Extending OpenMP for Machine Learning-Driven Adaptation

Chunhua Liao[1(✉)], Anjia Wang[2], Giorgis Georgakoudis[1], Bronis R. de Supinski[1], Yonghong Yan[2], David Beckingsale[1], and Todd Gamblin[1]

[1] Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
{liao6,georgakoudis1,bronis, beckingsale1,gamblin2}@llnl.gov
[2] University of North Carolina at Charlotte, Charlotte, NC 28223, USA
{awang15,yyan7}@uncc.edu

Abstract. OpenMP 5.0 introduced the `metadirective` directive to support compile-time selection from a set of directive variants based on OpenMP context. OpenMP 5.1 extended context information to include user-defined conditions that enable user-guided runtime adaptation. However, defining conditions that capture the complex interactions between applications and hardware platforms to select an optimized variant is challenging for programmers. This paper explores a novel approach to automate runtime adaptation through machine learning. We design a new `declare adaptation` directive to describe semantics for model-driven adaptation and also develop a prototype implementation. Using the Smith-Waterman algorithm as a use-case, our experiments demonstrate that the proposed adaptive OpenMP extension automatically chooses the code variants that deliver the best performance in heterogeneous platforms that consist of CPU and GPU processing capabilities. Using decision tree models for tuning has an accuracy of up to 93.1% in selecting the optimal variant, with negligible runtime overhead.

Keywords: OpenMP · Machine Learning · Runtime Adaptation

1 Introduction

Variant directives such as metadirective and declare variant are major new features introduced in OpenMP 5.0 [18] to improve performance portability by adapting OpenMP pragmas and user code at compile time. The OpenMP context, which consists of traits from active OpenMP constructs, devices, implementations or user-defined conditions, can guide adaptation. For example, the `metadirective` is conditionally resolved at compile time based on traits that define an OpenMP condition or context to select one of multiple directive variants. Based on a recommendation from a prior study [27], OpenMP 5.1 [19] added a new dynamic

S. Bhalachandra et al. (Eds.): WACCPD 2021, LNCS 13194, pp. 49–69, 2022.
https://doi.org/10.1007/978-3-030-97759-7_3

trait set that supports user-defined conditions. As a result, OpenMP programmers can now use dynamic conditions to guide the selection of directive variants. A canonical example is a user-defined loop iteration threshold (in a form of `N >=50000`) to decide if a parallel loop should execute on CPUs or GPUs.

While the `metadirective` enables runtime code adaptation, it falls to the programmer to determine the conditions upon which to select the best performing code variant. However, manually determining the appropriate conditions (such as the loop iteration threshold) is challenging. Meaningful values depend on complex interactions between applications and hardware platforms, thus by definition programmer choices are not portable. Further, there are many options and configurations of OpenMP compilation and the supporting runtime software that contribute to complexity, given the software stack configuration affects performance. Thus, users would benefit from automated mechanisms to select the best performing variant without manually specifying non-portable and error-prone runtime conditions.

In this paper, we explore a novel, portable approach of incorporating machine learning capabilities into OpenMP to automatically derive models used as dynamic conditions that guide directive variant selection. This paper makes the following contributions:

- A new directive and associated clauses to express essential semantics to achieve automated model-driven runtime adaptation of a given OpenMP region;
- Compiler transformations that enable runtime profiling, model building, and model-guided adaptation of an adaptive OpenMP region; and
- Extensions to a tuning runtime library that provides a small but powerful set of novel APIs to support the multiple stages needed for model-driven adaptation.

Experimentation shows that our adaptive OpenMP extension is able to select the best performing variant for the Smith-Waterman algorithm, which is particularly hard to tune, for a range of input sizes, on heterogeneous platforms with CPU and GPU processing capabilities.

2 A Motivating Example

We use the Smith-Waterman algorithm [22] to demonstrate the need for automated OpenMP adaptation. This dynamic algorithm finds the optimal local alignment of a subsequence within a larger DNA or RNA sequence by calculating a distance (or similarity) matrix. The scoring process has a wavefront computation pattern, as Fig. 1 shows, due to data dependencies between points of the matrix computation. The algorithm has $O(M \times N)$ time complexity in which M and N are the lengths of the two sequences. The space complexity is also $O(M \times N)$ due to matrices used for computing scores and backtracking.

Figure 2 shows a typical OpenMP CPU implementation of the Smith-Waterman algorithm's scoring step. It parallelizes the inner loop iterating on elements of each wavefront line. Similarly, Fig. 3 shows an OpenMP GPU offload version, which moves the data used on GPU before the outer loop, and copies back results after processing completes to reduce data transfer overheads.

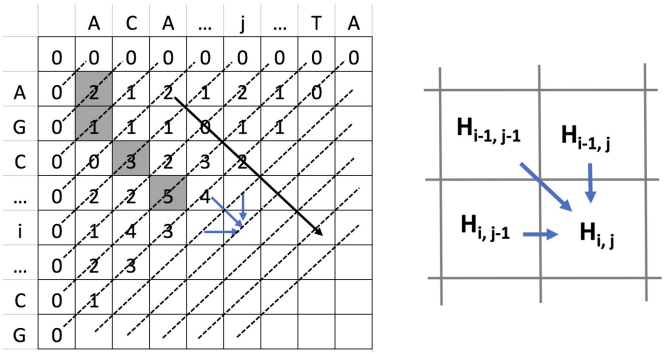

Fig. 1. Wavefront Computation Pattern of the Smith-Waterman Algorithm

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

Fig. 2. OpenMP CPU Implementation of the Smith-Waterman Algorithm

We compare the performance of three versions (serial CPU, OpenMP CPU and OpenMP GPU) for two sequences of equal input lengths, by ranging their length from 32 to 15,000 with a stride of 256. Our comparison uses one compute node of the Corona cluster of the Livermore Computing Center. It has two AMD EPYC 7401 processors, each with 24 cores clocked at 2 GHz, 250 GB memory, and four AMD MI50 GPUs. We compile with Clang 12.0.0 and ROCm v4.1.0, with the −O3 option. Figure 4 shows the scoring kernel execution time. The serial version performs the best for input sizes ranging from 32 to 6048. From 6304 to 8864, the OpenMP GPU version is the best choice. Finally, the OpenMP CPU version performs the best for input problem sizes ranging from 9120 to 15,000. It would be challenging for programmers to manually determine such conditions to select the best variants for different software and hardware configurations.

