Peruse and Profit : Estimating the Accelerability of Loops

Snehasish Kumar
School of Computing Sciences
Simon Fraser University
ska124@sfu.ca

Vijayalakshmi Srinivasan
IBM Research
viji@us.ibm.com

Amirali Sharifian
School of Computing Sciences
Simon Fraser University
amiralis@sfu.ca

Nick Sumner
School of Computing Sciences
Simon Fraser University
wsumner@cs.sfu.ca

Arrvindh Shriraman
School of Computing Sciences
Simon Fraser University
ashriram@cs.sfu.ca

ABSTRACT

There exist a multitude of execution models available today for a developer to target. The choices vary from general purpose processors to fixed-function hardware accelerators with a large number of variations in-between. There is a growing demand to assess the potential benefits of porting or rewriting an application to a target architecture in order to fully exploit the benefits of performance and/or energy efficiency offered by such targets. However, as a first step of this process, it is necessary to determine whether the application has characteristics suitable for acceleration.

In this paper, we present Peruse, a tool to characterize the features of loops in an application and to help the programmer understand the amenability of loops for acceleration. We consider a diverse set of features ranging from loop characteristics (e.g., loop exit points) and operation mixes (e.g., control vs data operations) to wider code region characteristics (e.g., idempotency, vectorizability). Peruse is language, architecture, and input independent and uses the intermediate representation of compilers to do the characterization. Using static analyses makes Peruse scalable and enables analysis of large applications to identify and extract interesting loops suitable for acceleration. We show analysis results for unmodified applications from the SPEC CPU benchmark suite, Polybench, and HPC workloads.

For an end-user it is more desirable to get an estimate of the potential speedup due to acceleration. We use the workload characterization results of Peruse as features and develop a machine-learning based model to predict the potential speedup of a loop when off-loaded to a fixed function hardware accelerator. We use the model to predict the speedup of loops selected by Peruse and achieve an accuracy of 79%.

Keywords

Accelerator, static analysis, machine learning

ICS '16, June 01-03, 2016, Istanbul, Turkey

DOI: http://dx.doi.org/10.1145/2925426.2926269

1. INTRODUCTION

Energy efficiency has become a key design constraint for computer systems. The continued shrinking of the gains from Dennard scaling negatively impacts energy efficiency and transistor speed. At the same time, the continued increase in transistor density allows systems to have more transistors, even though only a few can be powered at any given time, leading to the widely recognized phenomenon of "dark silicon" [42]. Although application specific integrated circuits (ASICs) clearly offer significant improvement in energy efficiency relative to general-purpose processors, it is not clear how applications in general can leverage fixed-function hardware [26]. Programmable hardware accelerators [23] provide more flexibility relative to ASICs and at the same time offer better energy efficiency relative to general-purpose processors. However, it is unclear what code regions in an application may profit from using these programmable hardware accelerators. Hardware accelerators, irrespective of the type, improve energy efficiency and performance by customizing the data and control circuitry for particular program regions. Accelerators arguably present a hard challenge for programmers due to their application or domain specific nature. The effectiveness of an accelerator for an application depends on the extent to which the application exhibits the characteristics that map efficiently to the accelerator. For instance, data-dependent branches constitute a challenge particularly for GPU execution models. In this paper, we characterize code regions in workloads with a primary focus on assessing programmer effort and hardware effort required to exploit accelerators. Akin to the question "What is parallelizable?" that gained widespread attention with the advent of multiprocessors, we set out to answer the question "What is acceleratable?". The question is particularly pertinent today when custom hardware seem to point the way forward for obtaining performance in our energy constrained future. Instead of focusing on any particular domain-specific hardware accelerator, we chose to characterize workloads independent of the accelerator architecture but still provide information to assess the suitability of the workloads for acceleration on heterogeneous system architectures. Heterogeneous architectures may possibly include multiple types of accelerators, including fixed-function accelerators, domain-specific programmable accelerators (e.g. GP-GPUs), and traditional multicore processors.

A key issue with program characterization is what code granularity the static analysis should focus on, i.e. function

boundaries, loops, or basic blocks. We choose to focus on loops as they typically dominate execution times and their iterative nature are naturally suited for accelerators [27]. However, not all loops are suitable for acceleration. For example, loops with loop-carried dependencies may limit performance improvements from specialized data-parallel hardware accelerators relative to a general-purpose hardware.In this paper, we identify program metrics that play an important role in the acceleratability of loops and systematically characterize and analyze application's loops.

Typically, workload characterization for a given architecture tends to be dynamic in nature. These dynamic approaches can use either hardware performance counter measurements on current multicores [20, 16, 49] or use simulation-based analysis. Such dynamic characterization approaches are dependent on the target ISA and the inputs, as also pointed out in prior work[57, 29], and may not expose the features of the workload.

We propose Peruse, an LLVM-based static program analysis framework, to characterize key accelerator-oriented features within the loop nests of applications. Peruse presents a mechanical approach to identify the suitability of the workload for acceleration based on their characteristics that we have identified and analyzed. To the best of our knowledge, Peruse is the first of such suitability assistants for accelerators, but it builds upon and complements a long line of research related to workload characterization and parallelization.

Researchers at IBM have used Peruse to find and isolate loops of interest before further analyses was applied. For example, for the Discrete Wavelet Transform benchmarks from the PERFECT suite, Peruse was used as a filter to discard loops not suitable for acceleration. Candidate loops were then profiled using Aladdin [24] for ASICs as well as running on GPUs to determine a suitable performance/watt target. Using Peruse to filter the loops not suitable enabled the dynamic analyses to target only the relevant (accelerator-friendly) loops.

Our contributions include: **(i)** Peruse : A program analysis tool to characterize loops within real-world applications. Leveraging LLVM's internal representation (IR), we are able to avoid introducing bias from analyzing only a single, specific target architecture. Using *Peruse* we are able to characterize individual loops to identify the nature of compute operations, data footprint, side-effects, branch divergence, and nesting depth among other characteristics that help to not just identify but also to understand the programmer and hardware effort required for acceleration. Building on prior efforts in auto-parallelization, we extract memory-dependence characteristics and postulate that they are also indicative of performance gains obtainable from fixed function hardware or other execution models. Peruse also discovers code characteristics such as idempotence [14] that are indicative of programmer effort required to port the code region to fixed-function hardware. **(ii)** An end-to-end framework using machine-learning models to predict the potential speedup of a loop when off-loaded to a fixed function hardware accelerator.

2. BACKGROUND

The design of efficient compute platforms, whether efficient in terms of execution time, power, memory, or some other contested resource, requires direct interaction between programmers and architects. Programmers need help to focus on specific program regions that can profit from running on specialized hardware and architects need help in designing the hardware accelerators that can improve the efficiency of applications. To achieve both these objectives, we need to understand the characteristics of the workloads themselves. By formalizing the properties of programs that limit efficient design, developers and architects are together able to boost both the efficiency of their applications and the productivity of the architects who design them. In this section, we discuss the context of our work, trying to answer the question of "what is acceleratable?", with respect to "what is vectorizable?" and "what is parallelizable?".

2.1 What is vectorizable?

An early instance of domain specific computation is vector processors that perform the same operation on arrays or vectors of data all at once. Compiler developers have developed auto-vectorization [54] techniques to enable applications to take advantage of vectors operations and answer the question: "is the code region vectorizable?". To answer this question, compilers look at the data dependence graphs of programs, which show where values are created and used within a program [37]. In the common case of vectorizing loops, a compiler will try to perform multiple iterations of a loop simultaneously using vector operations. For a loop to then be vectorizable, every iteration of the loop must perform the same operations or be transformable to such a loop. Furthermore, the values used by each iteration cannot be defined or overwritten by other iterations. Otherwise, vectorizing the loops could yield incorrect results. In practice, even nested loops and function calls prove to be challenging, although progress has been made against such limitations. Compilers examine the data dependence graphs of programs to recognize these patterns and identify loops that are amenable to vectorization.

