

© 2020 Thiago Santos Faria Xavier Teixeira

A LANGUAGE AND A SYSTEM FOR PROGRAM OPTIMIZATION

BY

THIAGO SANTOS FARIA XAVIER TEIXEIRA

DISSERTATION

Submitted in partial fulfillment of the requirements  
for the degree of Doctor of Philosophy in Computer Science  
in the Graduate College of the  
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Doctoral Committee:

Professor William Gropp, Chair

Professor David Padua, Director of Research

Professor Vikram Adve

Dr. Corinne Ancourt, MINES ParisTech

Professor Saman Amarasingue, Massachusetts Institute of Technology

## ABSTRACT

Hardware complexity has increased over time, and as architectures evolve and new ones are adopted, programs must often be altered by numerous optimizations to attain maximum computing power on each target environment. As a result, the code becomes unrecognizable over time, hard to maintain, and challenging to modify. Furthermore, as the code evolves, it is hard to keep the optimizations up to date. The need to develop and maintain separate versions of the application for each target platform is an immense undertaking, especially for the large and long-lived applications commonly found in the high-performance computing (HPC) community.

This dissertation presents Locus, a new system, and a language for optimizing complex, long-lived applications for different platforms. We describe the requirements that we believe are necessary for making automatic performance tuning widely adopted. We present the design and implementation of a system that fulfills these requirements. It includes a domain-specific language that can represent complex collections of transformations, an interface to integrate external modules, and a database to manage platform-specific efficient code. The database allows the system’s users to access optimized code without having to install the code generation toolset. The Locus language allows the definition of a search space combined with the programming of optimization sequences separated from the application’s reference code. After all, we present an approach for performance portability.

Our thesis is that we can ameliorate the difficulty of optimizing applications using a methodology based on *optimization programming* and automated empirical search. Our system automatically selects, generates, and executes candidate implementations to find the one with the best performance.

We present examples to illustrate the power and simplicity of the language. The experimental evaluation shows that exploring the space of candidate implementations typically leads to better performing codes than those produced by conventional compiler optimizations that are based solely on heuristics. Locus was able to generate a matrix-matrix multiplication code that outperformed the IBM XLC internal hand-optimized version by  $2\times$  on the Power 9 processors. On Intel E5, Locus generates code with performance comparable to Intel MKL’s. We also improve performance relative to the reference implementation of up to  $4\times$  on stencil computations.

Locus ability to integrate complex search spaces with optimization sequences can result in very complicated optimization programs. Locus compiler applies optimizations to remove

from the optimization sequences unnecessary search statements making the exploration for faster implementations more accessible.

We optimize matrix transpose, matrix-matrix multiplication, fast Fourier transform, symmetric eigenproblem, and sparse matrix-vector multiplication through divide and conquer. We implement three strategies using the Locus language to create search spaces to find the best shapes of the base case and the best ways of subdividing the problem. The search space representation for the divide-and-conquer strategy uses a combination of recursion and OR blocks. The Locus compiler automatically expands the recursion and ensures that the search space is correctly represented. The results showed that the empirical search was important to improve performance by generating faster base cases and finding the best splitting.

We also use Locus to optimize large, complex applications. We match the performance of hand-optimized kernels of the Kripke transport code for different input data layouts. The Plascom2 multi-physics application is optimized to find the best way to use a multi-core CPU and GPU. The use of Tangram, Hydra, and OpenMP provided an interesting search space that improved performance by approximately  $4.3\times$  on ZAXPY and ZXDOTY kernels. Lastly, in a similar fashion to how a compiler works, we applied a search space representing a collection of optimization sequences to 856 loops extracted from 16 benchmarks that resulted in good performance improvements.

*To Gabriela, Arthur, and Oliver.*

## ACKNOWLEDGMENTS

First and foremost, I thank Professors David Padua and William Gropp for being supportive and engaging advisors. They are truly a model of great researchers. I admire David's positiveness, creativeness, and attention to detail. He was always an excellent listener, despite my often incoherent and ill-formed explanation of ideas. I will miss our conversations, especially the occasional ones. I appreciate Bill's guidance on and insights into high-performance computing. He was always encouraging and patient and understood my confusing questions and statements before I even finished them.

Among other faculty at the University of Illinois at Urbana-Champaign (UIUC), I thank Professors Vikram Adve for taking the time to be on my qual and dissertation committees, Daniel Bodony, Jonathan Freund, Luke Olson, and Andreas Kloeckner for their support and insights through the meetings and reviews at the Center for Exascale Simulation of Plasma-coupled Combustion (XPACC). External to UIUC, I thank Professor Saman Amarasingue for participating in my dissertation committee, especially for the relevant feedback and interesting conversations. I also thank Corinne Ancourt for participating in the dissertation committee. She was always thoughtful in our meetings, provided significant guidance and crucial support for Pips, which was used in many of the experiments presented in this dissertation.

Of my countless helpful colleagues, I especially admire Dimitrios Skarlatos, Thomas Shull, and Luis Remis for their amazing humor, intelligence, and hard work. Thomas was always helpful and provided remarkable insights into this work. I thank Dimitrios for the relentless support, even in the most uncertain times.

For being inquisitive colleagues, understanding companions, and for their pleasant distractions, I thank Sam White, Andrew Reisner, Tarun Prabhu, Ronak Buch, Simon Garcia de Gonzalo, Antonio Maria Franques Garcia, Apostolos Kokolis, Michael Robson, and Nikhil Jain.

I owe much inspiration for learning and guidance to the Polaris research group. I want to thank all the members, both past and present. Many thanks to Saeed Maleki, Adam Smith, Albert Sidelnik, Carl Evans, Amarin Phaosawasdi, Justin Sazday, and Sweta Pothukuchi. I thank Adam and Miranda Smith for all the warming receptions at their house, where we were always welcome. They made our holidays much more special.

I was very fortunate to have worked alongside several incredibly talented researchers at XPACC. They contributed enormously to the ideas presented in this dissertation, especially

Matthias Diener, Michael Anderson, Michael Campbell, John Larson, Matt Smith, and Cory Mikida.

I cannot thank enough Celso and Marcia Mendes for their love, friendship, and support. They have been fantastic since the day I first arrived in Illinois, and we will always be grateful.

Gabriela, I could not have finished this without your constant emotional support, confidence, understanding, and tolerance. I am well aware of how much you have given up for this to happen. Thanks to Arthur and Oliver for embracing me with all the energy, excitement, and love necessary to get to the end. They made this journey much more entertaining.

Most of all, I owe especially deep gratitude to my sisters, Maria Tereza and Ana Luiza, and parents, Nelson and Walkiria, for their never-ending love, patience, and encouragement. They created the circumstances that allowed me to pursue my dreams.

*This material is based in part upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0002374 and by the National Science Foundation under Award 1533912.*

## TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	1
1.1	Problem Context	5
1.2	Related Work	7
1.3	Contributions	12
1.4	Summary, Scope, and Outline	12
CHAPTER 2	PROGRAM OPTIMIZATION	15
2.1	The Program Optimization Search Space	20
2.2	Exploring the Search Space	24
2.3	Our Approach to the Program Optimization Process	30
2.4	Summary	33
CHAPTER 3	THE SYSTEM DESIGN AND IMPLEMENTATION	34
3.1	Interface with the Transformation Modules	36
3.2	Interface with the Search Modules	42
3.3	Correctness	43
3.4	Consistency Checking	43
3.5	Database of Variants and Results	43
3.6	Summary	47
CHAPTER 4	THE LOCUS LANGUAGE	49
4.1	Overview	49
4.2	Examples	51
4.3	Syntax and Semantics	52
4.4	Extended Backus-Naur Form of the Language	58
4.5	Summary	58
CHAPTER 5	COMPILING AND OPTIMIZING LOCUS PROGRAMS	61
5.1	Data-Flow Analysis	61
5.2	Pruning the Search Space	64
5.3	Interdependent Search Statements	66
5.4	Expansion Passes for Interdependent Search Statements	67
5.5	Summary	71
CHAPTER 6	EXPERIMENTS ON LOOP-BASED TRANSFORMATIONS	72
6.1	Experimental Environment	72
6.2	Matrix-Matrix Multiplication	73
6.3	Evaluating Hierarchical Tiling	75
6.4	1D and 2D Stencils	78
6.5	3D Stencil	79



6.6	Fixed Versus Random Initial Search Configuration . . . . .	82
6.7	Summary . . . . .	83
CHAPTER 7 OPTIMIZING THROUGH DIVIDE AND CONQUER . . . . .		84
7.1	Divide-and-Conquer Search Space . . . . .	85
7.2	Code Generation . . . . .	87
7.3	Matrix Transpose . . . . .	87
7.4	Matrix-Matrix Multiplication . . . . .	93
7.5	Fast Fourier Transform . . . . .	96
7.6	Symmetric Eigenproblem . . . . .	102
7.7	Sparse Matrix-Vector Multiplication . . . . .	105
7.8	Summary . . . . .	110
CHAPTER 8 OTHER EVALUATIONS . . . . .		111
8.1	Kripke . . . . .	111
8.2	Optimization of Arbitrary Loop Nests . . . . .	113
8.3	Comparing Search Modules . . . . .	116
8.4	Plascom2 . . . . .	119
8.5	Summary . . . . .	122
CHAPTER 9 CONCLUSIONS . . . . .		123
9.1	Future Directions . . . . .	125
REFERENCES . . . . .		129

## CHAPTER 1: INTRODUCTION

This dissertation presents a language and a system for optimizing applications by generating highly efficient, platform-specific code. Platform-specific optimizations are a key factor for attaining high performance in current hardware. However, organizing, structuring, and carrying out a sequence of optimizations have proven extremely difficult. Performance is a complicated function of many factors, including the underlying machine architecture, operating system, compiler technology, the application instruction mix and memory behavior, and the input characteristics, which is often unknown before runtime. Applications optimized using our language and system were able to match performance and, in some cases, outperform hand-optimized implementations.

Based on our experiences with several applications, the developers of complex, long-lived applications usually mainly rely on two approaches: i) on the selection of code according to the platform, compiler, and even input through `#ifdefs` or conditional statements; ii) on the use of highly-efficient, vendor-provided libraries for the computational bottlenecks. The former makes the code logic harder to understand and hinders maintainability in the long term. The latter transfers part of the optimization burden to the libraries. This practice requires that libraries contain functions to implement the algorithms and data structures used in these bottlenecks and, to be portable, that the interface across multiple platforms to be the same. Library functions, however, are domain-specific and consist of contained computations that cover mostly only common cases. The use of libraries may also miss out on optimization opportunities across function calls.

We present a methodology to help performance experts as well as ordinary programmers optimize applications. The same methodology can be used to generate platform-specific, efficient code for libraries. The optimization should be explicit and separate from the code, whereas the application’s code should be as “clean” as possible and not have any platform- or compiler-specific optimizations. We developed a language to program complex optimization sequences. The *optimization programming* happens on an external file, separated from the application itself.

Despite deep knowledge about the underlying architecture, compiler, and application’s domain, the optimization experts hardly know exactly the optimization sequence and what optimization parameters will effectively make the code run close to the optimal speed. The programmer must do some experimentation to decide how to proceed during the optimization process. Performance modeling is a common path towards understanding which optimizations to use. It has the advantage of requiring no empirical evaluations. Models are,

nonetheless, very complicated to devise, domain-specific, and sensitive to small changes on the target machine.

We claim that empirical search combined with heuristics is the best approach for optimizing a broad range of applications. The use of heuristics incorporates domain-specific knowledge to the process and is essential to prune the optimization search space that can grow exponentially.

A *search space* consists of a finite collection of search variables. A search variable has a domain of potential values, and constraints are used to either restrict or order combinations of values between variables. An exploration of a search space involves finding values for each search variable that satisfy a finite set of constraints. Each combination of values found during the exploration is considered a *point* in that space (i.e., configuration). A search space for program optimization consists of search variables whose domain is discrete. The search variable’s values are used for the parameters of the optimizations (e.g., unrolling factor). We also use search variables to either restrict or order combinations of optimizations. We define optimization as any insertion to or transformation of the source code with the intent to improve performance. A point in the program optimization search space represents an optimization sequence used to generate a candidate implementation (i.e., code variant). An empirical evaluation consists of the execution of  $n$  candidate implementations from the search space.

We have developed Locus, a language and a system for program optimization. The language is used to implement optimization programs and includes special constructs to describe the search space to be explored through empirical search. It allows the expression of complex, large search spaces concisely. The optimization programs are easy to maintain and adapt. In order to make the optimization process easier, the application’s code is divided into labeled code regions later referenced in the optimization programs. These code regions can share the same optimization sequence and search space, or each can have its own.

The system allows the integration of tools developed internally and externally. The goal is to reuse as much as possible the work developed by others. We focused on fine-grained transformations that fully automate the cumbersome and error-prone task of code manipulation. The system includes interfaces to integrate modules for code transformation and for exploring the search space during the empirical search. The easy integration of multiple modules allows the comparison and selection of the best modules for each specific context.

Generating semantically equivalent candidate implementations that perform better than the “naive” one can be quite complicated. Figure 1.1 shows the naive version and one candidate implementation for matrix-matrix multiplication. The candidate code shown is an example of how complicated the variants generated can get after a sequence of only two

loop transformations. It represents a 2-level hierarchical tiling combined with loop reordering. Carrying out this process by hand is time-consuming, error-prone, and unproductive, especially because multiple candidates must typically be generated until the best one is found.

```

1  int main()
2  {
3      int i, j, k;
4      init_array(A, B, C);
5
6      #pragma @Locus loop=matmul
7      for (i=0; i < 4096; i++)
8          for (j=0; j < 4096; j++)
9              for (k=0; k < 4096; k++)
10                 C[i][j] = beta*C[i][j] + alpha*A[i][k]*B[k][j];
11
12     print_array(C);
13     return 0;
14 }

```

(a)

```

1  int main()
2  {
3      int i, j, k;
4      init_array(A, B, C);
5
6      #pragma @LOCUS loop=matmul
7      for(i_t = 0; i_t <= 7; i_t += 1)
8          for(k_t = 0; k_t <= 3; k_t += 1)
9              for(j_t = 0; j_t <= 1; j_t += 1)
10                 for(i_t_t = 8 * i_t; i_t_t <= ((8 * i_t) + 7); i_t_t += 1)
11                     for(k_t_t = 256 * k_t; k_t_t <= ((256 * k_t) + 255); k_t_t += 1)
12                         for(j_t_t = 32 * j_t; j_t_t <= ((32 * j_t) + 31); j_t_t += 1)
13                             for(i = 64 * i_t_t; i <= ((64 * i_t_t) + 63); i += 1)
14                                 for(k = 4 * k_t_t; k <= ((4 * k_t_t) + 3); k += 1)
15                                     for(j = 64 * j_t_t; j <= ((64 * j_t_t) + 63); j += 1)
16                                         C[i][j] = beta*C[i][j] + alpha*A[i][k]*B[k][j];
17
18     print_array(C);
19     return 0;
20 }

```

(b)

Figure 1.1: a) Annotated matrix-matrix multiplication source code. The identifier *matmul* to be referenced in the optimization program. b) Candidate implementation of the matrix-matrix multiplication automatically generated. This code represents loop reordering combined with a 2-level hierarchical tiling.

Even for compilers, the generation of such implementations can be difficult. The gap between the performance of hand-tuned and compiler-generated code has grown substantially. There are several aspects of code optimization that makes it harder for traditional compiler approaches. First, depending on the input, we may choose a completely different data structure from that of the initial implementation. The use of different data structures may alter

code behavior completely. Data structure selection depends on application-specific semantic information, which is difficult to obtain using traditional static analysis. Second, because of the limited compilation time, the optimization passes can only use quick heuristics. Knowing when high heuristic costs or even an empirical search can be tolerated must be justified by expected performance gains. Third, compilers have no access to application-specific information to help to identify candidate transformations and their parameters. We would not expect current compilers to identify candidates nor to be able to choose among these candidates. In short, no input and application-specific information and limited compilation time restrict performance attained by traditional compiler-generated code.

Our work’s ultimate goal is to provide a way to define optimization sequences and search spaces concisely, which will be used to automatically generate candidate implementations whose performance approaches that of the best hand-optimized code. It has been shown that it is possible to build automatic tuning systems to generate implementations whose performance competes with, and even exceeds that of, the best hand-tuned code [1, 2, 3, 4, 5, 6]. The lessons learned from these systems have inspired our implementation.

Our system follows these steps for optimizing an application:

- parse of the optimization program and convert it to internal representation;
- compiler passes that are applied to the internal representation of the optimization program to prune the search space;
- conversion of the search space to the format accepted by the selected search-module;
- search the space for the best implementation using the search module’s techniques. The candidates are selected, generated, executed, and evaluated according to the chosen metric;
- generates an optimization program implementing a single optimization sequence containing the steps to generate the best implementation for each code region evaluated. A code region can be a loop, straight-line code, or a function.

Users of the system can experiment with different search techniques without having to modify the optimization program. All the details regarding the use of the search techniques and representation are transparent to the user.

The optimization of applications plays an increasingly critical role in achieving high performance. Our thesis is that, by using the aforementioned approach, we can ameliorate the difficulty of optimizing applications by using *optimization programming and automated empirical search*. Our methodology is based on the definition of search spaces combined with

the programming of optimization sequences. Our system automatically selects, generates, and executes candidate implementations to find the one with the best performance.

The remainder of this chapter presents in more detail the challenges of the optimization process. We review the related developments leading up to our work and show a summary of our contributions.

## 1.1 PROBLEM CONTEXT

Hardware complexity has increased over time, and the race for more performance is the reason behind the addition of new features and accelerators’ development. New memory technologies, deep cache hierarchies, branch prediction, out-of-order execution, and speculative execution complicate modern highly complex platforms’ use and performance modeling. They make programming efficient codes harder and performance less predictable. Moreover, each hardware platform commonly requires a different sequence of optimizations to attain a high fraction of its nominal peak speed. Software developers must devote significant time to benefit from the computing power of modern CPUs and accelerators as the gap between the performance of hand-tuned and compiler-generated code has grown substantially.

As platforms evolve and new ones are adopted, programs must often be altered by the numerous optimizations needed for each target environment (hardware and software stack) to approximate maximum computing power. As a result, the code becomes unrecognizable over time, hard to maintain, and challenging to modify. Furthermore, as the code evolves, it is hard to keep the optimizations up to date. The need to develop and maintain separate versions of the application for each of the target platforms is an immense undertaking, especially for the large and long-lived applications commonly found in the high-performance computing (HPC) community.

An application code is portable if it runs on a diverse set of platforms without needing significant modifications and produces a similar output. Ideally, the application code would also be *performance portable* and achieve high performance across a variety of platforms [7, 8].

However, creating performance portable code is difficult because the search space is very large, and many important decisions affect an application’s performance on the target machines. These include the order of operations within large loops and the choice and layout of data structures. The optimal or near-optimal choices often differ depending on the target platform.

In practice, improving performance can be challenging because of the quantity of factors involved in the process. The first challenge is to define the search space, namely: choose which optimizations to carry out, the order of them, and their parameters. This requires the

understanding of the underlying architecture (e.g., cache sizes, number of cache levels, kinds of registers) in order to list the optimizations that would bring performance improvements. A deeper knowledge of the application and architecture often results in the definition of heuristics to reduce the search space to optimization sequences and their parameters that are most likely to generate faster candidates.

Following the search space description, the next step is to select which techniques (or a collection of) to use for traversing such space. Depending on its size, it may be necessary to partition the space to accelerate space exploration. The partition of the search space also facilitates the analysis of results as each partition contains fewer search variables than the original space. With fewer search variables, it may be easier to identify any correlation between optimizations and performance improvement. Techniques represent the space differently (e.g., flat, hierarchical), and optimal selection depends on the input space.