In general, the optimal choice among OpenMP variants varies significantly depending on the application kernels, input sizes, machines and compilers. Manual specifying conditions guiding the optimal choice is neither practical nor portable. Thus, we propose a new mechanism to automate adaptation without user intervention.

```
long long int nDiag = M + N - 1;
#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. 3. OpenMP GPU Implementation

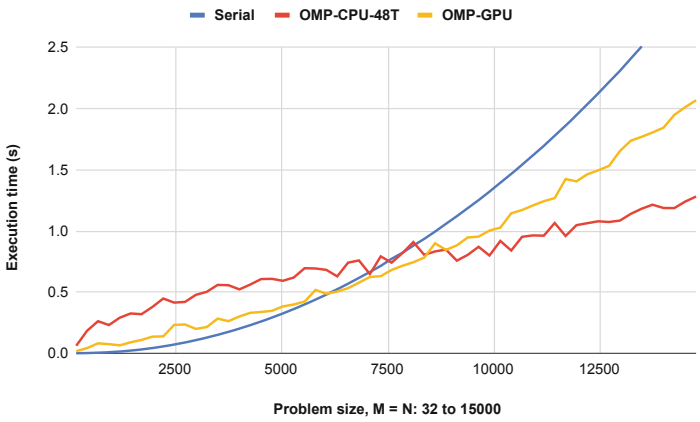

Fig. 4. Performance of Three Versions of Smith-Waterman Running on Corona

3 A Vision

We envision that future programming models, including OpenMP, will allow programmers to express rich semantics related to automated adaptation using machine learning techniques. Seamless integration of programming models and machine learning has multiple benefits. For one, direct support in a programming model will make machine learning techniques more accessible. Programmers will be relieved from manually assembling machine learning pipelines to optimize each program. Further, the integration will improve performance portability and productivity of programming systems.

As Fig. 5 shows, we extend OpenMP to enable machine learning-driven adaptation. Our extension uses a new directive, `declare adaptation`, to generate transformed (or lowered) code variants for each annotated code region. The lowered code implements an execution pipeline that includes profiling, model building and adaptation. Common functionality in those steps is supported by a runtime library to simplify the compiler transformation.

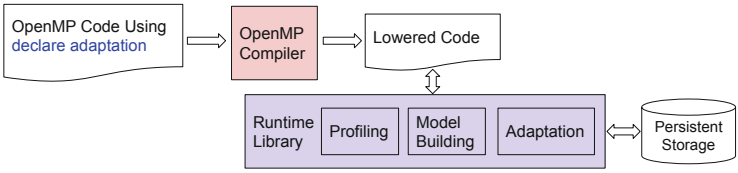

Fig. 5. Machine Learning-Driven Adaptive OpenMP

A generated executable file may run in different modes. The first run transparently collects profiling data for selected adaptive code regions. Once sufficient data is collected, the executable automatically builds a predictive machine learning model for each selected code region. Finally, the internally generated machine learning models guide runtime selection of the best variants for each region. Profiling, model building and model-driven adaptation may finish within the first run of a program, especially for those that use iterative algorithms, which often can easily generate sufficient training data.

The extended OpenMP also supports collecting profiling data across multiple runs, which is essential if a single run does not generate sufficient training data. Those profiling data accumulate in persistent storage to enable model building and adaptation in later runs. Also, previously trained models are saved too for reuse in later runs, avoiding unnecessary profiling and model building. In summary, the execution of an adaptive OpenMP program checks if previous profiling data or machine learning models are available in order to initialize adaptive execution. The following sections elaborate on the design and implementation of the `declare adaptation` directive.

4 The `declare adaptation` Directive

The proposed `declare adaptation` allows programmers to express semantics related to machine learning-driven automatic runtime adaptation. In our present design, `declare adaptation` works with metadirectives. Code regions annotated with `metadirecive` naturally provide multiple directive variants for adaptation. Future work will explore its composability with other directives.

When a code region enclosed by `metadirective` immediately follows the `declare adaptation` directive, each `when` and `default` clause is treated as a code variant that can be automatically selected. Internally, each code variant is assigned a unique variant ID, starting from 0.

Using `declare adaptation` overrides the context-selector-specifications of the `when` clauses, using instead user-provided features as part of the `adaptation` directive to model the performance of possible variants and select the predicted optimal one. Programmers can also entirely avoid specifying context selectors in the `metadirective`. The machine must support a valid execution context to enable the execution of all variants of the `metadirective` so the runtime can freely activate any of them for profiling, modeling and subsequent selection.

4.1 Syntax and Semantics of `declare adaptation`

`declare adaptation` has the following syntax:

