

Portal: A High-Performance Language and Compiler for Parallel N-body Problems

Laleh Aghababaie Beni, Saikiran Ramanan, Aparna Chandramowlishwaran
University of California, Irvine
 laghabab@uci.edu, ramanans@uci.edu, amowli@uci.edu

Abstract—There is a big gap between the algorithm one designs on paper and the code that runs efficiently on a parallel system. Our goal is to combine the body of work in compilers, performance optimization, and the domain of N-body problems to build a system where domain scientists can write programs at a high level while attaining performance of code written by experts at the low level. This paper presents *Portal*, a domain-specific language and compiler designed to enable high-performance implementations of N-body problems on modern multicore systems.

Our goal in the development of *Portal* is three-fold, (a) to implement *scalable, fast* algorithms that have $\mathcal{O}(n \log n)$ and $\mathcal{O}(n)$ complexity, (b) to design an *intuitive language* to enable rapid implementations of a *variety of problems*, and (c) to enable *parallel large-scale* problems to run on multicore systems. We target *N-body* problems in various domains from machine learning to scientific computing that can be expressed in *Portal* to obtain an out-of-the-box optimized parallel implementation. Experimental results on 6 *N-body* problems show that *Portal* is within a factor of 5% on average of expert hand-optimized C++ code on a dual-socket AMD EPYC processor. To our knowledge, there are no known libraries or frameworks that implement *parallel asymptotically optimal algorithms* for the class of *generalized N-body problems* and *Portal* aims to fill this gap. Moreover, the *Portal* language and intermediate algorithm representation are portable and easily extensible to different platforms.

Index Terms—Domain-specific language, LLVM, N-body problems, tree-based algorithms.

I. INTRODUCTION

Modern machines are becoming increasingly more complex resulting in even the most advanced compilers to fail to generate the best-optimized code. Moreover, Proebsting’s Law [1] states that improvements in compiler technology double the performance of *typical programs* every *18 years*. This has resulted in the trend of expert high-performance programmers who desire the best possible performance to write hand-tuned and hand-optimized code that outperforms compiler generated code. Unfortunately, this typically results in implementing a single algorithm or problem, for one or a small subset of architectures. Even if we just consider the domain of interest in this paper, there are hundreds of *N-body* problems and it is practically impossible to generate hand-optimized code for every single one of them. Furthermore, hand-tuning is not only tedious but also highly machine-specific. Given the trend that architectures are constantly evolving makes these hand-written codes obsolete.

Domain scientists who are experts in their particular domain often lack expertise in parallel programming. In general, scientists prefer to program in high-level languages which allow concise expression of their problem. Matlab and Python are two examples that are widely used in data analytics [2]. However, achieving performance requires computation at a low level and in-depth knowledge of the underlying architecture.

These are two examples of the natural tension between the software goals of performance and productivity. In the former, we have *performance programmers* who sacrifice productivity for performance and in the latter, we have *productivity programmers* whose main goal is rapid prototyping. This motivates the need for an infrastructure to enable *both high-performance and high productivity*.

To that end, we present *Portal*, a domain-specific language and compiler embedded in C++ for yet another uncharted domain of *N-body* problems. Our specific work on *N-body* optimizations and tuning follows from the vast body of prior work on hand-optimizing and tuning *N-body* algorithms and applications in scientific particle simulations [3]–[5]. The initial choice of the domain of *N-body* problems stems from its applications in various domains ranging from scientific computing simulations in molecular dynamics, astrophysics, acoustics, fluid dynamics all the way to big data and machine learning problems [6]–[8]. *N-body* methods were identified as one of the original seven dwarfs or motifs [9] and are believed to be important in scientific computations. There is potential for big impact in this domain and general *N-body* applications today are still orders of magnitude from optimal. *Portal* aims to fill this gap.

The first goal in the development of *Portal* is to choose optimal algorithms with time/accuracy guarantees. Big data motivates fast approximate algorithms. To that end, *Portal* is built on the top of the *algorithmic framework PASCAL* [10] which utilizes tree data-structures and user-controlled pruning or approximations to reduce the asymptotic runtime complexity from being linear in the number of data points to be logarithmic. *PASCAL* can generate prune and approximation conditions for any number of operators making it easily extensible to a large class of *N-body* problems. We describe the algorithmic choices and the abstractions in *PASCAL* in Section II.

Contributions and findings. This paper makes the following contributions.

- 1) **[Portal Language]** How to best express N -body problems at a *high-level* representation? We design the Portal language inspired by the mathematical formulation of N -body problems. Such a high-level representation described in Section III not only gives the compiler freedom to apply transformations and optimizations but also to choose an optimal algorithm.
- 2) **[Real Code]** How to translate from a *high-level* representation to an efficient *low-level* code? We develop a *domain-specific compiler* that chooses the optimal algorithm and generates optimized and parallel vector code for x86 architectures outlined in Section IV. Experimental results on 6 problems show that the programs generated by Portal are within a factor of 5% (on average) of expert hand-tuned code. Additionally, we also compare the lines of code of Portal programs against hand-optimized and library codes. For example, k-nearest neighbors is written in 13 lines of Portal code and achieves within 2 – 5% of expert hand-tuned performance (Section V).
- 3) **[Portal Validation]** We also validate Portal against three separate N -body problems (namely, 2-point correlation, naive Bayes classifier, and Barnes-Hut) that have not been hand-optimized and compare against state-of-the-art libraries/packages. The parallel code generated by Portal using optimal tree-based algorithms outperforms libraries/packages such as scikit-learn [11] and ML-PACK [12] by a factor of 15 – 165 \times for the computation of 2-point correlation and naive Bayes classifier. For the Barnes-Hut computation, we compare the performance of Portal against FDPS [13] which is a high-performance hand-optimized particle simulation framework in C++. Portal achieves 70% better performance compared to FDPS on a dual-socket AMD EPYC processor (Section V).

This work, based on DSLs has the potential to allow for interoperability and scalability of parallel fast tree-based N -body problems which we believe will play a major role as we approach *exascale computing* towards the end of the decade. We show that a DSL with an appropriately high-level formulation leads directly to both asymptotically fast algorithms and their efficient parallel implementations on multicore systems. Portal is available open-source at <https://gitlab.com/Nbody-Portal/Code>.

II. BACKGROUND: N -BODY PROBLEMS

N -body problems are those where an update to a single element in the system depends on every other element, making these computations asymptotically $\mathcal{O}(N^2)$. The most familiar example arises in physical simulations which has the following form.

$$\forall q, \sum_r K(x_q, x_r) \cdot s(x_r), \quad (1)$$

where $s(x_r)$ is the density of the reference point and $K(x_q, x_r)$ is an interaction kernel that specifies *the physics* of

the problem. For instance, the Laplace kernel, $K(x_q, x_r) = \frac{1}{\|x_q - x_r\|}$ models electrostatic or gravitational interactions.

It turns out, this style of N -body problems arises in other significant domains such as machine learning (ML) and computational statistics which capture powerful reductions beyond sum. Some examples of such problems are k-nearest neighbors, range search, expectation maximization, Euclidean minimum spanning tree, n-point correlation to name a few.

In the general form of N -body problems, a set of m operators $\{op_1, \dots, op_m\}$ are applied to m datasets ($\mathcal{D}_1 \dots \mathcal{D}_m$) using a kernel function, \mathcal{K} , as follows.

$$op_1, \dots, op_m \mathcal{K}(x_1, \dots, x_m) \quad (2)$$

where $x_1 \in \mathcal{D}_1, \dots, x_m \in \mathcal{D}_m$. The common theme that brings these problems from various areas under a single umbrella is the insight that their inner-loop computations are analogous and naively require $\mathcal{O}(N^m)$ operations to evaluate (where N is the size of each dataset and m is the number of datasets). This raises the question of how to compute these updates in an efficient scalable fashion.

