SKOPE: A Framework for Modeling and Exploring Workload Behavior

Jiayuan Meng†, Xingfu Wu†, Vitali Morozov‡, Venkatram Vishwanath†, Kalyan Kumaran‡, and Valerie Taylor†
Argonne National Laboratory†
{meng, morozov, venkatv, kumaran}@anl.gov
Texas A&M University‡
{wuxf, taylor}@cse.tamu.edu

ABSTRACT
Understanding workload behavior plays an important role in performance studies. The growing complexity of applications and architectures has increased the gap among application developers, performance engineers, and hardware designers. To reduce this gap, we propose SKOPE, a SKeleton framewOrk for Performance Exploration, that produces a descriptive model about the semantic behavior of a workload, which can infer potential transformations and help users understand how workloads may interact with and adapt to emerging hardware. SKOPE models can be shared, annotated, and studied by a community of performance engineers and system designers; they offer reusability in the frontend and versatility in the backend. SKOPE can be used for performance analysis, tuning, and projection. We provide two example use cases. First, we project GPU performance from CPU code without GPU programming or accessing the hardware, and are able to automatically explore transformations and the projected best-achievable performance deviates from the measured by 18% on average. Second, we project the multi-node scaling trends of two scientific workloads, and are able to achieve a projection accuracy of 95%.

1. INTRODUCTION
The constantly growing scale of scientific applications has been motivating the design and adoption of new algorithms and system hardware. However, the increasing scale and complexity in both workload and hardware lead to higher uncertainty in the performance outcome, which may eventually lengthen development cycles and reduce science productivity. These performance uncertainties often arise from the interplay between workload and hardware. For example, loop structures may affect cache footprints; parallelism strategies may affect memory bandwidth utilization. Therefore, it is often desired to explore potential performance of alternative implementations. Doing so requires understanding the workload behavior and how it may interact with new algorithm and hardware features.

Current performance analysis tools focus mostly on the performance characteristics of a given implementation, without extracting the workload’s intrinsic behavioral properties. Mini-applications are often used as testbeds to generate performance results on experimental hardware; their performance behaviors remain to be studied. Domain-specific languages and performance models often abstract high-level operations of applications, but they do not express the performance properties within those operations. Profiling tools measure hardware-specific performance responses but are not able to identify details about the data flow and their relationship with the control flow. All the above approaches require users to develop and run the workload on an existing hardware. Architecture simulators are able to reveal performance responses of various hardware configurations, but they treat workloads as black boxes and are time consuming. Application performance models [24, 40] summarize the asymptotic performance characteristics. They can estimate performance bounds at a coarse granularity by describing the overall computation intensity, i.e. the combined number of instructions and the number of memory accesses) and the memory footprint. However, they only express the characteristics of a given implementation without capturing the interrelationship between the control and data flow. As a result, the user still needs to manually explore performance potentials of alternative implementations.

To explore potential workload implementations, we model the workload’s behavioral properties, such as data flow, control flow, computation intensity, and communication patterns. These behavioral properties are intrinsic to the application and interdependent; the control flow may determine data access patterns, and data values may affect control flow. Given different input data may result in diverse performance outcome. The interaction of the properties also reveal transformation opportunities that may affect performance significantly. We call this effort SKOPE - a SKeleton framewOrk for Performance Exploration, a framework that helps users describe, model, and explore a workload’s current and potential behavior. Given a formalized description of the workload’s performance behavior, SKOPE automatically analyzes, tunes, and projects the workload’s performance for a target hardware. SKOPE offers readability and expressiveness in its frontend and flexibility in its backend functionality.

An overview of the framework is shown in Figure 1. Its frontend is the SKOPE language, which produces code skeletons, a uniform description of the semantic behavior of a workload, regardless of the original language in which the workload is written. With the formalized language, SKOPE can help the research community share, communicate, and study the performance of workloads collaboratively. It also provides a common interface for third-party software tools to programmatically analyze and utilize such information. A code skeleton is agnostic of any system hardware but can infer potential transformations and help users understand how workloads may interact with and adapt to emerging hardware. According to the semantics and the structures in the code skeleton, the backend explores various transformations, synthesizes performance characteristics of each transformation, and evaluates the transformation with various types of hardware models.

The key distinction of SKOPE from other application modeling techniques is that it encodes performance behaviors rather than performance characteristics; the former is the rules and properties behind the latter. As a result, SKOPE can holistically capture and model the potential interactions among various components of the workload. Our novelty lies in the language formalization and the vertical software architecture integration of code analysis, transformations, search algorithms, and hardware models into a unified modeling framework. This generic modeling framework provides an intuitive way for the user to model application behaviors, and a clean interface be-
between application models and hardware models. To our knowledge, there has been no convenient and standard way to express such behavioral properties; they often appear as code annotations, text documents, whiteboard graphics, and algebraic equations.

The paper is organized as the following. We first introduce the SKOPE language in Section 2. We then present SKOPE’s backend that uses a code skeleton’s intermediate representation to characterize, transform, and explore performance potentials of various implementations in Sections 3, 4, and 5. Finally in Section 6, we showcase two applications of SKOPE: one projects a computational kernel’s best-achievable performance on GPUs; the other projects the scaling trend of a full application on the IBM Blue Gene/Q supercomputer. Related works and future plans will summarize the paper.