The main task of the technique selected to explore the space is to suggest the values that represent a point in the search space. These values are used to create a candidate implementation. Creating the candidate implementation by hand is not feasible, and multiple of them are often generated. The next challenge is to find tools that can generate such candidates automatically. Typically, these candidates consist of multiple optimizations chained together, in which the order is important.

Another important aspect is to use candidates that are guaranteed to be correct. Dependence analysis is able to guarantee the correctness of the transformations applied by using execution-order constraints between statements. However, it has limitations for which codes the constraints can be accurately calculated. Because of that, some correct candidates may not be generated because dependence analysis cannot calculate the constraints and guarantee their correctness. The programmer, however, can skip this check whenever it is known that the code generated is correct.

Each candidate implementation suggested is evaluated empirically and returns a value that is used to compare the candidates. It is also used to help select the next candidate to be evaluated.

After the empirical search, the next challenge is to analyze the results and understand which optimizations are the most critical to attaining performance. This helps better understand the application behavior and the underlying architecture. The results from one process can be reused to guide future search processes using different compilers or even in different but similar platforms.

Finally, the best candidate implementation needs to be available for the users. Depending on the application's size and complexity, the best candidate includes optimizations to multiple code regions. And, since they are platform-specific, the optimizations used for each

code region for each target platform must be stored.

In summary, the optimization process can become very challenging as it is necessary to describe a search space of reasonable implementations, select the method to explore such space, automatically generate the candidates, analyze results, and deploy platform-specific optimized code to users.

## 1.2 RELATED WORK

There have been several projects to develop new programming models, languages and tools aimed at providing programmers with productive tools for achieving performance portability. As shown in Table 1.1, we classify the approaches into *high-level abstractions*, *non-programmable*, *code transformations*, *custom languages or language extensions*, and *alternative selection*.

### 1.2.1 High-level Programming Abstraction Tools

The high-level programming abstraction tools provide a limited set of abstractions for specific domains to represent algorithms. From these representations, the tools can generate optimized platform-specific binaries. Lift [9] requires programmers to write a high-level expression composed of algorithmic primitives using rewriting rules. Lift maps this high-level expression into a low-level expression in OpenCL. Halide [10] is a domain-specific language for complex image processing pipelines that decouples the algorithm representation from the schedule of the operations. The Pochoir stencil compiler [11] allows a programmer to write simple, functional specifications for stencils that are translated into highly optimized implementation. SPIRAL [6] includes a high-level mathematical framework that links the “high” mathematical level of transform algorithms and the “low” level of code implementations. It features multiple search methods to select among the low-level options, including exhaustive, random, dynamic programming, evolutionary, and hill climbing.

### 1.2.2 Non-programmable approaches

The non-programmable approaches differ from the other approaches in that they do not start with user-written code, but instead, the code is generated directly. These tools carry out all the tuning process without user intervention. It is completely automatic and typically uses heuristics to accelerate the search of the space of variants. These tools have a particular domain, are self-contained, and generate optimized library routines.



Table 1.1: Automatic program optimization approaches.

	Domain	Optimization Domain	Optimization Time	Search Method
High-level Abstractions				
Spiral	Signal processing	Rewriting rules	offline	Dynamic programming, Evolutionary (+others)
Lift	General purpose	Rewriting rules	offline	Bandit
Halide	Image processing	Scheduling	offline	Stochastic
Pochoir	Stencils	Cache-oblivious algorithm	offline	-
Non-programmable				
FFTW	Fast Fourier transform	Combination of solvers for FFTs	offline	Dynamic programming
Atlas	Dense linear algebra	Blocking, scheduling, and unrolling	offline	Exhaustive
OSKI	Sparse linear algebra	Sparse solvers	online	Heuristic
PHiPAC	Matrix multiplication	Adaptive library generator	offline	Heuristic
Code Transformations				
CHiLL	General purpose	Loop Transformations	offline	-
Pluto	General purpose	Loop Transformations	offline	-
POET	General purpose	Parameterized code transformations	offline	-
Orio	General purpose	Loop Transformations	offline	Nelder-Mead, Simulated Annealing
X Language	General purpose	Loop Transformations	offline	Exhaustive
Custom Languages or Languages Extensions				
Sequoia	General purpose	Memory hierarchy aware language	offline	-
Petabricks	General purpose	Algorithmic choices	offline	Evolutionary
Kokkos	General purpose	Abstractions for parallel execution and data management	offline	-
RAJA	General purpose	Abstractions for loops and data layouts	offline	-
Alternative Selection				
Nitro	General purpose	Variant selection	offline, online	Classification (SVM)
Active Harmony	General purpose	Parametric traversal	online	Nelder-Mead
OpenTuner	General purpose	Parametric traversal	offline	AUC Bandit

For instance, FFTW [12] is a comprehensive collection of fast C routines for computing the discrete Fourier transform (DFT). It does not implement a single DFT algorithm, but it is structured as a library of routines that can be composed in many ways or plans. The plan dictates which routines should be executed and in which order, taking into account the input size and type and which routines happen to be faster on the underlying hardware. ATLAS [5] automatically generates linear algebra routines, including highly efficient basic linear algebra routines. It first focuses on machine-specific features, especially the memory hierarchy, by generating an optimized matrix multiplication that fits in the fastest level cache. The other parameters, such as block and loop unrolling factors, are selected through an empirical search.

The goal of PHiPAC [13] is also to produce high-performance linear algebra libraries. Its design is similar to ATLAS.

OSKI [14] is a collection of low-level primitives integrated into automatically tuned computational kernels for sparse matrices. Unlike the previous two approaches, in which the tuning takes place offline, OSKI defers the tuning until the runtime to make decisions of the data structures and code transformations based on the input matrix and the underlying hardware.

### 1.2.3 Code Transformation Tools

The code transformation tools take as input an initial version of the code and the transformations to apply to it. This specification of transformations can take the form of annotations on the code [2, 3], scripts with commands that represent each transformation [1], or command-line parameters. Some of these tools allow the developers to implement their own program transformations [15].

CHiLL [1] contains loop transformation and code generation primitives. It takes as input the original code and a transformation script with bound parameters and generates a collection of code versions. POET [15] is an embedded scripting language for parameterizing complex code transformations to be empirically tuned. The POET language is designed to decouple the empirical tuning aspect of performance optimization from any library or compiler’s specifics. Locus is not restricted to AST-based transformations and is designed to combine transformations from diverse techniques. For instance, we combine transformations from Pips, which is a polyhedral-based compiler, with our own Rose-based transformations that are carried out through the AST. The Locus language is able to represent better complex search spaces on applications that have multiple code regions to be optimized. In Locus, the search space definition is part of the language.

Another example in this class is Orio [2], an annotation-based empirical performance-tuning system that takes annotated C source code as input, generates code variants of the annotated code, and empirically evaluates the performance of the generated codes, ultimately selecting the best-performing version to use for production runs. The X Language [3] provides pragmas that can perform loop transformations and code transformations defined as pattern-replacement rules. Pluto [16] transforms C programs for coarse-grained parallelism and data locality simultaneously. It uses the polyhedral model as an abstraction to perform high-level transformations on affine loop nests. Locus implements a separation of concerns and avoids the definition of the optimizations in the application’s code. Besides, our system is designed to allow the integration of modules for search space exploration. It abstracts from the user the details about using each of these modules, making it easier to evaluate different search methods each one has implemented.

### 1.2.4 Custom Languages and Language Extensions

Custom languages and language extensions have been proposed to provide abstractions that insulate algorithmic choices, loop patterns, and data layout from the underlying platform. The Sequoia [17] programming model assists the programmer in structuring bandwidth-efficient parallel programs that remain easily portable to new machines. The abstraction of tasks is used as self-contained computation units isolated in their own local address space, which helps express parallelism within a hierarchical organization.

Another programming language, along with a compiler, is PetaBricks [4]. The language incorporates fine-grained algorithmic choices in program optimization and allows the specification of different granularity and corner cases. The auto-tuner uses a choice dependence graph containing the choices for computing each task and encodes the implications of different choices on dependencies. It also contains a dynamic task scheduler and a runtime library to manage reading, writing, and configurations.

RAJA [18] provides portable abstractions for loops that include loop transformations, reductions, scans, atomic operations, data layouts, and views. Loop bodies and traversals are decoupled via lambda expressions (loop bodies) and templates (loop traversal methods) and support execution policies for different programming model backends. The Kokkos [19] library unifies abstractions for both fine-grained data parallelism and memory access patterns for performance portability in manycore architectures.

### 1.2.5 Alternative Selection

The alternative selection tools assume that either the user or some other tool provides a collection of variants as input. The main concern of the designer of tools in this class is the selection process. As the search space is very large, efficient techniques for selecting the best combination of code variants or parameters can drastically reduce the search time. Some of these tools make the decision online, whereas others have it defined offline. The advantage of making the decision online is that the input features can be used in the decision process [20]. The design of languages, compilers, and runtime systems for the development and selection of algorithmic variants has shown to be a successful way of optimizing irregular iterative and recursive problems on parallel platforms [4, 17]

The Nitro [20] framework focuses on how code variants and meta-information for variant selection are expressed and uses a classification technique to select the most appropriate variant during execution. The codes are represented through library calls rather than language extensions. Input features that significantly affect the variant selection can be calculated

by providing functions to the system. A two-phase approach is used. First, the application is processed to generate a model over the code variants also provided by the user. Second, during execution, the production version uses the model generated in the first phase and adapts the execution based on input data.

Active Harmony [21] permits application programmers to express application-level parameters and automates the process of searching among a set of alternative implementations. It has been combined with CHiLL [22] to search for the best sequence of loop transformation variants of computationally-intensive kernels. The OpenTuner [23] project presents a new framework for building domain-specific program auto-tuners. It features an extensible configuration and technique representation able to support complex and user-defined data types and custom search heuristics.

Some systems attempt to separate the role of the performance tuning expert from the application domain. This separation can allow the programmer to focus on application issues and performance tuning at different times. Alternatively, if they are not the same person, make these two tasks more independent and more productive once each expert focuses on their own domain. Multiple tuning specifications can be generated and tested for different architectures, making the application performance portable.

Sequoia, for instance, maintains a strict separation between the algorithm implementation and the machine-specific optimizations. The auto-tuning system in Petabricks outputs an application configuration file containing the choices selected. This file can be either used to run the application, or used by the compiler to build a binary with hard-coded choices. CHiLL also has an external file containing all the transformations to be applied to the code, which can be different depending on the machine specifics.

Another interesting feature of some of these systems is the automated validation of code variants. Petabricks has an automated consistency checking to ensure that the different algorithms solving the same problem produce consistent results. This helps the user to detect bugs and increase confidence in the code generated. Orio also supports an automated validation by comparing the numerical results of the multiple transformed versions. This technique is not provably correct but provides good testing coverage. It also complements provably correct techniques, as these proofs do not extend to the implementation of the compilers, runtime systems, or hardware. In real systems, users must confront faults and errors in these components as well.

### 1.3 CONTRIBUTIONS

The main contributions of this dissertation are:

- A programming language capable of representing complex spaces of candidates generated by multiple transformation sequences for mainstream programming languages (C, C++, Fortran);
- Full compiler support that makes the notation less verbose and enables the automatic optimization of optimization programs;
- Compiler passes to generate complex hierarchical space for divide-and-conquer algorithms properly;
- A system that brings together transformations, search space generation, and traversal of the space:
  - The system can integrate multiple external search modules (backends) that can be chosen according to the search space. It automatically converts between its internal search space representation to the format accepted by the search module. This search-module abstraction results in no changes in the optimization program when new modules are integrated;
  - It is also able to integrate multiple transformation modules that can be chosen according to the input source-code. Given the complexity of programming languages, some modules have constraints on the source-code being transformed, making it important to have multiple alternatives.
- A database to save candidate implementations and results that later can be reused by future searches. These data are also helpful in analyzing the performance attained. The best candidates are retrieved for deployment according to user demands;
- We apply these techniques to various applications, including linear algebra kernels, stencils, fast Fourier transform, symmetric eigenproblem, sparse matrix-vector multiplication, Kripke mini-app, Plascom2 multi-physics application, and to hundreds of loops extracted from 16 important benchmark suites.

### 1.4 SUMMARY, SCOPE, AND OUTLINE

This thesis’s central claim is that achieving and maintaining high performance over time, especially for complex, long-term applications, given the current trends in architecture and

compiler development, requires a platform-specific, search-based approach. The idea of a language for programming a search space of reasonable implementations and then exploring that space is modeled on what practitioners do when hand-tuning code. The language gives the experts more flexibility in trying different configurations, and the application’s code is kept cleaner, which helps maintainability.

The primary aim of this dissertation is to show why and how optimization programming on a separate language combined with the automatic generation of candidates can be used to make applications faster. The design and implementation of the system are discussed in Chapter 3. The interface to integrate both transformation and search modules, details about the correctness and consistency checking, and how we use the database to share results among multiple search modules and across multiple optimization processes are also discussed.

In Chapter 2, we discuss the program optimization search space. We also review the most common search techniques that are used to explore complex search spaces. Finally, we present in detail our approach to the optimization process.

The details, syntax, and how to represent search spaces in our language are presented in Chapter 4. A description of the compiler developed to optimize our language is in Chapter 5. In the same chapter, we present the use of data-flow analysis to prune the search space by removing unnecessary search statements, the important role of interdependent search statements, and the compiler passes developed to represent complex hierarchical spaces correctly.

Our system’s first target was the optimization of kernels that are frequently bottlenecks in diverse applications in scientific computing, artificial intelligence, and engineering. Using loop transformations, we were able to show the usefulness of the search by generating faster code than those produced by conventional compiler optimizations based solely on heuristics. The optimization of these kernels using our system are presented in Chapter 6.

Divide and conquer is an important strategy for solving many conceptually difficult problems. It uses recursion to break and solve smaller subproblems that, when small enough, can benefit from the faster access speeds of the caches. We optimize a series of important applications (e.g., FFT, matrix transpose, and symmetric eigenproblem) through divide and conquer using our proposed language and system. Locus searches for the best loop transformations and size for the base case. We evaluate multiple strategies for determining when to stop subdividing and in which order the problem’s dimensions should be subdivided. The results are shown in Chapter 7.

We extended our experimentation to other more complex contexts in Chapter 8. We use our approach to work as a compiler and optimize a collection of loops extracted from

16 important benchmarks. We use the same heuristic for all the loops, but the search space is defined according to each loop’s characteristics. This is important to generate only correct variants and to accelerate the empirical search. We also optimize Plascom2, a large, complex multi-physics application, to use GPU and CPU more efficiently. Kripke is a mini-app developed by Lawrence Livermore National Laboratory that consists of multiple code regions that are optimized independently. The authors provide a hand-optimized version whose performance we were able to match using our approach.

Chapter 9 looks forward to the future of the tuning systems and considers the common challenge to all systems: the problem of search. Specifically, how and when to stop the search and how to search large and deep hierarchical spaces. Furthermore, how to deal with optimizing for the myriad of specialized architectures that are becoming the norm.

## CHAPTER 2: PROGRAM OPTIMIZATION

The push for more performance has encouraged the development of computer architecture. For instance, the cost of memory accesses, which are increasingly dominating the cost of computing as processors get faster has spurred the development of deep cache hierarchies and new memory technologies to take advantage of data locality. In many cases, to take the best advantage of the memory hierarchy, programs must be written in ways that affect readability. Likewise, out-of-order execution, branch prediction, and speculative execution have been devised to harness a potential execution overlap among instructions. However, as the instructions execute in parallel, it is more complicated to predict and model the performance of different algorithms in these highly dynamic environments. Again, in this case, the way a program is written affects how well the program benefits from instruction-level parallelism. Additionally, the popularization of heterogeneous computing, which adds more performance and energy efficiency to the systems, brings in another architecture with its own characteristics, increasing even more the complexity of high-performance programs. As we can see, all these architecture advances make programming harder to achieve high performance and performance less predictable.

The performance gains achieved by computer architecture has attracted most of the attention over the last 30 years. Software developers expected performance gains by adopting new generations of processors that could execute programs at a faster clock rather than implementing architecture-specific optimizations. However, computer architecture was forced to change because of the end of Dennard Scaling and more attention from the software developers became necessary in order to harness all the computing power provided by the hardware. Furthermore, each modern computer architecture requires a potentially non-overlapping set of optimizations to attain a higher fraction of its nominal peak speed. This leads to other questions about performance portability and code maintainability. In particular, the developers must decide how to manage different optimized versions of the same code tailored to different architectures and keep them up to date as new algorithmic features are added.

Nevertheless, another degree of complexity is the development for parallel computers consisting of thousands of processors, in which load imbalance, communication, and synchronization costs complicate attaining a high ratio of the machine's performance peak.

The fundamental requirement to achieve high performance is an optimal mapping between architecture and algorithm. This mapping is not straightforward, though, as it can span from an entirely specific algorithm to a particular architecture or a very general one developed without any architecture insight. In the former case, high performance is achieved at the



expense of portability, given that the code is so specific that it cannot execute in different architectures. In the latter case, the portability comes at the expense of performance because it is challenging to exploit the resources available efficiently without any insight about the architecture.

Selecting the algorithm among many to solve a specific problem is the first and foremost step of the software development process. However, the selection is not straightforward and often input-specific. In some cases, multiple fine-grained algorithmic selections can be very advantageous, as shown in [4]. This composition complicates, even more, the algorithm selection task given the combinatorial explosion of possibilities. For instance, the intuition about the optimal sorting algorithm would provide different answers based on the number of elements to be sorted. For very few elements, insertion sort is faster, whereas for a medium number of elements, quicksort is faster, and for very large input, radix sort results in the best performance. It is also possible to compose the three different algorithms once quick sort and radix sort recursively divides the problem into subproblems until they are small enough to apply the insertion sort.

After algorithm selection, there are some guidelines to improve performance without a significant time investment by the developer, as discussed by Goedecker et al in [24]. The use of efficient libraries is a good example. Highly optimized numerical libraries are available (i.e., Intel MKL, Nvidia cuBLAS) and should be used whenever possible, sparing developers from implementing and tuning their own version for each machine.

Another example is choosing a data structure layout that increases data locality and reduces cache misses. A canonical example is the layout of data in a multidimensional array, which affects performance depending on how the array is linearized in memory. Row-major order puts consecutive elements of a row next to each other, whereas column-major order puts consecutive elements of a column next to each other. In some cases, different data structures could be optimal in different parts of a program.

The search for and use of appropriate compiler flags typically leads to speed up. At the same time, higher optimization levels can lead to performance deterioration. The heuristics used by compilers to apply transformation rules do not guarantee an improvement in all cases. Nevertheless, it is always worthwhile to experiment with different optimization levels as well as other options. Notably, architecture-specific compiler flags have a bigger chance of better results since the heuristics can assume a narrower scenario. Compiler directives or pragmas are also an effective approach for specific code regions where performance is crucial. Nonetheless, these directives are not standardized across compilers leaving the code less portable. In summary, the results achieved by experimenting with compiler flags and directives are not portable, and the flags rarely remain the same for new processors, requiring

the experiments to be performed all over every time a new processor comes out.

Many basic optimizations are generally architecture agnostic and automatically carried out by modern compilers. Among them are common subexpression elimination, strength reduction, loop-invariant code motion, constant value propagation, dead-code elimination, and induction variable elimination [25].

There are also more time consuming and invasive optimizations that may yield a bigger performance reward. These code transformations can be performed by the compiler or manually and generate a semantically equivalent program to the original but better suited to the target machine. Despite the better performance, after transformations, the code may become less readable and increase register pressure and program size, which should be taken into account depending on the final goal.