We re-use the algorithmic abstractions in PASCAL [10] which generalizes the N -body problems under a single umbrella. It consists of space partitioning trees, a prune/approximate condition generator, and a multi-tree traversal scheme which we describe in the following subsections.

A. Space-partitioning Trees

There exists a powerful class of space-partitioning tree-based algorithms that can reduce the complexity of N -body problems from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$ and even $\mathcal{O}(N)$ [7], [14]. These algorithms use techniques such as approximation and pruning to discard regions of space.

Two well-known algorithms in physics N -body simulations are the Fast Multipole Method (FMM) [7] and Barnes-Hut [14] that make use of low-dimensional spatial trees such as quadtrees in 2-D and octrees in 3-D. A commonly used tree in ML and data mining applications is *kd-trees* [15] which are typically based on the same principles as quad- and octrees but can handle high-dimensional data. This binary tree structure maintains bounding boxes for all the nodes in each level. Children are formed by recursively subdividing the bounding box space based on some splitting criterion. The partitioning stops when each child node contains no more than q points ($q > 0$). The bounding box information allows us to efficiently compute the center, minimum and maximum node-to-point and node-to-node distances during evaluation without accessing the actual points in each node, which is critical for performance.

B. Classification of N -body Problems

We classify N -body problems into two main categories – (a) approximation, and (b) pruning problems. Approximation problems are those in which the contribution by a subset of the data to the solution can be approximated by a smaller subset. This group of problems only encompass arithmetic operators and non-comparative kernels, such as Barnes-Hut and FMM.

For example, operators such as Σ or Π require the contribution of all the points. In such *approximation* problems, as the name suggests, there is a trade-off between performance and desired accuracy. Portal exposes this trade-off to the user as a tuning knob which can be customized based on the target application.

Pruning problems are those in which a part of the data and its associated computation can be discarded. One example is nearest neighbors whose inner operator is \min which only returns the minimum value. This provides opportunities to prune parts of the subtrees where there is no likelihood of finding the minimum value, without any loss of accuracy. Such pruning opportunities are deduced based on the set of operators and kernel function. Comparative operators such as \min or \max result in a pruning problem and same is true for comparative kernel functions such as $(x_r - x_q < h)$.

C. Multi-tree Traversal

After building the space-partitioning tree, PASCAL uses a multi-tree traversal in order to compute the N -body problem. Algorithm 1 describes the multi-tree traversal given two inputs: a set of nodes, \mathcal{N}^{all} and a rule set, \mathcal{R} . The rule set provides three main functionalities described below and the corresponding functions are highlighted in blue in Algorithm 1. Note that the traversal is called recursively on the children at each level of the tree as seen by the green function call in line 11.

Algorithm 1 MultiTreeTraversal

Require: Nodes set $\{\mathcal{N}_1, \mathcal{N}_2, \dots, \mathcal{N}_m\} \approx \mathcal{N}^{all}$, rule set \mathcal{R} .

```

1: if  $\mathcal{R}.$ Prune/Approximate( $\mathcal{N}_1, \mathcal{N}_2, \dots, \mathcal{N}_m$ ) then
2:   return  $\mathcal{R}.$ ComputeApprox( $\mathcal{N}_1, \mathcal{N}_2, \dots, \mathcal{N}_m$ )
3: if ( $\forall \mathcal{N}_i \in \mathcal{N}^{all}$  is leaf) then
4:    $\mathcal{R}.$ BaseCase( $\mathcal{N}_1, \mathcal{N}_2, \dots, \mathcal{N}_m$ )
5: else
6:   for all  $\mathcal{N}_i \in \mathcal{N}^{all}$  do
7:     if  $\mathcal{N}_i$  is leaf then  $\mathcal{N}_i^{split} = \mathcal{N}_i$ 
8:     else  $\mathcal{N}_i^{split} = \{\mathcal{N}_i.right, \mathcal{N}_i.left\}$ 
9:     PowerSet-Tuples =  $\{(\mathcal{N}'_1, \dots, \mathcal{N}'_m) | \mathcal{N}'_i \in \mathcal{N}_i^{split}\}$ 
10:    for all  $(\mathcal{N}''_1, \dots, \mathcal{N}''_m) \in$  PowerSet-Tuples do
11:      MultiTreeTraversal( $\mathcal{N}''_1, \dots, \mathcal{N}''_m$ )

```

BaseCase implements the direct point-to-point computation only for the leaf nodes of the tree. For instance, for the nearest neighbors problem, this would be equivalent to finding the corresponding point in the reference node which has the minimum distance for every point in the query node.

Prune/Approximate checks if the computation for a set of nodes can be *approximated* or *pruned* based on the N -body problem definition, using a prune/approximate generator. If so, the node and its descendants will not be visited. The prune/approximate generator [10] first categorizes the N -body problem based on the classification in Section II-B and then generates its condition accordingly.

For approximating the contribution of a node, we check if the minimum and maximum contribution of that node are very close (i.e. less than a user-defined threshold). If so, we know

that all the data points in that node have a similar contribution and therefore, *ComputeApprox* replaces the computation with the center contribution of each node multiplied by the density of that node which is equivalent to the number of data points in that node. In Barnes-Hut, this would result in replacing the influence of the points in the node with their center of mass.

In the case of pruning problems, PASCAL first builds a pipeline of pruning opportunities (deduced from comparative operators and kernel) which are then evaluated on the boundary points of each hyper-rectangle node to decide whether to prune that node or not. In some cases, the algorithm prunes entire sub-trees, so the nodes and their descendants will not be visited. Portal adapts the prune/approximate generator of PASCAL to generate the prune condition based on the Portal language specification.

Note that the N -body problems must satisfy two properties to enable one to choose the tree-based optimal algorithm – (1) operators satisfy the decomposability property over datasets. Decomposability applies to an operator if the dataset is decomposable across its subsets. For instance, the Σ operator adheres to the decomposability property because the sum of all interactions over the dataset can be decomposed into the sum of smaller subsets of that data, and (2) the kernel function should decrease monotonically with distance.

Portal leverages the above generalized algorithmic framework of PASCAL, which abstracts the actual computation from the traversal, and also abstracts the tree type which gives us the freedom to plug and play with different trees.

III. PORTAL DSL

The design of the Portal DSL is inspired by the mathematical formulation in (2). It aims to describe N -body problems in a well-structured high-level form that allows domain experts to focus on the specification of the problem rather than the algorithm or its associated implementation on a target platform.

Recall that a N -body problem is defined using a set of m operators applied to m datasets ($\mathcal{D}_1 \dots \mathcal{D}_m$) using a kernel function, \mathcal{K} , as seen in (2) - Section II. Each operator, dataset, and/or kernel function can be associated with a layer. Problems are built up by chaining multiple layers to specify a query. Changing the operator, dataset, or the order of layers can change the specification and meaning of the problem. Note that the same dataset may be reused in multiple layers. We explain the structure and semantics of the Portal language using a well known N -body problem in machine learning, *nearest neighbor*, which has the following form.

$$\forall_q \operatorname{argmin}_r \|x_q - x_r\| \quad (3)$$

Code 1 shows the Portal specification of nearest neighbors problem¹. **PortalExpr** in line 3 is the main object that holds the problem definition. The first or outer-most layer applies the \forall (FORALL) operator over the query (q) dataset. The second or inner-most layer applies the argmin (ARGMIN)

¹Appendix VIII describes the grammar of the Portal language.

operator over the reference (r) dataset, as well as the Euclidean distance kernel function, which calculates the distance between two points. A layer is added to the `PortalExpr` using the `addLayer()` method, which allows a user to build the structure of the problem (lines 4-5). The `execute()` function in line 6 generates and runs the tree-based N -body algorithm and `getOutput()` in line 7 returns the result of the computation in the format of a `Storage` object.