## 2. Behavior Description Language

The SKOPE language serves the frontend of the framework, and plays an important role in capturing a workload’s behavior. Given a workload, represented as the combination of source code and input data, an important goal of the language is to statically describe the workload’s runtime behavior by capturing how input data may affect the control and data flow. The resulting description, referred to as code skeletons, serves as a “roadmap” for exploring transformations. In other words, a code skeleton describes what the workload needs to do algorithmically, without specifying how it is done. The key language features include the following:

1. Extracting the workload’s control flow and data flow, as well as their interdependency. These properties are further parameterized so that the resulting code skeleton is generic and works for different inputs. The parameterization also enables the intermediate representation to be transformed without undermining semantic correctness.

2. Expressing behavioral potentials of an application. Users can optionally express high-level application structures and semantics (e.g., data structures, parallelism patterns) according to domain knowledge, even though that property is not been exploited in the current implementation.

3. Preserving the structural representation of the source code. By mimicking the original workload’s organization of code blocks, functions, and files, code skeleton collection is modularized and easy to read. This feature also makes it convenient to evolve code skeletons according to future changes in the source code.

SKOPE automates the generation of the code skeleton using a source-to-source translator. The translator statically analyzes C/Fortran code using the ROSE compiler infrastructure [35]; it collects information about instruction mix, array indices, and control flow structure at the level of basic blocks. Depending on the desired level of details, users can optionally edit or write the code skeleton manually to express high level semantics that cannot be reliably identified from static analysis. For example, in our case study to project GPU performance from sequential code, we manually identified parallel for loops in the code skeleton (Section 6.1). To understand the high level scaling behavior, we manually summarized a full application with 32,000 lines of Fortran code to 560 lines of a code skeleton (Section 6.2). Note that SKOPE minimizes the amount of user inputs; variables are symbolically preserved within the code skeleton and can be computed once the input is provided. In cases where performance depends largely on dynamically generated data values (e.g. indirect accesses, data-dependent control flow, etc), profiling is inevitable and we allow users to provide empirical values to the variables in the code skeleton. With a good understanding of the application, it only took 8 hours to manually write the code skeleton for the full application.

In this section, we first introduce the design philosophy behind the SKOPE language. We then present the major language constructs, followed by several examples.

### 2.1 Design Philosophy

The SKOPE language is not designed specifically for particular applications, hardware platforms, or modeling objectives. We embrace the following design principles:

1. **Expressiveness**: The language should be able to express all major factors that may affect performance: computation, control flow, data flow, and communication patterns. The expressiveness should not be limited by the original programming language in which the source code is written. The language also welcomes domain knowledge that may not be needed for computation correctness but is important for performance.

2. **Flexibility**: While our source-to-source translator provides a default level of details, not all parts of the code

<table>
<thead>
<tr>
<th>Table 1: Syntax of the SKOPE language.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Macros and Data Declarations</strong></td>
</tr>
<tr>
<td>Types: int / float / double / complex</td>
</tr>
<tr>
<td>Constant definition: symbol = expr</td>
</tr>
<tr>
<td>Array definition: :type array[N][M]</td>
</tr>
<tr>
<td>Variable def./assign: var = expr</td>
</tr>
<tr>
<td>Variable range: var_name = begin end (exclusive): stride</td>
</tr>
<tr>
<td><strong>Control Flow Statements</strong></td>
</tr>
<tr>
<td>Sequential for loop: for var range: (list_of_statements)</td>
</tr>
<tr>
<td>Reorderable for loop: stream var range: (list_of_statements)</td>
</tr>
<tr>
<td>Parallel for loop: forall var range: (list_of_statements)</td>
</tr>
<tr>
<td>Branches: if (conditional probability): (list_of_statements) else: (list_of_statements)</td>
</tr>
<tr>
<td><strong>Data Flow Statements</strong></td>
</tr>
<tr>
<td>Data read: id array[expr]/expr</td>
</tr>
<tr>
<td>Data write: st array[expr]/expr</td>
</tr>
<tr>
<td><strong>Communication Statements</strong></td>
</tr>
<tr>
<td>Comm. pattern: comm (communicator[coordinate]) {}</td>
</tr>
<tr>
<td>Send message: send peer array[range]/message_size</td>
</tr>
<tr>
<td>Receive message: recv peer array[range]/message_size</td>
</tr>
<tr>
<td>Broadcast: bcast array[range]/message_size</td>
</tr>
<tr>
<td>AllReduce: allreduce array[range]/message_size</td>
</tr>
<tr>
<td><strong>Characteristic Statements</strong></td>
</tr>
<tr>
<td>No. of operations: comp N</td>
</tr>
<tr>
<td>No. of FP operations: fp N</td>
</tr>
<tr>
<td>No. of Int. operations: int N</td>
</tr>
<tr>
<td>Functions and Multi-file Support</td>
</tr>
<tr>
<td>Function definition: def function[arg_list]/list_of_statements</td>
</tr>
<tr>
<td>Function calls: call function[arg_list]</td>
</tr>
</tbody>
</table>
need to be expressed with the same resolution. Users may optionally provide macro-level descriptions with empirically measured statistics. A workload can be described with code skeletons at different resolutions.

3. Structured Representation: The code skeleton should preserve a structure of the original source code for readability, modularity, and maintainability. This representation should allow modeled performance characteristics to be aggregated hierarchically.

4. Interoperability: Code skeletons are parsed into a formal intermediate representation that captures the branch probabilities, loop boundaries, and array access indices. This representation offers an interface to various performance transformations and can serve different purposes including processing and characterization.

