 and N=20,000-200,000, the execution overheads are measured. By average among all those configurations, the adaptive version is slower than individual GPU version by 0.28%, which is unnoticeable.

We also measured code size of different versions, as shown in Table 1. The object file of adaptive version is 18.37% to 34.9% larger than the individual non-adaptive versions. For the final executable files, the GPU executable files are

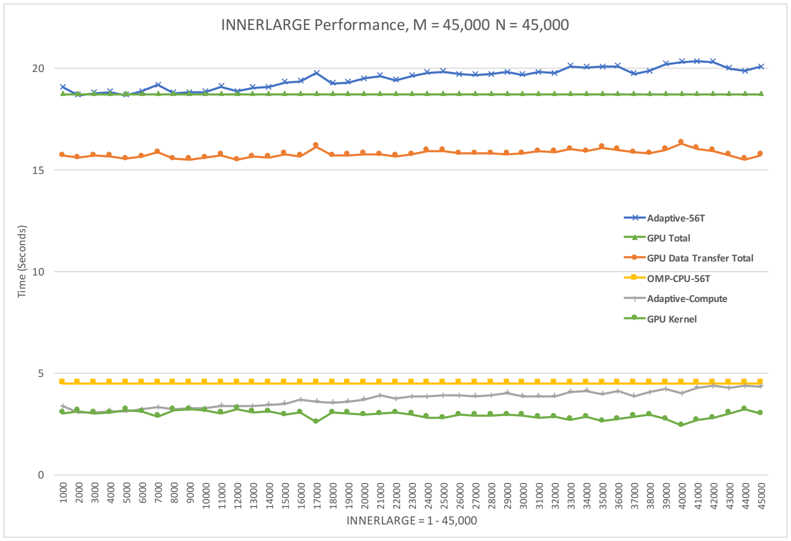

Fig. 13: Adaptive SW's Performance using Inner Loop Count (M ==N)

significantly larger since they incorporate more supportive object files for both CPU and GPU execution. As a result, our transformation has much less impact on the code size.

Smith Waterman Version		Object file size /KB	Executable file size /KB
Non-Adaptive	Serial	43	14
	CPU Parallel	49	18
	GPU	44	751
Adaptive		58	751

Table 1: Code Size Overhead of Adaptive Smith Waterman

7 Related Work

In [6], 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 [3]. 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 movement and memory requirement by doing backtracing on the GPU [9] and even exploring repeating work to accomplish the backtrace in linear space [4]. 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.

8 Conclusion

Metadirectives in OpenMP 5.0 allow programmers to easily apply multiple directive variants to the same code region in order to meet the need of different software and hardware contexts. However, the context must be resolved at compile time. In this paper, we have used the Smith-Waterman algorithm to demonstrate the need for runtime adaptation. We propose to relax the compile-time restriction to allow dynamic adaptation of user-defined contexts. Our experimental results with a prototype compiler implementation show that dynamic evaluation of user-defined conditions can provide programmers more freedom to express a range of adaptive algorithms that improve overall performance. In the future, we would like to explore more complex user-defined conditions and extend other context selectors to support dynamic adaptation of metadirective at runtime, including dynamic work partitioning between CPUs and GPUs.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, and supported by the U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357. The manual reference codes were supported by LLNL-LDRD 18-ERD-006. LLNL-CONF-774899. This material is also based upon work supported by the National Science Foundation under Grant No. 1833332 and 1652732.

References

1. Liao, C., Quinlan, D.J., Panas, T., De Supinski, B.R.: A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries. In: International Workshop on OpenMP. pp. 15–28. Springer (2010)
2. Liao, C., Yan, Y., de Supinski, B.R., Quinlan, D.J., Chapman, B.: Early experiences with the OpenMP accelerator model. In: OpenMP in the Era of Low Power Devices and Accelerators, pp. 84–98. Springer (2013)
3. Liu, Y., Huang, W., Johnson, J.: GPU accelerated smith-waterman. In: International Conference on Computational Science (ICCS) (2006)

4. de O Sandes, E., de Melo, A.: Smith-waterman alignment of huge sequences with gpu in linear space. In: 2011 IEEE International Parallel Distributed Processing Symposium. pp. 1199–1211 (May 2011). https://doi.org/10.1109/IPDPS.2011.114
5. OpenMP Architecture Review Board: OpenMP Application Programming Interface 5.0 (Nov 2018), `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`
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 (Nov 2018). https://doi.org/10.1109/P3HPC.2018.00007
7. Quinlan, D., Liao, C.: The ROSE source-to-source compiler infrastructure. In: Cetus users and compiler infrastructure workshop, in conjunction with PACT. vol. 2011, p. 1. Citeseer (2011)
8. Smith, T.F., Waterman, M.S., et al.: Identification of common molecular subsequences. Journal of molecular biology **147**(1), 195–197 (1981)
9. Xiao, S., Aji, A.M., Feng, W.c.: On the robust mapping of dynamic programming onto a graphics processing unit. In: 2009 15th International Conference on Parallel and Distributed Systems. pp. 26–33. IEEE (2009)