Portal code 1: Portal language specification of the nearest neighbor problem using pre-defined Euclidean distance metric.

```

1 Storage query("query_file.csv");
2 Storage reference("reference_file.csv");
3 PortalExpr expr;
4 expr.addLayer(PortalOp::FORALL, query);
5 expr.addLayer(PortalOp::ARGMIN, reference,
6   PortalFunc::EUCLIDEAN);
7 expr.execute();
8 Storage output = expr.getOutput();

```

As seen from code 1, each layer can consist of three components – (a) portal operator, (b) storage object, and (c) kernel/modifying function. We explain each component in detail in the following subsections.

A. Portal Operators

Portal operators are a set of distinct operators that filter the results of their layer and pass it to the next outer layer or the output. A list of these operators is defined in Table I. Each layer requires an operator to specify what sort of filtering happens on the data that layer computes on. We categorize operators into 3 groups – (a) single variable reduction operators, (b) multivariable reduction operators, and (c) all operator. This categorization is helpful in deciding the type of intermediate storage required for each layer. It allows Portal to allocate just the right amount of storage to propagate to outer layers.

TABLE I: Mathematical operators supported in Portal along with their categorization into *Single*, *Multi*, and *All* operators.

Category	Mathematical Operator	Portal Operator
All	\forall	FORALL
Single	\sum	SUM
Single	\prod	PROD
Single	argmin	ARGMIN
Single	argmax	ARGMAX
Single	\min	MIN
Single	\max	MAX
Multi	\cup	UNION
Multi	$\cup \text{arg}$	UNIONARG
Multi	argmin^k	KARGMIN
Multi	argmax^k	KARGMAX
Multi	\min^k	KMIN
Multi	\max^k	KMAX

Single variable reduction operators. These operators take a set of values as input and reduce the set to a single output. These operators are listed in the *Single* category in Table I.

For example, when the \min operator is applied to a reference dataset, it returns the minimum value which is a single output.

Multi variable reduction operators. These operators take a set of values as input and reduce them to a smaller set of values, typically with a specified length, k . These operators include argmin^k , argmax^k , \min^k , \max^k , \cup , and $\cup \text{arg}$. Each multivariable reduction operator requires us to specify a number k , which limits the number of values that get filtered, except \cup and $\cup \text{arg}$ operators. In our nearest neighbor example, using argmin is the same as using argmin^k when $k = 1$. If one desires to compute the k closest reference points to each query point, the Portal operator in the inner `addLayer()` method in line 5 of code 1 has to be modified to apply a multivariable reduction operator as shown below.

```

expr.addLayer((PortalOp::KARGMIN, k), reference,
  PortalFunc::EUCLIDEAN);

```

All operator. The \forall operator does not filter any values and given an input, it returns all the data. In the nearest neighbor example, the \forall operator is applied to the outermost layer to compute the nearest neighbors of **all** query points in the set q .

B. Storage

Each layer includes a `Storage` object, which corresponds to a dataset. `Storage` objects are designed to be the primary user-facing data structure. `Storage` objects can be created from C++ vectors (as shown below) or from CSV files (as shown in lines 1-2 of code 1).

```

// Construct Storage from C++ data-structure
std::vector<std::vector<float>> input;
// Fill in the input data structure with values
Storage query(input);

```

Portal determines the data layout of the `Storage` objects based on the dimensionality of the dataset to further optimize performance. Portal can choose between column- or row-major data layout. When the dataset has a smaller dimensionality (less than or equal to 4), Portal chooses a column-major layout. Datasets with a larger dimensionality default to a row-major layout. Portal makes this decision to enable efficient compiler auto-vectorization which is discussed in more detail in Section IV.

Portal generates a space-partitioning tree for each input `Storage` object. The output `Storage` object will be returned by calling the `execute()` function. Both input and output storage objects are available while Portal deletes all the intermediate storage objects after `execute()` function. The programmer can delete input/output storage object using the `clear()` function (not shown).

C. Kernel/Modifying Function

The last component constituting a layer is the kernel/modifying function. This function is required for the innermost layer and is commonly known as the *kernel function* in literature [6]. Other layers can also specify functions, however, we call them *modifying functions*. The kernel function specifies the *science* of the problem which depends on the distance

between pairs of points in the chosen parameter space. Portal implements a set of commonly used distance metrics (for ease of use), which can be used to compose kernel functions.

Portal code 2: Examples of pre-defined distance metrics.

```

1 expr.addLayer(PortalOp::FORALL, reference,
2   PortalFunc::MANHATTAN);
3 expr.addLayer(PortalOp::FORALL, reference,
4   PortalFunc::CHEBYSHEV);
5 expr.addLayer(PortalOp::FORALL, reference,
6   PortalFunc::MAHALANOBIS);
7 expr.addLayer(PortalOp::FORALL, reference,
8   PortalFunc::SQREUCDIST);

```

For instance, code 1 uses the pre-defined Euclidean distance metric to specify the kernel function for nearest neighbors which in this case is equal to the distance between points in set q and r in the Euclidean space. Code 2 shows examples of other pre-defined distance metrics such as Manhattan, Chebyshev, Mahalanobis, and Square Euclidean distances.

Portal also allows the user to define custom kernel functions. Code 3 shows how Euclidean distance, the same function used in our nearest neighbor example, can be defined using the `Expr` object in line 5. `Expr` can be specified using `Var` objects as input variables. These `Var`'s can be used in a wide variety of calculations using a specified set of operations that can be compiled and optimized by Portal. The `Var` objects are then linked to specific layers using the `addLayer` method as shown in code 3.

Portal code 3: Portal specification for user-defined distance metric and kernel function for nearest neighbors.

```

1 Storage query("query_file.csv");
2 Storage reference("reference_file.csv");
3 Var q;
4 Var r;
5 Expr EuclidDist = sqrt(pow((q-r), 2));
6 PortalExpr expr;
7 expr.addLayer(PortalOp::FORALL, q, query);
8 expr.addLayer(PortalOp::ARGMIN, r, reference,
9   EuclidDist);
10 expr.execute();
11 Storage output = expr.getOutput();

```

To allow for greater flexibility, users can also define their own external C++ functions. For example, if the user wants to use another library to specify a kernel function, the `addLayer` method accepts C++ functions. Since Portal is an embedded DSL, we can take advantages of these libraries and infrastructures provided by the host language. Although external C++ functions will not be optimized in the same way the internally specified Portal kernel functions are, they allow for greater flexibility on what problems Portal can solve. Additional bindings to other languages, such as MATLAB or Python, can be added in the future.

IV. PORTAL COMPILER

To synthesize optimal machine code from the high-level Portal language, the Portal compiler proceeds through the

major stages as illustrated in Fig. 1. First, Portal builds space-partitioning trees for the input datasets. The next step is to synthesize code for the three key functions in the multi-tree traversal (Algorithm 1). The *Prune/Approximate* and *ComputeApprox* functions in the tree traversal are generated using a modified version of the prune generator in the PASCAL algorithmic framework [10]. To use the prune generator in Portal, we modify it to get the Portal operators and kernel function as input and emit the *Prune/Approximate* and *ComputeApprox* functionality in Portal IR. This enables us to apply optimizations and transformations to the former functions in the compiler backend.

Portal core compiler lowers and synthesizes loops for the *BaseCase* given a `PortalExpr` and automatically manages the intra-layer storage injection. Then, it flattens multi-dimensional store, load, and allocations. After flattening, we perform numerical and strength reduction optimizations specific to N -body problems. Fig. 2 and 3 illustrate the IR for the key functions in the tree-traversal (Algorithm 1) and walk through the different stages of the compiler transformations for two N -body problems, namely nearest neighbor and kernel density estimation respectively. The reason for choosing these two problems is to illustrate the IR and Portal stages for both prune (nearest neighbor) and approximation (kernel density estimation) examples from the two categories of N -body problems.

Finally, backend code generation emits x86 machine code using LLVM. The parallelization is applied in the tree traversal which applies to all the N -body problems expressed in Portal. Note that in addition to the asymptotically optimal tree algorithm, Portal also generates the code for the brute-force algorithm. This is currently used for correctness checks. In the rest of this section, we describe each of these major compiler steps in detail.