5. Configurability: The same code skeleton should be able to express the workload with different input data. Factors related to the input that may significantly impact performance should also be parameterized in the code skeleton. Typical examples include array dimensions and control flow flags.

2.2 Syntax and Semantics

Table 1 summarizes key language constructs, represented by three major categories: data objects, statements, and functions.

- Data objects are constants, variables, and arrays. All data objects can be defined in the global scope or locally within a code block. A locally defined data object is visible only to statements within the scope to which it belongs. Input of the workload is also defined as data objects, which can be later parameterized to mimic runs with different flags or data set sizes.

- Statements contain structural statements that express control flow, data flow, and constructs, as well as characteristic constructs that summarize local properties such as instruction count and instruction mix. Structural statements open opportunities to explore transformations, while characteristic statements serve as building blocks to construct an overall performance behavior. Control flow statements are function calls, loops, and branches that can be nested within each other. Data flow statements describe loads and stores of individual array elements and the ordering of memory operations. Communication statements are point-to-point sends and receives, collectives, and barriers. Assignment instructions keep track of values of variables critical to control and data flow.

Since the resulting code skeleton is merely a “roadmap” of the workload’s behavior, the control flows are not actually executed. For example, a loop merely assigns weights to data objects Link 8 in Listing 4). In this case, the user is interested in understanding the communication behavior over a high-performance computing (HPC) cluster; therefore only message passing operations are written in detail. The time spent in computation is simply summarized by an empirical time measurement for processing a single element (Line 8 in Listing 4).

Note that there is no “golden skeleton” for a given workload. Both the algorithm and the data structures may change for the same workload, which leads to different code skeletons. Even for the same source code, users may target different levels of detail, performance components, and modeling techniques, leading to different code skeletons. However, given the same requirements, backend procedures, and the same source code to start with, two code skeletons corresponding to the same source code should generate the same performance insights.

3. Modeling Workload Behavior

Once a user provides the application’s code skeleton as a collection of files, the SKOPE parser parses all files and extracts data objects, functions, and statements. Eventually, the parser constructs an intermediate representation (IR) as the model of the workload’s behavior. This section introduces the intermediate representation and its basic operations.

3.1 Construct Intermediate Representation

Workload behavior defined in each function of the code skeleton is parsed into an intermediate representation referred to as a block skeleton tree (BST). Different from the abstract syntax tree (AST), which goes into detail about operators and operands, the BST has each statement as a node and emphasizes on the collective properties of statements within each scope. The statements are nested; for example, all statements in a loop body are child nodes of the loop statement. Inner nodes in the BST include loops and branches, while leaf nodes include all other types of statements, including function call statements. Each BST node also contains data objects locally defined at this scope. Figure 2 (left) shows the per-function BSTs corresponding to the Stassuij code described in detail in Section 6.1 (Listing 5). The main function contains two statement calls, with functions exhibiting nested loop structures and code blocks containing sequences of loads, stores, and computation statistics.

Advanced users can extend the language syntax by using SKOPE’s API to register a context-free expression and the corresponding callback handler. The handler will be invoked when the parse is processing the new syntax and will add a new BST node to the IR. With such an API, the user can include extra characteristics, such as instruction-level parallelism, memory-level parallelism, etc.

3.2 Model Execution Path

The initial per-function BSTs are constructed without any given context about their arguments. In order to model the execution of the workload, a holistic picture of the application has to be built according to the input. When input values
Listing 1: Matlab's source code.
```matlab
float A[N][K], B[K][M];
float C[N][M];
int i, j, k;
for (i=0; i<N; ++i){
    for (j=0; j<M; ++j){
        float sun = 0;
        for (k=0; k<K; ++k){
            sun += A[i][k]*B[k][j];
        }
        C[i][j] = sun;
    }
}
```

Listing 2: Matlab's code skeleton.
```matlab
float A[N][K], B[K][M]
2 float C[N][M]
forall i=0:N, j=0:M {
    comp 1 /* op. count */
    /* a streaming loop */
    for (k = 0:K)
        for (r = 0:C)
            st C[i][j] /* store */
    }
```

Listing 3: Source code for 2-D halo exchange.
```c
#define DBL MPI_DOUBLE
2 MPI_Comm com;
MPI_Request r[4];
MPI_Status s[4];
4 int src[4], dst[4];
6 double sb[4][NUM], rb[4][NUM];
10 int dims[2] = [64,64];
12 #define Communicator's size */
14 int p[2] = [0,0];
16 /* Creating the 2-D grid communicator */
18 MPI_Cart_create(MPI_COMM_WORLD, 2, dims, p, &com);
20 /* Register MPI ranks of neighbors */
22 MPI_Cart_shift(com, 0, -1, & src[1], & dst[1]);
24 MPI_Cart_shift(com, 0, 1, & src[2], & dst[2]);
26 MPI_Cart_shift(com, 1, -1, & src[3], & dst[3]);
28 for (int n = 0; n < NITERs; ++n) {
    solver(); /* Local computation */
    MPI_BARRIER(com); /* Start 2-D halo exchange */
    MPI_Isend(& sb[3][0], NUM, DBL, dst[3], 0, com, &r[6]);
    MPI_Isend(& sb[2][0], NUM, DBL, dst[2], 0, com, &r[4]);
    MPI_Isend(& sb[1][0], NUM, DBL, dst[1], 0, com, &r[2]);
    MPI_Isend(& sb[0][0], NUM, DBL, dst[0], 0, com, &r[0]);
    MPI_Waitall(4, &r[0], &s[0]);
}
```