```
#pragma omp declare adaptation [clause[[,]clause]...] new-line.
```

Semantically, `declare adaptation` allows OpenMP programmers to specify that the associated OpenMP region is transformed into adaptive code using online performance profiling and model-driven adaptation. The compiler generates a lowered multi-variant code region, leveraging runtime functions to support profiling, model building and tuning, as Fig. 5 shows.

The possible associated clauses are the following:

- `model(model_type_name)`,
- `feature([modifiers]: list)`,
- `model_name(region_id)`,
- `use_model(region_id)`, and
- `variant_mapping(list-of-mapped-model-region-variant-ids)`.

The parameter of the optional `model` clause indicates the type of machine learning model to use. If this clause is not specified, the model type is implementation defined. Values of `model_type_name` are supervised machine learning models for classification problems, such as `logistic_regression`, `decision_tree`, `random_forest`, `artificial_neural_network` and `support_vector_machine`.

The mandatory `feature` clause specifies a list of variables that serve as model features. Any program variable in scope may be used as a feature of the machine learning model. In addition, we assume that a set of special OpenMP variable identifiers, including `omp_num_threads` and `omp_num_teams`, are available to enable modeling of the OpenMP context. The clause may be repeated as often as necessary to describe all variables that the model should use as features.

Further, the `feature` clause accepts two optional modifiers that specify additional information for listed items. An example is `feature (range[0:30000], min_sample_points(25): N)`. The `[lower_bound:upper_bound]` argument of the `range` modifier specifies a range expression for the variables in `list`. Both bounds are inclusive values of either integer or floating point types that define a search space of feature values. The integer argument of the `min_sample_points` modifier is a hint on the number of data points to sample for those features. This modifier guides the implementation in determining if sufficient data have been gathered for adaptation. A possible formula for an implementation is $Min\_dataset\_size = (min\_f1 \times min\_f2 \times \cdots \times min\_fn) \times code\_variant\_count$. For example, a code region with 3 code variants and 2 feature variables of `min_sample_points(10)` has a suggested training set size of $10 * 10 * 3 = 300$ data points.

The `model_name` and `use_model` clauses specify the model for guiding this region's adaptation and they form an exclusive clause set. This means that at most one of them can be used within a `declare adaptation` directive. If neither clause is specified, the effect is as if `model_name` is specified with an implementation-defined unique `region_id`. The `model_name` clause indicates that the associated region is a primary region for profiling, model building, and model-driven adaptation. An example primary region is a loop doing intensive computation. The user must specify a unique identifier for the region in the `model_id`

argument. The `use_model` clause indicates that this region is an associated region to a primary region, so it should use the choices made by that corresponding primary region's model. An example associated region is a data transferring region preparing data for later computation. Its required argument specifies the ID of its primary region. Multiple regions may use the primary region's model.

The `variant_mapping` clause is valid and required only if the `use_model` clause is specified. It establishes a code variant mapping between a primary region and the associated region. The required mapping allows the region to have a different number of variants from the primary region. It specifies which variant an implementation should use based on the model decision of the primary region. The size of the list of mappings is equal to the number of code variants of the associated region. Each list item is a code variant ID of the corresponding primary region. For example, `variant_mapping(2,3)` means that the associated region has 2 code variants (with IDs 0 and 1) that are mapped to variant IDs 2 and 3 of the corresponding primary region. The associated region must not have more code variants than the region specified by `region_id`.

4.2 Examples Using `metadirective`

We demonstrate the use of `declare adaptation` within the Smith-Waterman algorithm. Figure 6 shows two nested loops that comprise the similarity score computation kernel. Our version specifies three code variants using a `metadirective`: serial, OpenMP CPU threading, and OpenMP GPU offloading (line 7–11). We use `declare adaptation` on line 6, right before `metadirective`, to specify a decision tree model trained on a single feature (`nDiag` derived from the lengths of the two sequences), which is the number of the wavefront lines of the similarity matrix. For two sequences with size M and N, the following relationship holds: $nDiag = M + N - 1$.

The choice of `nDiag` instead of the inner loop bound `nEle` is based on experiments of a prior study [27] which reports that `nDiag` is a good indicator for tuning. Choosing `nDiag` means that for a given pair of M and N values, a single code variant is activated for the entire execution of the program. So adaptation happens at a coarse granularity across different executions of the entire program. In comparison, if we choose `nEle`, adaptation happens at a fine granularity, across different wavefront lines. This fine-grain adaptation requires data transfers inside the outer loop, which introduces excessive data copy overhead across wavefront lines as Fig. 6 shows. Our experiments on Corona confirms that this overhead results in severe performance degradation compared to its baseline serial version using input sizes of 2000 by 2000 (56 s for the fine-grain adaptation version vs. 0.04 s for the serial version), hence the motivation for coarse grain adaptation.

A further optimization is using a data region that encloses both the outer and inner loop. Thus, data is copied between devices only when entering and exiting that region. We render the data region's execution adaptive by adding two more adaptive `metadirective` definitions (line 2–5 and 25–27 in Fig. 7). Variant selection for those two regions corresponds to the decision made for the primary region (line 13–21). When the primary region's variant ID 2 is active at runtime,

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

#pragma omp declare adaptation model(decision_tree) feature(nDiag)
#pragma omp metadirective  \
   when (:)                    /* variant 0: serial*/ \
   when (:parallel for)        /* variant 1: CPU threading */ \
   default(target teams distribute parallel for map (to:a[0:m], ...) \
        map(tofrom: H[0:asz], ...) /* variant 2: GPU offloading */
  for (j = 0; j < nEle; ++j)
    similarityScore(si-j, sj+j, H, P, &maxPos);
}
```

Fig. 6. Basic Use of `declare adaptation` with `metadirective`

which selects GPU execution, the two associated regions are activated (using `variant_mapping(2)`). This example also shows that when an associated region executes before its primary region, the corresponding feature variables should be available at the entry point of this associated region for model evaluation. Also, the values of those variables should not change before entering the primary region, thus stay invariant. Then it is possible for the runtime to activate the mapped variants in both regions. Otherwise, a primary region should execute before its associated regions to forward its model decision.