2.2 What is parallelizable?

Seeded by the demand for high performance computing (HPC) in scientific and engineering domains, parallel computing has long proven itself as a means of exploiting multiple machines, processors, or cores to speed up applications. Indeed, with the plateau of CPU clock speed improvements, there is a growing need for concurrent and parallel solutions in non-HPC applications. In either case, this fueled the need for developers and compilers to analyze code in order to exploit parallelism. This required understanding what makes code *parallelizable*.

Similar to the question of vectorizability, parallelizability relies heavily on recognizing the data dependencies within a program and within loops in particular. For instance, if a later iteration of a loop uses or updates a value computed in an earlier iteration, then executing those iterations in parallel could lead to a race condition that produces non-deterministic results. The presence or patterns of such conflicting memory accesses crucially determine whether and/or how a loop may be parallelizable. Thus, techniques like dependence testing have been developed to identify such conflicting access patterns [22].

However, the restrictions imposed upon vectorizable code do not all apply to parallelization. This is in part due to the lock-step nature of vector operations, while general-purpose parallelism focuses on task-oriented parallelism. While ev-

ery iteration of a loop must perform the same operations to be vectorizable, a parallelizable loop may exhibit widely varying behavior in each iteration as long as no conflicting dependencies are introduced. Recognizing this relaxation of the vectorizability constraints allows developers to still reap the benefits of parallelization when vectorization is difficult.

2.3 What is acceleratable?

The energy efficiency and performance benefits from hardware accelerators now pose a new question to both software developers and compilers alike: "What is acceleratable?" Previously, the hardware execution model has been fixed, but it is now possible to customize the model for an application using techniques like fixed function accelerators. Thus, this final question is arguably broader, encompassing both existing and hypothetical execution models, yet it is commonly asked today for many workloads as general purpose multiprocessor chips hit the power wall. This question is easier to address when coupled with a known execution model, and the traditional approach has been to port an application to answer the question. While this provides a concrete answer for that particular execution model, it is dependent on the expertise of the developer, and it poses a large time sink with no guarantee of a profitable end result. Recent work has looked at how to estimate performance, via dynamic characterization, for various execution models such as OpenMP style parallel[18, 44], GPU style[8] as well as for fixed function hardware[24]. On the other hand, there are many proposals[35, 23, 65] for innovative accelerator architectures that promise improved performance and increased energy efficiency. Peruse aims to reconcile the gap between the execution models and workload characterization. It provides a convenient first-hand look into which regions of code are likely to be feasible and profitable for acceleration.

3. PERUSE FRAMEWORK

The primary purpose of Peruse is to aid in understanding opportunities for acceleration in a given application. The design of Peruse integrates LLVM IR analysis infrastructure with reporting techniques to study a wide variety of applications. Determining whether a code region is amenable for acceleration, requires us to consider the important characteristics of particular architectures. For instance, in CPU-based execution models (independent of Out-of-Order vs In-order) a key source of energy inefficiency is instruction fetch [26], and hence to determine the profitability of accelerating a code region we consider the number of static instructions. Other characteristics influenced by the execution model include features such as branches, which directly impact throughput accelerators such as GPUs [55], and data footprint, which impacts fixed-function hardware models that employ local stores. We define and discuss the features in Sections 3.6 and 3.7. The key goal of Peruse is to extract the key characteristics of applications that lend themselves to acceleration and to use them to automatically filter out the loops clearly not suited for acceleration.

Peruse is implemented as an LLVM IR analysis pass to obtain characteristics of loop nests, individual loops, and functions called within the loops. This analysis is source language independent, as it operates upon the LLVM Intermediate Representation (IR). LLVM IR for a program can be generated using clang (for C/C++/ObjC). Peruse builds upon extensive work done by the compiler community for code optimization and exposes the key features from the code analysis passes to help the programmer understand the benefits and limits of porting the code region to accelerators. It also incorporates components from earlier work on idempotent code generation[14] and auto-parallelizing compilers[1] to aid in analysis of wider code characteristics. The rest of this section presents the key design decisions in Peruse.

3.1 Static vs Dynamic Characterization

A key aspect of workload characterization is whether the characterization is *static*, i.e. examining only the structure and semantics of the application, or *dynamic*, i.e. examining the behavior of an application on a particular input. Typically, workload characterization for a given architecture tends to be dynamic in nature. These dynamic approaches can be further classified into native and simulation-based analysis. Native dynamic characterization observes the application's behavior on a given architecture while monitoring events exposed via a predefined interface, e.g. hardware performance counters on general purpose processors. Simulated dynamic characterization offers the flexibility of observing any desired metric and changing the architecture representation as required, but it is slow relative to native execution and limited by the model's accuracy. More important, both dynamic approaches are dependent on the target ISA as also pointed out in prior work[57], The features of the workload itself may not be exposed due to the constraints imposed by the ISA. Finally, the robustness of the dynamic characterization is also input dependent. Input dependent characterization could impact important features such as branch divergence as well as observed memory dependencies.

With Peruse, we choose to characterize the workload via static program analysis. While earlier static analysis has primarily focused on compiler transformations[61, 51] and bug checking[5, 50], Peruse employs static analysis to characterize loops to determine whether they may be suitable for a particular architecture. While the dynamic analysis based approaches may precisely characterize how a program behaves under a particular input, they only under approximate the overall program behavior. Thus, the results may even unsafely contradict characterizations produced via some alternative input. In order to produce a characterization that safely represents all possible inputs static techniques are required.

3.2 IR vs Source

In order to achieve language independence, we implemented Peruse on top of the LLVM IR. Operating on top of the IR has several advantages over operating on source code or even abstract syntax trees. Most notably, analyzing the IR frees the implementation from depending on any one language. As long as the language can be compiled to LLVM IR, Peruse can produce a characterization for it. Given that this includes any language compiled by clang, 403.gcc, or third party front ends, the flexibility is substantial. This well established and canonical IR also simplifies analysis.

3.3 Loops vs full program analysis

Prior work examines acceleration opportunities at the function level[17, 34] and at the level of loops within a program [56]. In order to capture the most salient and relevant features, Peruse focuses on loops because they represent fre-

quently executing paths that dominate the runtime of an application. We study the runtime behaviour to justify our choice of only analysing loop nests for accelerability. To study the workloads, we build an LLVM based tool which performs efficient path profiling[7]. We pick 12 workloads[1] from our set to analyse and show a) path bias exists and b) loop nest contained (i.e "loopy") paths are frequent.

The workloads are executed with a varied set of input provided by their respective suites, and their path profiles (i.e the frequency of execution of each static path) are collected. The workloads we study show limited input sensitivity with respect to path profiles. This is an important note since the profiles generated by traditional profilers like *gprof*, do change based on the inputs provided. Additionally, results from sampling based profilers may also be perturbed by micro-architectural effects. From our analysis of paths in each workload we find that loopy paths are important in all workloads. Paths are ranked using the frequency of the path being executed multiplied by the number of instructions in the path ($P_{wt} = P_{freq} \times |P_{ins}|$). We summarize our observations in Table 1. Column $\sum P_{wt}$**%** shows the percentage weight of the top five highest weighted paths. In all but 1 of 12 applications, the weight is greater than 30%. For this application, we enumerated lower ranked paths until we reached a coverage of at least 40% and observed that **all** these paths are loop nest contained paths.

All workloads apart from *453.povray* identify loopy paths as the highest weighted path for the selected function. For *453.povray*, the object methods are stored as function pointers and the appropriate function invoked at runtime. The main loop iterates over objects in the scene, thus the parent function is invoked in a loop.

Overall, we find on average 25% of the work is performed by a single path in a loop. Furthermore, we find that 11 of the 12 application paths are part of a loop nest. Finally, in the workloads where the paths are not loop nest contained, we find them to be invoked in a loop.