The most common code transformations are loop transformations, as loops are often sources of hotspots due to their repetitive nature. For example, loop unrolling unwinds the sequence of statements in the loop body to reduce or eliminate the overhead of executing instructions that control the loop and reduce branch penalties. Loop interchange switches the positions of two tightly nested loops. Depending on the loop order and on the target architecture, the switch may create parallelization opportunities through automatic vectorization (CPU) or efficient use of threads (GPU). Strip-mining divides the iteration space of a loop into chunks. A combination of strip-mining and interchange is called loop tiling. Its goal is to enhance cache performance by improving spatial and temporal locality in a nested loop [26]. Loop transformations require parameters that rely once again on the architecture. For instance, unrolling and tiling require the unrolling factor and the blocking dimensions, respectively, which are related to the processor pipeline depth and cache size.

After all, the search space is large, and we only mentioned a few optimizations. There are many important decisions over the software development process to exploit all the available target machine opportunities. With this in mind, studies have shown that an empirical search over the search space can be very effective. This strategy consists of generating the candidate implementations and executing each of them on the target machine. Instead of creating complex models that could predict the performance in an ever-changing myriad of architectures, the empirical search relies on just running each candidate, collecting the evaluation criteria, and selecting the best performant one. Nonetheless, since the search space is extensive, the exhaustive traversal can be prohibitive. Many works have presented effective ways of finding good solutions without traversing the entire search space.

Auto-tuning software means that the software has been encoded to automate the painful and error-prone process of generating and exploring the search space. A process that is largely repeated to encounter a candidate implementation with better performance than the

original version. Auto-tuning can resort to an empirical process, as explained before, use a predictive model or a combination of both. It may take into account the input data, architecture details, and compiler characteristics to optimize more aggressively than traditional compilers. Auto-tuning has been used to select loop transformations, data structures, parallelization, and algorithmic parameters.

Compilers have been successful in some of the critical tasks of code generation, particularly instruction scheduling, pointer analysis, instruction selection, and inter-procedural analysis. In the ideal scenario, compilers should automatically generate the optimal code on any architecture, without any programmer intervention. Nevertheless, the compiler by itself cannot fully satisfy this role. Its conservative nature precludes the generation of some high-end optimizations without permission from the programmer. It is also very complicated to develop static analyses and optimizations that suit all architectures for arbitrary code. The compilation time is also a constraint for many applications and limits the number of optimizations that can be considered out of the extensive number of possible ones. The increasing complexity of the architecture and the optimization space's size tend to make compilers deliver unsatisfactory performance. Even the use of advanced optimization flags like `-O3` is not enough to narrow this gap. Also, the compiler technology has not solved all problems, and, for instance, there are no effective compilers that automatically generates MPI code out of a sequential version.

On the other hand, the profitability of compiler optimizations has been shown by several publications. There are numerous reports of performance improvement through data locality, instruction parallelism, better usage of registers, and vectorization [27, 28, 29, 30, 31].

Applying all these optimizations by hand, however, is not an easy task. It is error-prone, repetitive, and the outcome is substantially less readable than the original version. Most of the optimizations are architecture-dependent, which makes performance portability an issue. Performance improvement in one architecture does not mean similar improvements in other architectures. Even small design changes may expose different bottlenecks and require different optimizations. Moreover, given the combinatorial explosion of the search space, selecting the optimizations and their variations to be implemented turns out to be a critical part of the tuning process.

The software development process is driven by the requirements defined by the developers based on user needs. They are divided into functional and non-functional requirements. The functional requirements are defined as features or as components implemented in the system. The non-functional requirements are used to assess the system's operation, rather than specific behaviors.

Developers are primarily concerned with the correctness and capabilities of the software,

which result from the implementation and verification of the functional requirements. Performance is a non-functional requirement and during the software development is often dealt with a “fix-it-later” attitude [32]. This attitude only changes if the software performance is below expectation and happens in the very late stages of the development, just before releasing the software. Postponing is always a reasonable decision. Performance tuning is indeed a significant investment and very time consuming, due mainly to the need of understanding and interacting with many different hardware details. Furthermore, it is hard to convince someone to put effort into it without telling them how much they are missing out.

In the high-performance computing community, this scenario is not different. The scientists make decisions based on maximizing scientific output, not application performance. Performance is a means to an end, not an end in itself. For instance, as reported in [33], when a scientist was informed that the application performance was improved by a factor of two, he responded that, instead of saving computing time, the saved time could be used to get a higher-fidelity approximation of the problem being solved.

Scientists will not sacrifice maintainability for modest performance improvements. They are not interested in applying machine-specific performance tuning because they will likely lose the benefits of their effort when they port to the next platform. Moreover, they will end up with a less understandable code [33].

The automation of applying the optimizations and exploring the search space is interesting in many ways. It could popularize the performance gains delivered by current and future architectures that only experts can achieve today. It could also improve productivity during the development process. By making optimization easier and accessible, it could be applied to the development of many other applications that are not big enough to justify a performance expert. Moreover, as Dennard scaling came to an end and no alternative has taken its place, the optimization of applications to improve performance will become even more important and necessary.

Auto-tuning has already been successful in improving performance for specific cases as presented by many research papers published over the years [2, 4, 6, 12, 13, 14, 15, 23, 34, 35].

Notwithstanding the success in specific areas and the knowledge of how to harness performance from current architectures, the use of auto-tuning has not been widely adopted. Besides the examples of ATLAS being used in MATLAB and SPIRAL to generate routines of Intel MKL libraries, to the best of our knowledge, no general-purpose auto-tuning systems are being used on production-level codes.

## 2.1 THE PROGRAM OPTIMIZATION SEARCH SPACE

One of the compiler technology goals is the effective translation of high-level abstractions provided by the programming languages to low-level code that makes efficient use of architecture components.

For instance, register allocation assigns program variables to CPU registers during code generation. Accessing operands in registers is much faster than accessing them from memory. However, the number of registers is limited, and their efficient use is of critical importance to overall performance. The register allocation can be reduced to the graph coloring problem, which in general is NP-complete, and requires the use of heuristics that, as have been shown, can produce efficient code. These heuristics are machine-independent, and, even though the number and organization of the registers available differ among architectures, they are general enough to work with different setups.

The development of data-flow analysis has enabled many machine-independent optimizations. The data-flow analysis derives information about the flow of data along program execution paths [25]. It is, for example, used to implement global common sub-expression elimination, find and remove code that does not affect the program output, and replace expressions that always evaluate to the same constant at compile time.

We divided the optimizations according to 3 classes of machines: uniprocessors, shared-memory architectures, and distributed-memory architectures.

In some cases, the same optimization can be applied to different classes of machines with different granularity. For example, many optimizations that enable parallelism can be used to apply vectorization for uniprocessors, partition code into threads for shared-memory architectures, and create processes for distributed-memory architectures. There is a hierarchical relation among these classes, a distributed machine is composed of nodes that contain multiprocessors accessing the same memory and are comprised of multiple cores (uniprocessors).

### 2.1.1 Uniprocessor Optimizations

The optimizations presented here comprise optimizations to improve performance on the usage of a single processor (core). Notwithstanding that these optimizations are very effective and important to achieve high performance, they have been frequently given less importance due to larger interest in the use of parallel systems for big problems.

The relevance in selecting the appropriate data layout between row-major and column-major has already been mentioned. Another important and more complex example is the

decision about using *array of structures* or *structure of arrays*. In the array of structures, each structure has all of its fields close together when laid out in memory. In the structure of arrays, each field is represented by one array where each element belongs to a different structure. This way, all elements of the same field from different structures are laid out in memory contiguously. The array of structures is more intuitive, but for problems that require reading all elements of the same field from multiple structures, using the structure of arrays is more efficient due to the use of coalesced memory accesses. The access of non-contiguous data requires the use of gather and scatter operations and can affect the vectorization efficiency as well as the memory subsystem by increasing spatial locality. Coalesced memory accesses, especially on accelerators, can substantially improve performance [36].

Loop unrolling can reduce the instruction branch overhead as it enables the execution of more instructions per branch. However, if the number of iterations is small, which is usually only known at runtime, the overhead of always executing the tail section required for correctness in case the number of iterations are not multiple of the unrolling factor can overcome the benefits of reducing the number of branches.

Instruction parallelism is vital to keep the processor occupied. Therefore, the dependent instructions should be distant from the source instruction at least equal to the latency of the source one. To increase the instruction parallelism, one technique is to unroll the loop body statements and reorder the statements so that no instructions need to wait for their source's execution. Generally, the memory reads are placed as early as possible in the code to hide their latency.

Loop interchanging is perhaps the single most important loop transformation, according to Wolf [37]. It switches the inner and outer loop positions. A sequential nested loop can only be converted to the parallel form if any of the loops carry no dependence relations. If the goal is vectorization and one loop carries all the dependence relations, the interchange of that loop with the outermost one allows the other inner loops to be executed in parallel and vectorized. In case the goal is the parallelization of a loop nest using threads (shared-memory optimization), the opposite could be considered where each thread executes one iteration of the outer loop (no dependence relations) at a time, and all the iterations of the inner loop would be executed sequentially by each thread respecting the dependence relations.

Loop skewing changes the iteration space by adding the outer loop index to the inner loop index changing the dependence distances. It can improve parallelism and enable other transformations. Tiling reschedules the iteration space such that the iterations are executed in blocks, which improves data locality and uses the cache more efficiently. The tiling can be performed on many levels: from the last level cache up to the registers.

A single loop can be divided into two or more loops through loop fission or loop distri-

bution. Loop fission can break up a large loop into a smaller one that could be useful for architectures with minimal instruction cache. It can also improve the usage of the data cache by transforming a loop that refers to many different arrays into multiple loops, each accessing only part of the arrays.

The opposite of loop fission is loop fusion. Fusing loops can improve data locality if they refer to the same data. In the same way, a bigger loop body may allow common subexpression elimination as well as better instruction scheduling. On the other hand, it increases register pressure and reduces instruction locality. Fusion is only legal if it preserves all the dependence relations.

Sometimes one optimization is necessary to allow another one to proceed. For instance, loop reversal changes the order of the iterations by executing them backward. Thus, it negates the dependence distance for the loop and can be used to allow loop fusion where it might otherwise fail.

The organization of the program in procedures is essential for code maintenance. A procedure call, however, incurs overhead because it needs to, at procedure entry and exit, save and recover the register states from memory. One way to avoid the procedure call is by using inlining. The body of the procedure is copied to the place where the procedure or function was called. Inlining considerably increases the code size, though, and might lead to performance problems for architectures with a small instruction cache.

Memory latency and bandwidth are increasingly becoming the bottleneck as the processors get faster. In order to avoid the processor waiting for the data to arrive, the data can be requested long before it is needed, and other computations can be carried out in the meantime. Data prefetching prevents the processor from becoming idle while waiting for memory requests. Nonetheless, if the data is loaded long before the use, it will occupy resources, such as registers, unnecessarily and may even slow down the application. Therefore, prefetching should be carefully used [38].

Arrays referred to by different names are aliased if the allocated memory regions to which they point overlap. The aliasing prohibits the compiler from performing optimizations because it may be referring to the same memory location when writing and reading through different array variables. The compiler aborts efficient instruction scheduling, software pipelining, and vectorization when there is no guarantee that the arrays are not aliased. There are ways to let the compiler know that the arrays are not an alias to each other. These options, however, are often compiler-specific and not portable.



### 2.1.2 Optimizations to Shared-Memory Architectures

After improving locality, vectorization, and other optimizations already mentioned, the optimized sequential code still needs to be further modified in order to take advantage of the parallelism available in current architectures.

In shared-memory architectures, to execute tasks in parallel, it is necessary to use special libraries (e.g., Pthreads [39]) or runtime systems (e.g., OpenMP [39], Cilk [40]). No compiler automatically generates a multithreaded version from arbitrary code, and programmer intervention is required. Even though they all try to make this task as easy as possible, it is still complex to parallelize an application.

As an example, OpenMP requires compiler support and allows adding parallelism to existing source code through directives. It makes it easy to execute loops in parallel and provides options to split the loop iterations among the queries statically or dynamically and in variable chunks. These parameters affect performance and depend on the characteristics of the loop body and the number of threads.

In the Cilk language, the programmer is responsible for defining tasks that the system can execute in parallel. The parallel tasks are exposed by invoking them using the *spawn* keyword. The programmer is also responsible for enforcing the dependencies among the *spawned* parallel tasks. The *sync* keyword is used to perform synchronization among them. In the example of parallel loops with OpenMP, the spawn and synchronization are placed implicitly by the compiler and the runtime system.

These challenges get even more complicated as the number of processing units increases, such as in many-core architectures like Intel Xeon Phi and Nvidia GPUs.

Not only parallelism has increased the burden on programmers developing applications for shared-memory architectures. In cache-coherent NUMA systems, the pages are allocated on the memory attached to the processor on which the thread that first accessed it resides. Accessing data allocated on different NUMA nodes incurs a longer latency, and this first-touch policy affects performance when threads that first “touched” the page moves to another NUMA processor.

### 2.1.3 Optimizations to Distributed-Memory Architectures

Distributed-memory architectures are comprised of many nodes interconnected by fast networks. To efficiently use these architectures, the programmer needs to consider the overhead of communication between nodes. For some applications, it can be trivial, and, in some cases, there is no communication at all. For other applications, the overlap between



computation and communication is the only way to use these architectures efficiently.

The challenges of optimizing for distributed-memory architectures are similar to the ones faced by the shared-memory architectures. Load balance, data and task decomposition, and computation and communication overlap are critical to harnessing the computing power available. A popular programming paradigm for distributed architectures is message passing. In this paradigm, the processes exchange data through messages sent to each other. The Message Passing Interface (MPI) is the most widely used implementation of this paradigm.

Another paradigm is the *partitioned global address space* (PGAS). In PGAS, a global memory address space that is physically partitioned between the nodes is logically contiguous. Programming becomes similar to what is done for shared-memory architectures. The PGAS is the basis of Unified Parallel C and Coarray Fortran.

The optimizations for distributed-memory architectures are added on top of the aforementioned optimizations for uniprocessors and shared-memory architectures. Multi-level optimizations using MPI for the communication and synchronization among different nodes and OpenMP for intra-node parallelism is a common approach for demanding applications.

## 2.2 EXPLORING THE SEARCH SPACE

The search problem can be viewed as

$$\begin{aligned} & \text{minimize } f(x) \\ & \text{subject to } x \in \Omega. \end{aligned} \tag{2.1}$$

The function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  is the *objective function* that we want to minimize. The vector  $x$  is a vector of  $n$  independent variables:  $x = [x_1, x_2, \dots, x_n]^\top \in \mathbb{R}^n$ . The variables  $x_1, \dots, x_n$  are referred to as *search variables*, and the set  $\Omega$  is a subset of  $\mathbb{R}^n$  that is the *constraint set* or *feasible set*. The problem involves finding the vector  $x$  of the search variables over all possible vectors in  $\Omega$  that results in the smallest value of  $f(x)$ . The problems that look for the maximization of  $f(x)$  can be represented equivalently.

A point  $x^* \in \Omega$  is a *local minimizer* of  $f$  over  $\Omega$  if there exists  $\varepsilon > 0$  such that  $f(x) \geq f(x^*)$  for all  $x \in \Omega$  except  $\{x^*\}$  and  $\|x - x^*\| < \varepsilon$ . A point  $x^* \in \Omega$  is a *global optimizer* of  $f$  over  $\Omega$  if  $f(x) \geq f(x^*)$  for all  $x \in \Omega$  except  $\{x^*\}$ . If  $x^*$  is a global minimizer of  $f$  over  $\Omega$ , we write  $f(x^*) = \min_{x \in \Omega} f(x)$  and  $x^* = \operatorname{argmin}_{x \in \Omega} f(x)$ . If the minimization is unconstrained, we simply write  $x^* = \operatorname{argmin}_x f(x)$  or  $x^* = \operatorname{argmin} f(x)$  [41]. In other words, for a real-valued function  $f$ , the notation  $\operatorname{argmin} f(x)$  denotes the point  $x$  in the domain that minimizes the function  $f$ . If there is more than one such point, we select one arbitrarily.

Each search variable can be seen as a dimension to the search space. The more dimensions, the more difficult and complex it is to explore the space. The challenge posed by the high dimensionality of the search space is referred to as the *curse of dimensionality* and was originally introduced by Bellman [42].

Gradient, Newton's, conjugate gradient, and quasi-Newton are iterative methods that start with an initial point and then generate a sequence of iterates. The hope is that the sequence converges to a local minimizer. For this reason, the selection of the initial point is crucial as it is often desirable for it to be close to a global minimizer. These methods require first derivatives, and, in the case of Newton's method, the convergence, particularly the second-order convergence, depends on the existence of a second derivative.

The objective function of the program optimization search problem, however, operates on a discrete domain. That is,  $f : \mathbb{Z}^n \rightarrow \mathbb{Z}$ , and the vector  $x$  of  $n$  independent variables is  $x = [x_1, x_2, \dots, x_n]^\top \in \mathbb{Z}^n$ .

### 2.2.1 Global Search Algorithms

There are other methods global in nature in the sense that they attempt to search throughout the entire constraint set. These global search algorithms use only objective function values and do not require derivatives. They are, consequently, applicable to a much wider class of search problems. In some cases, they are even used to generate the starting points for the iterative methods. Some of the methods are also used for finite but enormous constraint sets. Examples of global search algorithms are Nelder-Mead, simulated annealing, particle swarm, and genetic algorithm.

The *Nelder-Mead simplex algorithm* is a derivative-free method that uses the concept of a simplex. A simplex is a geometric object determined by an assembly of  $n + 1$  points,  $p_0, p_1, \dots, p_n$ , in the  $n$ -dimensional space such that two points in  $\mathbb{R}$  do not coincide, three points in  $\mathbb{R}^2$  are not colinear, four points in  $\mathbb{R}^3$  are not coplanar, and so on. The simplex in  $\mathbb{R}$  is a line segment, in  $\mathbb{R}^2$  is a triangle, and in  $\mathbb{R}^3$  a tetrahedron; in  $\mathbb{R}^n$  it encloses a finite  $n$ -dimensional volume. At each iteration, the function  $f$  is evaluated at each point of the simplex. The point with the largest function value (for the function minimization process) is replaced by another point. The process of modifying the simplex continues until it converges toward the function minimizer.

*Simulated annealing* is an instance of a randomized search method that searches the feasible set of a search problem by considering randomized samples of candidate points in the set. Typically, we start the process by selecting a random initial point  $x^{(0)} \in \Omega$ . Then, we select a random next-candidate point, usually close to  $x^{(0)}$ . For any  $x \in \Omega$ , there is a set

$N(x) \subset \Omega$  such that we can generate a random sample from it. We usually think of  $N(x)$  as a set of points on the “neighborhood” of  $x$ . A simple version of the simulated annealing algorithm is shown in 2.1.

The main problem with the naive implementation of simulated annealing is that it may get stuck in a region around a local minimizer. One alternative to prevent this is to use a larger set of possible points to pick from. Another alternative is to “climb out” such a region by accepting a new point that is worse than the current one.

---

**Algorithm 2.1:** Simulated Annealing Algorithm.

---

```

1  $k \leftarrow 0$ ;
2 Select an initial point  $x^{(0)} \in \Omega$ ;
3 while no stopping criteria are satisfied do
4   Pick a candidate point  $z^{(k)}$  at random from  $N(x^{(k)})$ ;
5   if  $f(z^{(k)}) < f(x^{(k)})$  then
6      $x^{(k+1)} \leftarrow z^{(k)}$ ;
7   else
8      $x^{(k+1)} \leftarrow x^{(k)}$ ;
9   end
10   $k \leftarrow k + 1$ 
11 end

```

---

*Particle swarm optimization* (PSO) is another randomized search technique that was inspired by social interaction principles. In PSO, instead of updating a single candidate solution  $x^k$  at each iteration, we update a *population* (set) of candidate solutions, called a *swarm*. Each candidate solution in the swarm is called a particle. The PSO algorithm aims to mimic animals and insects’ social behavior, such as a swarm of bees or a flock of birds that apparently disorganized populations of moving individuals tend to cluster together as each individual seems to be moving randomly.