5 Implementation

We design and implement a compiler-runtime system that translates OpenMP programs with the `declare adaptation` directive into adaptive executables. Figure 8 shows that our source-to-source compiler (based on ROSE [21]) translates an OpenMP program that uses `declare adaptation` into lowered adaptive OpenMP code. We then translate that representation into a final executable using Clang/LLVM. The lowered adaptive OpenMP code and our runtime system (based on Apollo [4,26]) implement runtime profiling, model building and model-guided adaptation. The runtime uses the OpenCV machine learning library [7] to build machine learning models from profiling data. To support reuse of profiling data and ML models across executions, the runtime system loads and stores training data and models between main memory and persistent storage (e.g., the file system).

5.1 Compiler Support

We use ROSE to prototype our compiler implementation. Developed at LLNL, ROSE [21] is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for Fortran and C/C++ applications.

```
//Copy the data to GPU if the GPU version will be used later.
//Primary region's variant #2 is mapped to variant id #0 here.
#pragma omp declare adaptation use_model("scoring_loop") variant_mapping(2)
#pragma omp metadirective \
 when(: 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);

// The primary region with 3 variants
#pragma omp declare adaptation model_name("scoring_loop") \
   model(decision_tree) feature(nDiag)
  #pragma omp metadirective \
   when (: )  \
   when (: parallel for private(j))  \
   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
//Primary region's variant #2 is mapped to variant id #0 here.
#pragma omp declare adaptation use_model("scoring_loop") variant_mapping(2)
#pragma omp metadirective \
 when( : target exit data map(from: H[0:asz], P[0:asz], maxPos))
```

Fig. 7. Optimized Use of `declare adaption`

ROSE supports OpenMP 3.0 [13] and part of 4.0 [15]. More recently, it was used to prototype the dynamic extension of metadirective [27].

Our prototype compiler includes an extended OpenMP parser and internal AST to support `declare adaptation`. It also translates an AST that represents a `metadirective` region affected by `declare adaptation` into one that uses a switch-case statement to enable machine-learning based adaptation. We lower that AST into source files that use OpenMP 4.5 directives (using CPU threading and GPU offloading directives). Finally, Clang/LLVM compiles the lowered code and links it with the Apollo runtime library to generate the final executable.

The lowered code uses several runtime interface functions to support all stages in the model-driven adaptation workflow. The workflow first collects execution time of variants associated with user-specified features of a code region. It then processes those data into feature vectors suitable for machine learning and feeds those training data into OpenCV to generate the model. Finally, it evaluates at runtime the generated model to select code variants.

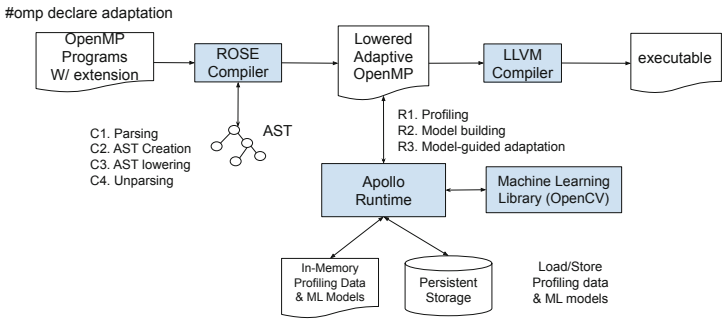

Fig. 8. Design and Implementation of Adaptive OpenMP

Figure 9 shows the lowered code for the input code in Fig. 7. Each code variant of a `metadirective` region under `declare adaptation` control is placed in a `case` statement of a `switch` statement. We synchronize the primary and the associated adaptive regions using a region name identifier and variant ID numbers. In this example, the two corresponding regions that copy data between the CPU and GPU are only activated when the primary region's code variant 2 (GPU offloading version) is activated. The lowering step leverages runtime support to reuse the same generated code for different stages of the workflow. For example, we use the `getPolicyIndex()` function at line 27 to pick a policy to support both training and production runs. Details of the runtime support are explained in the next subsection.

5.2 Runtime Support

We extend Apollo [4,26] to serve as our runtime library. Apollo was originally applied as an auto-tuning extension of RAJA [12] that uses pre-trained, reusable machine learning models to tune data-dependent kernels at runtime. Nevertheless, Apollo's modular design simplifies support of runtime adaptation for non-RAJA codes, OpenMP in our case.

For adaptive code regions, an internal C++ Region class tracks the associated features, manages training data and activates the model. Each code region can have multiple code variants, such as one for CPU and another for GPU. Apollo treats each variant as a distinct execution policy of the region to measure its execution time. The runtime uses these measured times to train a machine learning model for suggesting the best execution policy, which corresponds to the fastest code variant.