Listing 4: Code skeleton for 2-D halo exchange.
```c
#define sb[4][NUM]
2 double rb[4][NUM]
: N_P X = 64, N_P Y = 64
4 for n = 0: NITERs {
    /* Average computation time */
    if p = 0: NUM + NUM
    for i = 0: NUM + NUM
        MPI_BARRIER(com);
        /* Define a communication pattern */
        MPI_Isend(& sb[3][0], NUM, DBL, dst[3], 0, com, &r[6]);
        MPI_Isend(& sb[2][0], NUM, DBL, dst[2], 0, com, &r[4]);
        MPI_Isend(& sb[1][0], NUM, DBL, dst[1], 0, com, &r[2]);
        MPI_Isend(& sb[0][0], NUM, DBL, dst[0], 0, com, &r[0]);
        MPI_Waitall(4, &r[0], &s[0]);
    }
```

Figure 2: Intermediate representation corresponding to Stassuij (Listing 5). On the left, we show the block skeleton trees (BSTs) of individual functions in Stassuij. On the right, these individual BSTs are combined to model the workload execution.

4. PROGRAM BEHAVIOR ANALYSIS

Data Usage

Data usage includes memory footprint, cache locality, and data access patterns. SKOPE analyzes the workload’s data usage to infer the utilization of memory bandwidth, effectiveness of caching, and the working set size: all are critical performance factors. We adopt the data flow analysis described in [29] and provide a brief summary below.

Given a code skeleton that contains abstractions about individual array accesses, we employ bounded regular sections (BRS) to study data access patterns [16]. For example, a two-level nested for loop has a load statement, ld A[r][c], with loop counters r and c to index an array A. The set of elements accesses can be represented as a pattern, \( \mathbb{P} = \{A[r]: r \in [r^\text{lo}; r^\text{hi}]; c^\text{lo}; c^\text{hi}\} \), where the ranges are constructed by the lower bound, upper bound, and stride of the nested for loop.

The union of all patterns within a code block determines the corresponding memory footprint (or working set size). Together with computing statistics, such as the instruction count, we can deduce the computation intensity of the code block. With hardware information such as issue width, clock frequency, and peak memory bandwidth, we can further calculate the memory bandwidth utilization.

The degree of data reuse can be deduced by intersecting patterns among consecutive code blocks, which can be used to infer caching efficiency or locality. All these analyses are used in our case study in Section 6.1.

4.2 Parallelism

SKOPE allows users to express loop-level parallelism. The `forall` loops can be imperfectly nested, even across function calls; backend procedures determine where to exploit loop parallelism and how to transform the loops.

Task-level parallelism can also be determined by performing data dependency analysis across code blocks. BRS analysis is used here to test whether data patterns in two code blocks intersect with each other. If not, the two code blocks can potentially execute in parallel; otherwise, the two code blocks have to be sequentially executed. As a result, SKOPE’s intermediate representation can infer high-level behaviors such as...
master-slave, block-synchronize, and iterative operations.

4.3 Control Flow

Control flow mainly dictated by loops, function calls, and branches constructs. In particular, loop boundaries and branch outcomes often remain unknown until the run time. Nonetheless, a code skeleton has to capture sufficient information to reflect the number of loop iterations and the chances of executing a given path, which may significantly impact the workload size and therefore the execution time.

Although a code skeleton does not execute the actual statements to determine branch outcomes, it can estimate the overall control flow better than conventional static analysis because of the availability of context information such as input data. In scientific applications, most loop boundaries are known once the input is determined; however, branches are more dynamic. While the probability of executing a path can be calculated by propagating the conditional probability from the root BST node, it may not be straightforward to determine the conditional probability of a branch outcome. Four scenarios are likely present in the code and our benchmarks described in Section 6 cover all scenarios:

1. Static branches. The conditional branch-taken probability can be estimated directly from the code. For example, given i as an iterator of a consecutive loop, a branch condition in the form of \( \text{mod}(i, 4)==0 \) has a 25% chance to be taken. Such branches appear in GTC application.

2. Input-dependent branches. Conditions of these branches depend on input flags or array sizes. Once the input is specified, the condition can be evaluated in the same way as static branches. All our benchmarks have this type of branches.

3. Statistics-dependent branches. For theses, conditions depend on statistics of the workload. One example is the percentage of nonzero elements in a sparse matrix. Users must measure the statistics and then specify the branch outcome probabilities accordingly. Stassuij benchmark contains such branches.

4. Dynamic branches. The likelihood of them can be determined only by executing the program. In this case, we rely on users to provide an empirical conditional probability. This type of branch occurs in HotSpot example.

In fact, we observe that most branches in our benchmarks fall into the first three categories. Even for the last case, users can obtain the conditional probabilities of branch outcomes by profiling on any accessible architecture. Afterwards, the results can be used for performance studies on other architectures.

5. PERFORMANCE EXPLORATION

Information about control flow and data flow enables the SKOPE backend to explore various transformations. Each transformation is analyzed to synthesize performance characteristics. These characteristics are used as inputs to the hardware model of the user-selected system. Performance projections are then generated. Eventually, all transformations are compared, and the one with the best projected performance is presented to the user. At this point, the user observes two outputs: the projected performance of the application, and the transformations needed to achieve such performance. The rest of this section discusses this process in more detail.