The PSO starts with an initial randomly generated population from the feasible set. Associated with each point in the population is a velocity vector. Then, the objective function is evaluated at each point. Based on this evaluation, a new population of points together with a new set of velocities is created. Each particle keeps track of the best position visited so far with respect to the value of the objective function. The particles “interact” with each other by updating their velocities according to their individual best as well as the global best. The usual stopping criteria consist of reaching a certain number of iterations or a certain value of the objective function.

The *genetic algorithm* is another randomized population-based search technique that has its roots in the principle of genetics. It starts with an *initial population* of points from  $\Omega$ ,

denoted  $P(0)$ . Then, the objective function is evaluated at points in  $P(0)$ . Based on this evaluation, a new set of points  $P(1)$  is created. The creation of  $P(1)$  involves certain “genetic” operations called *crossover* and *mutation*. The purpose of the crossover and mutation is to create a new population with an average objective function that is lower than that of the previous population. The process is repeated iteratively until an appropriate stopping criterion is reached.

### 2.2.2 Flat versus Hierarchical Search Spaces

In practice, the implementations available use two main ways to represent a space: flat and hierarchical. The *flat* representation forms a Cartesian product of every variable on the space, which has a multiplicative effect on the number of possibilities. On the other hand, the *hierarchical* representation results in a smaller number of choices because variables in independent sub-spaces do not form a Cartesian combination among them. The variables in a subspace are conditional on the selection of another variable. Only when this variable is selected, the others are evaluated. The hierarchical space can be flattened, but this may lead to worse space exploration performance as the variables on all subspaces are always considered. OpenTuner [23] uses a flat representation, whereas Hyperopt [43] uses a hierarchical representation.

For example, let a search space  $\mathcal{S}$  for program optimization of code on Figure 1.1a that consists of three search variables and their corresponding domains:  $x_1 \in \{\{1, 2, 3\}, \{1, 3, 2\}, \{2, 1, 3\}, \{2, 3, 1\}, \{3, 1, 2\}, \{3, 2, 1\}\}$ ,  $x_2 \in \{2, 4, 6\}$ , and  $x_3 \in \{Reorder, Unroll\}$ , where  $x_1$  represents the orders of a triple nested loop,  $x_2$  the unroll factor of the innermost loop body, and  $x_3$  the selection of either reordering, represented by  $x_1$ , or unrolling, represented by  $x_2$ , the loop nest.

In a flat representation, the search space consists of the Cartesian product of the domain of the three search variables. That is,  $\mathcal{S} = \{\{\{1, 2, 3\}, 2, Reorder\}, \{\{1, 2, 3\}, 4, Reorder\}, \{\{1, 2, 3\}, 6, Reorder\}, \{\{1, 2, 3\}, 2, Unroll\}, \dots, \{\{3, 2, 1\}, 6, Unroll\}\}$ , and  $|\mathcal{S}| = |x_1| \times |x_2| \times |x_3| = 6 \times 3 \times 2 = 36$  possibilities.

On the other hand, when using the hierarchical representation, it is possible to represent the constraint of variable  $x_3$  over variables  $x_1$  and  $x_2$ . It turns out that the search space now can be partitioned into two independent sub-spaces. That is,  $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2\}$ , where  $\mathcal{S}_1$  and  $\mathcal{S}_2$  represent the domain of  $x_1$  and  $x_2$ , respectively. Now,  $|\mathcal{S}| = |x_1| + |x_2| = 6 + 3 = 9$  possibilities. The hierarchical representation allows the partition of the search space according to the constraints among the variables, which results in representations with a smaller number of possibilities.

### 2.2.3 Implementations

OpenTuner is a framework for building domain-specific program autotuners. It includes a diverse collection of techniques that can handle many types of search spaces. Many search techniques run simultaneously, and the ones that deliver lower values of the objective function (e.g., deliver better-performing codes) are allocated more evaluations, whereas the ones that perform poorly are used less. The techniques included are differential evolution, variants of the Nelder-Mead and Torczon hill-climbers, various genetic algorithms, pattern search, particle swarm optimization, and random search. The techniques share results through the results database to benefit from improvements achieved by each other. This sharing occurs in technique-specific ways. For instance, evolutionary methods add results found by other techniques as members of their population. A meta-technique is the *root* technique with which the search driver interacts with. It is responsible for dividing the evaluations among the sub-techniques. Besides, multiple meta-techniques can be combined together.

The core meta-technique used by OpenTuner is the *multi-armed bandit with sliding window, area under the curve credit assignment* (AUC Bandit). It is used in addition to other simple meta-techniques, such as round-robin. The multi-armed problem is the problem of picking levers to pull on a slot machine with many arms, each with an unknown payout probability. The sliding window makes it consider only a subset of history when making decisions.

The AUC Bandit encapsulates a fundamental trade-off between *exploitation* (use of the technique that resulted in the best values of the objective function so far) and *exploration* (use the others, less performing techniques to gather information). It assigns each evaluation to technique,  $t$ , according to the formula

$$\arg \max_t \left( AUC_t + C \sqrt{\frac{2 \lg |H|}{H_t}} \right) \quad (2.2)$$

where  $|H|$  is the length of the sliding history window,  $H_t$  is the number of times the technique has been used in that window,  $C$  is a constant to control the exploration/exploitation trade-off, and  $AUC_t$  is the credit assignment term quantifying the performance of the technique in the sliding window.

The *area under the curve credit assignment mechanism* draws a curve looking at the history for a technique and whether that technique has yielded a new global best. If the technique has yielded a new global best, an upward line is drawn, otherwise a flat line is drawn. The area under the curve (scaled to a maximum value of 1) is the total credit attributed to the

technique. This credit assignment mechanism is calculated through the formula

$$AUC_t = \frac{2}{|V_t|(|V_t| + 1)} \sum_{i=1}^{|V_t|} iV_{t,i} \quad (2.3)$$

where  $V_t$  is the list of uses of  $t$  in the sliding window history,  $V_{t,i}$  is 1 if using technique  $t$  the  $i$ th time in the history resulted in a speedup, otherwise 0.

Hyperopt implements multiple *hyperparameter optimization algorithms* that take as input the search space and an experimental history  $H$  of values of the objective function and returns suggestions for which configuration to try next. Hyperopt implements a tree-structured Parzen estimator (TPE), adaptive tree-structured Parzen estimator (ATPE), random, and simulated annealing algorithms. It is also possible to use a mix of these techniques by telling the percentage of evaluations given to each of the techniques on the mix.

Bayesian optimization methods can be differentiated by their *regression models* and *acquisition functions*. Typically, a probabilistic regression model  $\mathcal{M}$  is initialized using a small set of samples from the domain  $\mathcal{X}$ . New locations within the domain are selected by an acquisition function  $\mathcal{S}$  that uses the current model as a fast surrogate for the expense objective  $f$ . Each suggested function evaluation produces an observed result  $y_i = f(x_i)$ . This result is appended to the historical set  $\mathcal{D} = \{(x_1, y_1), \dots, (x_i, y_i)\}$ , which is used to update the regression model for generating the next suggestion. The initialization sampling strategy is an important consideration for these methods. Viable approaches include random and quasi-random sampling [44].

TPE is a probabilistic regression model that creates two hierarchical processes,  $\ell(x)$  and  $g(x)$ , acting as generative models for all search variables [45]. These process model the domain variables when the objective function is below and above a specified quantile  $y^*$ ,

$$p(x|y, \mathcal{D}) = \begin{cases} \ell(x) & \text{if } y < y^* \\ g(x) & \text{if } y \geq y^*, \end{cases} \quad (2.4)$$

where  $\ell(x)$  is the density formed by using evaluations  $\{x^{(i)}\}$  such that objective function value  $f(x^{(i)})$  was less than  $y^*$  and  $g(x)$  is the density formed by using the remaining observations. The tree structure preserves the specified conditional relationships, while other variable interdependencies may not be captured. TPE naturally supports domains with specified conditional variables.

The acquisition function defines a balance between exploring new areas in the objective space and exploiting areas already known to have favorable values. Expected Improvement

(EI) is the one used by TPE. It defines the nonnegative expected improvement over the best previously observed objective value at a given location  $x$ .

## 2.3 OUR APPROACH TO THE PROGRAM OPTIMIZATION PROCESS

We rely on an approach that decouples the performance expert role from the application expert role (separation of concerns). The *baseline version* defined by the developer should be as readable as possible and avoid any platform- or compiler-specific optimizations. The application of the transformations to the baseline is controlled from an external file in what we refer to as *optimization programming*. This approach allows the use of architecture-specific optimizations while keeping the code maintainable in the long term. A database of different variants for different architectures (and possibly different problem sizes and other input parameters) is also maintained.

There have been several tools to ease the burden of the optimization process on the programmer. They commonly require manual refactoring [2, 4, 17], and provide little control over the steps being carried out. They also are not prepared to coexist with other tools and cannot be incrementally adopted [33].

In the following list, we describe what we believe are the requirements for making auto-tuning more accepted in applications. These requirements are based on our experiences with several applications, and in particular, with a large, complex multi-physics application, being developed as part of the Center for the Exascale Simulation of Plasma-Coupled Combustion [46]:

1. A straightforward, clean version of the code that is understandable by the computational scientist and program developer. This is the *baseline* code and is the one that may be modified by the code developers.
2. The code should run in the absence of any tool so that the developers are comfortable that their code will run even if the optimization system fails for some reason.
3. A clean way to provide extra semantic information is needed; for example, indicate that a loop is short or long or that a set of functions is always called in the same order.
4. Code must run with good performance on multiple platforms and architectures, at least as long as the same algorithm is appropriate.
5. Because in practice, code tuning and optimization are difficult to make fully automatic, there needs to be a way for a performance expert to provide additional, possibly target-specific, information about optimizations.

6. Because auto-tuning can be expensive (since many candidates may need to be examined), the system must store the results of the auto-tuning step(s) and use previously found optimized code whenever possible.
7. Changes to the baseline code should ensure that “stale” optimized versions of the code are not used and preferably replaced by updated versions.
8. Hand-tuned optimizations should be allowed.
9. Using (as opposed to creating) the optimized code *must not* require installing the code generation and auto-tuning frameworks, as these are often difficult to install and use on many platforms.
10. The system should make it possible to gather performance data from a remote system, permitting the framework to run on a system on which the auto-tuning and code transformation tools can be installed.

From these requirements, we made the following design decisions:

- Use of annotated code, written in C, C++, or Fortran, with high-level information that marks regions of code for optimization (addresses 1 and 2).
- Use annotations that only cover high-level, platform-independent information (addresses 3).
- Maintain platform and tool-dependent information (e.g., loop-unroll depth) in a separate *optimization file* (addresses 5).
- Maintain a database of optimized code, organized by target platform and other parameters (addresses 4 and 6).
- Maintain, in the database, a hash of the relevant parts of the code for each transformed section, and this hash is confirmed before making use of a previously stored auto-tuned version (addresses 7).
- Allow the insertion of hand-tuned versions of the code into the database to be used by the system in the same way that code generated by the auto-tuning step (addresses 5 and 8).
- Separate the steps of determining optimized code and populating the database from extracting code from the database to replace labeled code regions in the baseline version (addresses 9).



- Provide some support for running variants on a remote system, meaning that the full system need not be installed on the target system; this is especially important when the target is a supercomputer (addresses 9 and 10).
- Allow the inclusion and use of a preferred, custom version. For some applications and libraries, there is already a preferred, hand-optimized version (e.g., many of the kernel operations in PETSc have manually unrolled loops [47, 48, 49]). A small change to our approach accommodates this by placing the *baseline* version into the database; this is the version used by the auto-tuning tools (addresses 2).

Not included above is the integrated support for debugging code; that is, methods that relate the auto-tuned code directly to the original baseline code in a way that can be presented by a debugger. While desirable and beneficial, we do not view this as essential, in part because, at high levels of optimizations, compilers often perform complex code transformations that are not well-reflected in the source code that a debugger may present. While this does complicate debugging, it is not a new problem. In fact, because our system stores both the baseline code and the transformed code (created through source-to-source transformations), it should be *easier* to debug code in this approach than relying on transformations handled entirely within a compiler.

We implemented these design decisions in Locus [50, 51]. Locus is a semi-automatic approach to assist performance experts and code developers in the performance optimization process of programs developed in mainstream programming languages (C, C++, and Fortran). The system is non-prescriptive, which means that if none of the optimizations can be applied or improve performance, the baseline (original version) is used instead.

Regions of interest in the baseline version are marked and given an identifier. The optimization program uses this identifier to specify where to apply each transformation. Multiple regions with the same identifier will receive the same optimization steps (not necessarily generating the same resulting code).

Locus combines expert knowledge with empirical search, automates much of the optimization process, and gives total control to the developers. The system defines an interface to use external transformation and search modules. The idea is to have a collaborative environment where existing modules can be integrated into a single system. This enables the comparison among modules and the selection of the one that generates the most performant code.

Locus does not require the installation of any specific module. Only the ones used require installation. The full toolset integrated with Locus, however, requires a long and cumbersome installation process that is hard to replicate in all target machines. Therefore, the Locus

system uses a database of platform-specific variants that separates the code generation from the uses of the generated optimized code. It also supports a remote empirical search in which the driver that explores the search space and the code generation is executed on a different machine than the one where the variants are assessed.

## 2.4 SUMMARY

In this chapter, we discuss the complexities faced by the developers when optimizing a program. We describe widely used platform-specific optimizations for different classes of machines. We also present algorithms for exploring the search space automatically. A detailed description of our approach to the program optimization process is given at the end.

## CHAPTER 3: THE SYSTEM DESIGN AND IMPLEMENTATION

The Locus system design and implementation embody our philosophy about the program optimization process (Section 2.3). A summary of our approach is as follows: a clean version of the code should be defined, this clean version should be executable in the absence of any optimization tool; extra semantic information about the application and the target platform are provided in a separate file using a specialized language — we use the Locus language for the work reported here; a database is used to save the steps to generate and the resulting code of the best candidates for each platform; the consistency between the clean version and each of the best candidates on the database must be guaranteed.

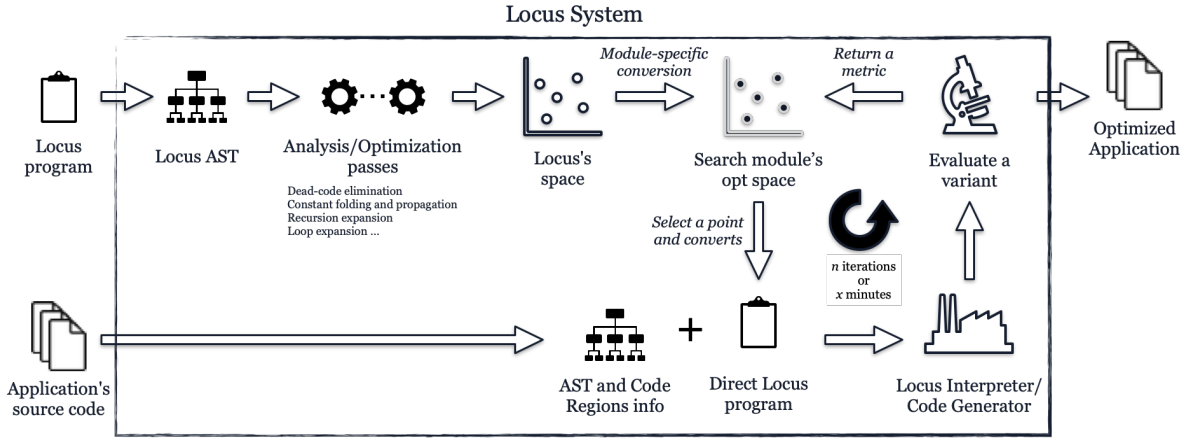


Figure 3.1: Locus workflow depiction.

The granularity of optimization, according to Smith [52], depends on the size and semantics of the program component being optimized. Traditional static compilation applies optimizations to programming language constructs, such as: within basic blocks, loop nests, or procedures. Locus leaves the decision of where to apply optimizations to the user by allowing the definition of arbitrary code regions as long as the transformation modules used are able to operate at the chosen granularity.

The system has an interface to integrate optimization and search modules. This pool of modules allows the comparison and selection of the best module for each specific context. There is no one-size-fits-all module solution. The transformation modules have restrictions on the source-code that they are able to transform, and can only transform specific types of language constructs. The chosen search module carries out the empirical search exploring spaces that are very hard to model accurately. The selection of the best search module highly depends on the input space, but it is still hard to predict which search module is the best because performance can be a complex function of many parameters.

The searching process could happen offline, once per architecture/application pair, at runtime, or in some combination. The cost of an offline search can be amortized over many uses, while a runtime search can benefit from information only available at runtime. Executing the search process at runtime, however, incurs substantial overhead. Locus limits itself to choose from the database the best candidate implementation for the given input and architecture during deployment. Although we focus on an offline search process in this work, Locus can be easily extended to carry out the search process during runtime. A hybrid approach could be useful when no implementation optimized for similar input and architecture is found on the database. The combination of searching offline, saving information and best implementation in the database, and running a search guided by the offline results is what we consider the best solution. The hybrid approach would work as follows: the database of optimized implementations would be consulted first, a “satisfactory” implementation is used if found. Otherwise, a search process is invoked. The database is then used to provide the initial configurations of the search process. Using an initial configuration that performed well with similar input and architecture, instead of a random start, often helps the search to converge faster to the best result. The results from the runtime search are fed back to the database to be used by future searches in a similar context.

Locus has two possible workflows: *direct* and *search*. The *direct* workflow applies one sequence of transformations and generates one candidate implementation. The *search* workflow applies when the Locus program contains search statements and typically involves multiple optimization sequences and candidate implementations that must be evaluated to select the best.

Figure 3.1 presents the search workflow of the Locus system. There are two inputs to the system: the application’s source code and the optimization program (Locus program). The former is parsed and translated into an internal representation that includes an abstract syntax tree (AST) and code region data structure. The latter is translated into an internal AST that represents the search space. Then, it goes through the optimizations to remove any unnecessary code, especially the unnecessary search statements. The search space is then translated from Locus notation into the chosen search module’s notation. An iterative search process can now start for  $n$  iterations or for a time limit: the search module selects one point from the space; this point is represented by a data structure that contains information of each search variable on the space. The search variables information replaces their respective search statements on the Locus program AST. The replacement of the search statements results in a *direct* Locus program. Then, the Locus interpreter executes the *direct* Locus program and generates a candidate implementation. The candidate implementation is compiled, executed, and evaluated. The evaluation returns a metric that is fed back to the search module to

assist in selecting the next point. For efficiency, during the search process, we do not actually generate a file containing a *direct* Locus program. Instead, its AST representation is directly fed to the interpreter.

The system executes all the optimization sequences defined using `CodeReg` (more information in Chapter 4) in the Locus program, whose name matches any code region label in the source-code. If no code-region label in the source-code matches the name of an optimization sequence, that optimization sequence, including the search statements defined on that sequence, is ignored for the Locus system execution.

In the remainder of this chapter, we present in more detail the interface to integrate optimization and search modules, and about the database for variants and results.