Apollo exposes a small set of runtime API functions to support data collection, model building and model-guided adaptation through two concepts: training models and tuning models. Training models are special models that activate different code variants to collect data during training runs, while tuning models are generated machine learning models to select optimal code variants (or equivalently execution policies) to activate during production runs. The active model

```
/* 1. Translation of the first dependent region*/
/* Create or obtain the main region*/
/* Parameters: unique region id, feature count, and variant count. */
Apollo::Region *region1 =
Apollo::instance()->getRegion("scoring-loop", 1,  3);
/* feature vector of size 1 */
region1->begin({(float)nDiag} );

// Get the policy to execute from Apollo
int policy = region1->getPolicyIndex();
if (policy ==2)
{
  #pragma omp target enter data map(to:a[0:m-1], b[0:n-1]) \
    map(to: H[0:asz], P[0:asz], maxPos)
}
region1->end();

for (i = 1; i <= nDiag; ++i) {
  /* ....some code omitted here... */

  /* 2. Translation of main adaptation  region*/
  Apollo::Region *region = Apollo::instance()->getRegion(
  "scoring-loop", 1, 3, 1);
  region->begin({ (float)nDiag });

  /* calling a training or real model to select a code variant */
  int policy = region->getPolicyIndex();

  switch (policy)    {
    case 0:  /* variant 0: serial */
      {  /*  code omitted here  */}
    case 1: /* variant 1: CPU threading */
      {
      #pragma omp parallel for
      for (j = 0; j < nEle; ++j)
        similarityScore(si-j, sj+j, H, P, &maxPos);
      break;
      }
    case 2: /* variant 2: GPU offloading */
      {
      #pragma omp target teams distribute parallel for map (...)
      for (j = 0; j < nEle; ++j)
        similarityScore(si-j, sj+j, H, P, &maxPos);
      break;
      }
    default:
       /* .. error handling here... */
  }
  region->end();
}

/* 3. Translation of the 2nd dependent region, code omitted here*/
```

Fig. 9. Lowered Code Enabling Profiling, Model Building and Adaptation

field of the Region class can be set to a training or tuning model. Thus, Apollo re-uses the same API interface function, `getPolicyIndex()` to return either a training or optimal code variant, which simplifies the compiler transformation.

Apollo provides two builtin training models (Random and Round-Robin) to support profiling code variants. A training run with a given input data may invoke a region multiple times and at every invocation the Random model randomly selects a code variant of the region to measure performance. Similarly, the Round-Robin model cyclically selects each code variant for performance profiling. By default, Apollo averages the measured execution times for each code variant when collecting measurements during training. The tuning models include the Static model (returns a fixed policy choice) and a set of machine learning models supported by OpenCV such as Decision Tree, Random Forest and Support Vector Machine.

We extend Apollo in several ways. Specifically, we add support for collecting and accumulating profiling data across multiple executions to ensure there is sufficient training data for model building. Original Apollo requires an explicit function call to trigger model building. We automatically trigger model building when sufficient data have been collected based on the semantics of `declare adaptation`. Additionally, we apply the Static model as a training model to support coarse grain adaptation, by using a fixed code variant throughout an entire program execution for a given input data size. Lastly, we add a new configuration option to use the accumulated total execution time instead of the average time as the input performance feature for OpenCV-generated models to enable coarse grain adaptation.

Overall, our implementation uses six runtime functions to support adaptation. `Apollo* Apollo::instance()` is used to initialize the runtime and obtain a handle to it.

`Apollo::Region* Apollo::getRegion (string& region, int feature_count, int policy_count, int model_type)` obtains a managed code region's internal C++ object by its name. If the region object exists, the function directly returns it. Otherwise, the runtime creates and initializes it, using the specified feature count, policy count, and machine learning model type. Each code region object's active model field is initialized to a tuning or training model. At first, the function tries to load an existing tuning model file saved on disk for the region. If the model file does not exist, a default training model (Static, Random or Round-Robin) is configured for the region. Similarly, if training across multiple executions is requested, the runtime tries to initialize the region object's training data field by loading an existing training dataset for the region from disk.

The `Apollo::Region::begin(std::vector<float>)` indicates the beginning of a managed code region. The parameter of this function is a vector of features of float type. The length of the vector matches the number of features of the code region. This function starts a timer for the managed code region.

`Apollo::Region::getPolicyIndex()` calls the active model associated with the code region to return a preferred policy ID. If the model is a training model, it picks a variant for profiling. Otherwise, a tuning model (such as a decision

tree model) selects an optimal code variant by evaluating the model with the set features associated with the region as inputs.

`Apollo::Region::end()` stops the timer for the managed code region and adds information (such as the measured execution time, the executed policy, and the feature vector) into the region's training data field. Additionally, if the average execution time is used as training data, it checks if sufficient profiling data have been collected for the region, in which case the function triggers data processing and model building using the collected data. Also, it stores the generated model for later use.

`Apollo::~Apollo()`, the destructor of the Apollo runtime object, is implicitly called when a program ends. If the accumulated total execution time of regions is used to train models, this function will check if sufficient training data have been collected and trigger model building for later re-use. It saves any collected training data and generated models to disk.

6 Evaluation

6.1 Software and Hardware Configurations

We evaluate the effectiveness of the proposed OpenMP extension using the adaptive Smith-Waterman algorithm shown in Fig. 7. The corresponding serial, OpenMP CPU threading, and OpenMP GPU offloading versions are used as baseline, non-adaptive versions. Picking `nDiag` as the feature requires multiple runs using different problem sizes to collect training data. The minimal sample points per feature (specified using `min_sample_points(val)`) is configured to have three values: 25, 50, or 100. During the training runs, the input problem size range is fixed to be between 32 to 15,000. Three different strides (128, 256 and 512) in that range are used to generate sufficient training data for the three sampling configurations.

For each input problem size, all code variants are measured in the same batched run for collecting training data. The training run is repeated five times and median values are used as performance measurements. Decision tree models are created and stored in `yaml` files for later reuse. Once the model files are available, the execution of the program enters the production run mode. Different input problem sizes (160 to 15,000 with a stride of 256) are picked to evaluate the generated models in production runs.