Note that the framework provides components and options that can be used as building blocks for performance exploration tools. The users of the framework can use a provided API to connect the components into a workflow that specifies which transformations and characterizations to apply, which hardware models to use, and how to order the operations. The user is able to plug in any hardware models which interact with the synthesized characteristics. The user is also able to develop workflows to explore design spaces. It is up to the user to provide design space exploration strategies, hill-climbing or generic algorithms are possible options. Two workflows exemplify the use of our framework are presented in Section 6.

5.1 Transformations

The BST intermediate representation describes which variables are involved in loop boundaries, branch conditions, and function calls. It also tells how they are used for data accesses. Given such behavioral information, a backend procedure is able to propose code skeletons corresponding to alternative implementations while preserving the semantics. Our framework provides the following built-in set of transformations.

1. Spatial loop tiling. A forall loop can be partitioned into groups of iterations; each group is processed independently. This is achieved by replacing the BST node corresponding to the forall loop with two nested nodes; the parent node representing the outer loop across tiles, and the child node representing inner loop across iterations within the same tile.

2. Temporal loop tiling. A sequential loop can be broken into stages to promote data reuse within each stage. This transformation can be achieved by dividing the loop into a two-level nested loop and inserting necessary data movement and synchronization statements in between.

3. Loop fusion. Two loops can be fused to improve data reuse given that their iteration-wise dependencies are preserved. From the data access patterns of the producer and consumer loops, our framework symbolically computes which producer iterations are required for a given consumer iteration, thus ensuring semantic correctness. The BST can be transformed by moving the producer’s loop body into the consumer loop and inserting an assignment statement that maps the consumer’s loop variable to that of the producer’s loop.

4. Loop unrolling. Loop unrolling may impact performance significantly when it comes to frequently executed basic blocks with a small number of instructions. Loop unrolling is modeled by reducing the loop size and replicating the BST nodes corresponding to the loop body.

5. Caching. With the information about array access indices and the expression of these indices, the framework uses the array section analysis described in Section 4.1 to compute the data footprints within a code block. Data reuse is discovered when two memory accesses have overlapping footprints. The framework then enumerates multiple possibilities about which data should be kept in cache and how many times they are reused; their overall footprint becomes the cache footprint. A cache strategy is valid only if the cache footprint is smaller than the cache capacity. The approach models only fully associative caches or explicitly managed scratch pads. Integration of more sophisticated cache models are in our future plans.

6. Task scheduling. Dependent tasks can be inferred from the code skeleton when two sibling nodes have overlapping data footprints. By distinguishing whether data is read or written, the framework determines whether the dependency is true or false. This enables the framework to build a task graph. We can then model different task scheduling effects.

Often, several phases are needed to explore combinations of transformations. Users can define their own transformation workflow. Our framework enumerates transformations recursively by collecting the set of local transformations when visiting each node and then exploring combinations of child nodes’ transformations in the parent node. Note that when combining two transformed child nodes, the parent node may apply further optimization to exploit data locality among them.
Listing 5. Stassuij's code skeleton.
```python
def main():
    # A sparse matrix T
    double T[ELEMS] /* CSR format */
    int J[N+1][ELEMS]

    # Three dense matrices w/t
    * complex numbers
    double A[N][m][N][m], B[N][m][N][m], C[N][m][N][m]
    double D[N][m][N][m]
    /* Compute B = A * T */
    call IspinEx(j, i, T[*])
    /* Exchange elements to get C */
    call SpinFlip(H, B, C)

    def IspinEx(j, i, T[*])
    for all j=0:N, i=0:NS, r=0:2
        id(j)
        id(T[j])
        Compute dot product /*
        stream m = j[1]:j[1] {  
        id T[j]
        id A[i][m][i][r]
        comp 11
        }  
        comp 6
        at B[j][1][1][r]
        }  
        def SpinFlip(M, A, B, C)
        {  
        /* Loop with a different shape */
        for all j=0:N, k=0:NS, m=0:4, r=0:2
            id M[j][k]
            /* Indirect accesses */
            id A[M][m][k][r]
            at C[M][m][k][r]
            comp 36
        }  
```

```python
/* Average number of iterations */
2 = for the loop at line 33
  /* avg_loop_size[24][14]

/* The values for elements in */
array J ranges from 0 to 156
  /* range[J] = (0:132)

/* Average data strides of indirect */
12 = array accesses at line 42-43
  /* avg_stride[41,42,43] = 5.2
```

Listing 7. Stassuij's transformed code.
```python
work groups(1, NS/NSG, 1, 1)
/* (s): memory access */
/* (c): cache access */
/* alloc: space requirement */
alloc(c) M:NSG*[0][4:6]
alloc(c) J[0:NSG]
/* Work items in a work group */
work items 1 = 0:NSG*[0.4:0.5]
/* Derived loop taken of the */
/* function */

/* M[0][m][n][r]

/* Temporal tiling for locality */
StageSize = 11
alloc(c) [0:avg_j.ntd=StageSize]
alloc(c) [0:avg_j.ntd=StageSize]
for j = 0:NT
  /* Compute dot product */
  /*
  alloc(c) B[0][0][NSG+4][2]
  id(c) J[j]
  alloc(c) J[j]=1
  for n = 0:avg_j.ntd {  
  id(c) T[n]
  id(c) T[n]
  id(m) A[n][m][i][r]
  }  
  /* Exchange elements */
  alloc(c) A[0][NSG+3][2]
  id(m) A[0][NSG+3][2]
  alloc(c) J[j][i][r]
  alloc(c) B[j][i][r]
  alloc(c) C[j][i][r]
  }  
```

5.2 Characterizations and Projections

A BST can be analyzed to estimate its performance characteristics. Key characteristics include number of floating-point operations, memory accesses, caching effectiveness, degree of parallelism, and communication overhead. Basic statistics can be aggregated recursively by traversing the BST. At the end of this step, characteristics for each BST node are recorded.