Workload Name	Function Name	$\sum P_{wt}$ %	Loopy?
401.bzip2	handle_compress	35	Yes
464.h264.refref	dct_luma_16x16	55	Yes
470.lbm	LBM_performStreamCollide	100	Yes
444.namd	calc_pair_energy_fullelect	78	Yes
482.sphinx3	vector_gautbl_eval_logs3	100	Yes
429.mcf	price_out_impl	77	Yes
450.soplex	vSolveUrightNoNZ	33	Yes
458.sjeng	gen	24	Yes
403.gcc	bitmap_operation	57	Yes
456.hmmer	P7Viterbi	100	Yes
453.povray	All_Sphere_Intersections	89	No

Table 1: Path bias of loop nest contained paths

3.4 Feature Selection

In order to address whether or not an existing CPU implementation of an algorithm is acceleratable, we consciously ensured that Peruse does not require any extra code annotation or information about how the program could be rewritten. Indeed, the burden of porting the program without knowing whether the program will run efficiently on an accelerator is precisely why a tool like Peruse can be useful for programmers to calibrate their initial expectation. Having guidance on the acceleratability in advance can better streamline efforts to rewrite applications for a particular target. Based on this intuition, Peruse instead examines several general application features that are selected to capture the characteristics of the workload.

While some features may affirm that, for instance, a loop is suitable for acceleration on a particular execution model, others can instead provide evidence that a loop is *unsuitable* for acceleration on a particular model. For example, runtime memory allocation is challenging to implement for accelerator architectures. Such evidence is even more important than recognizing suitability if there are no easy solutions at the algorithmic or implementation level to bypass or eliminate such features. For example, in a stencil computation, loop carried memory dependencies are bound to be present. Such computation cannot be offloaded to an OpenMP style parallelizing execution model. Sections 3.6 and 3.7 present the chosen characteristics in more detail.

3.5 Query based filtering

While synthetic benchmarks such as *Polybench* contain few loops on average, real world applications contain hundreds of loops. Peruse discovered an average of 886 loops in each benchmark of the SPEC2006 benchmark suite (max 4635 – *403.gcc*, min 23 – *470.lbm*). Manually inspecting the results and characteristics extracted for each of these loops in each benchmark is tedious and not scalable. Thus, it is beneficial to also *automate the search* for prime acceleration candidates based on a set of criteria derived for a given execution model. This allows Peruse to shortlist a small number of loops that can be further investigated manually or automatically classified by a machine learning model as described in Section 5.

To retain flexibility and make it easy to consider varied execution models for the same application, Peruse exposes a query interface where the user can specify which features and characteristics should be used to filter and prioritize the reported results. Multiple characteristics may be specified as shown in the pseudocode below. The query to select the top 15 candidate loops for a data parallel, thoughput oriented execution model could be:

```
SELECT * FROM loops WHERE IsInnermost = True
AND Mem-Deps-Count = 0 ORDERBY Loop-Data-Tile
DESCENDING ORDERBY Branch-Ins-Count
ASCENDING LIMIT 15
```

These queries can specify which characteristics must be true or false, how results should be ordered, and how many results should actually be provided to the user.

3.6 General Features

Peruse assigns unique identifiers to loops and loop nests and gathers general characteristics for all loops encountered in the benchmark. The list of features extracted by Peruse is shown in Table 2.

1. Annotated Parallel : Loops may be marked up with `#pragma omp parallel` or `#pragma ivdep` to indicate the absence of loop carried dependencies. This boolean is set, dependent on support from the compiler frontend, when such a directive is present in the source code.

[1]We do not analyse overly simple kernels such as those from PolyBench. We also exclude benchmarks which use language features such as C++ exceptions as they cannot be accelerated easily.

Group	Fields
Loop characteristics	Innermost, Annotated Parallel, Trip Count, Loop Exits and Loop Nest Depth
Instruction characteristics	Static instructions, Branches, Atomics, Intrinsics, Big Operations, Fence , Side Effect and Carried Memory Dependencies
Code characteristics	Memory Allocations, Data footprint, Compute-to-Communicate Ratio, Function Calls, Vectorizable, Idempotent
Data structure characteristics	Data Structure Containers, Accessors and Mutators, Strided Accesses

Table 2: General Workload Characteristics Analyzed by Peruse

2. Atomics : Indicates the presence of atomic instructions such as CMPXCHNG or RMW instructions. Similarly for Intrinsics and Fence Count.
3. Big Operations : Long latency floating point operations.
4. Strided Accesses : Indicates the number of arrays accessed in a strided manner with respect to the loop induction variable.

3.7 Accelerator Specific Features

Peruse also uses some more in-depth analysis to obtain additional features from the loops to determine suitability for acceleration.

3.7.1 Data Footprint and C-to-C ratio

The loop data footprint indicates the total amount of distinct local and global memory referenced, either read or written, in the loop body. This metric is useful to gauge the amount of local storage to provision for while exploring new accelerator architectures. It is also useful to gauge the amount of work being done. For example, a large amount of data being accessed could potentially favor accelerating the loop on a throughput-oriented architecture. The compute to communication (C-to-C) ratio provides a similar intuition. Again this metric could help determine whether a given loop is a good candidate to port to an architecture such as GPU, where the data transfer cost must be amortized by the compute performed.

3.7.2 Function calls and data structures

When Peruse encounters a function call within a loop body it characterizes the function itself provided the source code is available. Peruse collects features of the functions such as the number of arguments, recursive nature, purity, idempotence and general instruction characteristics to augment the previously collected information for the loop body. For scalability, however, Peruse only characterizes function calls one level deep. Functions called transitively by functions called in the loop body are not characterized but are indicated in the characteristics of the original function call. Should these function calls be to standard data structure interfaces such as STL data structures, Peruse can even recognize these calls and classify them as data structure accessors and data structure mutators.

3.7.3 Vectorizability

Based on the integrated LLVM vectorizer, introduced in LLVM 3.3, Peruse tests the loop to determine whether it is possible to vectorize. If not, Peruse reports the limiting factor for vectorization. Auto-vectorization in LLVM is nascent and may not yet be at par with aggressive implementations in compilers such as IBM's XLC, but it gives a reasonable estimate of limiting factors.

3.7.4 Idempotence

Idempotence in computing refers to the ability to freely execute a section of code without side effects. Entry points of these sections act as implicit checkpoints, and thus idempotent operations provide the unique ability to recover from an operation via re-execution rather than checkpointing. This is an important feature and has been frequently discussed in prior work[41, 52, 60]. In this light, there has been significant work in static analysis for idempotent processing and code generation[14]. Peruse integrates open sourced work on idempotent processing[14] to test loop bodies and function bodies for idempotence.

3.7.5 Memory Dependencies

When a store to memory precedes another store or read to the same location, the second access depends upon the first. Informally, this indicates that the accesses cannot be easily reordered without changing the meaning and behavior of the program. For instance, if a memory access in one iteration of a loop depends on an access in an earlier iteration, then those two iterations cannot easily be made to execute concurrently.

We implement hierarchical dependence testing as used by AESOP [1]. Hierarchical dependence testing was described by Burke et al.[9] as a means to efficiently implement existing dependence tests while extending them to interprocedural analyses. The dependencies between array accesses at varying nesting depths in a loop are described using direction vectors (first introduced by Wolfe[64]). A direction vector for a pair of memory accesses is a list of symbols that describes their relation with respect to the induction variable. The symbols commonly used in direction vectors are:

1. `*` : No dependence for any iteration
2. `0` : Dependence within the same iteration
3. `<` : Dependent on a prior iteration
4. `>` : Dependent on a future iteration

4. PERUSE ANALYSIS CASE STUDIES

In this section, we present case studies of selected applications from SPEC2006 (C and C++ benchmarks), Polybench [48], and CORAL[11]. The selected benchmarks are *seidel* (3 loops, from Polybench), *lulesh* (10 loops, from CORAL), *470.lbm* (23 loops, from SPEC2006), *482.sphinx3* (588 loops, from SPEC2006) and *444.namd* (623 loops, from SPEC2006). The benchmarks are listed in ascending order of the number of loops they contain, with *seidel* having the least and *444.namd* having the most. We demonstrate how Peruse is able to extract useful characteristics for these benchmarks to guide us to interesting loops in the code that the programmer should focus on for acceleration.