### 3.1 INTERFACE WITH THE TRANSFORMATION MODULES

One significant challenge in developing the system was to integrate different and unrelated transformation modules. Although the current implementation only deals with source-to-source transformations, it is still necessary to modify the source code (e.g., insert pragmas, labels) before invoking the transformations. The integration programmer is responsible for implementing the modifications to direct the modules to where on the source code and how the transformation should be carried out. After the transformation, the resulting source-code must also be translated back to the internal representation (AST and code region data structure). This internal representation is used as an intermediate representation between the transformations so that the transformed code can be fed from one module to another.

Locus could be extended to optimize code by other transformations than source-to-source, which we further discuss as future work in Section 9.1.

A very common workflow for the integrated modules is:

1. *mark* the code region according to the module specification;
2. *unparse* the code (generates source code);
3. *apply* the optimization;
4. *parse* the code again and *translate* it back into Locus internal source-code representation (abstract syntax tree and code region data structure).

Several transformations were integrated using this sequence. A wrapper function for each translator implements it using the system interface. Before calling the transformation module, the abstract syntax tree is modified to mark the code region that the module is

supposed to optimize. These marks (e.g., labels, pragmas) depend on the module integrated. After the optimization has finished, the code regions must be identified again as the result is parsed back into the internal abstract syntax tree. The interface is comprised of operations to modify the internal abstract syntax tree, such as:

- `replaceCoregWithPragma(·)`: replaces a Locus code-region annotation (e.g., `#pragma Locus loop=11`) with a *pragma*. The pragma’s arguments and position on the code are used by the transformation. For instance, the interchange transformation from the module `RoseLocus` is invoked by preceding a loop nest with `#pragma interchange 3,2,1`. The loop statement following the pragma is the loop that the module should transform. The argument `3,2,1` represents the new order of the loops in the nest. A sequence of numbers separated by comma starting from 1 is used to represent the loop nest;
- `replaceCoregWithLabel(·)`: replaces a Locus code-region annotation (e.g., `#pragma Locus loop=11`) with a label. A label is a name followed by a colon (e.g., `loop1:`) in C, C++, and Fortran. The label position on the code is often important. For instance, a label is used to mark the loop to be transformed by loop tiling from the `Pips` module;
- `addPragma(·)`: inserts a pragma preceding a statement that is inside a code region;
- `addLabel(·)`: inserts a label preceding a statement that is inside a code region;
- `replacePragmaWithCoreg(·)`: replaces a pragma with the original Locus code-region annotation. This is the inverse of the `replaceCoregWithPragma(·)`. The code-region information is added back to the transformed code to be used by the subsequent transformations;
- `replaceLabelWithCoreg(·)`: replaces a label with the original Locus code-region annotation. This is the inverse of the `replaceCoregWithLabel(·)`. The code-region information is added back to the transformed code to be used by the subsequent transformations;
- `removePragma(·)`: removes a pragma that precedes a statement;
- `removeLabel(·)`: removes a pragma that precedes a statement.

There are four collections of transformation modules currently available: `Pips`, `RoseLocus`, `Pragmas`, and `BuiltIn`. Table 3.1 presents the transformations available on each module. Transformations may receive a reference to a loop or statement through a string representing

hierarchical indexing (see Section 4.3), an integer for accessing a loop from a perfectly nested one (outermost is 1), or a loop reference to the AST representation of the source code. Next, we discuss how they interface with the system and how to use their optimizations.

### 3.1.1 Pips

A source-to-source compilation framework for analyzing and transforming C and Fortran 77 programs [53], Pips has a Python interface for invoking the optimizations. The loop optimizations are invoked on loops marked with labels. Locus adds to the code region a label and removes the pragma that originally identified the region using the `replaceCoreg-WithLabel(·)` operation. The changed AST is unparsed in a temporary file, and Pips is invoked. After finishing the optimization, the resulting code is read and parsed, and its AST and code region data structure are rebuilt.

The loop transformations available from Pips in Locus are: unrolling, tiling, *GenericTiling*, fusion, and unroll-and-jam. Loop tiling assumes a perfectly nested loop and receives as an argument a list of tiling factors for each loop in the nest.

Tiling is an effective optimization to improve data locality and expose parallelism, unleashing the potential of hierarchically organized systems. Loop nests can be tiled hierarchically to fit the target machine’s organization, and the shape and size of tiles can significantly affect locality, intertile communication, and parallelism.

The representation of loop iteration spaces as polyhedron enables the representation of loop transformations as linear transformations. The indices of a loop of depth  $d$  are represented as  $H = [i_1, \dots, i_d]^\top$ , and the values assumed by the indices using a system of affine inequalities as  $AH \leq b$ , which represents the iteration space as a polyhedron delimited by hyperplanes. A matrix  $T$  translates each point of the source iteration space with indices  $H$  onto a point on the target space with indices  $H' = [k_1, \dots, k_d]^\top$ . That is,  $TH = H'$ . The resulting polyhedron representing a transformed iteration space becomes  $AT^{-1}H' \leq b$  [37].

*GenericTiling* accepts a matrix representing the tiling transformation that allows advanced tiling strategies. Table 3.2 presents the tiling matrix,  $T$ , for four tiling schemes: square, diamond, skewing-1, and skewing-2 [54, 55].

Dependencies are partial order requirements that must be respected between loop iterations. They determine which transformations are valid. *GenericTiling* accepts *checkvalid* to check whether the tiled code respects the dependencies. Two extra parameters are also accepted to enhance parallelism: *tile direction* ( $td$ ), which specifies the scanning directions of the tiles, and *local tile direction* ( $ld$ ), which specifies the scanning directions of the iterations inside each tile. The *tile direction* values are `tp` that uses the hyperplane direction

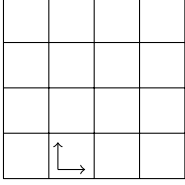
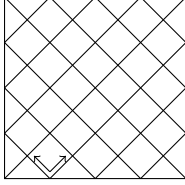
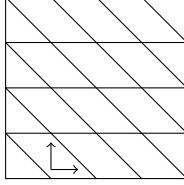
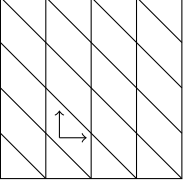
Table 3.1: Transformation Modules available on Locus. *Loop* argument can be a string for hierarchical indexing (see Section 4.3), an integer representing the loop on the nest (outermost loop is 1), or a reference to the AST.

Module	Transformation	Arguments Description
Pips	<code>GenericTiling(loop, factor [,td, ld, checkvalid])</code>	<i>factor</i> : list of lists representing tiling matrix; <i>td</i> : { $tp^d$ , $ti$ }; <i>ld</i> : { $li^d$ , $ls$ , $lp$ }; <i>checkvalid</i> : { $true^d$ , $false$ }.
	<code>Tiling(loop, factor)</code>	<i>factor</i> : list for tiling diagonal matrix.
	<code>Unroll(loop, factor)</code>	<i>factor</i> : integer.
	<code>UnrollAndJam(loop, factor)</code>	<i>factor</i> : integer.
	<code>Fusion(loop)</code>	
RoseLocus	<code>Unroll(loop, factor)</code>	<i>factor</i> : integer.
	<code>UnrollAndJam(loop, factor [,checkvalid])</code>	<i>factor</i> : integer; <i>checkvalid</i> : { $true^d$ , $false$ }.
	<code>Stripmine(loop, factor [,checkvalid])</code>	<i>factor</i> : integer; <i>checkvalid</i> : { $true^d$ , $false$ }.
	<code>Tiling(loop, factor [,checkvalid])</code>	<i>factor</i> : list representing tiling factors for each loop in the nest; <i>checkvalid</i> : { $true^d$ , $false$ }.
	<code>Interchange(order [,checkvalid])</code>	<i>order</i> : string or list representing loop nest order, outermost loop is 0; <i>checkvalid</i> : { $true^d$ , $false$ }.
	<code>Distribute(loop [,checkvalid])</code>	<i>checkvalid</i> : { $true^d$ , $false$ }.
	<code>LICM([loop])</code>	
	<code>ScalarRepl([stmt])</code>	
Pragma	<code>IsDepAvailable([loop])</code>	returns <code>true</code> or <code>false</code> .
	<code>OMPFor(loop, schedule, chunk)</code>	<i>schedule</i> : { <code>static</code> , <code>dynamic</code> , <code>guided</code> , <code>runtime</code> , <code>auto</code> }; <i>chunk</i> : integer.
	<code>Ivdep([loop])</code>	
	<code>Vector([loop])</code>	
BuiltIn	<code>ListOuterLoops([loop])</code>	returns references to outermost loops.
	<code>ListInnerLoops([loop])</code>	returns references to innermost loops.
	<code>LoopNestDepth([loop])</code>	returns an integer for the nest depth.
	<code>IsPerfectLoopNest([loop])</code>	returns <code>true</code> or <code>false</code> .
	<code>Altdesc(source [,stmt])</code>	<i>source</i> : file path with the content to be inserted; <i>stmt</i> : string for hierarchical indexing, defaults to replace the entire code region.

<sup>d</sup> Default value.



Table 3.2: Tiling schemes.

	Square	Diamond	Skewing-1	Skewing-2
				
Tiling Matrix	$\begin{pmatrix} x & 0 \\ 0 & x \end{pmatrix}$	$\begin{pmatrix} x & -x \\ x & x \end{pmatrix}$	$\begin{pmatrix} x & -x \\ 0 & x \end{pmatrix}$	$\begin{pmatrix} x & 0 \\ -x & x \end{pmatrix}$

associated with orthogonal directions, and `ti` that uses the colinear direction to the partitioning vectors. The *local tile direction* values are `li` that uses the original basis, `ls` that uses one of the partitioning vectors, and `lp` that uses the hyperplane direction associated with orthogonal directions. The interested reader may consult Pips documentation [56] for a thorough explanation.

### 3.1.2 RoseLocus

We implemented our own set of annotation-based source-to-source loop transformations using the Rose infrastructure [57]. Loop unrolling, loop tiling, loop interchange, strip-mine, loop distribution, unroll-and-jam, loop invariant code motion, and scalar replacement are available. The interface with RoseLocus is similar to that with Pips. However, instead of labels, pragmas are added to the code regions.

We have also implemented `IsDepAvailable()` to check whether the transformations' dependence analysis can compute the data dependencies of the source code. The data dependencies are used by some transformations to check whether the transformation would be valid. If `IsDepAvailable()` returns *false*, we avoid invoking transformations whose result must be verified by the dependence analysis (i.e., *checkvalid* argument is *true*).

If no `loop` or `stmt` argument is provided to `IsDepAvailable()`, `ScalarRepl()`, and `LICM()`, the transformations are carried out on the first loop or statement in the code region.

### 3.1.3 Pragma

Pragmas and directives are easy ways to execute loop iterations in parallel or provide hints to the compiler to help vectorization. Although they are easy to use, the selection of pragmas and parameters depends on the application and underlying architecture. Besides, there is no standard for directives across compilers. In our effort to achieve performance portability, we believe that pragmas and directives should not be added directly to code.

Compiler-specific pragmas and directives can be added using the module `Pragma`. In the experiments, `ivdep` and `vector always` are used for enhancing vectorization with the ICC compiler by calling `Ivdep(·)` and `Vector(·)`, respectively. If no `loop` argument is provided, the `ivdep` and `vector always` are inserted immediately before the first loop in the code region.

It is possible to execute loops in parallel by calling `OMPFor(·)` to insert `omp parallel for` pragma. `OMPFor(·)` accepts the schedule and the chunk size as parameters [58]. The *static* schedule divides the iterations into chunks of *chunk size* and assigns to threads in a round-robin fashion. The threads execute a chunk of *chunk size* iterations in the *dynamic* schedule and then requests for another chunk until no iteration remains to be assigned. The *guided* schedule works in the same way as the dynamic, however, the chunk size starts large and shrinks to the indicated chunk size as chunks are scheduled. The *auto* schedule delegates the decision to the compiler and/or runtime system. The *runtime* option uses the schedule and chunk size defined in OpenMP internal control variables.

### 3.1.4 BuiltIn

It contains functions to analyze or transform the source code by manipulating the internal AST. Functions to get information about loops are: `ListInnerLoops(·)` and `ListOuterLoops(·)` return a list with references to the innermost and outermost loops, respectively, from a code region; `IsPerfectLoopNest(·)` returns whether the loop is perfectly nested; `LoopNestDepth(·)` returns the depth of a loop nest. If no `loop` argument is provided, functions get information from the first loop in the code region.

`AltDesc(·)` is used to replace a statement in the code region with external code snippets. Its functionality is similar to having macros in the program. It is mostly used to incorporate hand-optimized kernels into a code region. If no `stmt` argument is provided, the entire code region is replaced by the content of the *source* file.

## 3.2 INTERFACE WITH THE SEARCH MODULES

Three functions must be implemented to enable the integration of a search module:

1. *convertOptUniverse()*: responsible for automatically converting the Locus search space of all code regions to the module’s search space. The space conversion usually includes converting search statements and defining parameters and options that will be used during the search process;
2. *search()*: responsible for starting the search process. The search can only start after the conversion is finished;
3. *convertChosenPoint()*: a conversion back to the Locus representation is necessary for each point chosen during the search. Point information is represented by a data structure that contains a selected single value for each search variable (each search variable is represented by a search statement on the Locus program) and is used to generate a *direct* Locus program. The *direct* Locus program can now be interpreted, and the transformations carried out.

Two modules were integrated to explore the search space: Opentuner [23] and Hyperopt [59]. Next, more details about each tool integration are provided.

### 3.2.1 OpenTuner

The search statements for *OR* blocks, *OR* statements, and *enum* are converted to OpenTuner’s *EnumParameter* (more information about search statements in Chapter 4). The optional statements are represented using the *BooleanParameter*. The other search statements have straightforward representations between the two spaces. For instance, the *permutation* is represented by *PermutationParameter*.

### 3.2.2 HyperOpt

The search statements for *OR* blocks, *OR* statements, numerical search statements, and *enum* are represented by the `randint` parameter. Hyperopt accepts the use of conditional search space. These conditional, hierarchical spaces are constructed in Locus by using nested *OR* blocks. The use of search statements dependent on other search statements is treated in the same way as in OpenTuner.

### 3.3 CORRECTNESS

The transformations available use dependence analysis to verify that the transformed code is correct. In some cases, the dependence analysis cannot guarantee correctness, but it can be turned off whenever the programmer is confident that the transformed code is correct.

### 3.4 CONSISTENCY CHECKING

Before applying an optimization sequence to an application’s code region, the system checks whether the code in that code region has changed. We assume that a change in the code region’s code means that the optimization sequence is outdated. To check whether the code region’s code has changed, we keep a hash key for each code region in the Locus database, and before the transformation or replacement, the hash keys are compared to the ones stored. If the keys do not match, we assume that the optimization sequences were planned for a different code region, which may result in incorrect code, and the system aborts. The user needs to update the hash keys in the Locus program if the optimizations steps are still applicable.

The hash function to generate the keys are language-dependent. The current function removes all the white-spaces characters and comments before generating the SHA-1 key.

Furthermore, during the search, each candidate implementation evaluated can be checked by custom a function to ensure it is valid. This custom function is provided by the user and usually application-dependent.

### 3.5 DATABASE OF VARIANTS AND RESULTS

During the optimization process, the programmer needs to deal with a relatively large amount of information. It includes information about the host, compiler, application, input details, and metrics for each candidate implementation generated and evaluated. A database is proposed to store information about the optimization process. The goal is to have important information easily accessible to facilitate performance analysis and the selection and use of the best candidates found during the empirical searches.

A database of platform-specific codes for different code regions that allows the use of pre-generated auto-tuned code is an important requirement for making auto-tuning more accessible. Locus allows using a direct optimization program to reuse results from previous empirical evaluations, making the optimization results accessible to non-experts. The

Locus database saves the optimization sequence and the resulting code for each candidate implementation evaluated during the empirical search.

Despite the system’s goal to facilitate the use of multiple transformation modules, in our experience (and especially the experience of our computational scientist users), the installation of the modules and their dependencies has shown to be complex and often not available for all systems. The users of a direct optimization program containing the optimization sequences would still need to install all the modules invoked in the direct program. However, the system also can generate a direct Locus program that loads from the database the best pre-generated auto-tuned code of each code region into the application source code, sparing the user from the complicated installation of all the modules needed to generate the best candidate implementations.

The database organizes information about the optimization process of an application. The process is commonly comprised of multiple independent empirical evaluations. The result of each evaluation is added to the database using a unique identifier.

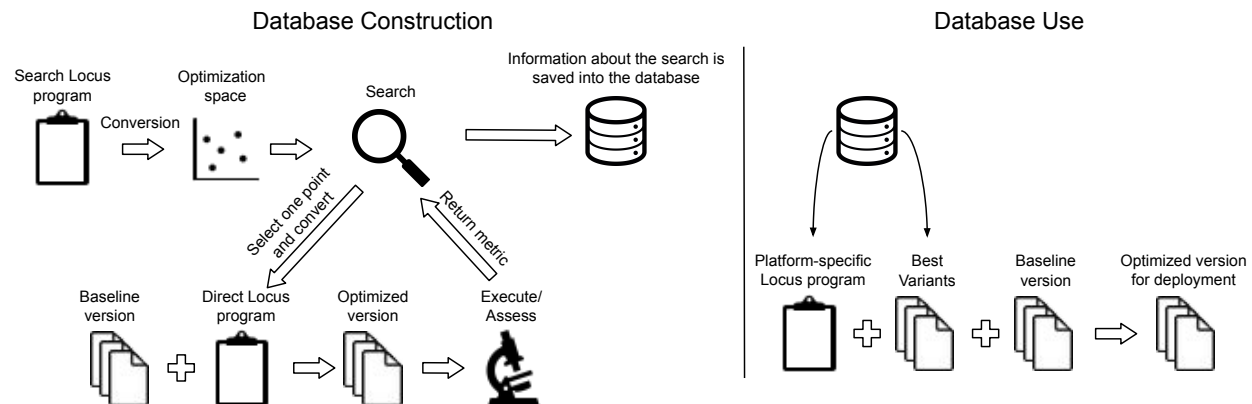


Figure 3.2: On the left-hand side, the steps to add platform-specific optimized candidates to the database during the search process. During the deployment, shown on the right-hand side, the best platform-specific candidates for each code region and a direct Locus optimization program that loads these variants are automatically generated.

Figure 3.2 shows the database construction (left-hand side) and database use (right-hand side). The empirical evaluation results are added to the database during the search process. It can also generates (or updates if there is one already) a direct Locus program that loads the implementations from the database.

The main purposes of the database are:

- **Information about the searches:** Because of the non-deterministic behavior of the search tools, it is often necessary to run multiple independent searches instead

of a single one containing a larger search space. For instance, when comparing the application’s performance across multiple compilers, the search will be guided to use the one that generates the fastest results. Therefore, if one compiler stands out in the process, the others will rarely be evaluated. This behavior is the correct one, but if one wants to evaluate the performance of multiple compilers, the only guaranteed way is to have one search for each compiler. The experimental results for each compiler are store in the same database for future comparison.

The database also stores experimental results carried out using different search modules. Besides, each variant evaluated is often executed multiple times, and all the metrics are saved in the database.

- **Reuse search configurations:** The execution of multiple independent searches opens the possibility of sharing best configurations when executing different evaluations among compilers, architecture, and search tools.

The best configurations for the same application can be extracted from the database and reused by future empirical searches. We claim that the proper selection of the initial configuration to be evaluated affects the search’s variability. The empirical search has converged faster even by reusing as initial configurations the fastest configuration found in another architecture. We revisit this in Chapter 6.6.

This feature is especially useful for independent evaluations of applications on the same architecture. Revisiting the compiler example described earlier, if we use one empirical search for the evaluation of the application using each compiler, reusing the best configurations from other searches ensures that all the compilers were invoked to the same set of fastest configurations, and the results are not skewed by the non-deterministic behavior of the search tools.

Chapter 7 presents the use of this feature to compare different compilers, search tools, and architectures.