The characteristics of each BST node can then serve as input to hardware performance models. There are no restrictions on the type of hardware model to employ, whether it is analytical or empirical, detailed or abstract. The models can be asymptotic, such as the Roofline Model [45] which computes performance bounds, or detailed, whose goals are projecting the performance more accurately [17]. The user can simply wrap up the hardware performance model as a callback function, which is invoked for each BST node representing a function call. The hardware model takes all the synthesized characteristics of that BST node as the input and calculates the estimated performance accordingly. The projected performance of the main function is the estimated overall performance.

6. Evaluation and Case Studies

The SKOPE framework is implemented in Python, which provides packages to easily define a new language syntax and parse it. The backend is organized into modules for characterization, transformation, and performance projection. Each module provides a pool of operations. The transformation module includes various BST operations mimicking potential program transformations described in Section 5.1. The characterization module contains operations for data flow analysis and statistics aggregation. The projection module provides a pool of hardware models and specifications reflecting various machines and system configurations. The modules are then used as building blocks to construct performance exploration workflows serving different purposes.

We showcase the use of SKOPE with two examples, each has a different goal and targets a different hardware. In Section 6.1, we use SKOPE to project the GPU performance from a CPU code without GPU programming or accessing the hardware. For this purpose, the code skeleton expresses the details about loop boundaries, parallelism patterns, and data access patterns. We show that SKOPE is able to auto-explore various
transformation, synthesize their characteristics, and compare their performance using a detailed, analytical GPU hardware performance model. In Section 6.2, SKOPE is used to project multinode scaling trends for two scientific applications. We demonstrate that a code skeleton can be written at a coarse granularity as well, and users can choose to emphasize one aspect of the workload, which is "communication" in this case. A different, empirical hardware model is used in this example, and SKOPE is able to model data distribution and the resulting communication overhead given a different number of nodes and input sizes. Through these two examples, we demonstrate the wide applicability of SKOPE and its flexibility to serve different performance modeling and projection purposes.

Note that the GPU performance models and the communication subsystem models are not new. However, conventional approaches require profiling, tracing, or execution on the physical hardware to obtain inputs for these models. We demonstrate that SKOPE provides a unified framework that integrates various application behavior models and hardware performance models to produce performance insights such as performance characteristics, predicted execution time and scaling, analyzed bottlenecks, and suggested transformations.

6.1 Automatic Exploring GPU Performance

The goal of GPU performance projection is to estimate the potential performance for a piece of legacy code if ported to a particular GPU. By casting the projection before porting and executing the code on a physical GPU, significant effort can be saved in coding, debugging, and tuning. It even helps users select which GPU hardware to acquire.

It is widely known that program performance on GPUs depends largely on code implementation; therefore, it is critical to explore various transformations with regard to loop partition, data allocation, caching strategies, and so forth. These factors affect latency hiding, degree of multi-threading, global memory accesses, data coalescing, to which performance is often sensitive. Not only must the performance model change the code structure while preserving semantics; it also must assess the effects resulted from the transformations.

With the SKOPE framework, we re-implemented a GPU performance projection tool [28] and constructed a workflow to explore GPU program transformations using the BST. The customized transformation engine is composed of four sequential steps to judiciously search the transformation space: GPU kernel creation → Kernel fusion → Loop partition → Cache optimization. In each step, a resulting BST from the previous stage is traversed, producing several more transformations. For example, a parallel for loop can be blocked into several thread blocks, a sequential loop can be staged for the purpose of cache locality, two loops can be merged into one, and a code block can choose which data array to cache. All of these examples can be represented by configuring, creating, and merging BST nodes.

For each transformed BST, we synthesize a set of program characteristics according to the need of the hardware model. The characteristics include the number of SIMD thread groups, cache and memory footprint, degree of coalescing, and computation intensity. This is accomplished by another traversal of the BST. Eventually, the performance is projected using GPU performance models for each thread block. We note that these two generations of GPUs belong to different GPU architectures (Quadro and Tesla, respectively) and therefore we employ different GPU performance models for each [17, 38]. Both models compute the time spent in computation and memory accesses according to the SIMD width, number of cores, and memory latency and bandwidth, as well as the true thread block characteristics. The models then determine the overlap between computation and memory access. Eventually, they output the estimated execution time. We report the best projected performance and its associated transformed code skeleton. The skeleton is generated by traversing the transformed BST and print out each statement hierarchically.

We exemplify GPU performance projection using two computational kernels. For Stassuij<br>

![Figure 3: GPU performance study for Stassuij](image)

With the SKOPE framework, we re-implemented a GPU performance projection tool [28] and constructed a workflow to explore GPU program transformations using the BST. The customized transformation engine is composed of four sequential steps to judiciously search the transformation space: GPU kernel creation → Kernel fusion → Loop partition → Cache optimization. In each step, a resulting BST from the previous stage is traversed, producing several more transformations. For example, a parallel for loop can be blocked into several thread blocks, a sequential loop can be staged for the purpose of cache locality, two loops can be merged into one, and a code block can choose which data array to cache. All of these examples can be represented by configuring, creating, and merging BST nodes.