Two machines, Corona and Pascal, are used for the experiments, with their details shown in Table 1. For the OpenMP CPU version, we use the number of threads matching the number of physical cores on a machine to avoid system noise caused by oversubscribing CPU cores.

6.2 Performance Results

Figure 10 and 11 show the execution time of different versions of the Smith-Waterman algorithm on the two machines. The adaptive version uses the decision

Table 1. Software and Hardware Configurations

	LLNL Corona	LLNL Pascal
CPU	AMD EPYC 7401 2.00 GHz	Intel Xeon E5-2695 v4 2.10 GHz
Cores	2 sockets × 24 physical cores	2 sockets × 18 physical cores
Main Mem	256 GB	256 GB
GPU	AMD Radeon Instinct MI50	NVIDIA Tesla P100
Device Mem	16 GB	16 GB
OS	TOSS 3	Red Hat Enterprise Linux 7.6
Clang/LLVM	12.0.0	11.0.0
Compiler Options	−O3	−O3
GPU Driver	AMD ROCm 4.1.0	NVIDIA CUDA toolkit 10.2.89

tree model generated using the minimum sample points per feature set to 50. It is clear that the performance of the adaptive version, denoted with a green line, closely matches the best choices, especially for Pascal. On Corona, the adaptive version does not pick the serial version, which is the fastest, for input size range between 32 and 5,000. However, the execution time of the predicted variant is very close to serial, so performance is near-optimal anyway.

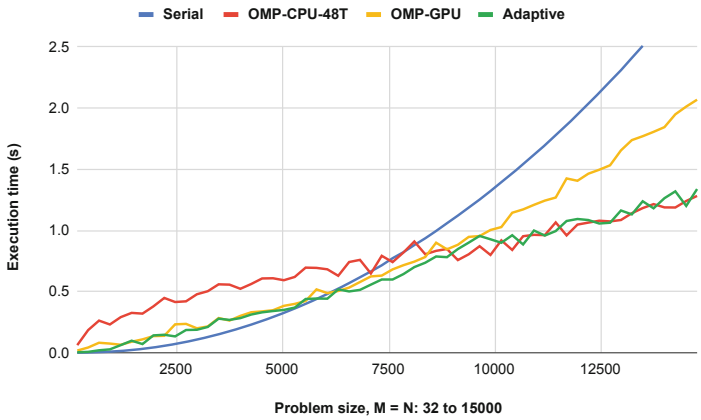

Fig. 10. Execution Time of Different Versions of Smith-Waterman on Corona

6.3 Accuracy of Prediction Models

The accuracy of the generated models is evaluated by comparing the predicted best code variants against the ground truth of optimal variants for a set of production runs using the selected input problem sizes. Note that we purposely select a different set of 58 input problem sizes (160 to 15,000 with a stride of 256) in the production run, which are unseen in the training runs. To generate

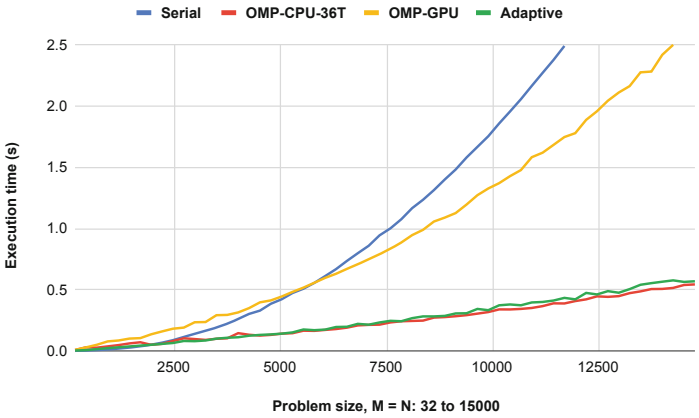

Fig. 11. Execution Time of Different Versions of Smith-Waterman on Pascal

the ground truth, we run the baseline versions using the same input problem sizes selected for the production runs to identify the fastest execution variant.

Table 2 shows the accuracy evaluation results. For the three values of minimal sample points per feature (25, 50 and 100), the created decision tree models show the best accuracy of 79.31% for Corona and 93.10% for Pascal (among table's columns named `Median`).

Table 2. Prediction Accuracy of Smith-Waterman under Multiple Configurations

Training samples	25			50			100		
Platform	Median	Majority Vote	Majority Vote (sklearn)	Median	Majority Vote	Majority Vote (sklearn)	Median	Majority Vote	Majority Vote (sklearn)
Corona	72.41%	77.59%	72.41%	79.31%	82.76%	84.48%	75.86%	77.59%	75.86%
Pascal	93.10%	93.10%	94.83%	93.10%	93.10%	93.10%	93.10%	93.10%	93.10%

We investigated possible causes for the limited accuracy of the models generated on Corona. It is observed that the serial version's timing information collected in the training data is not exactly the same as the corresponding baseline version without code instrumentation. The OpenMP CPU and GPU versions do not show such a problem. We suspect that code instrumentation (using runtime API calls) prevents the compiler from applying some optimizations on the serial version of the code. Both OpenMP versions already use outlining which hurts optimizations, so additional instrumentation causes much less negative impact. To test this hypothesis, we re-run the experiments with compiler optimizations turned off (using the −O0 compilation flag). The adaptive version then made 55 correct choices out of 58 input problem sizes, which leads to an accuracy of 94.83%. Only three sizes have wrong predictions. These three wrong predictions happen near the crossover points in Fig. 10 where different policies have similar performance.