A. Lowering

The first step of our compiler is the lowering process that synthesizes a set of nested loops given a `PortalExpr` object. The order of the loops follows the same as the language specification (e.g. outermost layer mapping to the outermost loop) since the ordering defines the structure of the N -body computation. Loops are defined by their minimum and maximum values, and all loops implicitly stride by 1.

Next, Portal builds an argument list for each layer and assigns the initial values for each operator and reduction filter. For all inner layer operators, intermediate values of operations are stored in the intermediate storage objects, which are initially assigned with their default values. For example, for the `min` operator, the initial value of the intermediate storage is set to the highest value for that specific numeric type as its default value (e.g. `DBL_MAX` for double precision data).

After that, the kernel/modifying functions are lowered into Portal IR. Finally, Portal lowers the mathematical functionality of each operator at the end of the corresponding synthesized loop. For example, for the `min` operator, Portal generates a

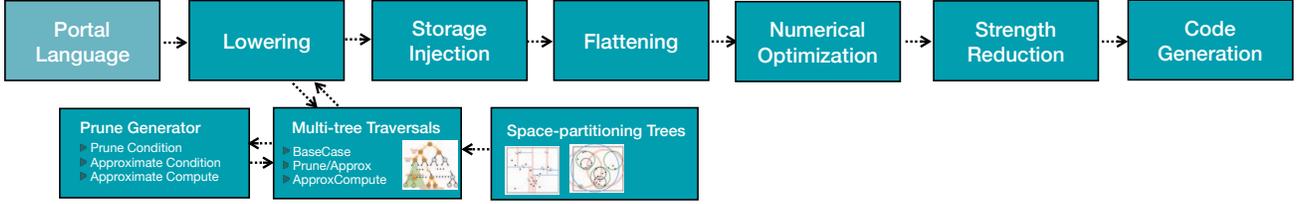


Fig. 1: Portal block diagram. The core of the compiler lowers the N -body problem defined in the Portal language to imperative code. Portal constructs a loop nest and injects storage for each layer of the loop according to their corresponding operators. The backend code generator emits machine code via LLVM.

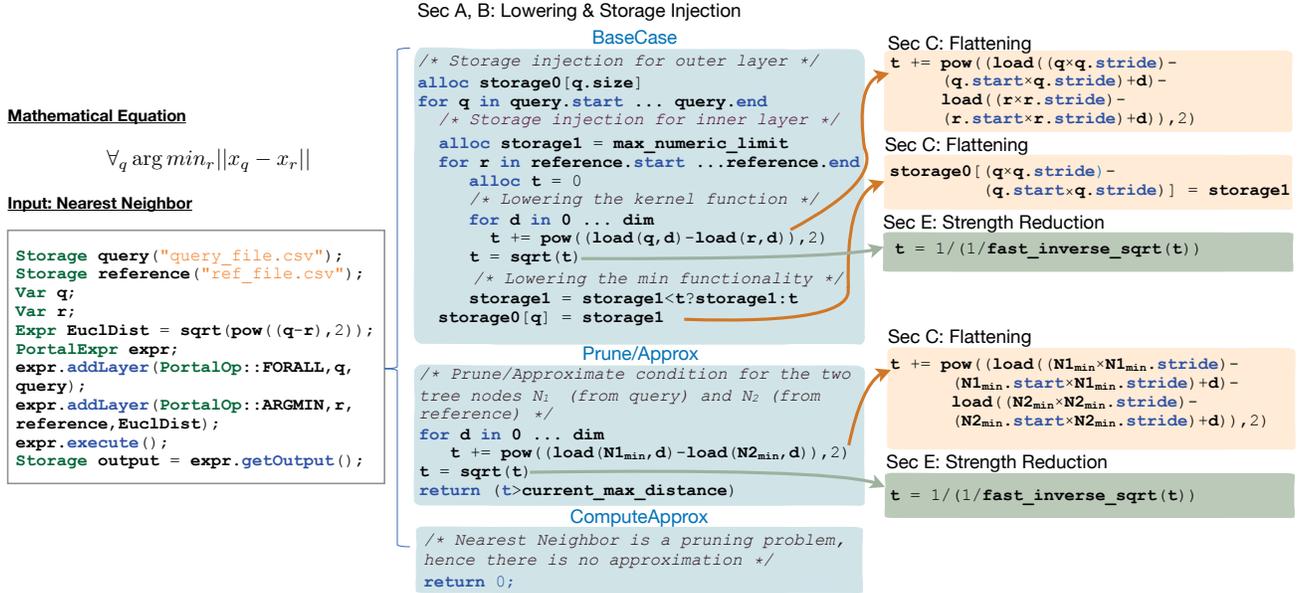


Fig. 2: The IR representation of the nearest neighbor problem illustrating the IR of the three main functions in the tree traversal (BaseCase, Prune/Approximate, and ComputeApprox) along with the different transformation applied to it. The nearest neighbor returns zero for ComputeApprox since it is a prune N -body problem. Note that there is no numerical optimization applied for this problem since the nearest neighbor doesn't use Mahalanobis distance. Portal uses metadata information from the tree, such as min, max, and center of the nodes of the tree for computing the Prune/Approximate condition efficiently. Refer to [10] for more details on the generation of the Prune/Approximate conditions. The blue colored rectangles (middle) show the IR after lowering and storage injection (subsections IV-A and IV-B), while yellow and green colored rectangles (right) present the IR after flattening and strength reduction respectively (subsections IV-C and IV-E).

comparison imperative code at the end of loop synthesis, to update the minimum computation of that layer.

B. Storage Injection

Storage injection creates the output and all intermediate data storage for each layer. The intermediate storage is necessary to pass data between layers; each inner layer filters and passes its results to the next outer layer as intermediate storage. As Portal recursively moves across layers in order to synthesize nested loops, it dedicates storage for each layer depending on the layer's operator and category listed in Table I. The storage injected for *single variable reduction operators* is only one unit of data, as the output of that layer is only a single value, while the storage for *multivariable reduction operators*

is equal to the size of the multivariable defined by the operator (such as k for KARGMIN). As a special case, the \forall operator injects a storage object equal to the size of that layer's dataset. This is because \forall doesn't serve to filter through the data, rather to output all the calculations.

For example in the nearest neighbor problem, the inner layer searches for the `argmin` on the reference dataset. Since `argmin` is a single reduction operator, we inject one memory location per output of this layer. The outer layer applies a \forall operator on the query data, which results in a storage injection as large as the query set.