For the selected benchmarks described below, we choose an illustrative configuration in which Peruse orders the loops by decreasing order of data accessed in the loop body and we only report the top 10 loops. Loops are given unique integer identifiers from 0–N depending on the number of loops discovered in the application by Peruse. We also demonstrate

Listing 1: 2D Stencil compute for Seidel

```
74 for (t = 0; t <= tsteps - 1; t++)
75   for (i = 1; i<= n - 2; i++)
76     for (j = 1; j <= n - 2; j++)
77       A[i][j] = (A[i-1][j-1] +
78          A[i-1][j] + A[i-1][j+1]
79          + A[i][j-1] + A[i][j]
80          + A[i][j+1] + A[i+1][j-1]
81          + A[i+1][j] + A[i+1][j+1])/9.0;
```

Table 3: Memory dependencies in seidel

Source	Types	Direction Vectors
seidel.c:77 → 77	WAR	(*,0,0) (*,0,<) (*,0,>) (*,<,0) (*,<,<) (*,<,>) (*,>,0) (*,>,<) (*,>,>)

the "Query Interface", which helps filter out loops based on a subset of desired characteristics, and show how clustering loops using similar characteristics may be beneficial. Finally, we present observations for the SPEC2006 benchmark suite.

4.1 Seidel

This benchmark is a 2-D Seidel stencil computation written in C. It is a part of the Polybench benchmark suite, which is used to evaluate polyhedral compilers and provides a rigorous test of memory dependency resolution due to the various array indices accessed in each step of the iteration as shown in Listing 1. The other 2 loops present in this microbenchmark are used for initialization and printing the arrays on which they operate.

Peruse identifies the three loops present in the microbenchmark. From the generated response the programmer can immediately determine that the innermost loop pattern; ten array elements in a strided fashion with a total memory footprint, distinct local and global memory accesses, of 80 bytes. The memory dependencies for this benchmark are represented as direction vectors as described previously in Section 3.7.5 and indexed with a line pair tuple. Peruse reports cross-references the dependencies against the corresponding source code line as shown in Table 3, the first column indicates the source filename and the line number pair for the dependencies. The second column indicates the types of dependencies present in between these source line numbers and the third column shows the direction vectors of the individual array accesses across iterations and the direction of their carried dependence. We also note that there are nine long latency instructions, floating point multiply operations, while there are 37 compute instructions and 47 total instructions in the loop body[2]. The HTML report generated by Peruse also specifies the line numbers at which the loops and interesting characteristics within the loops, such as long latency instructions, occur in the source. This enables the user to quickly refer to the source for further clarification. *Seidel* is a benchmark with very low cyclomatic complexity and $\simeq$ 100 lines of code.

4.2 LULESH

This benchmark is derived from the C language version of LULESH 1.0[38] and extracts the `CalcKinematicsForElems` method and its children. It is a benchmark written to stress specific features such as compiler auto-vectorization. It is also carefully structured to exploit spatial locality and caching by using a Structure-Of-Arrays for storing data while iterating over them to perform molecular simulations on an unstructured mesh.

Peruse identifies a loop (L_0), which accesses 204 bytes of data with a high *compute-to-communication ratio* (see Section 3.7.1) of 6.0 as the first candidate based on the query. This loop also calls a function called `CalcElemVolume`. Peruse further characterizes the function called and we find that it is an idempotent function, (see Section 3.7.4), with 154 static instructions. This loop also contains another loop (L_1) with a small static trip count of eight. The loop body of L_1 is found to be idempotent in nature. Peruse identifies for L_0 a lack of memory allocations, which makes it suitable for accelerator architectures.

Compiler Optimizations. Peruse, by default, does not perform any transformations to the source LLVM IR other than those that are absolutely necessary to perform the analysis. However, in the case of LULESH, we see that after turning on optimizations enabled by `-O1` in LLVM, the loop L_1 is fully unrolled. This results in an increase in the data accessed by L_0 from 204 bytes to 620 bytes. With optimization level `-O2` the function `CalcElemVolume` is also inlined into the body of loop L_0. Given the variability of observed metrics with optimization level, Peruse offers the flexibility of running arbitrary optimization transformation passes prior to the analysis and it can be used by the developer to find opportunities as shown in the case of LULESH.

4.3 470.lbm

This is a SPEC2006 benchmark that implements the Lattice-Boltzmann Method (470.lbm) to simulate incompressible fluids in 3D. The benchmark contains $\simeq$ 1100 lines of code and is compiled to LLVM IR using clang. The query identifies the top 10 loops in descending order of the amount of data being accessed in the loop body.

The largest amount of data is found to be accessed in L_{10} at 1008 bytes, shown in Table 4. The loop ranked second (L_{11}), accesses 936 bytes of data. To evaluate the suitability of L_{10} vs L_{11} we compare and contrast some of the characteristics reported by Peruse (shown in Table 4). Loop L_{11} has characteristics favorable for acceleration such as the absence of non-loop branches and an idempotent loop body. However, Peruse reports the presence of loop carried dependencies (total 61) and a trip count of 10,000.

For L_{10}, Peruse reports the presence of non-loop control flow branches (see Table4). One of the branches diverges to successors of 208 and 113 instructions. Another branch guards successors of size 320 and 4 instructions. Since Peruse also indicates that this loop only has 1 exit, we know that this branch is not used to evaluate a break condition. Based on the line number indicated by Peruse (`470.lbm.c:241`) we inspect the code to find that it resets variables `ux,uy` and `uz` to default values based on a condition. While presenting multiple execution paths, this does not indicate the presence of the classical problem of branch divergence for GPUs due to the short divergent path length. Peruse indicates the presence of 12 loop carried memory dependencies. With L_{10} having 1300× the iteration count of L_{11} as well as a lower number of loop carried memory dependencies, L_{10} is selected as a better candidate loop for acceleration.

[2]All instructions are LLVM IR instructions.

Listing 2: Data bound loop in 482.sphinx3

```
465 while (ww < wEnd)
466 {
467       /* wwf2 = ww*f2 */
468       wwf2.r =
              f2->r*ww->r - f2->i*ww->i;
469       wwf2.i =
              f2->r*ww->i + f2->i*ww->r;
470       /* t1 = f1+wwf2 */
471       t1->r = f1->r + wwf2.r;
472       t1->i = f1->i + wwf2.i;
473       /* t2 = f1-wwf2 */
474       t2->r = f1->r - wwf2.r;
475       t2->i = f1->i - wwf2.i;
476       /* increment    */
477       f1 += 2*k; f2 += 2*k;
478       t1 += k; t2 += k;
479       ww += k;
480  }
```

In order to further investigate the suitability of the aforementioned loops, we inspected the source at the indicated locations (`470.lbm.c:186` for L_{10} and `470.lbm.c:353` for L_{11}). Both candidate loops were found to be marked for OpenMP execution after certain variables were marked as private. The pragma for L_{10} indicates that 2 variables are marked as private whereas it was 15 for L_{11}. The `#omp pragma` was wrapped in an `#ifdef` block that was not defined while it was compiled into LLVM IR. Prior work[27] also indicates that the prime candidate loop for offload is L_{10}. This demonstrates Peruse's ability to independently highlight acceleratable loops.