We also tried another method to process and label the raw data. The original method has two steps: 1) picking the median execution time of 5 runs for each variant for a given input size, 2) finding the best variant using the median values. The new method first finds the best variant within each batched run including three code variants using a given input size. Then a majority vote is used to decide the final best variant out of 5 repeated batched runs. The second method leads to better accuracy for Corona. For example, accuracy increases to 82.76% when using 50 samples per feature on Corona (Table 2). On Pascal, either of those methods shows similar accuracy. Therefore, we deem the second method as more accurate. Out of curiosity, we feed the identical training data in the second method into another machine learning package, Python scikit-learn v0.24.2. The prediction accuracy numbers overall are similar to what Apollo generates on two machines. Nevertheless, the loss in accuracy is small and our ML approach results in near-optimal execution decisions, evidenced by the performance measurements.

6.4 Overhead Analysis

There are three kinds of overheads in the adaptive version: the one-off overhead to perform the training run for data collection, the one-off overhead for model building, and the instrumentation and model evaluation overhead in production runs. The observed overheads depend on many factors, including the number of data points, the input size of a program, and the choice of the machine learning model, To measure those overheads, we pick the configuration of using 50 sample points and three input problem sizes from 32 to 15,000, which are 4,128, 8,480, and 12,576.

Table 3. Execution Time of Baseline, Training and Production Runs on Corona

	Baseline Run			Training Run			Production Run	
M == N	Serial	OMP-CPU	OMP-GPU	Serial	OMP-CPU	OMP-GPU	Execution Time	Predicted Variant
4128	0.214	0.552	0.286	0.309	0.573	0.289	0.304	OMP-GPU
8480	0.967	0.871	0.803	1.094	0.798	0.793	0.757	OMP-GPU
12576	2.164	1.042	1.585	2.562	1.051	1.644	1.077	OMP-CPU

Table 3 shows the measured execution time for different runs using different configurations on Corona. Results on Pascal are similar, so we omit them. Table 4 shows overhead in percentage numbers for training runs and production runs. The serial variant's training runs have significantly high overhead compared to the corresponding baseline runs. For example, it took 0.309 s while its baseline version took only 0.214 s for the input size of 4,128, indicating an overhead of 44.74%. Again, the reason is that code instrumentation prevents certain compiler optimizations being applied, which has a more negative performance impact on the serial version than the OpenMP versions. We measured the training

overhead of the serial version using −O0 compilation. The overhead then reduces significantly to 9.85% for the input size of 4,128.

The time cost of building the models is negligible. It took only 0.00684 s on average. The corresponding 95% confidence interval is 0.00684 ± 0.0038 s. It only happens once for a configuration. The code instrumentation and runtime adaptation in the production runs have overhead up to 6.28%.

Table 4. Overhead Percentage

	Training Run			Production Run
M==N	Serial	OMP-CPU	OMP-GPU	Predicted Variant
4128	44.74%	3.76%	1.05%	6.28%
8480	13.14%	−8.37%	−1.22%	−5.81%
12576	18.39%	0.84%	3.73%	3.28%

For the input size of 8480, there are three negative overhead numbers for the two training runs using OMP-CPU and OMP-GPU, and the production run. We looked into confidence interval values for the relevant measurements. The results show that the measured execution times of training and production runs do have significant overlapping with their baseline runs. For example, the baseline OMP-GPU has a confidence interval of 0.803 ± 0.0444 s while its production run's confidence interval is 0.757 ± 0.0553 s. As a result, we conclude that there is no statistically significant overhead.

Overall the implementation has negligible impact on execution time for training and production runs using CPUs or GPUs.

7 Related Work

Machine-learning based compiler optimization has been studied extensively for decades. Wang et al. [25] provide a comprehensive survey of machine learning techniques used to guide compiler optimization. Ashouri et al. [2] summarize machine learning techniques used to tackle two particular compiler optimization problems: optimization selection and phase-ordering. A notable project, Milepost GCC [9], combines production-quality GCC with machine learning to adapt to different architectures and predict profitable optimizations. Luk et al. [16] profile execution variants to build linear regression models in order to determine the optimal splitting ratio between CPU and GPU computation. Grewe et al. [10] uses decision tree models to decide if it is profitable to run OpenCL kernels on GPUs. Hayashi et al. [11] used offline, supervised machine-learning techniques to select preferred computing resources between CPUs and GPUs for individual Java kernels using a JIT compiler. DeepTune [8] uses raw code to develop a deep neural network to guide optimal mapping for OpenCL programs.

Given the flexibility of OpenMP, there is growing interest in autotuning of OpenMP programs to enable performance portability across different platforms. Liao et al. [14] apply source code outlining to enable autotuning of OpenMP loops from large applications. Sreenivasan et al. [23] introduce a lightweight OpenMP pragma autotuner to optimize scheduling policies, chunk sizes, and thread counts. In [20], 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.

Autotuning techniques are also well-studied for high performance computing, but dedicated mostly for loop transformation and for performance optimization, such as those in earlier works including POET [28] and CHILL [5]. Recent work, such as OpenTuner [1], provides a general-purpose optimization tool that could help users find the best configuration to improve the performance over a group of compilation parameters as search space. CLTune [17], as a generic tuner for OpenCL kernel, adopts a similar strategy. Active Harmony [24] is a runtime tuning framework for searching tuning variables for the configuration that delivers optimal performance. Indicatively, 3D-FFT has shown 1.76× speedup when using online tuning methods implemented with Active Harmony. Another Active Harmony-based tool, named ANGEL [6], is developed to tune multiple functions for balancing the trade-off between computing time and power consumption. Bari et al. in [3] present ARCS framework for tuning OpenMP program targeting on optimizing power consumption.