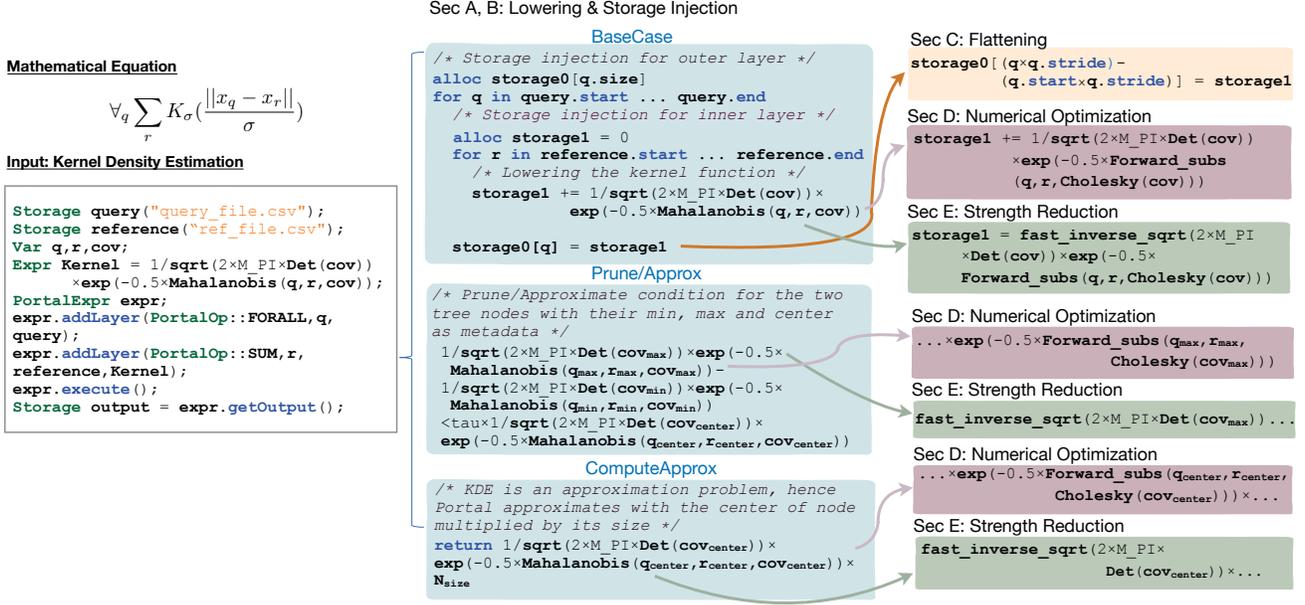


Fig. 3: The IR representation of kernel density estimation (KDE) problem illustrating the IR of the three main functionalities in the tree traversal (BaseCase, Prune/Approximate, and ComputeApprox) along with the different transformation applied to it. The Kernel (\mathcal{K}) in the mathematical equation is the Gaussian kernel. KDE also benefits from the numerical optimizations for the Mahalanobis distance computation. Portal uses metadata information from the trees such as min, max, center, and size of the node for computing Prune/Approximate condition and the associated approximation, *ComputeApprox*. This metadata is denoted with the subscript notation. For example, the metadata q_{center} is the center of a node q of the tree. The blue colored rectangles (middle) show the IR after lowering and storage injection (subsections IV-A and IV-B), while yellow, purple, and green colored rectangles (right) present the IR after flattening, numerical optimization, and strength reduction respectively (subsections IV-C, IV-D, and IV-E).

C. Flattening

The Portal compiler flattens multi-dimensional loads and stores into one-dimensional load and store operations. Similarly, for nested loop arguments, Portal flattens the arguments using the base offset and strides from each loop to compute a one-dimensional version of its arguments.

D. Numerical Optimization

After flattening, Portal performs two optimization passes that exploit domain (N-body) specific knowledge. The first such pass is numerical optimization, which implements a fast calculation between a point x and a distribution D , known as Mahalanobis distance. It generalizes the measure of how many standard deviations away x is from the mean of distribution D . If we re-scaled each x to have unit variance, this distance then would correspond to the standard Euclidean distance in the transformed space. About 60% of basic statistical inference N -body problems presented in [16] have a form of Mahalanobis distance calculation. Therefore, optimizing this distance metric can benefit a large subset of N-body problems.

Mahalanobis distance between two points, q and r is defined as, $(x_q - \mu_r)^T \Sigma^{-1} (x_q - \mu_r)$ where μ and Σ are the mean and covariance respectively. Naively evaluating this distance requires computing the inverse of the covariance matrix which

is an expensive linear algebra operation that can take on the order of m^3 operations where m is the matrix dimension. However, we can exploit the fact that the covariance matrix of any real random vector is symmetric positive semi-definite.

That is, we rewrite the covariance matrix with a combination of Cholesky decomposition and forward substitution which reduces the complexity to $m^2/2$. First, we factorize the covariance matrix using Cholesky decomposition to obtain a lower triangular matrix as follows.

$$(x_q - \mu_r)^T \Sigma^{-1} (x_q - \mu_r) = (x_q - \mu_r)^T (LL^T)^{-1} (x_q - \mu_r)$$

where L is a lower triangular matrix. If $Y = x_q - \mu_r$, we can further re-write the right-hand side of the above equation as,

$$Y^T (LL^T)^{-1} Y = (L^{-1}Y)^T L^{-1}Y.$$

Since L is a lower triangular matrix, we can use forward substitution to compute $X = L^{-1}Y$, in turn, reducing the original exponent to $X^T X$, a simple and cheap inner product. Moreover, we can now compute the determinant of the covariance matrix, $|\Sigma|$ by simply multiplying the diagonal entries of the lower triangular matrix, L which is another expensive

operation that arises in such computations.

E. Strength Reduction

Operations such as are power (`pow`), square-root (`sqrt`), and reciprocal square-root ($1/\sqrt{x}$) that arise in many N -body problems [16] have long latencies. The second optimization pass is strength reduction which replaces such expensive operations with faster, albeit less accurate versions. If the `pow` operation has an exponent less than 4, Portal replaces this with a chained multiplication. For computing $1/\sqrt{x}$, we use the fast inverse square root that is provided by LLVM, which can result in up to $4\times$ faster performance compared to the naive version with an error of 0.17% [17]. For approximation problems with inherent time/accuracy trade-off, this optimization pass provides an additional tuning knob.

There are two potential ways of calculating \sqrt{x} – (a) to multiply x with its fast inverse square root ($x * 1/\sqrt{x} = \sqrt{x}$), and (b) inverse of fast inverse square root ($1/(1/\sqrt{x}) = \sqrt{x}$). The former is faster, however when $x = 0$ then the former returns NaN while the latter returns 0 as desired.

F. Code Generation

Finally, we perform low-level optimizations and emit machine code for the defined N -body problem. Portal backend uses LLVM for low-level code generation. We first perform a set of standard passes, such as constant-folding and dead-code elimination. The Portal IR is then lowered to LLVM IR. For the most part, there is a one-to-one mapping between Portal and LLVM IR. However, for some filters, we implement additional data-structures. For example in the case of multivariable reduction filters such as \min^k , we implement an ordered array of size k to keep a sorted list of the minimum distances calculated so far. Keeping these values sorted allows for efficient computation and fewer comparisons in each iteration/update. After generating LLVM IR, the compiler links external functions, including user-defined external kernel functions. Before finishing the compilation, this function is finally wrapped around another function that the user and our methods can easily call upon.

The parallelization happens in the tree traversal using OpenMP. In Portal, we exploit a combination of data and task parallelism. Initially, we spawn OpenMP tasks recursively until all the threads are saturated, at which point we switch to data parallelism. The generated code is auto-vectorized by the compiler. To enable efficient auto-vectorization, recall that Portal chooses between column- and row-major data layout based on the dimensionality of the dataset.

To make sense of this, we break down our nearest neighbor example. The *BaseCase* computation between two leaf nodes consists of three nested loops; the outermost loop iterates over all the points in the query set, the middle loop iterates over all the points in the reference set, and the innermost loop iterates through all the elements within the two chosen query and reference data points to calculate the distance. The loop extent of the innermost loop is determined by the dimensionality of the datasets.

For low dimensional data, the compiler can unroll the innermost loop, thereby exposing vectorization opportunities at the level of the middle loop. To exploit this, we store the data in a column-major layout such that every row stores values from the same dimension for different points. In other words, each data point is stored as a column while in row-major layout, each data point is stored as a row. When using column-major layout, each cache line loads data from different data points. Since the middle loop is the target of vectorization for low dimensional data, this results in less wait for memory loads and consequently better vectorization performance. On the other hand, for high dimensional data, the compiler does not unroll the innermost loop due to large loop counts. Therefore, to exploit vectorization in the innermost loop, we use a row-major data layout where every row stores one data point.

V. EVALUATION AND DISCUSSION

In this section, we first evaluate the code generated by Portal against hand-optimized PASCAL implementations of 6 N-body problems. We then demonstrate Portal’s ability to generate optimal code on three additional N-body problems not implemented in PASCAL. For the ML problems, we compare their performance against widely used open-source libraries and frameworks such as MLPACK [12] and scikit-learn [11]. For Barnes-Hut computation, we compare against hand-optimized C++ code from the FDPS framework [13]. All evaluations are performed on the state-of-the-art AMD EPYC 7551 multicore processor with a total of 128 cores.