Loop	Line	Data (bytes)	Instructions	Memory Deps.	Long Latency Ops
L_{10}	186	1008	647	12	251
L_{11}	353	936	605	61	248
		Branches		Trip Count	Idempotent
L_{10}		4		1300000	No
L_{11}		0		10000	Yes

Table 4: L_{10} and L_{11} from 470.lbm

4.4 482.sphinx3

The *482.sphinx3* benchmark is a C benchmark based on a widely recognized speech recognition engine by Carnegie Mellon University consisting of ≃23K lines of code. The most interesting loop highlighted by Peruse has a compute-to-communicate ratio less than 0.25 making it unsuitable for accelerator models with high offload communication cost. Static analyses also shows that there are no statically allocated arrays, and there are significant floating point operations. This is corroborated in the Listing 2 shown. Furthermore, the source listing shows that the memory accesses are still sequential because the data is heap allocated as an array of structures. Based on these observations, the loop may be amenable to OpenMP style data partitioning or may benefit from a Processing-In-Memory acceleration architectures along the lines of VIRAM[36].

4.5 444.namd

The *444.namd* benchmark is a C++ benchmark from the SPEC benchmark suite and is a parallel program for the simulation of large biomolecular systems. It is a fairly complex benchmark containing ≃5300 lines of code with 623 loops.

Listing 3: Loop instantation in 444.namd

```
#define NORMAL(X)
#define EXCLUDED(X)
#define MODIFIED(X) X
#include "ComputeNonbondedBase2.h"
#undef NORMAL
#undef EXCLUDED
#undef MODIFIED
```

The SPEC website indicates the inner loop code is present in `ComputeNonbondedUtil.C` but we find that the loop itself is present in `ComputeNonbondedBase2.h`. The code for the loop is instantiated using snippets similar to Listing 3 in multiple places in the source. While structuring the source in this manner may have the benefit of factoring out similar code without runtime overhead, it leads to complex, unmaintainable code that is hard to analyze. It is challenging to reason about the generated code while manually looking for opportunities for acceleration. Peruse does not face these challenges while analyzing the generated LLVM IR.

Peruse, identifies the top ten loops as variants of the loop defined in `ComputeNonbondedBase2.h`. The loops access between 404–444 bytes of data with 277–317 instructions in the loop body. Based on the presence of multiple loops with similar characteristics discovered by Peruse, the developer can synthesize specialized hardware as required to simultaneously target all the candidate loops.

Other large benchmarks. `403.gcc` and `400.perlbench` are also challenging to analyze. These benchmarks employ more advanced forms of preprocessor expansion as described for `444.namd`. This results in a large amount of preprocessor generated code that is difficult for humans to analyze. For example, the average number of arguments for the functions profiled by Peruse in `403.gcc` was 1746 and `400.perlbench` was 457.

4.6 SPECCPU2006 Results

We now summarize our analysis of the benchmarks from the SPEC2006 suite, starting with an overall categorization shown in Table 5 based on the total number of loops present in each of these applications. For this analysis, we omitted the applications written in Fortran, and analyzed only the 19 benchmarks written in C/C++. Among the 7 benchmarks with 1000 or more loops, `403.gcc` topped the list with 4635 loops. Among the 5 benchmarks with 0-250 loops, `470.lbm` had only 23 loops in all. Peruse was able to identify all the loops successfully, and these results match the data presented in [46]. Overall, from Peruse's characterization of the loops we observe the following:

1. Large variation in the number of loops per application (min 23 – `470.lbm`, max 4635 – `403.gcc`)
2. Standard STL data structures are accessed in 7 out of 19 benchmarks
3. 13 of 19 applications have more than 75% of loops as innermost loops

# of Loops	Benchmarks
1000+	403.gcc, 483.xalancbmk, 464.h264.refref, 400.perlbench, 453.povray, 445.gobmk, 447.dealII
251-1000	456.hmmer, 450.soplex, 444.namd, 482.sphinx3, 471.omnetpp, 433.milc, 458.sjeng
0-250	401.bzip2, 473.astar, 462.libquantum, 429.mcf, 470.lbm

Table 5: Number of loops in SPEC2006

Listing 4: Peruse filter query

```
{
  'be-true'   : ['innermost'],
  'be-false'  : [ ],
  'where'     : [{'loop-exit' : 1 },
                 {'mem-alloc' : 0 }],
  'order-by' :
                [{'loop-data-tile': true},
                 {'memory-deps'   : false}],
  'limit' : 10
}
```

Query Interface. For large workloads like those in SPEC-CPU2006, it is challenging to examine the characteristics of each individual loop to decide what to accelerate. To alleviate this problem, Peruse provides an interface, dubbed the "Query Interface," to perform user-guided filtering on the loops extracted from the source. Peruse was configured to filter and order loops according to the following criteria:

1. Be the innermost loop
2. Have only one loop exit
3. Does not have any memory allocations
4. Order by the amount of data accessed in an iteration
5. Order by the number of loop carried memory dependencies

This query represents some of the basic desirable characteristics for a loop to be accelerated on a throughput oriented architecture or an OpenMP like parallelization model. Further filtering can be done based on the presence and number of loop carried memory dependencies. The query was specified in the Peruse configuration file in the form of a JSON object. The query is shown in Listing 4. The `be-true` and `be-false` conditions check for characteristics that are True and False respectively. Any number of characteristics can be specified in the array. The `where` array includes objects that define conditions to be checked. The `order-by` array stores the characteristics by which the loops that pass the prior filters are ordered. The boolean value True indicates a descending order and vice-versa. While the absence of loop carried memory dependencies are a strict requirement for all data parallel execution models, due to the conservative nature of dependency analysis which may lead to false positives, we use it as an ordering constraint.

As discussed previously in Peruse's characterization of `470.lbm` in Section 4.3 IV-C, L_{10} is found to be a suitable candidate. In addition to being the loop with the largest amount of data accessed, L_{10} also satisfies two other criteria imposed by the query, namely, a) being the innermost loop and b) having only one loop exit. On examining the source for `470.lbm`, it was found that the developers provided `#omp parallel` pragmas for the loop identified by Peruse. The previously observed loop carried dependencies are either false positives or resolved by variable privatization. This observation serves to validate Peruse's ability to find appropriate loops for a given user specified criteria.

For `471.omnetpp`, the loop with the largest amount of data accessed is found at `cpar.cc:1032`, consuming 518 bytes of data per iteration. However, this loop does not meet the criteria of having a single loop exit; it has 17. The loop that does meet the criteria, in descending order of data being accessed as well as satisfying both conditions, is found at `cpsquare.cc:225`. This loop is found to have 260 bytes consumed by 137 compute instructions.

Acceleratable Loops. To assess the suitability of the loops for acceleration, we used the query interface of Peruse to extract loops with a chosen set of characteristics. These sample characteristics shown in Listing 4 are typically suited for accelerators that do not share the address space with the host, and the data is shipped to the accelerator from the host. Table 6 shows the categorization of the loops within the applications based on the query.

In Table 6 we show the topmost loop for each application that matched the query criteria. Based on the detailed characteristics presented by Peruse for each of the loops, we observe that only 8 out of the 19 loops are amenable for acceleration based on this query. Among the loops amenable for acceleration, `445.gobmk, 462.libquantum`, and `444.namd` have data dependent branches and a large number of floating point operations. Thus, they may benefit from using the *OpenMP* model. `433.milc, 401.bzip2`, and `482.sphinx3` do not have data dependent branches and may benefit from SIMD-style acceleration. It is interesting to note that Holewinski et al.[27] use dynamic trace analysis to demonstrate the potential for auto vectorization and make a similar observation. Similarly, our assessment of suitability for acceleration of the loops in `470.lbm` and `464.h264.refref` based on Peruse matches the observations in prior work[27] as well.

Among the loops not suited for acceleration based on this query, `400.perlbench, 403.gcc, 453.povray, 456.hmmer`, and `458.sjeng` are impacted by control flow dependence, and `456.hmmer` and `403.gcc` are also impacted by loop carried memory dependencies. Similarly, for `450.soplex`, the top candidate loop extracted by Peruse contains a large number of branches, and in general the loops in the program are small. Prior work also corroborates that without restructuring the `450.soplex` algorithm it is not acceleratable [10]. As demonstrated by this analysis, the query interface of Peruse can be used to further refine and filter the selected loops to match the set of desired characteristics based on the nature of the hardware accelerator.