For each transformed BST, we synthesize a set of program characteristics according to the need of the hardware model. The characteristics include the number of SIMD thread groups, cache and memory footprint, degree of coalescing, and computation intensity. This is accomplished by another traversal of the BST. Eventually, the performance is projected using GPU performance models for each thread block. We note that these two generations of GPUs belong to different GPU architectures (Quadro and Tesla, respectively) and therefore we employ different GPU performance models for each [17, 38]. Both models compute the time spent in computation and memory accesses according to the SIMD width, number of cores, and memory latency and bandwidth, as well as the true thread block characteristics. The models then determine the overlap between computation and memory access. Eventually, they output the estimated execution time. We report the best projected performance and its associated transformed code skeleton. The skeleton is generated by traversing the transformed BST and print out each statement hierarchically.

We exemplify GPU performance projection using two computational kernels. For Stassuij, a Fortran77 application that performs Monte Carlo calculation for light nuclei. It contains two functions: IspinEx and SpinFlip. IspinEx multiplies a 132 × 132 sparse real-type matrix with a 132 × 2048 dense complex-type matrix. The sparse matrix is compressed in CSR format with three vectors: T, J, and I. The resulting matrix is then permuted in SpinFlip by exchanging elements of each permutation pattern is stored by indirect referencing in a separate array. M.

Performance projection for Stassuij has two challenges. First, the workload size depends on the sparsity of the matrix. Second, the indirect indices make data access patterns irregular, which affects the ability of GPUs to.coalesce multiple memory requests. We provide three hints in Listing 6: avg_loop_size specifies the average number of nonzero values for each row in the sparse matrix, obtained from user's domain knowledge; avg_stride specifies the average data stride among addresses pointed to by the indirect indices, obtained by profiling the input array statically; and vrange specifies the value range of elements in array I, which is used for indirect indexing. With those hints, SKOPE can evaluate the computation intensity and the memory access latency and determine whether computation or memory access is the bottleneck, and project the performance. Exploring transformations include varying work-group sizes, loop blocking for data locality, data allocation, caching, and kernel fusion. Hundreds of transformed code skeletons are examined, and the best-performing one is reported to a user. Figure 3 compares the projected performance with the measured performance of manually optimized code.

Using this SKOPE-based workflow, we can also conduct an early evaluation on the performance potential of fusing dependent kernels. For Stassuij, IspinEx and SpinFlip can be fused into one by using loop indices of the forall loop in SpinFlip. The benefit of kernel fusion is often data locality; however, it may also incur changes in data access patterns. By obtaining bounded regular sections of each kernel, we can determine dependent kernels and the cache footprint as a result of fusion. This transformation is evaluated with the performance model as well. The corresponding code skeleton is shown in Listing 7. The work_groups and work_items keywords are extended SKOPE syntax corresponding to the OpenCL terminology [25]. The outermost loop is tiled into work-groups that can be executed independently in parallel, and each work-group further contains parallel work-items that operate over shared memory. Our technique identifies that the load at Line 35 becomes irreguarly stripped in this fused kernel as a result of indirect indices involved in computing i (Line 22). Performance projection shows that the increased memory access latency caused by such irregular data accesses cancels the benefit of data reuse in the cache memory accesses. The projected performance is validated by comparison with the manually optimized code.

We have also applied the same workflow to HotSpot, a finite-element ordinary equation solver originally written in C [7]. In HotSpot, a kernel that performs stencil operations is executed...
Several iterations of the same kernel can be fused into one. The tradeoff here is the overhead in replicated computation and the benefit of cache reuse. Our technique is able to project the best scheme. HotSpot has a total of 27M floating-point operations per iteration. The best fused kernel achieves 15 GFlops and 45 GFlops on FX5600 and C1060, respectively.

iteratively. Fusing loop partitions (e.g., work-groups) across several iterations may achieve better locality. However, each loop partition also needs to replicate computation in order to produce “halo” regions locally that otherwise would be computed elsewhere. Thanks to the abstracted array accesses in SKOPE, our data flow analysis is able to identify how much computation must be replicated given the number of iterations to fuse. No hints are needed for HotSpot. As Figure 4 shows, our technique can effectively model the tradeoff of data locality and replicated computation and determine the best degree of kernel fusion.

Our results on performance projection for two GPU generations, FX5600 and C1060, show that the projected best-achievable performance deviates from the measurement by an average of 18%, with a maximum error of 30%. Since users often want to learn whether GPUs can accelerate their program by orders of magnitude, such accuracy can still be valuable.

6.2  Projecting Scaling Trend on HPC Systems
Scientific application developers often need to study how their applications scale on HPC systems, given different problem sizes and processor counts. Sometimes such insight is sought before their parallel code is fully functional. In this case, the hardware is accessible, and the application can already be compiled and executed on a single processor with relatively small input size. The challenge is to estimate multinode performance from a single-node execution.

We model the application performance with two components: single-node computation and internode communication. Because the goal is to project the scaling trend for the given implementation, no transformation is necessary, and, there is no need to present the detailed structure of the target application. In fact, since the application is able to run on a single node, we include the measured computation time in the code skeleton. However, communication overhead has to be obtained through modeling, and therefore communication operations have to be expressed in more detail. Using SKOPE, we encode communication operations such as send, recv, and allreduce. The communicator size and the message size were parameterized according to the input data and the configuration of the MPI runs.