A. Experimental Setup

Architecture and Compiler. For our evaluation, we choose a dual-socket AMD EPYC 7551 processor. Each socket has 64 cores, for a total of 128 cores and a theoretical double precision peak performance of 2611.2 GFlops/s. We use clang compiler version 6.0.0 and LLVM version 6.0.0. We use Python v3.7.0 for scikit-learn v0.20.0 and MLPACK 3.0.3.

Benchmarks. We present results on six real-world datasets characterized in Table II. These include Yahoo! front page module user click log dataset, v1.0 (Yahoo!), Higgs boson’s signals and background process dataset (HIGGS), Individual Household Electric Power Consumption dataset (IHEPC), US Census data from 1990 (Census), and KDD Cup 1999 dataset (KDD) from the UCI ML repository [18]. The 3-dimensional dataset (Elliptical) is generated for evaluating the Barnes-Hut algorithm where particles are angularly uniformly (in spherical coordinates) distributed on the surface of an ellipsoid with an aspect ratio

TABLE II: Description of the datasets. N is the number of points and d is the dimensionality.

Dataset	N	d
Yahoo!	41904293	11
IHEPC	2075259	9
HIGGS	11000000	28
Census	2458285	68
KDD	4898431	42
Elliptical	10000000	3

TABLE III: Summary of the characteristics of the nine N -body problems chosen for evaluation (EM consists of two N -body sub-problems: E-step and log-likelihood). The kernel functions are evaluated on the border points given by N_r^{border} and N_q^{border} . $N_r^{diameter}$ represents the span of the widest dimension in hyper-rectangle N_r . Metadata such as min, max, and center represent the minimum, maximum, and center data point in a hyper-rectangle. τ , τ_1 , τ_2 , and σ represent threshold values defined based on the N -body problem or controlled by the user. For the EM computation, μ_k, Σ_k are the parameters of Gaussian component, k , with mean vector μ_k and covariance matrix Σ_k , while π_k is the mixing weight. Other parameters such as M_q and M_r in Barnes-Hut are defined based on the problem specification which in this case refer to the masses of the interacting objects. * denotes iterative algorithms.

N -body Problems	Operators	Kernel Function	Pruning/Approximation Condition
k -Nearest Neighbors	$\forall, \arg \min$	$\ x_q - x_r\ $	Prune $\Leftrightarrow \ x_q - x_r\ \geq \tau, \forall x_r \in \mathcal{N}_r^{border}$
Range Search	$\forall, \bigcup \arg$	$I(h_{\min} < \ x_q - x_r\ < h_{\max})$	Prune $\Leftrightarrow \ x_q - x_r\ > h_{\max}$ or $\ x_q - x_r\ < h_{\min}, \forall x_r \in \mathcal{N}_r^{border}$
Hausdorff Distance	max, min	$\ x_q - x_r\ $	Prune $\Leftrightarrow \tau_1 \geq (\ x_q - x_r\ : \tau_2 \leq \ x_q - x_r\), \forall x_q \in \mathcal{N}_q^{border}, \forall x_r \in \mathcal{N}_r^{border}$
Kernel Density Estimation	\forall, \sum	$\mathcal{K}(\frac{\ x_q - x_r\ }{\sigma})$	Approximate $\Leftrightarrow \mathcal{K}_{\max} - \mathcal{K}_{\min} < \sigma \times \mathcal{K}_{\text{center}}$
Minimum Spanning Tree*	$\forall, \arg \min$	$\ x_q - x_r\ $	Prune $\Leftrightarrow \ x_q - x_r\ \geq \tau, \forall x_r \in \mathcal{N}_r^{border}$
E-step in EM*	\forall, \forall	$r_{nk} = \frac{\pi_k \mathcal{N}(x_n \mu_k, \Sigma_k)}{\sum_{j=1}^K \pi_j \mathcal{N}(x_n \mu_j, \Sigma_j)}$	Approximate $\Leftrightarrow (r_i^{\max} - r_i^{\min}) < \sigma \times r_i^{\text{center}}, i = 1, \dots, K$
Log-likelihood in EM*	$\sum, \log \sum$	$\pi_k \mathcal{N}(x_n \mu_k, \Sigma_k)$	Approximate $\Leftrightarrow \log \sum_{i=1}^K \pi_i \mathcal{N}(x_{\max} \theta_i) - \log \sum_{i=1}^K \pi_i \mathcal{N}(x_{\min} \theta_i) < \sigma \times \log(\sum_{i=1}^K \pi_i \mathcal{N}(x_{\text{center}} \theta_i)) $
2-Point Correlation	\sum, \sum	$I(\ x_q - x_r\ < h)$	Prune $\Leftrightarrow \ x_q - x_r\ \geq h, \forall x_r \in \mathcal{N}_r^{border}$
Naive Bayes Classifier	$\forall, \arg \min$	$\mathcal{N}(x_n \mu_k, \Sigma_k)$	Prune $\Leftrightarrow \mathcal{N}(x_n \mu_k, \Sigma_k) > \tau, \forall x_n \in \mathcal{N}_r^{border}$
Barnes-Hut	\forall, \sum	$f = \frac{GM_q M_r}{(\ x_q - x_r\)^2}$	Approximate $\Leftrightarrow N_r^{diameter} < \tau \times \ x_q - N_r^{\text{center}}\ $

1:1:4. The elliptical dataset generates an adaptively refined octree.

B. Comparison with PASCAL

Table III shows a summary of key characteristics of the six N -body problems chosen for comparison namely, k -nearest neighbors (kNN), range search (RS), kernel density estimation (KDE), Hausdorff distance (HD), minimum spanning tree (MST), and expectation maximization (EM). The choice of these six is because they cover a variety of different N -body problems including pruning vs approximation and direct vs iterative problems. They also include problems from different metric spaces (e.g. Euclidean vs Mahalanobis). PASCAL achieves orders of magnitude higher performance compared to widely used libraries and frameworks such as MLPACK, scikit-learn, MATLAB, and Weka [10] for all six problems. Therefore, we compare against PASCAL's hand-optimized implementations.

Table IV presents the running time on five datasets for the code generated by Portal and hand-optimized PASCAL implementations (called *expert*) for the six problems. Both implementations use the same kd -tree data-structure (built using median partitioning strategy by splitting along the widest

dimension) and the same multi-tree traversal template (Algorithm 1). For both Portal and expert, we use the number of cores that deliver the highest performance for each dataset and problem combination. We also empirically tune the algorithmic parameter, leaf size and level of tree parallelization which control the size and number of tasks created. Since we rely on the OpenMP work-stealing scheduler to balance the work across all 256 threads (at the highest concurrency) and 8 NUMA domains, it is critical to tune these parameters to achieve scalability, especially at this scale on a multicore system.

Across a range of problems and datasets, our Portal language is able to *express* and our Portal compiler is able to *generate* implementations using optimal tree-based algorithms that are within 5% (on average) of state-of-the-art performance. We observe the largest deviation of 8–9% for EM because of external function calls such as the calculation of covariance that is required for evaluating the kernel function shown in Table III.

In addition to running time, we also report the number of lines of code (LOC) in the last three columns of Table IV. Portal programs require significantly fewer LOC than hand-

TABLE IV: Comparison of the Portal running time (in seconds) against PASCAL for 6 N -body problems across 5 datasets. Each dataset spans 3 columns that report their respective performance and % difference. The last 3 columns show LOC as an indicator for user productivity. Note that for MST and EM in Portal only require 12 and 30 lines respectively for specifying the N -body problem. The rest of the code implements the iterative logic which is written in native C++ code. The last column (\times shorter) shows the factor that Portal code is shorter than PASCAL.