Our work differs from the aforementioned studies in that we define combined language, compiler and runtime support methods to directly incorporate machine learning into a programming model, which enables automated model-driven runtime adaptation. Our approach significantly enhances portability and productivity of OpenMP.

8 Conclusion

In this paper, we have proposed a new OpenMP extension, `declare adaptation`, for programmers to express semantics related to machine learning-driven runtime adaptation. This directive is used with `metadirective` to guide the selection of an optimal choice of an OpenMP code region with multiple variants, using a machine learning model automatically built from user-specified features. Experimentation shows that this new extension improves the performance portability and productivity of OpenMP by alleviating the problem of manually deciding adaptation conditions for different software and hardware configurations. Additionally, this approach makes machine learning techniques more easily accessible to HPC developers.

In the future, we plan to expand the `declare adaptation` directive to apply to more types of OpenMP directives besides `metadirective`. Leveraging the prototype for the combined compiler and runtime support, we intend to migrate

the implementation to a production quality compiler, such as Clang/LLVM, and also evaluate our approach on more applications and more diverse platforms.

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 (LLNL-CONF-826432). The OpenMP language extension work was supported by the U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Research. The compiler and runtime work were supported by LLNL-LDRD 21-ERD-018.

Artifact Availability Statement

Summary of the Experiments Reported: We ran Smith Waterman algorithm on LLNL's Corona and Pascal supercomputer. The detailed software configurations are given in the experiment section of the paper.

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: All author-created data artifacts are maintained in a public repository under an OSI-approved license.

Proprietary Artifacts: None of the associated artifacts, author-created or otherwise, are proprietary.

List of URLs and/or DOIs where artifacts are available:

https://doi.org/10.5281/zenodo.5706501

References

1. Ansel, J., et al.: OpenTuner: an extensible framework for program autotuning. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, pp. 303–316 (2014)
2. Ashouri, A.H., Killian, W., Cavazos, J., Palermo, G., Silvano, C.: A survey on compiler autotuning using machine learning. ACM Comput. Surv. (CSUR) **51**(5), 1–42 (2018)
3. Bari, M.A.S., et al.: ARCS: adaptive runtime configuration selection for power-constrained OpenMP applications. In: 2016 IEEE International Conference on Cluster Computing (CLUSTER), pp. 461–470, September 2016. https://doi.org/10.1109/CLUSTER.2016.39
4. Beckingsale, D., Pearce, O., Laguna, I., Gamblin, T.: Apollo: reusable models for fast, dynamic tuning of input-dependent code. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 307–316. IEEE (2017)
5. Chen, C., Chame, J., Hall, M.: CHiLL: a framework for composing high-level loop transformations. Technical report, Citeseer (2008)

6. Chen, R.S., Hollingsworth, J.K.: ANGEL: a hierarchical approach to multi-objective online auto-tuning. In: Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers, pp. 1–8 (2015)
7. Culjak, I., Abram, D., Pribanic, T., Dzapo, H., Cifrek, M.: A brief introduction to OpenCV. In: 2012 Proceedings of the 35th International Convention MIPRO, pp. 1725–1730. IEEE (2012)
8. Cummins, C., Petoumenos, P., Wang, Z., Leather, H.: End-to-end deep learning of optimization heuristics. In: 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 219–232. IEEE (2017)
9. Fursin, G., et al.: Milepost GCC: machine learning enabled self-tuning compiler. Int. J. Parallel Prog. **39**(3), 296–327 (2011)
10. Grewe, D., Wang, Z., O'Boyle, M.F.: Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 1–10. IEEE (2013)
11. Hayashi, A., Ishizaki, K., Koblents, G., Sarkar, V.: Machine-learning-based performance heuristics for runtime CPU/GPU selection. In: Proceedings of the Principles and Practices of Programming on the Java Platform, pp. 27–36 (2015)
12. Hornung, R.D., Keasler, J.A.: The RAJA portability layer: overview and status (2014)
13. Liao, C., Quinlan, D.J., Panas, T., de Supinski, B.R.: A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 15–28. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13217-9_2
14. 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
15. Liao, C., Yan, Y., de Supinski, B.R., Quinlan, D.J., Chapman, B.: Early experiences with the OpenMP accelerator model. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 84–98. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40698-0_7
16. Luk, C.K., Hong, S., Kim, H.: Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 45–55. IEEE (2009)
17. Nugteren, C., Codreanu, V.: CLTune: a generic auto-tuner for OpenCL kernels. In: 2015 IEEE 9th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip, pp. 195–202. IEEE (2015)
18. 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
19. 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
20. 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

21. 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)
22. Smith, T.F., Waterman, M.S., et al.: Identification of common molecular subsequences. J. Mol. Biol. **147**(1), 195–197 (1981)
23. 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
24. Tapus, C., Chung, I.H., Hollingsworth, J.K.: Active harmony: towards automated performance tuning. In: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, SC 2002, pp. 1–11. IEEE Computer Society Press, Washington, DC (2002)
25. Wang, Z., O'Boyle, M.: Machine learning in compiler optimization. Proc. IEEE **106**(11), 1879–1901 (2018)
26. Wood, C., et al.: Artemis: automatic runtime tuning of parallel execution parameters using machine learning. In: Chamberlain, B.L., Varbanescu, A.-L., Ltaief, H., Luszczek, P. (eds.) ISC High Performance 2021. LNCS, vol. 12728, pp. 453–472. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78713-4_24
27. 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
28. Yi, Q., Seymour, K., You, H., Vuduc, R., Quinlan, D.: POET: parameterized optimizations for empirical tuning. In: 2007 IEEE International Parallel and Distributed Processing Symposium, pp. 1–8. IEEE (2007)