Offload Profitability. As explained in Section 3.7.1, one of the key features to estimate the cost of offloading a loop to an accelerator is the compute to communication (C-to-C) ratio. Subsequent to the qualitative analysis done to select the loops shown Table 6, we used Peruse to statically characterize the C-to-C ratio for these selected loops. Dependent on the target execution model's memory system interface, a lower or a higher C-to-C ratio may be preferred. For example, for a constrained memory system interface such as that

Workload	Filename	Line	Comments
401.bzip2	blocksort.c	854	
445.gobmk	engine/reading.c	4171	
464.h264.refref	mbuffer.c	3475	
470.lbm	lbm.c	186	Candidate loop matches the query provided and has characteristics amenable for acceleration.
462.libquantum	qec.c	241	
433.milc	m_mat_an.c	39	
444.namd	ComputeNonbondedBase2.h	12	
482.sphinx3	new_fe_sp.c	465	
473.astar	Library.cpp	427	
447.dealII	auto_derivative_function.cc	305	
429.mcf	pbla.c	59	Candidate loop has a large number of function calls reported by Peruse, some due to their object oriented nature (C++ implementation).
483.xalancbmk	TraverseSchema.cpp	4393	
471.omnetpp	libs/sim/cpsquare.cc	225	
450.soplex	spxshift.cc	671	
400.perlbench	scope.c	699	
458.sjeng	neval.c	493	Candidate loop has large number of data dependent control flow branches.
403.gcc	dbxout.c	2455	
456.hmmer	core_algorithms.c	1534	
453.povray	fpmetric.cpp	235	

Table 6: Finding acceleratable loops in SPEC2006

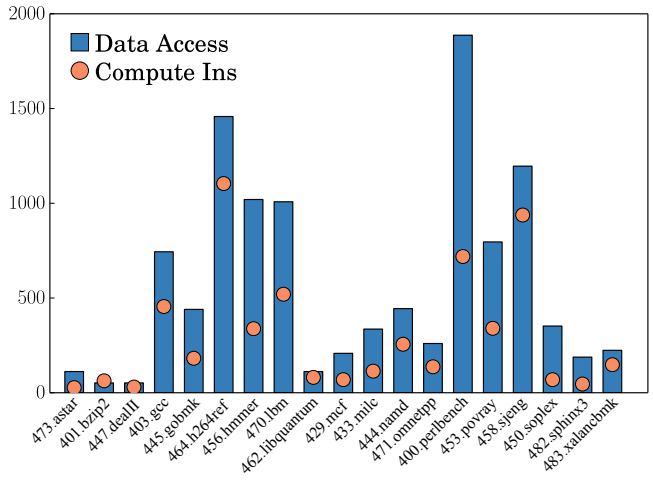

Figure 1: Data accessed (bytes) and compute instructions per iteration of candidate loop

for SIMD execution models, a high C-to-C ratio helps amortize the cost of loading a wide register once by performing multiple operations on it.

Figure 1 shows the bytes accessed by a single loop iteration along with the number of compute instructions. As they are set to the same numerical scale, we see for all applications with the exception of `401.bzip2` (18% more) the ratio of compute to communicate is less than 1, thus favoring the throughput oriented model. On examining the `401.bzip2` source (shown in Listing 5), it can be seen that applying SIMD specific transformations is almost trivial. For the remaining benchmarks, based on the observations presented in Table 6, we can determine the nature of the suitable execution model (OpenMP, SIMD, or Threads/warps).

To summarize, these case studies illustrate that static characterization, especially of the loops within applications is an effective means to assess suitability of these loops for acceleration. Performing such static analyses allows us to analyze large applications from the SPEC suite and extract

Listing 5: 401.bzip2 selected loop source code

```
853 for (; i >= 3; i -= 4) {
854     s = (s >> 8) | (block[i] << 8);
856     j = ftab[s] -1;
857     ftab[s] = j;
858     ptr[j] = i;
        /* Loop unrolled 4x */
871 }
```

interesting loops for further assessment of acceleration potential.

5. PREDICTING ACCELERABILITY

Peruse is useful to identify loops that are candidates for acceleration. For an end-user it is more desirable to get an estimate of the potential speedup due to acceleration. In this section, we describe an end-to-end framework to predict the potential speedup of a loop nest when off-loaded to a fixed function hardware accelerator.

Recently, Shao et al.[24] presented Aladdin, a pre-RTL, power-performance accelerator modeling framework. Aladdin provides a faster design space exploration relative to a conventional RTL-based synthesis flows. However, there are scalability limitations for Aladdin when used for large general-purpose applications with large input sets. Scalability limitations are inherent to any simulation-based approach to determine area/power/performance benefits of acceleration. To alleviate this, we have developed a machine-learning based approach to predict the potential speedup. This model is trained using features from static and dynamic (but target-independent) workload characterization and classification based on Aladdin's output. The model is then used to predict the speedup of any loop(s) from any general-purpose application. Instead of an absolute value for the speedup of the workload, we label the speedup as one of {*Poor*, *Moderate*, *Good*, *Excellent*} based on the range {0–4×, 5–10×, 11–20×, 20×+}, respectively.

Figure 2 shows the end-to-end work flow for the machine-learning based speedup prediction framework. For each ap-

plication we extract the most amenable loops for acceleration based on Peruse-based static analyses. An ISA independent dynamic trace is subsequently generated for these selected loops using the WIICA[57] LLVM based tracing tool. In addition, the dynamic trace is fed as input to Aladdin[24] to obtain the cycle time, area, and power for the loops. The machine learning model is trained using the static and dynamic characteristics of these loops as features, and the output of Aladdin as classes.

Evaluation Methodology.

The overall speedup prediction framework is independent of the choice of the baseline. For illustration, we have chosen a simple single-issue in-order model as a baseline for estimating speedup. The execution time of the dynamic trace is determined by assuming that instructions can be issued every cycle except memory access instructions which require 3 cycles and floating point (divides and sqrt) operations require 10 cycles each. The cycle time of the fixed function hardware accelerator is obtained by running the dynamic trace along with appropriate Aladdin configurations as mentioned in [24].

We use Weka[25], a popular Java based machine learning toolkit, to select and train a machine learning model. The training features include 27 dynamic features (instruction counts by type), 11 static features (selected characteristics from Table 2) and 5 configuration features which pertain to the layout of the fixed function hardware unit designed by Aladdin.

The training set is composed of 136 loops from *Polybench* and *SHOC*[13]. The training data set consists of 3264 data points when the loops are run with 24 different configurations. The training data contained 180, 856, 1022 and 1206 samples for {*Poor*, *Moderate*, *Good*, *Excellent*} classes, respectively.

Machine-Learning Model Selection.

We evaluated the accuracy of 6 classifiers using 10-fold cross-validation (a technique used to determine how well the training data will generalize to an independent result set). To select a model, we also take into account the time taken to train. Based on the data shown in Table 7 we chose Multiclass Alternating Decision Trees (LADTree – LogitBoost ADTree). The Multiclass alternating decision tree technique was proposed by Holmes et. al[28] and builds upon prior work by Freund and Mason[19]. The original alternating decision tree algorithm uses a combination of decision trees with boosting that generates classification rules that are often smaller and easier to interpret than when using conventional boosting methods such as AdaBoost[53]. Holmes et al extended the original binary classification algorithm to an effective multi-class algorithm which splits the problem into several two-class problems.

We validated the statistical significance of the selected model using a paired t-test[63] for a interval of 0.05 with the NaiveBayes classifier as baseline. The NaiveBayes classifier is chosen as baseline as its accuracy is not affected by the class distribution. Overall, the selected model had high precision, moderate recall with the area under the ROC (Receiver Operator Characteristic [62]) curve of 0.92.

Model Validation.

We use the LADTree model to predict the acceleration speedup potential for loops from `470.lbm` and `433.milc`. These loops were selected by Peruse as candidates for acceleration as shown earlier in Table 6. For validation, dynamic traces of these loops are fed to Aladdin to obtain the cycle time, area and power for 24 different configurations resulting in 48 unique data points. Table 8 shows that the model has a prediction accuracy of 79% (correctly predicts 38 out of 48 instances).