	Census			Yahoo!			IHEPC			HIGGS			KDD			LOC		
	Expert	Portal	% Diff	Expert	Portal	\times Shorter												
k -NN	22.8	23.9	4	84.6	85.2	2	8.7	9.1	4	186.0	191.0	3	21.4	22.6	5	867	13	67
KDE	1087	1129	3	133.7	139.5	4	39.2	41.7	6	411.8	430.9	4	926.5	946	2	626	15	42
RS	42.2	44.4	5	214.5	223.1	4	15	16.1	7	122	130.1	6	20.1	21.1	4	673	13	52
MST	374.1	391.7	4	918.4	946.1	3	200.8	211	5	478.3	486.2	2	273.6	281	3	956	54	17
EM	76.3	82.6	8	224.5	242.8	8	78.6	85.3	8	198.8	216.7	9	32.4	35.3	8	1681	104	16
HD	40.9	43.1	5	122.8	129.7	5	38.4	40.1	4	236.6	243.8	3	36.2	38.3	5	689	13	53

written hand-optimized code. For example, the Portal version of k -nearest neighbors is written in 13 lines of code and achieves within 2 – 5% of expert performance. EM, a soft clustering algorithm composed of two N -body sub-problems is the longest Portal program written in 104 LOC, which consists of 30 lines of Portal code and 74 lines of native C++ code which is $16\times$ fewer LOC compared to expert. Note that we do not report the tree construction, tree traversal, and prune generator LOC for expert since these modules can be reused when implementing a new problem. In summary, these results show the potential of Portal to express a wide range of N -body problems while achieving competitive performance with reduced programming effort.

C. Validation

We also validate Portal with three other N -body problems namely, 2-point correlation, naive Bayes classifier and Barnes-Hut that are not implemented in PASCAL. We refer the reader to the last three rows of Table III for the operators and the kernel functions defining these three problems. Since no framework like Portal exists at this time for generalized N -body problems (to the best of the author’s knowledge), we compare against open-source ML libraries/packages and optimized Barnes-Hut framework. We compare against scikit-learn [11] for 2-point correlation and MLPACK [12] for naive Bayes classifier. This is due to the fact that MLPACK does not implement 2-point correlation but both MLPACK and scikit-learn implement the naive Bayes classifier. However, MLPACK delivers the best performance among the two, so we only report MLPACK performance.

Scikit-learn is an open source project with more than 173,000 downloads, with 92 releases by 1265 contributors. MLPACK is a C++ machine learning library with emphasis on speed and ease of use with more than 137 contributors. Both scikit-learn and MLPACK implement a tree-based algorithm. For Barnes-Hut, we compare against FDPS, a high-performance framework specifically designed for parallel

particle simulations which provides a hand-optimized C++ implementation [13].

We choose kd -tree as the tree type for fair comparison across all implementations, and octree for Barnes-Hut. Table V shows that Portal achieves $66 - 165\times$ better performance than scikit-learn for 2-point correlation and is $15 - 47\times$ faster than MLPACK for naive Bayes classifier across different datasets. Portal achieves 70% better performance compared to FDPS for the Barnes-Hut computation on 10 million particles.

Portal is also comparable to ML libraries/packages written in popular languages such as python in terms of LOC. For instance, 2-point correlation is written in 12 lines of python code in scikit-learn. Portal code, on the other hand, is written in 13 lines of code providing similar productivity compared to high-level languages.

In summary, these results validate the potential of our approach resulting in performance that is orders of magnitude faster than state-of-the-art libraries. Additional algorithms can be expressed in this style with minimal programming effort resulting in out-of-the-box parallel optimized implementations.

VI. RELATED WORK

Domain-specific languages and compilers. In the past few years many domain-specific languages (DSLs) such as DeepDSL [19] for deep learning, Diesel [20] for linear algebra and neural nets, Saiph [21] for computational fluid dynamics, Indigo [22] for image reconstruction, and Halide [23] for image processing have demonstrated that DSLs not only produce terse, extensible, and composable programs but also achieve state-of-the-art performance across different hardware. This is due to a representation where the choices for *how* to execute a program are separated from the definition of *what* to compute. This distinction shows in part the separation between the computation from the problem specification, which gives the compiler flexibility to do the computation in the most efficient manner. Portal, a DSL for yet another uncharted domain of N -body problems is inspired by the same philosophy.

TABLE V: Comparison of the Portal performance (in seconds) against state-of-the-art (S-O-A) libraries/packages. S-O-A for 2-point correlation (2-PC), naive Bayes classifier (NBC), and Barnes-Hut (BH) are scikit-learn [11], MLPACK [12], and FDPS [13] respectively. Note that Barnes-Hut is limited to 3-dimensional data, hence there is a dash (-) for higher dimensional datasets. The elliptical dataset is a 3-dimensional dataset specifically chosen for Barnes-Hut.

	Census			Yahoo!			IHEPC			HIGGS			KDD			Elliptical		
	S-O-A	Portal	speedup	S-O-A	Portal	speedup	S-O-A	Portal	speedup	S-O-A	Portal	speedup	S-O-A	Portal	speedup	S-O-A	Portal	speedup
2-PC	3529	53	66	37043	250	148	4281	26	162	17823	151	117	5134	31	165	5412	94	57
NBC	1337	87	15	3629	198	18	1699	88	19	5231	261	20	981	47	21	1026	194	5
BH	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	473	278	1.7

DSLs can be stand-alone or embedded. For instance, OptiML [24], DeepDSL [19], and Saiph [21] are embedded in host language, Scala, while SCOPE [25] is a stand-alone DSL. Embedded DSLs inherit the language constructs of their host language and add domain specific primitives that allow programmers to express their problem at a higher level of abstraction. It also enables ease-of-adoption which are the reasons for our choice of an embedded DSL.

N-body algorithms. N -body algorithms in physics are some of the most well-studied parallel computing problems. The most popular and widely used fast algorithms for classical N -body problems are Barnes Hut [14] and FMM [7]. They use trees to approximate distance computations and achieve sub $\mathcal{O}(N^2)$ asymptotic time.

While parallel N -body algorithms in physics have received significant attention, the same is not true for other domains such as machine learning (ML). There are a number of ML libraries freely available, unfortunately, each of them lacks in one or both of the two ways, (a) efficient optimal algorithms and (b) parallelism and scalability on modern machines. For instance, MLPACK [12] which is a state-of-the-art C++ ML library offers fast algorithms but is not parallel or distributed making it infeasible to scale to large datasets. Other popular libraries emphasize ease of use but scale poorly such as Weka toolkit [26] and SHOGUN toolbox [27]. Even others implement fast algorithms but in languages such as Python resulting in poor performance such as scikit learn [11] and mply [28].

To our knowledge, none of the above libraries or currently available frameworks implement parallel ML algorithms. The main exception is OptiML [24], which is a DSL for parallel ML written in Scala which generates CUDA code for heterogeneous platforms. First, we are not parallelizing the brute-force algorithms for these computations, but instead, are parallelizing fast tree-based algorithms. Second, we are targeting the specific domain of N -body problems, not entire ML which allows us to aggressively apply domain-specific optimizations which are critical to bridging the performance gap between hand-tuned and traditional compiler generated code. As a result, our approach attempts to fill this huge void in the domain of N -body problems and also combines all the advantages described above that are inherent to DSLs.

PASCAL N -body algorithmic framework. PASCAL [10] is a parallel tree-based algorithmic framework which includes a library of hand-tuned and hand-optimized N -body problems. In PASCAL, the user writes the code for the functions in the tree-traversal (Algorithm 1) for new problems in addition to implementing domain-specific optimizations. In contrast, Portal is a DSL embedded in C++. Portal generalizes PASCAL since it generates the code for these functions from a high-level representation (Portal language) and the backend applies N -body specific optimizations and transformations to generate x86 code.