- **Deployment:** The database saves for each application’s code region the candidate implementations evaluated empirically on each search. During the application’s deployment, the database is used to get each code region’s best variant. This amortizes search time among users and avoids a long and cumbersome installation process of the modules used by Locus to generate the variants. Besides, the installation of the modules is hard to replicate in all target machines.

### 3.5.1 Implementation

The database implementation follows the diagram presented in Figure 3.3. The diagram can be synthesized as follows:

- Each application is comprised of one or more code regions;
- Optimizing one application requires the execution of one or more search processes;
- Each search, in turn, uses a search space defined in a Locus program;
- The Locus program has the search space segmented by each code region;
- During the search, each configuration is materialized into a variant;
- Each variant execution is an experiment;
- Each experiment may have multiple metrics and are saved in the *Experiment Results* table.

Hashes are used to generate unique keys for the application’s code regions, the Locus programs, and configurations. The hash function is language-dependent; the current version removes all the white-space characters and comments before generating the SHA-1 key. Because it is based on the Locus program syntax, the hashing function used does not solve the problem of returning different keys for semantically equivalent optimization sequences. Therefore, minimal changes in the Locus program result in the generation of different keys. The results in the database linked to programs that are semantically equivalent but have different keys must be aggregated manually by the programmer.

The hash function for the code regions includes the file name they belong to, their code region label, and an index value from 0 to the number of code regions in the same source file. The index value is used to set apart code regions if there is more than one in the same file with the same label. Identical code regions that have the same label will receive the same sequence of optimizations, but their experimental results will be separated on the database. Different code regions can also have the same label and receive the same sequence of optimizations; the use of *pragma query* allows the customization of the sequence of transformations according to the source code (see Section 8.2).

Changes in the baseline version of the application may result in the candidates in the database incorrect. The database contains a hash of the baseline code region from which each candidate implementation was generated. The *BuiltIn.AltDesc* uses that hash to check

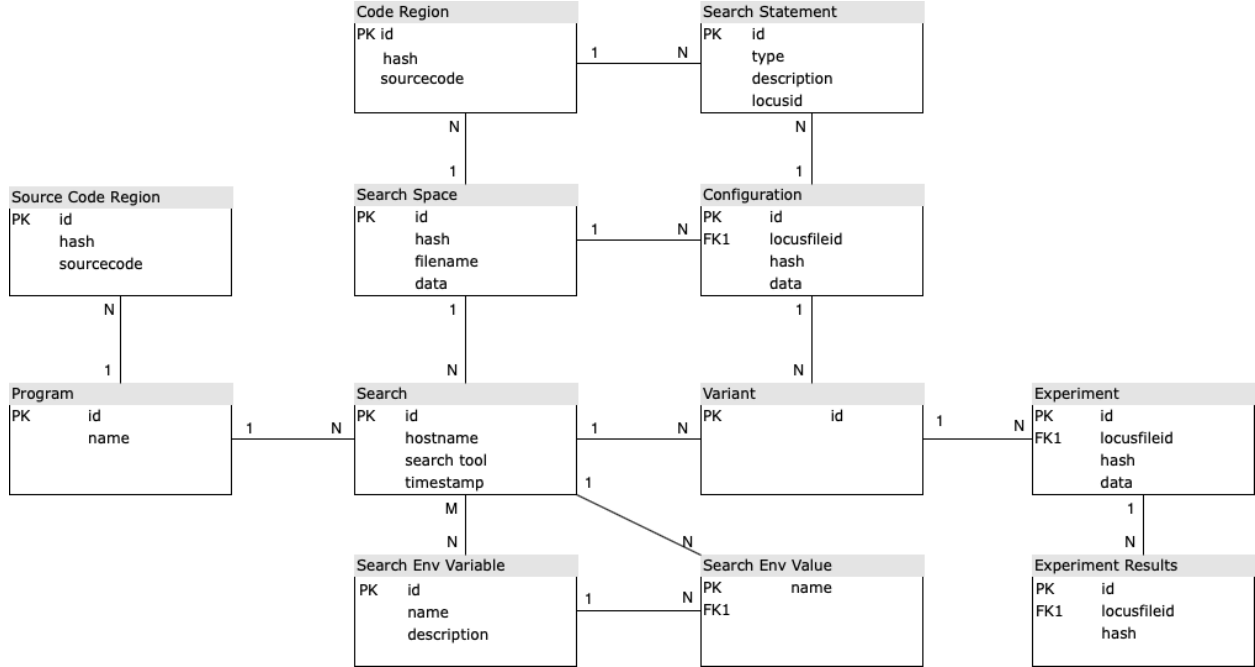


Figure 3.3: Database entity relationship diagram.

whether the current code, which the variant is being incorporated into, is the same as the one the variant was generated from.

The empirical evaluation is often dependent on input parameters (e.g., problem size, matrices shape). The system allows the inclusion of information about the search that better represents the context in which the variants were generated. This information is later used to load the proper variants that will be deployed for similar contexts. They are provided as *key,value* pairs using environmental variables and saved into the *Search Variables* and *Search Values* tables. For instance, the optimized code for matrix multiplication heavily depends on the shape of the matrices, which is one information that can be added as tags. During deployment, if the database does not have an optimized code that matches exactly the *key,value* requested, a heuristic can be used to get the most appropriate from the available ones.

### 3.6 SUMMARY

The Locus system consists of an approach to the program optimization process. Its main goal is to help the developer make complex and long-lived applications prone to perform faster on multiple platforms.

The system consists of interfaces to include search modules to conduct the empirical search



and transformation modules to modify the application's source code. A database is used to store the candidate implementations and the steps to generate them. It is also used to share with others the results from the empirical searches. The Locus system contains an important set of tools to generate faster, platform-specific code automatically.

## CHAPTER 4: THE LOCUS LANGUAGE

We devised a domain-specific language that allows the definition of an optimization sequence along with a search space. The programs developed in our custom language have optimization sequences intertwined with the search variables. Therefore, the search statements that represent the search variables are part of the optimization sequences’ programming logic. This results in a concise representation of search spaces combined with optimization sequences in what we call *optimization programming*.

The mixing of search spaces with the programming of optimization sequences is exclusive to Locus. In previous work, search spaces were defined apart from the sequences used to optimize programs. They define the search space in a meta-program to the optimization sequence representation, and the values selected during the search replace their respective placeholders in that representation [1, 2, 22, 34]. The reasons for this separation are two-fold: (i) the search technique accepts only search variables as input; (ii) the values assigned to the search variables from a search space are only known during the search process. Locus allows them to be defined together as it handles the search modules automatically, extracting the search space from the Locus program and representing it into the module’s notation.

### 4.1 OVERVIEW

The source-code is annotated to define regions on the source-code to be optimized (e.g., Figure 1.1a). These *code regions* can be loop nests, functions, or sequences of consecutive statements. An annotation contains a label that is referred to in the Locus program. An annotated code region can be optimized by creating a *CodeReg* construct using the same label defined in the annotation. The heading `CodeReg P`, where `P` is a name composed of letters, digits, and the underscore character (i.e., `<NAME>` on Figure 4.5), indicates that the optimization steps in the `<block>` that follows are to be applied to the regions labeled `P`.

To name optimization sequences that are not tied to any code region, there is the *OptSeq* construct. `OptSeq H(·)` defines a collection of transformations that can be invoked using `H(·)`. These collections of transformations can be reused and invoked by `CodeReg` and `OptSeq`.

The language accepts *pragmas* that are used to provide extra information for the search space translation and code generation. An important one is the *@pragma query* that tells the search space translator that the information provided by `OptSeq` calls in the subsequent

```

1  import "RoseLocus";
2  OptSeq printstatus(type) {
3    print "Tiling_selected:_" + type;
4  }
5  OptSeq Tiling2D() {
6    tileI = poweroftwo(2..32);
7    tileJ = poweroftwo(2..32);
8    RoseLocus.Tiling(loop="0", factor=[tileI, tileJ]);
9    return "2D";
10 }
11 OptSeq Tiling3D() {
12   RoseLocus.Tiling(loop="0", factor=[4,4,8]);
13   return "3D";
14 }
15 CodeReg matmul {
16   tiledim = 4;
17   tiletype = Tiling2D() OR Tiling3D();
18   printstatus(tiletype);
19   if (tiletype == "2D") {
20     RoseLocus.Unroll(loop=innermost, factor=tiledim);
21   }
22 }

```

Figure 4.1: Locus program example for optimizing a matrix-matrix multiplication.

line are static across the search process. These `OptSeq` calls are then executed before the translation to the search module's notation occurs, and the values returned replace the calls that originated them. This information may be used by search statements and useful for compiler passes to optimize the Locus program itself (Chapter 5 better explains the Locus program's optimization).

A transformation receives a reference to where in the code region to perform the transformation. References point to, for instance, a loop in a nest or a compound statement. The reference can be a parameter to an `OptSeq`, which could make it generic enough to be reused for optimizing different code regions.

The code region being optimized may change as the optimization sequence is being carried out. The user is responsible for making sure that the references passed to each transformation are consistent with the state of the transformed code at that point in the optimization sequence.

The Locus system ignores `CodeReg` definitions whose name does not match any label in the source code of the application being optimized. The absence of any code region in the source-code to be optimized by a `CodeReg` also results in the removal from the search space of any search statements defined in that `CodeReg`.

```

1  import "RoseLocus", "Pips";
2  dim=4096;
3  Search {
4    buildcmd = "make_clean_all";
5    runcmd = "./matmul";
6  }
7  CodeReg matmul {
8    perm = permutation([0,1,2], init=[0,2,1]);
9    RoseLocus.Interchange(order=perm);
10   tileI = poweroftwo(2..dim, init=512);
11   tileK = poweroftwo(2..dim, init=128);
12   tileJ = poweroftwo(2..dim, init=2048);
13   Pips.Tiling(loop="0", factor=[tileI, tileK, tileJ]);
14   tileI_2 = poweroftwo(2..tileI, init=256);
15   tileK_2 = poweroftwo(2..tileK, init=32);
16   tileJ_2 = poweroftwo(2..tileJ, init=32);
17   Pips.Tiling(loop="0.0.0.0", factor=[tileI_2, tileK_2, tileJ_2]);
18   {
19     None;
20   } OR {
21     tileI_3 = poweroftwo(2..tileI_2);
22     tileK_3 = poweroftwo(2..tileK_2);
23     tileJ_3 = poweroftwo(2..tileJ_2);
24     Pips.Tiling(loop="0.0.0.0.0.0.0", factor=[tileI_3, tileK_3, tileJ_3]);
25   }
26 }

```

Figure 4.2: Locus program for optimizing Matrix-Matrix Multiplication. The program applies loop interchange and a 2- or 3-level hierarchical tiling. It also provides the values to be used as the initial configuration for the search process.

## 4.2 EXAMPLES

Figure 4.1 is an example of a Locus program that tiles a reference version of matrix-matrix multiplication (Figure 1.1a). It shows a search space that includes two `OptSeq`: one, `Tiling2D()`, for tiling the two outermost loops and another, `Tiling3D()`, to tile the three outermost loops, which, in this case, is the whole loop nest. The transformation sequence that applies to the loop labeled `matmul` in Figure 1.1a must have header `CodeReg matmul`. Using the Locus `OR` statement, this sequence specifies that `Tiling2D()` and `Tiling3D()` will be used to create two different collections of points in the space of transformations. `Tiling2D()` will create 25 points on the space for the tiles of dimension 2 by 2, 2 by 4, ..., 2 by 32, ... 32 by 32, whereas `Tiling3D()` will create one additional point on the search space for a tile of dimension 4 by 4 by 8. When `Tiling2D()` is applied, the innermost loop is also unrolled after tiling. There are multiple tiling factor parameters in `Tiling2D()` represented using ranges of values. There are 31 sizes for each dimension, leading to  $31 \times 31$  possible tile shapes for `Tiling2D()`, whereas there is a single tile shape in `Tiling3D()`. The shape of a tile is a tuple representing the size of each dimension.

The optimization program in Figure 4.2 has a definition of a `CodeReg` with NAME as `matmul`. This program first changes the loop order according to the permutation on the

perm variable (e.g., from *ijk* to *ikj*) by calling `RoseLocus.Interchange`. Then, the loop nest is tiled twice by calling `Pips.Tiling` two times. In the end, there is an OR block that has the possibility of tiling again and generating a 3-level hierarchical tiling. Each tiling uses a range of values to cover different shapes; the best shape depends on the memory hierarchy, which is machine-specific.

### 4.3 SYNTAX AND SEMANTICS

A Locus program consists of the following structure:

- Preprocessor commands;
- `CodeReg` and `OptSeq` constructs;
- Variables;
- Statements and expressions;
- Comments.

Preprocessor commands include the `import` used to access optimization sequences defined in separate files. In a Locus program, a semicolon is a statement terminator. Comments start with `#`. A variable is a name given to a storage area that the program can manipulate. The name of a variable can be composed of letters, digits, and the underscore character but must begin with a letter or an underscore. Locus is case-sensitive and dynamically typed. Variables need not be declared or defined in advance. To create a variable, just assign it a value. A variable may be assigned different values of different types during its lifetime. The assignment operator `=` is used to assign values from right-side expression to left-side operand. In practice, a variable is a symbolic name to reference an object. Locus creates objects implicitly for each data structure and types it accepts. Data structures accepted are lists and tuples. Types accepted are for integers and floating-point numbers, and string literals (constants).

A code region definition consists of the keyword `CodeReg`, and a unique name followed by a body that consists of a block of statements. It contains no parameters nor return type. `CodeReg` is invoked by the system whenever there is a definition on the Locus program whose name is the same as an annotated code region on the source code.

An optimization sequence definition consists of the keyword `OptSeq`, a unique name, and parameters followed by a body containing a block of statements. Contrary to `CodeReg`, op-

timization sequences may contain parameters and return values. Since Locus is dynamically typed, no return type is provided.

The rest of this section provides more information about the language.

**Operators.** The arithmetic operators accepted are: addition ( $a + b$ ), subtraction ( $a - b$ ), multiplication ( $a * b$ ), modulus ( $a \% b$ ), power ( $a^b$  is  $a ** b$ ), division ( $a / b$ ), and floor division ( $a // b$ ).

The relational operators ( $a == b$ ,  $a != b$ ,  $a > b$ ,  $a < b$ ,  $a >= b$ ,  $a <= b$ ), logical operators ( $\&\&$  for and,  $\|$  for or, *not* for negation), and bitwise operators ( $\&$  for and,  $|$  for or,  $\sim$  for inversion,  $\wedge$  for xor,  $<<$  for left shift,  $>>$  for right shift) follow the same rules as in Python 3<sup>1</sup>.

**Search Statements.** Locus has multiple *search statements* to expose and define the optimization space. The search statements are:

1. OR Blocks;
2. OR statements;
3. Optional statements;
4. enum, integer, float, permutation, poweroftwo, loginteger, and logfloat statements.

An OR can be used between blocks of code (set of statements inside curly brackets (e.g., `{optA;} OR {optB;}))` and between statements (e.g., `transfA OR transfB;`). They are used to describe alternative optimization sequences. The statements or blocks are selected at run time, and it is not guaranteed that all possibilities will be evaluated; this decision depends on the search module.

In the same way, the Locus language has *optional* statements. Any statement (including OR statements) marked with a preceding `*` may or may not be executed and adds a dimension to the search space. The *OR* statements have precedence over the *optional* construct. The semantics of an optional statement is the same as having an OR statement in which one of the options is `None`. To keep these semantics, assignments cannot be optional statements.

A collection of values can be represented by `enum(<value>, ...)`, `permutation(<value>, ...)`, and `[integer, float, permutation, poweroftwo, loginteger, logfloat] (<min> .. <max>)`. In the search space, the enum exposes to the search all values received; it can be

---

<sup>1</sup><https://docs.python.org/3/library/operator.html>

used for specifying the performances of different compiler flags (e.g., `enum(-O2, -O3)`). The `permutation` exposes to the search all the permutations of a list. In our experiments, we used `permutation` to specify all orders of a loop nest as a parameter of the loop interchange optimization. The `poweroftwo` exposes to the search all values that are power of two on the given range. `integer` exposes to the search all the integer values in the range. Note that “..” is used to represent a range. The `integer` can be used to find the best dimensions when tiling a loop nest, and `poweroftwo` can reduce the search time as it covers fewer values for the same range. `loginteger` and `logfloat` expose to the search space all the scaled logarithm values to base 2 between the minimum and maximum values provided.

The arguments of these search statements can be another search statement. This is especially useful for defining constraints that depend on previously selected statement. For instance, in a sequence that tiles and then unrolls, the unroll factor is limited by the number of loop iterations, which in turn is defined by the block size selected using another search statement. If the block size was selected as 32, the search space for the unroll factor should be constrained by that value. The use of constraints reduces the size of the space and potentially speeds up the search process.

**Data Structures.** Locus can represent two kinds of data structures: lists and tuples (in a similar fashion to Python). The lists are represented by enclosing elements within square brackets, and tuples by enclosing elements within parenthesis. Lists are mutable and tuples are immutable. A new tuple can be generated from an existing one using the `+` operator with another tuple and assigning the result to a variable that will reference the created tuple. An element can be appended to an existing list by using the `append(v)` method. An element can be removed from an existing list by using the `remove(v)` method.

**Types.** Besides lists and tuples, there are two other types of constants: numbers and strings. As with tuples, these types are immutable. Strings are surrounded by double-quotes. Strings can be concatenated using the `+` operator and assigning the result to a new variable. Signed integers and floating-point real values are the types of numbers accepted.

**Built-in Functions.** The Locus compiler provides the following functions:

- `seq(start, stop[, step])`: generates a sequence of numbers. The arguments must be integers. If the `step` argument is omitted, it defaults to 1. `step` cannot be zero. For a positive `step`, the contents of a range  $r$  are determined by the formula  $r[i] = start + step * i$  where  $i \geq 0$  and  $r[i] < stop$ . For a negative `step`, the contents of the

range are still determined by the formula  $r[i] = start + step * i$ , but the constraints are  $i \geq 0$  and  $r[i] > stop$ ;

- `len(v)`: returns an integer that represents the length of  $v$ . The argument  $v$  must be a list or a tuple;
- `str(v)`: returns a *string* version of  $v$ . The argument  $v$  must be a number;
- `int(v)`: returns an integer representation of  $v$ . The argument  $v$  must be a number or string representing an integer literal in base 10;
- `float(v)`: returns a floating-point number constructed from a number or string  $v$ . If the argument  $v$  is a string, it must conform to the format of *<SIGNEDFLOAT>* from Figure 4.5;
- `getenv(key, default=None)`: return a value of the environment *key* if it exists, or *default* if it does not. The arguments and return values are strings;
- `print v`: print to *stdout* the argument  $v$ , which must be a string or a sequence of string separated by commas.

**Control Flow.** The control flow statements available are `if` and `for`. They can be used to decide, during run time, how to proceed with the optimization according to previous decisions. In the `CodeReg` and `OptSeq` block, it is possible to execute different optimizations sequences depending on the result of an expression or a search statement selection. For instance, the compiler to be used can be expressed as a search statement assigned to a variable (e.g., `compiler = enum("gcc", "icc")`), and an `if` can be used to execute different sequences of optimizations according to the value assigned to the variable during the search process.

**Scope.** Each block of code has its own scope. The variables declared in the body of a `CodeReg`, an `OptSeq`, or a control-flow statement are accessible only in the block representing their body and in all inner blocks of that block. The formal parameters of an `OptSeq` definition are considered local to the block of the `OptSeq` body. Variables declared outside of a `CodeReg` and `OptSeq` are considered global and are accessible anywhere in the program.