Model	Train Accuracy (10-fold CV)	Normalized model creation time
Naive Bayes	23.52%	3.5
Bayes Net	54.17%	5.5
SVC (RBF)[12]	51.66%	468
IB1[2]	74.82%	1
MLP	82.69%	1155
LADTree[28]	80.63%	61.5

Table 7: Model Selection

Class	True+	False-	Precision	Recall	ROC
Poor	0	0	0	0	?
Moderate	0.667	0.095	0.5	0.667	0.611
Good	0.455	0.081	0.625	0.455	0.526
Excellent	0.935	0.176	0.906	0.935	0.929
Wt. Avg.	0.792	0.144	0.791	0.792	0.858

Table 8: Test Accuracy by Class

It is important to note the high precision for classifying *Excellent* loops which are of primary interest to the user. The area under the ROC curve for the *Excellent* class is also near perfect (0.929) indicating that the predictions are made with low false positive rate which is highly desirable. Considering that Peruse already filters out poor candidate loops, it is no surprise that there are no instances of the *Poor* class in the testing set. All the 10 misclassified test instances belonged to the *Moderate* or the *Good* class.

6. RELATED WORK

Peruse builds upon and complements a long line of research related to workload characterization and parallelization.

Historically, much of this work stems from efforts at automated parallelization, and loop transformations. Traditionally, techniques such as feedback directed optimization have been used as part compiler optimizations to determine opportunities for loop unrolling, loop tiling, or software pipelining. Static program analysis for determining opportunities for loop transformations have also been explored [40]. Techniques like dependence testing have long been used to identify when different memory accesses do not conflict [22] and are thus amenable to concurrent execution. These techniques still form a backbone for automated parallelization today [1]. Similarly, idempotent regions of code may be executed or speculatively re-executed multiple times while always producing the same result, which is also useful in parallelization[15, 14]. Note that these properties of *parallelizable* code are frequently also subsets of the requirements for code to be *acceleratable* on a particular machine model. Thus, the results of these techniques are useful when determining whether or not a workload is amenable for a particular machine model. As the case studies show, Pe-

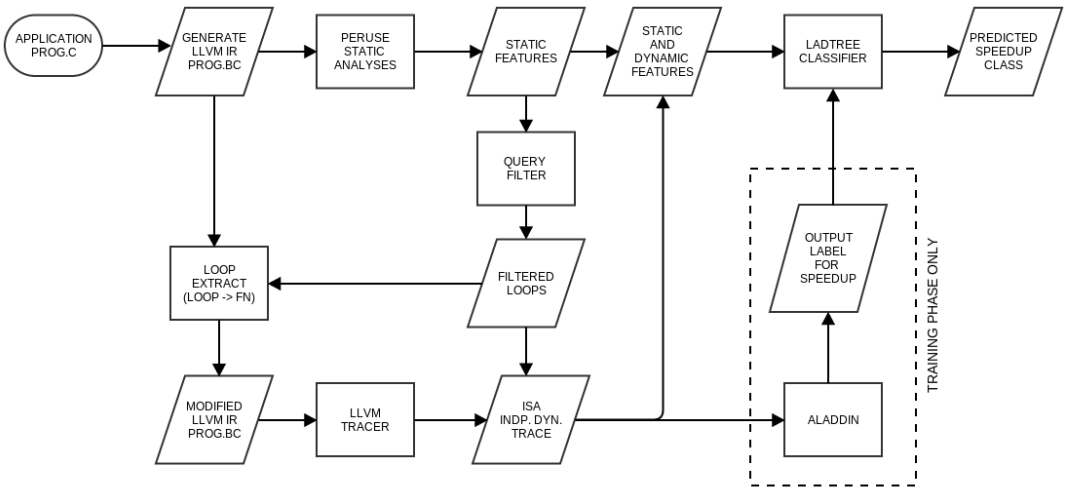

Figure 2: End to end machine learning based framework for accelerability estimation

ruse takes advantage of this knowledge and ensures that it is conveniently available.

More recently, efforts at both parallelization and acceleration have targeted specific semantic patterns, *idioms*, with the goal of recognizing and accelerating these semantic units [47, 33, 45, 21]. Note that Peruse does not seek to automatically discover or facilitate such parallelization, or acceleration of semantic units, but it instead helps a developer to determine in advance whether a program will be suitable for acceleration using static analyses.

Recent tools for loop-level characterization such as ELI-C [56] enable clustering workloads based on dynamic program structure characteristics for a given architecture and machine model. As demonstrated in the case studies, Peruse uses static program analysis based characteristics to cluster workloads and assess suitability for acceleration based on the chosen execution model.

Most recent of all are works that help to facilitate the search for a good accelerator design. For instance, Kim et al. [34] examine the dynamic computation and memory access behaviors of a program to determine whether accesses or computation pose the greatest opportunity or barrier to acceleration. Shao et al. [57] have used instruction set architecture independent characteristics to explore the design space of possible accelerators for a program and machine model [24]. Note that Peruse is not merely orthogonal but actually *complementary* to these techniques.

Prior work[30, 3, 39, 59] demonstrates that program characteristics can be correlated with performance for general-purpose architectures. Predicting parallel speedup estimates using machine learning has been studied by Ipek et al.[31] and more recently using an analytical model by Jeon et al.[32]. Predicting performance using machine learning for accelerators (FPGAs) has been recently studied by Meswani et al[43]. Baldini et al[6] and Ardalani et al[4] study performance prediction for GPUs based on micro-architectural behaviour of workloads on CPUs. The end-to-end framework developed in this work also uses similar machine-learning models to predict the speedup from acceleration of loops for fixed-function hardware accelerator.

7. CONCLUSIONS

We developed Peruse, a program analysis tool to quickly enable programmers and hardware developers to focus their efforts on code regions amenable for specialization, especially loops and loop nests. Peruse's program analyses open up opportunities for large scale exploration of current applications to identify code regions with common acceleration characteristics and help incrementally move specific code regions from general-purpose processors to accelerators. Our case studies show that Peruse can highlight program characteristics that identify opportunities for acceleration and provide a first-order assessment of the suitability of an application for a given accelerator execution model. Using Peruse we analyzed unmodified applications from the SPEC CPU benchmark suite and Polybench, HPC workloads, and identified not only the loops amenable for acceleration but also the potential challenges when porting the code to run on accelerators.

Further, we used the workload characterization results of Peruse as features and developed a machine-learning based model to predict the potential speedup of a loop when offloaded to a fixed function hardware accelerator. We have developed an end-to-end framework in which (a) Peruse is used to characterize and extract the most amenable loops for acceleration, (b) WIICA[58] is used to generate ISA independent dynamic trace, (c) Aladdin[24] is used to obtain the cycle time, area, and power for a fixed-function hardware accelerator for each of these loops, and (d) a machine-learning based model, trained using features from static and dynamic (but target-independent) workload characterization and classification based on Aladdin's output, is used to predict the potential speedup of loop from fixed-function hardware acceleration. Our results show that the model predicts the speedup of loops with an accuracy of 79%.

8. REFERENCES

[1] T. C. A. Kotha and R. Barua. Aesop: The autoparallelizing compiler for shared memory computers. Technical report, Department of Electrical and Computer Engineering, University of Maryland, College Park, April 2013.

[2] D. W. Aha, D. Kibler, and M. K. Albert. Instance-based learning algorithms. *Machine learning*, 6(1):37–66, 1991.

[3] M. Annavaram, R. Rakvic, M. Polito, J. Y. Bouguet, R. Hankins, and B. Davies. The fuzzy correlation between code and performance predictability. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 93–104, Dec 2004.

[4] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu. Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 725–737. ACM, 2015.

[5] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 1–8, New York, NY, USA, 2007. ACM.