Two applications are studied using the extended SKOPE language. We use SKOPE to project weak scaling or strong scaling trends according to how an application is used in production. We first study the weak-scaling trend of the Gyrokinetic Toroidal Code (GTC), a 3D particle-in-cell application that calculates turbulent transport in magnetic fusion [13]. GTC is written in Fortran90 and is parallelized with MPI and OpenMP. Its main loop involves a sequence of kernel functions. Several major kernels involve a sequence of sendrecv and allreduce operations with different sizes over three MPI communicators. For the weak-scaling study, the computation workload for a given node remains the same regardless of the problem size. We therefore summarize the computation characteristics of each kernel by measured computation time. Among the kernels, CHARGEI and PUSH both consume significant run time. Therefore, these two kernels are expressed in more detail, each with several computation phases interleaved with collective or point-to-point communication operations. The communication pattern again depends on the input and the configuration of the runs.
7. RELATED WORK

Researchers have proposed several approaches to analytically models a workload’s characteristics, including LogP [9], PRAM [21], and BSP [44]. However, these approaches focus on the asymptotic performance bounds based on operation counts and overall data footprint, with equations constructed to reflect a particular implementation. Furthermore, they do not capture sufficient details about a workload’s behavior to explore transformations.

There are also several attempts to capture workload structures through profiled measurements, static analysis, or mini-applications [4, 6, 18, 31, 37, 40]. In particular, the Aspen language organizes overall performance characteristics for major kernels within a workload so that their performance models can be conveniently composed in a modular fashion. However, the above techniques often assess the control flow and data flow at a function or module level, and the captured behavior is largely input-dependent and implementation-specific.

As a result, they are unable to fulfill performance tasks such as autotuning and projecting expected performance beyond performance bounds. SKOPE provides the flexibility for users to express different parts of the workload in different levels of detail; it also provides functionalities to systematically explore transformations and interact with various hardware models.

Figure 6: Strong scaling projection for SORD

In fact, one can use a workload’s code skeleton to generate an implementation-specific, coarse granularity summary of the application characteristics in the form of LogP equations or Aspen script. In other words, SKOPE is a superset of asymptotic performance models, with additional metainformation on workload behavior and more versatility in backend applications, and without sacrificing the ease of use. In fact, SKOPE may require less effort to write since users need not manually inspect the code and derive equations. Code structures and basic statistics are automatically derived with a source-to-source translator.

Domain-specific languages, libraries, and programming models [1, 10, 41] provide a high-level structural description of some applications and offer different ways underneath to transform and optimize the application. However, they themselves still face the question of how to adapt to a new hardware. Performance often has to be tuned on a per-hardware basis as well. SKOPE is complementary to these techniques; it can help these techniques model future workloads and discover beneficial transformations.

Annotation-based tools are trying to mitigate the programmatic difficulties of various transformations and employ different models. This work extends previous effort by formalizing a modular analysis domain or a particular hardware.

8. CONCLUSION AND FUTURE WORK

We have proposed SKOPE, a framework that helps users describe, model, and explore workload behavior. SKOPE summarizes an application’s parallelism, control flow, data flow, computation intensity, and communication patterns. It also allows users to include additional domain knowledge that is not necessary for program correctness but can be important for performance. Users can choose to describe the application in different levels of detail.

The key benefit of SKOPE is the ability to study how applications may adapt to a target hardware before production runs. We demonstrate case studies that use SKOPE to model and project performance of legacy code when ported to graphics processors. We also show how SKOPE can be used to study the scaling trends of scientific applications on supercomputers.

SKOPE can help performance engineers and hardware designers set goals for tuning performance, select optimizations to apply, and, above all, identify which hardware platforms to choose and whether it is worthwhile to invest the effort. It also has the potential to be integrated into software tools such as compilers or runtime schedulers for performance optimization. We plan to make SKOPE publicly available.

Future work includes extending the SKOPE implementation to embrace a wider selection of hardware models so that it can be used to study various computer architectures. We also plan to extend the skeleton language to model applications with complex data structures.

Acknowledgment We thank the ALCF application and operations support staff for their help. We also appreciate the help provided by the application teams whose codes are used herein. Gail Pieper’s and James Collins’ careful editing of the manuscript is gratefully acknowledged. We are also thankful for the feedback from Qing Yi, Jichi Guo, and Yao Zhang.

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

9. REFERENCES


[3] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Using performance at runtime [2, 19, 32]. These tools are optimization techniques that require users to rewrite parts of the code and execute the workload on the physical hardware in order to gather performance data. They often employ a specific performance model and do not enable performance projection for future systems and potential application transformations.

Several cross-platform project and tuning tools study performance on in-place systems [5, 26, 27, 39, 42, 46]. They often rely on machine learning techniques and base their projection on surrogates, therefore lacking analytical insights about the projected performance. Moreover, these techniques focus on characterizing particular implementations of an application, rather than expressing the application structure to enable various transformations and employ different models.

Our previous work on GPU performance projection [28] introduced a transformation pipeline to model GPU optimizations. This work extends previous effort by formalizing a modeling language and providing a software architecture that models, transforms, searches, and projects workload performance. We show that this framework is versatile, models various performance behaviors, and employs various hardware models.

Domain-specific performance modeling techniques have been proposed for different hardware systems [3, 8, 14, 22, 23, 30]. Our transformation framework is not limited to a particular application domain or a particular hardware.
modeling to design large-scale systems. *Computer*, 42, November 2009.