**Import.** It is possible to import optimization sequences created by others. This is an important feature as experts can share their recipes for common code regions running on similar architectures. It is also used to import modules and optimization definitions to be



used on CodeReg and OptSeq. All the definitions of CodeReg, OptSeq, and global variables from the imported files become accessible to the program. The syntax for importing a file is `import <path/filename>;`.

**Search Block.** The Search block is used to give the system commands on how to compile and run each candidate implementation being evaluated during the search process. The commands assigned to each statement are expressions that will be converted to a string before their execution. Lines 3-6 in Figure 4.2 shows an example of a search block. The string "make clean all" is assigned to `buildcmd` to compile each candidate implementation, and the string `./matmul` is assigned to `runcmd` to execute each candidate implementation. The statements allowed are:

- `prebuildcmd`: used for assigning a command to be executed before the search process;
- `buildcmd`: used for assigning a command to compile each candidate implementation;
- `runcmd`: used for assigning a command to execute each candidate implementation compiled using the *buildcmd*;
- `buildorig`: the same as *buildcmd*, but it is executed only once on the baseline version before the search process starts;
- `runorig`: the same as *runcmd*, but it is executed only once on the baseline version before the search process starts;
- `checkcmd`: used for assigning a command to be executed for each candidate after their execution. It is used for verification of the results of the execution.

The search process requires the use of, at least, `buildcmd` and `runcmd`, the others are optional.

**Initial Search Configuration.** The Locus syntax allows the specification of an initial value for each search statement. These values may be used as the as the first candidate implementation of the search process. The Locus system default, however, is to select the values randomly (among the possible options) for each search statement to serve as the first candidate implementation of the search process. The rules used to define the initial search configuration are:

- the first block of each OR block is used;

- the first statement of each OR statement is used;
- optional statements are assumed to be executed;
- search statements (except OR blocks, OR statements, and optional statements) have an extra parameter, namely `init`, to provide the value. Otherwise, the first value of each sequence is used.

The code in Figure 4.2 illustrates the representation of initial values that may be given to the search module to be assessed as the first candidate implementation. In this case, the initial values are given for the `perm` variable, and the variables `tileI`, `tileK`, `tileK`, `tileI_2`, `tileK_2`, and `tileJ_2`. The first candidate implementation will include the first block of the OR (only a `None`;) resulting in a 2-level tiled variant.

**Recursion.** `OptSeq` constructs accept the use of recursion. It is important to remember that the recursion happens during the execution of the Locus program. At that point, search statements were already replaced by the values selected by the search module. Therefore, the same selection for each search statement will be used in all calls of the respective `OptSeq`, including the recursive calls. In case it is necessary to have different values at each recursion level, the recursion must be expanded before the conversion to the search module notation. This conversion is further discussed in Section 5.4.2.

**Hierarchical Indexing.** Locus uses hierarchical indexing to be able to refer to loops and statements on the source code. This indexing is made of numbers separated by periods. Each number represents a code block level, starting from 0. The value of the number represents the order of the statement or loop at that level. For instance, “0.0.0” refers to the innermost loop in Figure 1.1a. In case of two innermost loops, the second one would be referenced as “0.0.1”. Using this indexing, we can reference any statement or loop in a code region.

Figure 4.3 presents an example of source code and the index for each statement. At each code block level, one extra number is added to the index. The index is incremented for statements, and loops, that are part of the same level.

For transformations that change the code structure, such as loop distribution and fusion, the indexing needs to refer to the current state of code and not to its original version. This raises the burden on the programmer to ensure at each transformation that the indexing is still valid. For instance, if the loops indexed by “0.0.0” and “0.0.1” are to be fused, after the fusion, the new, fused loop is referred to as “0.0.0” as it takes the position of the first loop. Conversely, if loop distribution is carried out, the indexes of the statements after the

```

0          for (t = 0; t < niter; t++) {
0.0        for (i = 1; t < x-1; i++) {
0.0.0      for (j = 1; j < y-1; j++) {
0.0.0.0    for (k = 1; k < z-1; k++) {
0.0.0.0.0  STMT_A;
            }
        }
0.0.1      for (j = 1; j < y-1; j++) {
0.0.1.0    for (k = 1; k < z-1; k++) {
0.0.1.0.0  STMT_B;
0.0.1.0.1  STMT_C;
            }
        }
0.0.2      for (k = 1; k < z-1; k++) {
0.0.2.1    STMT_D;
        }
    }
}

```

Figure 4.3: Hierarchical indexing example.

loop being distributed are incremented according to the number of new loops created by the distribution.

#### 4.4 EXTENDED BACKUS-NAUR FORM OF THE LANGUAGE

Figures 4.4 and 4.5 show the extended Backus-Naur form of the language. In the extended form presented, we use the following syntax rules and extensions: optional items are enclosed in square brackets; items occurring 0 or more times are suffixed with an asterisk (\*); items occurring 1 or more times are suffixed with the plus symbol (+); items occurring 0 or 1 times are suffixed with a question mark (?); terminals appear surrounded by single quotes; grouped items are enclosed in simple parentheses.

#### 4.5 SUMMARY

The Locus language can define search spaces of optimizations sequences. In this chapter, we explain the syntax and semantics of the language. The language accepts special statements to represent the search variables from a search space. The values of these search variables can be used by other search statements, control-flow statements, and passed as arguments to the transformations. The mixing of search spaces with transformation sequences creates a rich environment for defining complex spaces for optimizing programs.

```

<start> ::= (<coderegdef> | <optseqdef> | <globalscp> | <searchdef> | <pragmastmt>)*
<searchdef> ::= 'Search' <searchblock>
<searchblock> ::= '{' searchstmt ';' searchstmt '*' ';' '}'
<searchstmt> ::= <prebuildstmt> | <measureorigstmt> | <buildcmdstmt> | <runcmdstmt> | <runorigstmt> | <buildorigstmt> |
    <checkcmdstmt>
<prebuildstmt> ::= 'prebuildcmd' '=' <test>
<measureorigstmt> ::= 'measureorig' '=' <truestmt> | 'measureorig' '=' <falsestmt>
<buildcmdstmt> ::= 'buildcmd' '=' <test>
<runcmdstmt> ::= 'runcmd' '=' <test>
<runorigstmt> ::= 'runorig' '=' <test>
<buildorigstmt> ::= 'buildorig' '=' <test>
<checkcmdstmt> ::= 'checkcmd' '=' <test>
<coderegdef> ::= 'CodeReg' <NAME> <block>
<optseqdef> ::= 'OptSeq' <NAME> '(' [<arglist>] ')' <block>
<importstmt> ::= 'import' <STRING> (, <STRING>)* ';'
<pragmastmt> ::= '@pragma' <STRINGINNER>*
<block> ::= '{' <subblock> '}'
<subblock> ::= <block> ('OR' <block>)+ | <block> | <stmt>
<stmt> ::= <setstmt> | <compoundstmt>
<setstmt> ::= <smallstmt> '(' ';' <smallstmt>)* ';'
<smallstmt> ::= (<exprstmt> | <printstmt>)
<globalscp> ::= <exprstmt> ';' | <importstmt>
<printstmt> ::= 'print' [ <test> '(' ';' <test>)* '[', ']' ]
<exprstmt> ::= <testlist> '=' <starstmt> | <starstmt>
<starstmt> ::= '*' <orexprstmt> | <orexprstmt>
<orexprstmt> ::= <testlist> ('OR' <testlist>)*
<compoundstmt> ::= <ifstmt> | <forstmt>
<ifstmt> ::= 'if' <test> <block> ('elif' <test> <block>)* ['else' <block> ]
<forstmt> ::= 'for' '(' <exprstmt> ';' <test> ';' <exprstmt> ')' <block>
<test> ::= <ortest>
<ortest> ::= <andtest> ('|'| '&andtest'))*
<andtest> ::= <nottest> ('&&' <nottest>)*
<nottest> ::= 'not' <nottest> | <comparison>
<comparison> ::= <expr> (('>' | '<' | '=' | '>=' | '<=' | '!=') <expr>)*
<rangeexpr> ::= <expr> '..' <expr> ['..' <expr>]
<expr> ::= <xorexpr> ('|' <xorexpr>)*
<xorexpr> ::= <andexpr> ('^' <andexpr>)*
<andexpr> ::= <shifexpr> ('&' <shifexpr>)*
<shifexpr> ::= <arithexpr> (('<<' | '>>') <arithexpr>)*
<arithexpr> ::= <term> (('+' | '-') <term>)*

```

Figure 4.4: The extended Backus-Naur form of the language (part 1/2).

```

<term> ::= <factor> | <term> ('*' | '/' | '//' | '%') <factor>
<factor> ::= ('-' | '~' | '+') <factor> | <power>
<power> ::= <molecule> ['**' <factor>]
<molecule> ::= <funccall> | 'enum' '(' <testlist> ')',
| 'poweroftwo' '(' <rangeexpr> [ <initvalrg> ] ')',
| 'integer' '(' <rangeexpr> [ <initvalrg> ] ')',
| 'permutation' '(' (<test> | <listmaker>) [ <initlistmk> ] ')',
| 'seq' '(' <test> ',' <test> [ ',' <test> ] ')',
| 'len' '(' <test> ')',
| 'str' '(' <test> ')',
| 'int' '(' <test> ')',
| 'float' '(' <test> ')',
| 'getenv' '(' <test> [ ',' [ 'default' '=' <test> ] ')',
| <molecule> '.' <NAME>
| <atom>
<funccall> ::= <molecule> '(' [<arglist>] ')', | <molecule> '[' [<subscriptlist>] ']'
<atom> ::= <tuplemaker> | <listmaker> | <truestmt> | <falsestmt> | <nonestmt> | '(' <test> ')' | <NAME> | <number> | <string>+
<tuplemaker> ::= '(' <test> ((',' <test>))+ [ ',' ] ')',
<listmaker> ::= '[' [ <test> (',' <test>)* [ ',' ] ] ']',
<testlist> ::= <test> (',' <test>)* [ ',' ]
<subscriptlist> ::= <test> (',' <test>)* [ ',' ]
<arglist> ::= (<argument> ',' )* (<argument> [ ',' ])
<argument> ::= <test> | <test> '=' <test>
<initvalrg> ::= ',' 'init' '=' <expr>
<initlistmk> ::= ',' 'init' '=' <listmaker>
<number> ::= <INT> | <FLOAT>
<truestmt> ::= 'True'
<falsestmt> ::= 'False'
<nonestmt> ::= 'None'
<string> ::= <ESCAPEDSTRING>
<DIGIT> ::= '0'..'9'
<INT> ::= <DIGIT>+
<SIGNEDINT> ::= ['+'|'-'] <INT>
<DECIMAL> ::= <INT> '.' <INT>? | '.' <INT>
<EXP> ::= ('e'|'E') <SIGNEDINT>
<FLOAT> ::= <INT> <EXP> | <DECIMAL> <EXP>?
<SIGNEDFLOAT> ::= ['+'|'-'] <FLOAT>
<COMMENT> ::= /# [ ^\n ] */
<STRINGINNER> ::= /. * ? /
<STRINGESCINNER> ::= <STRINGINNER> / (?<!\) (\) * ? /
<ESCAPEDSTRING> ::= " <STRINGESCINNER> "
<LETTER> ::= 'A'..'Z' | 'a'..'z'
<NAME> ::= ('_' | <LETTER> ) ('_' | <LETTER> | <DIGIT>)*

```

Figure 4.5: The extended Backus-Naur form of the language (part 2/2).

## CHAPTER 5: COMPILING AND OPTIMIZING LOCUS PROGRAMS

A Locus program represents a search space of transformation sequences. Optimizing Locus programs affects search time by removing unnecessary search statements. The smaller the search space, the shorter the search time. Optimization also affects code generation by reducing the time to execute the program implementing the optimization sequence leading to the candidate implementation. This program is called the direct Locus program.

This chapter presents techniques to optimize Locus programs and compiler passes to represent correctly hierarchical search spaces constructed based upon the interdependence among search statements.

### 5.1 DATA-FLOW ANALYSIS

Dead-code elimination, constant propagation, and constant folding depend on the data-flow analysis. Data-flow analysis refers to techniques that derive information about the flow of data along program execution paths [25].

Before running the data-flow analysis, the statements are partitioned into basic blocks. For Locus, a basic block ends at the statement preceding a function call or the beginning of a compound statement in the form of a loop, a conditional statement, a Block, or an OR Block.

Once the Locus program is partitioned into basic blocks, we represent the control flow between them by a graph. The nodes of the control-flow graph (CFG) are the basic blocks. There is an edge from basic block B to basic block C if and only if it is possible for the first instruction in block C to be executed immediately after the last instruction in basic block B. We say that basic block B is a *predecessor* of C, and C is a *successor* of B. Two extra nodes are often added to the graph: *entry* and *exit*. They do not correspond to executable statements. There is an edge from the *entry* to the first executable node of the flow graph, i.e., to the basic block that contains the first statement of the program. The last basic block executed of the program is always a predecessor of the exit node. Figure 5.1 presents an example of a Locus program on the left and the program's CFG on the right.

Data-flow analysis algorithms associate every program point with a *data-flow value* that represents an abstraction of the set of all possible program states that can be observed at that point. The set of possible values is the domain for the data-flow analysis. For example, the domain of data-flow values for reaching definitions is the set of all subsets of definitions in the program. A definition is a statement that assigns a value to a memory location. Each

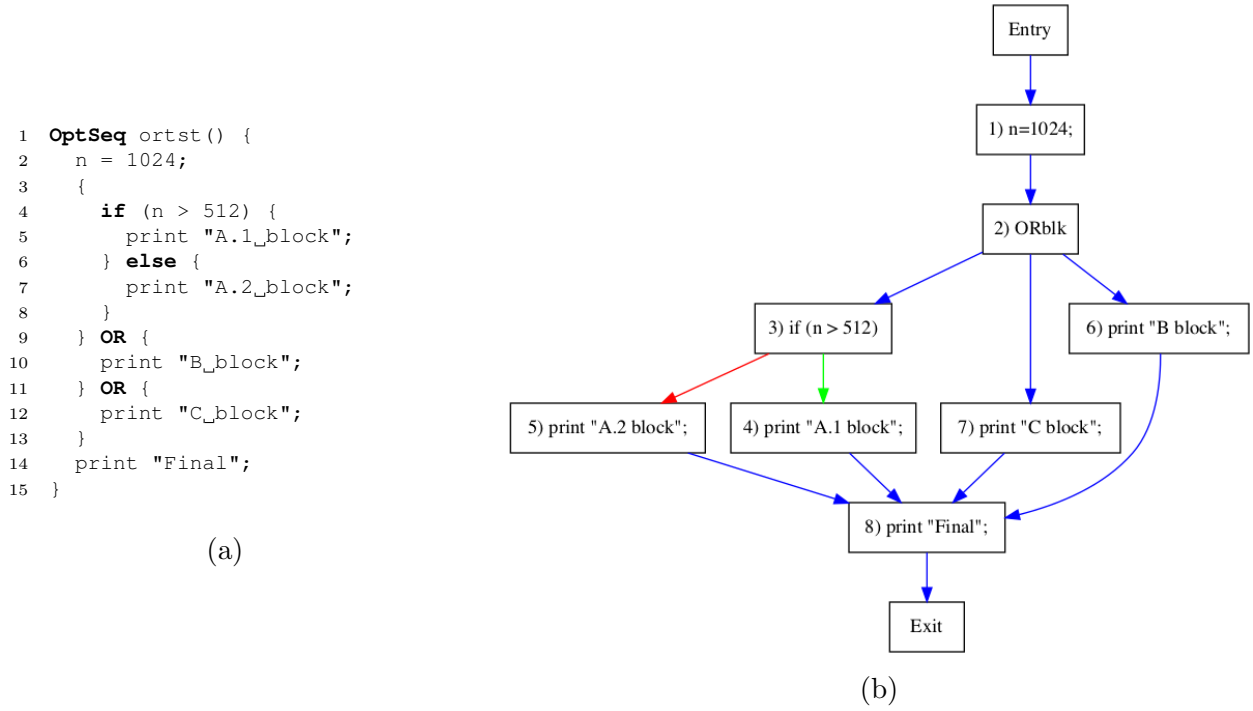


Figure 5.1: (a) An example of a Locus program. (b) CFG generated from the Locus example.

point in the program is associated with the exact set of definitions that can reach that point.

The data-flow values before and after each statement  $s$  are the sets  $IN[s]$  and  $OUT[s]$ , respectively. The goal of data-flow analysis is, for all statements  $s$ , to find the IN and OUT sets by solving a set of constraints. There are two sets of constraints:

- Transfer functions: the data-flow values before and after a statement are constrained by the semantics of the statement. For example, suppose data-flow analysis involves determining the definitions that reach point  $p$ . A definition  $d$  of variable  $x$  reaches a point  $p$  if there is a path from the point immediately following  $d$  to  $p$ , and there is no other definition to  $x$  along the path. The definition still reaches the next point  $p + 1$  if no definition of  $x$  occurred at point  $p$ . This relationship between the data-flow values before and after the statement is known as a *transfer function*. The transfer functions may flow forward along the execution paths or flow backward up the execution paths. The transfer function of a statement  $s$  is denoted by  $f_s$ , and, in a forward-flow, takes the data-flow value before the statement and produces a new data-flow value after the statement. That is,  $OUT[s] = f_s(IN[s])$ . In a backward-flow, the transfer function takes the data-flow value from after the statement to a new data-flow value for before the statement:  $IN[s] = f_s(OUT[s])$ .

- Control-flow: the control-flow is simple within a basic block. For a basic block  $B$  that consists of  $s_1, s_2, \dots, s_n$  in that order, then the control-flow of  $s_i$  is the same as the control-flow into  $s_{i+1}$ . Therefore,  $\text{IN}[s_{i+1}] = \text{OUT}[s_i]$ ,  $\forall i = 1, 2, \dots, n - 1$ . In between basic blocks, the constraints are more complex as it needs to take into account the multiple paths that the program could follow. For reaching definitions, there must exist at least one path along which the definition reaches a program point. Hence,  $\text{OUT}[P] \subseteq \text{IN}[B]$  whenever there is a control-flow edge from basic block  $P$  to basic block  $B$ .  $\text{IN}[B]$ , however, needs to be no larger than the *union* of the reaching definitions of all the predecessor blocks. The union is referred to as the *meet operator* for reaching definitions. In a data-flow analysis, the meet operator is used to create a summary of the contributions from different paths at the confluence of those paths.

The data-flow analysis technically involves values at each point of the program. However, in practice, it can be done at the basic block level in order to save time and space. Thus, the data-flow values immediately before and after each basic block are denoted by  $\text{IN}[B]$  and  $\text{OUT}[B]$ , respectively. The constraints involving  $\text{IN}[B]$  and  $\text{OUT}[B]$  can be derived from those involving  $\text{IN}[s]$  and  $\text{OUT}[s]$  for the various statements  $s$  in  $B$  as follows: consider a basic block  $B$  that consists of statements  $s_1, s_2, \dots, s_n$  in that order. Then,  $\text{IN}[B] = \text{IN}[s_1]$  if  $s_1$  is the first statement of basic block  $B$ , and  $\text{OUT}[B] = \text{OUT}[s_n]$  if  $s_n$  is the last statement of  $B$ . The transfer function of a basic block  $B$  denoted as  $f_B$  can be derived by composing the transfer function of the statements in the block. That is, let  $f_{s_i}$  be the transfer function of statement  $s_i$ . Then,  $f_B = f_{s_n} \circ \dots \circ f_{s_2} \circ f_{s_1}$ . The relationship between the beginning and end of the block is  $\text{OUT}[B] = f_B(\text{IN}[B])$ . The control-flow constraints between basic blocks can be rewritten by substituting  $\text{IN}[B]$  and  $\text{OUT}[B]$  for  $\text{IN}[s_1]$  and  $\text{OUT}[s_n]$ , respectively.