[6] I. Baldini, S. J. Fink, and E. Altman. Predicting gpu performance from cpu runs using machine learning. In *Proceedings of the 26th International Symposium on Computer Architecture and High Performance Computing*, pages 114–122. ACM, 2006.

[7] T. Ball and J. R. Larus. Efficient Path Profiling. In *PROC of the 1996 MICRO*, 1996.

[8] M. Boyer, J. Meng, and K. Kumaran. Improving gpu performance prediction with data transfer modeling. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1097–1106, May 2013.

[9] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, pages 162–175, New York, NY, USA, 1986. ACM.

[10] P. Chen, L. Zhang, Y.-H. Han, and Y.-J. Chen. A general-purpose many-accelerator architecture based on dataflow graph clustering of applications. *Journal of Computer Science and Technology*, 29(2):239–246, 2014.

[11] C. B. codes. Coral collaboration - oak ridge, argonne, livermore, 2013.

[12] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[13] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 63–74, New York, NY, USA, 2010. ACM.

[14] M. de Kruijf and K. Sankaralingam. Idempotent code generation: Implementation, analysis, and evaluation. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–12. IEEE, Feb. 2013.

[15] M. A. de Kruijf, K. Sankaralingam, and S. Jha. Static analysis and compiler design for idempotent processing. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation - PLDI '12*, volume 47, page 475, New York, New York, USA, June 2012. ACM Press.

[16] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5(1):1–33, 2003.

[17] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–460. IEEE Computer Society, 2012.

[18] T. Fahringer. *Automatic Performance Prediction of Parallel Programs*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[19] Y. Freund and L. Mason. The alternating decision tree learning algorithm. In *ICML*, volume 99, pages 124–133, 1999.

[20] K. Ganesan, L. John, V. Salapura, and J. Sexton. A performance counter based workload characterization on blue gene/p. In *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, pages 330–337. IEEE, 2008.

[21] S. Garcia, D. Jeon, C. Louie, and M. B. Taylor. Kremlin: Rethinking and rebooting gprof for the multicore age. In *PLDI '11: Proceedings of the Conference on Programming Language Design and Implementation*, 2011.

[22] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. *ACM SIGPLAN Notices*, 26(6):15–29, June 1991.

[23] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):0038–51, 2012.

[24] Y. S. S. B. R. Gu and Y. W. D. Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures.

[25] 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*, 11(1):10–18, 2009.

[26] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 37–47, 2010.

[27] J. Holewinski, R. Ramamurthi, M. Ravishankar, N. Fauzia, L.-N. Pouchet, A. Rountev, and P. Sadayappan. Dynamic trace-based analysis of vectorization potential of applications. In *PLDI*, pages 371–382. ACM, 2012.

[28] G. Holmes, B. Pfahringer, R. Kirkby, E. Frank, and M. Hall. Multiclass alternating decision trees. In *Machine Learning: ECML 2002*, pages 161–172. Springer, 2002.

[29] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3):63–72, 2007.

[30] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 114–122. ACM, 2006.

[31] E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee. An approach to performance prediction for parallel applications. In *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*, Euro-Par'05, pages 196–205, Berlin, Heidelberg, 2005. Springer-Verlag.

[32] D. Jeon, S. Garcia, C. Louie, and M. B. Taylor. Kismet: Parallel Speedup Estimates for Serial Programs. In *OOPSLA '11: Conference on Object-Oriented Programming, Systems, Language and Applications*, 2011.

[33] M. Kawahito, H. Komatsu, T. Moriyama, H. Inoue, and T. Nakatani. A new idiom recognition framework for exploiting hardware-assist instructions. *ACM*

SIGPLAN Notices, 41(11):382, Oct. 2006.

[34] M. A. Kim and S. A. Edwards. Computation vs. memory systems: pinning down accelerator bottlenecks. In *Computer Architecture*, pages 86–98. Springer, 2012.

[35] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. Meet the walkers: accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 468–479. ACM, 2013.

[36] C. Kozyrakis, J. Gebis, D. Martin, S. Williams, I. Mavroidis, S. Pope, D. Jones, D. Patterson, and K. Yelick. Vector iram: A media-oriented vector processor with embedded dram. In *Proc. Hot Chips XII*, 2000.

[37] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM.

[38] L. L. N. Lab. Livermore unstructured lagrangian explicit shock hydrodynamics (lulesh) - https://codesign.llnl.gov/lulesh.php, 2013.

[39] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, pages 236–247, March 2005.

[40] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 136–146. IEEE Computer Society, 2009.

[41] J. Menon, M. De Kruijf, and K. Sankaralingam. igpu: exception support and speculative execution on gpus. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 72–83. IEEE Computer Society, 2012.

[42] R. Merritt. Arm cto: power surge could createâĂŹdark siliconâĂŹ. *EE Times, Oct*, 2009.

[43] M. Meswani, L. Carrington, D. Unat, A. Snavely, S. Baden, and S. Poole. Modeling and predicting application performance on hardware accelerators. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 73–73, Nov 2011.

[44] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox. PaceâĂŤa toolset for the performance prediction of parallel and distributed systems. *International Journal of High Performance Computing Applications*, 14(3):228–251, 2000.

[45] C. Olschanowsky, A. Snavely, M. R. Meswani, and L. Carrington. PIR: PMaC's Idiom Recognizer. In *2010 39th International Conference on Parallel Processing Workshops*, pages 189–196. IEEE, Sept. 2010.

[46] V. Packirisamy, A. Zhai, W.-C. Hsu, P.-C. Yew, and T.-F. Ngai. Exploring speculative parallelism in spec2006. In *ISPASS*, pages 77–88. IEEE, 2009.

[47] B. Pottenger and R. Eigenmann. Idiom recognition in the Polaris parallelizing compiler. In *Proceedings of the 9th international conference on Supercomputing - ICS '95*, pages 444–448, New York, New York, USA, July 1995. ACM Press.

[48] L.-N. Pouchet and U. Bondugula. Polybench 3.2, 2013.

[49] T. K. Prakash and L. Peng. Performance characterization of spec cpu2006 benchmarks on intel core 2 duo processor. *ISAST Trans. Comput. Softw. Eng*, 2(1):36–41, 2008.

[50] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Practical static race detection for c. *ACM Trans. Program. Lang. Syst.*, 33(1):3:1–3:55, Jan. 2011.

[51] W. Pugh. Uniform techniques for loop optimization. In *5th International Conference on Supercomputing (ICS'91)*, pages 341–352. ACM, 1991.

[52] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke. Sage: self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 13–24. ACM, 2013.

[53] R. E. Schapire. A brief introduction to boosting. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'99, pages 1401–1406, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[54] P. B. Schneck. Automatic recognition of vector and parallel operations in a higher level language. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 772–779, New York, NY, USA, 1972. ACM.

[55] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. Palm: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors, 2011.

[56] E. M. Shaccour and M. M. Mansour. ELI-C A Loop-level Workload Characterization Tool. In *Third International Workshop on Performance Analysis of Workload Optimized Systems (FastPath2014)*, Mar. 2014.

[57] Y. S. Shao and D. Brooks. ISA-independent workload characterization and its implications for specialized architectures. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 245–255. IEEE, Apr. 2013.

[58] Y. S. Shao and D. Brooks. ISA-independent workload characterization and its implications for specialized architectures. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 245–255. IEEE, Apr. 2013.

[59] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36(5):45–57, Oct. 2002.

[60] H.-W. Tseng and D. M. Tullsen. Data-triggered threads: Eliminating redundant computation. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 181–192. IEEE, 2011.

[61] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, Apr. 1991.

[62] Wikipedia. Receiver operating characteristic : http://en.wikipedia.org/wiki/receiver_operating_characteristic.

[63] Wikipedia. Student's t-test : http://en.wikipedia.org/wiki/student%27s_t-test.

[64] M. J. Wolfe. *Optimizing supercompilers for supercomputers*. MIT press, 1990.

[65] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 249–260. ACM, 2013.