PASCAL has been compared with four state-of-the-art software/libraries (written in different languages such as C++, Java, and Python) including MLPACK, Matlab, Weka, and Scikit-learn [10]. PASCAL shows significantly better performance in comparison to all of these libraries/software. PASCAL performance comes from a combination of algorithmic, numerical, and domain-specific optimizations as well as parallelization. Since the former represents the current state-of-the-art for these class of problems, we choose it as the expert baseline for comparison with Portal. Portal inherits the algorithmic abstractions from PASCAL and automates the optimizations and transformations for the entire class of N -body problems.

VII. CONCLUSIONS AND FUTURE WORK

Portal is a high-performance domain-specific language and compiler for generalized N -body problems. We show how a DSL with an appropriately high-level mathematical formulation leads directly to both asymptotically fast algorithms and their efficient parallel implementations on x86 platforms. Moreover, Portal enables terse expression of the problem, thereby reducing the lines of code written by experts up to $67\times$ while obtaining performance comparable to expert tuned code. The Portal DSL and intermediate representation are independent of the underlying architecture which makes it easily extensible to different platforms. In future work, we intend to add support for additional backends including GPUs.

We foresee Portal to enable scientific discovery not only for N -body problems in scientific computing and machine learning but in a number of other domains such as computer graphics, computational geometry, and applied mathematics that can

be expressed in Portal to obtain an out-of-the-box optimized parallel implementation.

VIII. APPENDIX

Portal grammar is presented in code snippet 4. The `<name>` in the grammar is the same as variable names in the C++ language. The `<call>` function in the grammar is for pre-defined functions in Portal such as Mahalanobis, Cholesky, etc. as well as user-defined function calls (similar to C++ function calls). Note `?` represents an optional operator, meaning that the symbol on its left can appear zero or one time. Similarly, the symbol on the left of `*` could repeat zero or more times, and the symbol on the left of `+` could repeat one or more times.

Portal code 4: Grammar specification for Portal.

```

<PortalProgram> → <StorageDef>+ <VarDef>+
                                     <PortalExprDef>
<StorageDef> →
    "Storage" <name> ("<file_name>") |
    "Storage" <name> ("<std_vector>")
<VarDef> → "Var" <name>
<PortalExprDef> → <ExprDef> <AddLayer>+
                                     <Execute>
<ExprDef> → "PortalExpr" <name>
<AddLayer> → <name> ".addlayer ("<OP>", "<name>",
    <name>", "<Kernel?>")"
<Kernel> → sqrt (Kernel) | pow (Kernel) |
    <expression> | <call> (a*) | ...
<Execute> → <name> ".execute ()"
<OP> → "FORALL" | "SUM" | "PROD" | "ARGMIN" |
    "ARGMAX" | "MIN" | "MAX" | "UNION" |
    "UNIONARG" | "KARGMIN" | "KARGMAX" |
    "KMIN" | "KMAX"

```

IX. ACKNOWLEDGMENTS

This work was supported by the U.S. National Science Foundation under award number 1533917. We would like to thank UCI HPC computing facility for providing access to the AMD EPYC processor and their technical staff for assistance.

REFERENCES

- [1] K. Scott, "On Proebsting's law," Technical Report CS-2001-12, Department of Computer Science, University of Virginia, Tech. Rep., 2001.
- [2] C. Ozgur, T. Colliau, G. Rogers, Z. Hughes, B. Myer-Tyson et al., "Matlab vs. python vs. r," *Journal of Data Science*, vol. 15, no. 3, pp. 355–372, 2017.
- [3] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive Fast Multipole Method on heterogeneous architectures," *Communications of the ACM (CACM)*, vol. 55, no. 5, pp. 101–109, May 2012.
- [4] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc, "Optimizing and tuning the Fast Multipole Method for state-of-the-art multicore architectures," in *Proc. IEEE Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, Atlanta, GA, USA, April 2010.
- [5] A. Chandramowlishwaran, K. Madduri, and R. Vuduc, "Diagnosis, tuning, and redesign for multicore performance: A case study of the Fast Multipole Method," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, New Orleans, LA, USA, November 2010.
- [6] A. Gray and A. W. Moore, "N-body problems in statistical learning," in *In Advances in NIPS*. MIT Press, 2000.
- [7] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *Journal of Computational Physics*, vol. 73, pp. 325–348, 1987.
- [8] J. K. Salmon and M. S. Warren, "Fast parallel tree codes for gravitational and fluid dynamical N-body problems," *International Journal of High Performance Computing Applications*, vol. 8, no. 2, pp. 129–142, 1994.
- [9] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," EECS Department, University of California, Berkeley, Technical Report UCB/EECS-2006-183, December 2006.
- [10] L. A. Beni and A. Chandramowlishwaran, "PASCAL: A Parallel Algorithmic SCALable Framework for N-body Problems," in *European Conference on Parallel Processing*. Springer, 2017, pp. 482–496.
- [11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg et al., "Scikit-learn: Machine learning in Python," *The Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
- [12] R. R. Curtin, J. R. Cline, N. P. Slagle, W. B. March, P. Ram, N. A. Mehta, and A. G. Gray, "MLPACK: A scalable C++ machine learning library," *Journal of Machine Learning Research*, vol. 14, pp. 801–805, 2013.
- [13] M. Iwasawa, A. Tanikawa, N. Hosono, K. Nitadori, T. Muranushi, and J. Makino, "Implementation and performance of FDPS: a framework for developing parallel particle simulation codes," *Publications of the Astronomical Society of Japan*, vol. 68, no. 4, 2016.
- [14] J. Barnes and P. Hut, "A hierarchical $\mathcal{O}(n \log n)$ force-calculation algorithm," *Nature*, vol. 324, no. 446-449, December 1986.
- [15] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Comm. ACM (CACM)*, vol. 18, no. 9, pp. 509–517, September 1975.
- [16] A. G. Gray, "Bringing tractability to generalized N-body problems in statistical and scientific computation," Ph.D. dissertation, PhD thesis, Carnegie Mellon University, 2003.
- [17] C. Lomont, "Fast inverse square root," *Technical Report*, vol. 32, 2003.
- [18] K. Bache and M. Lichman, "UCI machine learning repository," 2013.
- [19] T. Zhao and X. Huang, "Design and implementation of DeepDSL: A DSL for deep learning," *Computer Languages, Systems & Structures*, vol. 54, pp. 39–70, 2018.
- [20] V. Elango, N. Rubin, M. Ravishankar, H. Sandanagobalane, and V. Grover, "Diesel: DSL for linear algebra and neural net computations on GPUs," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM, 2018, pp. 42–51.
- [21] S. Macià, S. Mateo, P. J. Martínez-Ferrer, V. Beltran, D. Mira, and E. Ayguadé, "Saiph: Towards a DSL for High-Performance Computational Fluid Dynamics," in *Proceedings of the Real World Domain Specific Languages Workshop 2018*. ACM, 2018, p. 6.
- [22] M. Driscoll, B. Brock, F. Ong, J. Tamir, H.-Y. Liu, M. Lustig, A. Fox, and K. Yelick, "Indigo: A Domain-Specific Language for Fast, Portable Image Reconstruction," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 495–504.
- [23] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Seattle, Washington, USA: ACM, 2013, pp. 519–530.
- [24] A. Sujeeth, H. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, and K. Olukotun, "OptiML: an implicitly parallel domain-specific language for machine learning," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 609–616.
- [25] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "SCOPE: easy and efficient parallel processing of massive data sets," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1265–1276, 2008.
- [26] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: An update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, November 2009.
- [27] S. Sonnenburg, G. Rätsch, S. Henschel, C. Widmer, J. Behr, A. Zien, F. d. Bona, A. Binder, C. Gehl, and V. Franc, "The SHOGUN machine learning toolbox," *The Journal of Machine Learning Research*, vol. 11, pp. 1799–1802, 2010.
- [28] D. Albanese, R. Visintainer, S. Merler, S. Riccadonna, G. Jurman, and C. Furlanello, "mipy: Machine learning python," 2012.