### 5.1.1 Reaching Definitions

It is important to know where each variable  $x$  may have been defined in a program for many reasons. For example, a compiler can know whether  $x$  is a constant when evaluating an expression at point  $p$ . Because of being a constant,  $x$  can be replaced by its value avoiding a load operation during execution. Reaching definitions is one of the most common and useful data-flow analysis.

A definition  $d$  *reaches* a point  $p$  if there is a path from the point immediately following  $d$  to  $p$ , such that  $d$  is not “killed” along that path. A definition of  $x$  is *killed* if there is any other definition of  $x$  anywhere along the path. Intuitively, a definition  $d$  of some variable  $x$  that reaches point  $p$  might be the place at which the value of  $x$  used at  $p$  was last defined.



A definition of a variable  $x$  is a statement that assigns a value to  $x$ . Program analysis must be conservative, and we must assume that it may assign to it even though we are not totally sure that a statement  $s$  is assigning a value to  $x$ .

---

**Algorithm 5.1:** Computing Reaching Definitions.

---

```

1 OUT[ENTRY] =  $\emptyset$ ;
2 foreach basic block  $B$  other than ENTRY do OUT[B] =  $\emptyset$ ;
3 while changes to any OUT occur do
4   foreach basic block  $B$  other than Entry do
5     IN[B] =  $\bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$ ;
6     OUT[B] =  $\text{gen}_B \cup (\text{IN}[B] - \text{kill}_B)$ ;
7   end
8 end

```

---

Consider a definition  $d: t = y + z$ . This statement “generates” a definition  $d$  and “kills” all the other definitions in the program that define variable  $t$ , while leaving all the remaining definitions unaffected. Reaching definitions is a forward-flow analysis, and its transfer function is  $\text{gen}_B \cup (x - \text{kill}_B)$ , where  $\text{gen}_B = d$  is the set of definitions generated by the statement, and  $\text{kill}_B$ , the set of all other definitions of  $t$  in the program.

The iterative algorithm to compute the reaching definitions is in Algorithm 5.1. The first two lines initialize the data-flow values. Line 3 starts the loop that iterates until convergence, and the inner loop at Line 4 applies the data-flow equations to every basic block other than entry. Intuitively, the algorithm propagates the non-killed definitions as far as possible, simulating all possible executions of the program. It will eventually halt, as the OUT[B] for every B never shrinks. The added definitions stay there forever. Since the set of all definitions is finite, eventually, there must be an iteration of the while loop during which nothing is added to any OUT, and the algorithm then terminates. It is safe to terminate then because if the OUT’s have not changed, the IN’s will not change in the following passes. Consequently, no changes to the IN’s mean that OUT’s also stay the same in subsequent passes.

## 5.2 PRUNING THE SEARCH SPACE

The Locus program contains a representation of the search space. Each use of a search statement on the program has a multiplicative factor on the space size. Hence, it is advantageous to remove from the Locus programs any search statements that are never used. These statements, then, will not take place in the search space, and a smaller space often results

in less time required for the empirical search.

Constant propagation, constant folding, and dead code elimination are applied before the Locus representation of the space is converted to the search module's space. These optimizations are applied iteratively until the statements converge to a state in which no more changes occur.

<pre> 1 <b>CodeReg</b> optimizer () { 2   @pragma query 3   perfect = IsPerfectLoopNest(); 4   <b>if</b> (perfect) 5   { 6     Interchange(<b>permutation</b>[0,1,2]); 7   } 8   Tiling(factor=<b>poweroftwo</b>(2..1024)); 9   Distribute(...); 10  Unroll(factor=<b>integer</b>(2..64)); 11 }</pre>	<pre> 1 <b>CodeReg</b> optimizer () { 2   perfect = <b>False</b>; 3   Tiling(factor=<b>poweroftwo</b>(2..1024)); 4   Distribute(...); 5   Unroll(factor=<b>integer</b>(2..64)); 6 }</pre>
(a)	(b)

Figure 5.2: Example of using a *pragma query* followed by constant propagation and dead code elimination. The code on Lines 4-7 is removed, including the permutation term, because of perfect value is False.

These optimizations can drastically reduce the search time by reducing the space and improving the quality of the candidate implementations suggested to be assessed. For instance, in Section 8.2, there is one example in which optimizations are only applied if the loop nest depth is greater than 1. Therefore, we can exclude from the search space all the search variables that are not used when optimizing loop nests with a depth equal to 1.

Constant propagation substitutes values of known constants in expressions at compile time. Constant folding recognizes and evaluates constant expressions at compile time rather than computing them at runtime. Terms in constant expressions are typically simple literals (constants), but they may also be variables whose values are known at compile time. Dead or useless statements compute values that never get used. Dead code may appear as the result of transformations. One advantage of constant folding and constant propagation is that it often turns statements into dead code. Dead code elimination removes these statements.

The optimizations often occur after the execution of calls preceded by a *pragma query* whose results are used by search statements and control-flow statements. These *query calls* are assumed to have a deterministic result throughout the search process if used by any search statement. The parameters of the search statements need to be known when the search is defined. Therefore, these *query calls* are executed, and their values replace the search statement parameters.

Figure 5.2 presents an example of a program that uses the query `IsPerfectNestLoop()`

to check whether the loop is perfectly nested. For this example, let's assume that the loop being optimized is not perfectly nested, and the query call returns `False`. By invoking the constant propagation, the `If` on Line 4 becomes `if(False)`. Dead-code elimination is now able to remove the block of code on Lines 4-7. The result is on the right side of the figure. The `permutation` search statement represents 6 possible interchange orders for the loop being optimized. Optimization of this Locus program was not only able to remove the useless code but also to remove the `permutation` search statement that would have incurred a 6 fold increase of the search space.

### 5.3 INTERDEPENDENT SEARCH STATEMENTS

When designing a search space to be explored, it is common to have interdependence between search statements. This interdependence is hard to represent concisely and, with no assistance, requires a verbose, hard to maintain description. We present compiler passes that automatically generate complex spaces with interdependent search statements allowing the users to program them concisely. The concise representations are easier to program and maintain and are less error-prone. Besides, these compiler passes can reduce the space dimensionality by cutting unnecessary search statements. Each search statement represents a search variable in the search space. This dimensionality reduction often results in faster search processes.

There are two important constraints when *programming* search spaces: the translation from the space in Locus IR to the notation accepted by the search modules only considers the static information of the code, and the translation must finish before the search starts. This section addresses the challenges of programming search spaces when using control-flow statements and recursion to represent interdependence between search statements. These control-flow statements and recursion require runtime information (i.e., when the Locus program is being executed during the search) to create the semantically correct search space properly.

The use of search statements that depend on other search statements is not natively supported by the search modules that we have integrated, OpenTuner and HyperOpt. Locus supports it, but since all the search variables in the search space need to be precisely defined, the minimum and maximum possible values that reach the search statements must be computed ahead of the search space definition. A data-flow analysis through the use-def chains of the Locus program is performed to get the boundary values. Nonetheless, during the search, for each point selected during the empirical evaluation, it is still necessary to check whether the values selected for the dependent search statements are valid.

```

1 tileI = poweroftwo(2..512);
2 Pips.Tiling(loop="0", factor=[tileI]);
3 tileI_2 = poweroftwo(2..tileI);
4 Pips.Tiling(loop="0.0.0.0", factor=[tileI_2]);

```

Figure 5.3: An example of interdependent search statements.

For instance, in Figure 5.3, the factor of the second-level tiling (`tileI_2`) is limited by the factor of the first-level tiling (`tileI`). In the search space definition, the maximum value of the search statement referenced by `tileI_2` is defined as the maximum of the statement referenced by `tileI`. The search module, however, is not aware of that and may suggest points in the space in which the value referenced by `tileI_2` is bigger than the value referenced by `tileI`. This violates the interdependence criteria. Hence, the system needs to check that the value selected for `tileI_2` is smaller or equal to the one selected for `tileI`; otherwise, this point is invalidated (the candidate implementation is not generated), and the search process moves on to the next point.

#### 5.4 EXPANSION PASSES FOR INTERDEPENDENT SEARCH STATEMENTS

Search statements related to control-flow statements need to be expanded before the search starts to make part of the search space. Search statements declared inside a loop body must have, represented on the search space, one instance for each iteration of the loop<sup>1</sup>. With a known number of iterations, or, in other words, with only constants on the loop header, a Locus pass is able to fully unroll a for-loop, replicate the loop body, and generate the proper space with one instance of each search statement for each iteration. This pass, however, is only necessary in cases that there are search statements on the body.

Nonetheless, there are cases in which the number of iterations of a loop is also part of the search. Locus allows search statements on the loop header. For instance, the header of the loop on the left-hand side of Figure 5.4 has `i < integer(1..3)` as a condition. Besides, the loop body contains three search statements (`poweroftwo`). The number of iterations can range from 1 to 3, which will only be defined during the search process, at a time that is too late to replicate on the space representation the number of search statements according to the number of iterations. For the search space to be represented appropriately, our solution is a pass that expands the for-loop for all possible number of iterations.

We developed compiler passes to transform the Locus IR and ensure that space is generated

---

<sup>1</sup>Otherwise, the search statement should be declared outside of the loop and only the value assigned to the statement referenced in the loop body.

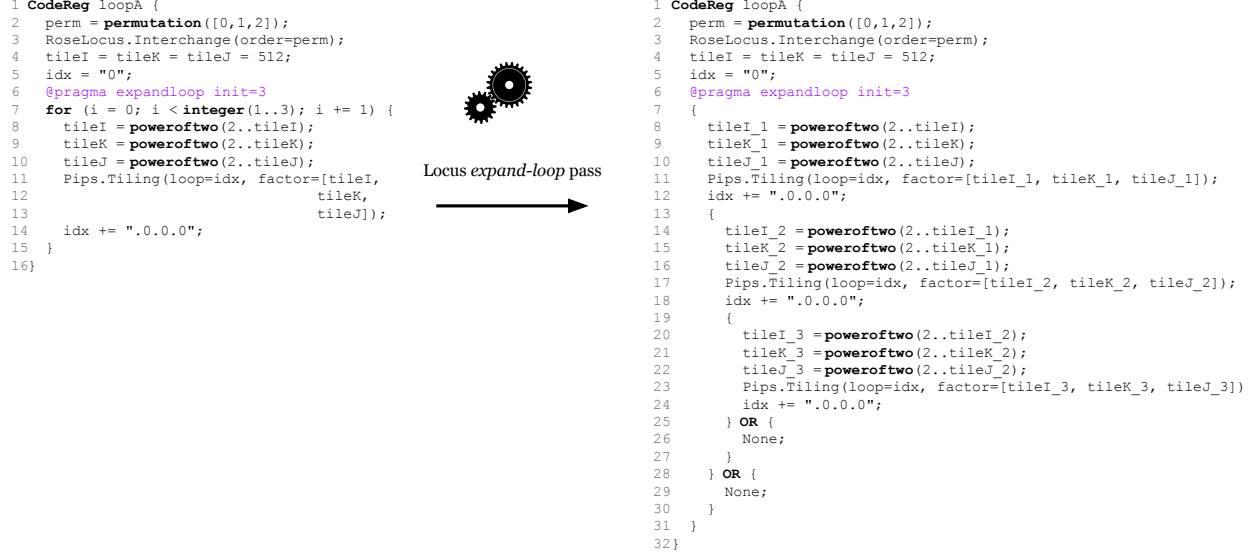


Figure 5.4: Loop-expansion pass for Locus program that contains a search statement in the condition expression of the loop-header.

correctly to represent the semantics of the code. The implementation details for handling interdependent search statements using control-flow statements and recursion are discussed next. Besides the loop-expansion pass, we also present a pass to expand recursion that is useful when representing a tree of choices. To represent a tree of choices, the recursion in the Locus program needs to be unfolded before the translation of the space. Otherwise, the same decision will be taken for all levels and branches of the tree. Although we show the result of the pass using Locus notation, for efficiency, the passes carry out the expansion in the IR level and do not actually generate another Locus program.

#### 5.4.1 Loop Expansion

A for-loop in Locus can contain search statements in its header and body. The body's search statements need to be replicated for each iteration defined in the header to represent the search space correctly. It must be replicated before the conversion from the Locus IR to the search module's notation.

With a known number of iterations (constants on the loop header) and the body containing search statements, the Locus pass unrolls the for-loop, replicates the loop body, and generates the proper space with one instance of each search statement for each iteration.

With search statements on the loop header and search statements on the body, Locus must generate a Cartesian product of all search statements on the loop header. Regarding

a for-loop header as `for(init; cond; step)`, the first step of the pass is to generate one option for each pair of values of the `init` and `step` expressions if they contain search statements. The second step, for each of these options, creates, in another level, a hierarchical OR block according to the values on the `cond` expression. The second step is depicted in Figure 5.4 since the `init` and `step` expressions are fixed. The loop-carried dependencies between search statements are also correctly solved by the second step. The expansion of the loop defines that the first level hierarchical tiling always occurs (Lines 8-11). The second and third levels, however, are optional because of the empty blocks (contain only a `None`; placeholder) on Lines 26 and 29, respectively. If the block on Line 26 is selected during the search, the second-level tiling (Lines 14-17) and third-level tiling (Lines 20-23) statements are ignored. Similarly, if the block on Line 29 is selected, the third-level tiling is ignored (but second-level tiling occurs).

The goal of the code on the left of Figure 5.4 is to tile a triple-nested loop, as shown in Figure 1.1a, hierarchically. The three search statements on the body (`poweroftwo`) find the best tile dimensions for each loop. The number of iterations defines the number of levels that the loop nest should be tiled.

The definition of the candidate to be the first evaluated (i.e., initial search configuration) can make the search process converge faster. Locus allows the definition of values for each search statement to be used as the first candidate implementation of the search process. In the case of OR Blocks, the first block is the one used by the first candidate. Therefore, when expanding the loop, by selecting which code to be on the first block of each OR block, it is possible to define which “iteration” of the loop should be used for the first candidate. The loop expansion pass is invoked by using the `pragma expandloop [init=<iteration>]`. The “iteration” to be used for the first candidate is defined by assigning an integer value to the `init` keyword. This value is optional, and the default is the first iteration. In Figure 5.4, the third “iteration” is used for the first candidate.

In summary, we use the for-loop with search statements in its header and body to generate a search space to explore for the best number of levels in hierarchical tiling.

### 5.4.2 Recursion Expansion

The divide-and-conquer strategy recursively divides a problem into smaller subproblems and solves them using (typically) simpler methods. The main challenge in these algorithms is to identify the best dimensions (or shape) for the base case so that the data fits in the cache and the best number of base cases for parallel execution, assuming that a new task is created for each recursive invocation to be executed in parallel. The dimensions of the

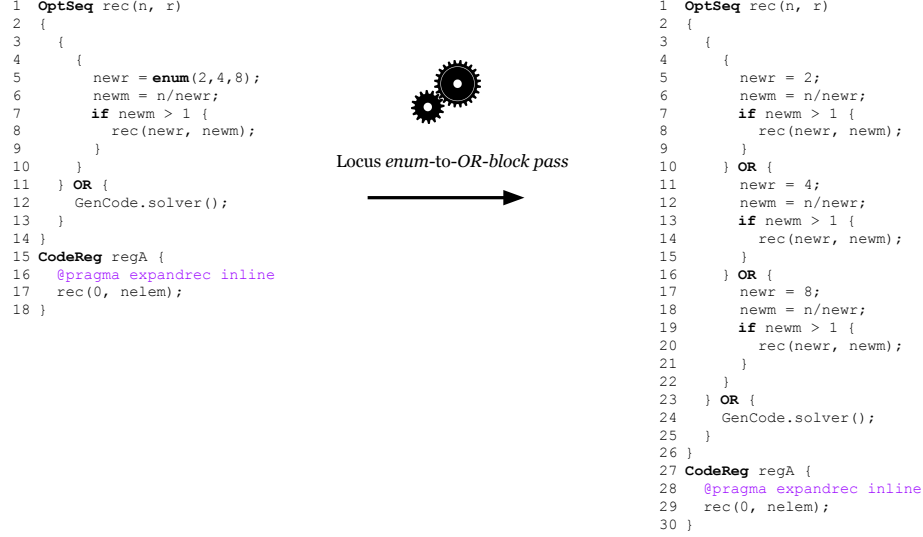


Figure 5.5: *Enum-to-OR-block* pass is required before the recursion expansion and inline.

base case are the result of recursively splitting the input. There is then the decision about how far to split between each dimension and the splitting order. This search space can be expressed as a tree of choices. We revisit this discussion in Chapter 7.

We can concisely represent a tree of choice in Locus language by using recursion combined with OR blocks. The tree of choices is automatically generated by a compiler pass that executes in 4 steps. The pass is invoked through the pragma *expandrec inline* placed right before the call of a recursive Locus function. To perform this tree generation, the values on which the recursion depends must be known.

In the first step, it is checked whether any of the recursive calls have an argument that is, or refer to, a search statement (e.g., *enum*). If yes, the block of code in which the recursion is included is replicated into a set of OR blocks, one for each possible value or combination of values in the arguments. Figure 5.5 presents an example of expanding a recursive optimization sequence with the argument (*newr*) that refers to an *enum* statement. The result is on the right-hand side, in which the block containing the recursive call is replicated into an OR block, and each search statement is replaced with one of the possible values from the original search statement.

In the second step, the recursion is unfolded and inlined. As the recursion is successively unfolded, before inlining each call, constant folding and propagation, common subexpression elimination, and dead-code elimination are executed to remove potentially unnecessary new recursive calls.

In the third step, blocks of code belonging to OR blocks that do not invoke any *OptSeq*

are removed (dead blocks containing *OptSeq* were removed in previous iterations of the data-flow optimizations). The fewer blocks in an OR block, the smaller the search space. For instance, on the right-hand side of Figure 5.5, recursion calls at Lines 8, 14, and 20 are only unfolded if the previous line’s condition is true. Otherwise, by executing constant folding and propagation, common sub-expression elimination, and dead-code elimination, we can eliminate the if block, avoiding to unfold unnecessary recursion calls.

Finally, the expanded recursive optimization sequence representing a tree is inlined and replaces the call right after the pragma.

Because the tree of choice can grow very fast, we also implemented a feature to generate an *approximate* recursion expansion. Based on a given percentage value  $x$ , at each level of the recursion expansion, only  $x\%$  of recursion invocations are actually expanded.

## 5.5 SUMMARY

The Locus compiler optimizes Locus programs to reduce search and code generation times. Data-flow optimizations can remove unnecessary statements, including search statements, whose result is a smaller search space.

The Locus compiler also contains passes to represent correctly hierarchical search spaces constructed based on the interdependence among search variables. It provides an important role by automatically optimizing programs to reduce the search time and code generation time and accurately generating complex search spaces through its passes.



## CHAPTER 6: EXPERIMENTS ON LOOP-BASED TRANSFORMATIONS

When optimizing programs for improving performance, the most gains come from regions of the program that require the most time. These regions often correspond to iterative loops. In this chapter, we concentrate on using loop transformations to improve the performance of linear algebra operations, represented by matrix-matrix multiplication, and stencils.

Selecting the transformations and applying a sequence of loop transformations can be a very complicated task, almost impossible to be done by hand. It is difficult even for a general-purpose compiler considering its limited compilation time and the large space of possible transformations for a given kernel on a given machine.

Locus helps define transformations sequences and search spaces for exploring sequences of loop transformations and their factors. It also includes searching for the best way to run loops in parallel and the best compiler-specific pragmas for vectorization.

### 6.1 EXPERIMENTAL ENVIRONMENT

We evaluated the performance of the code generated by Locus on two platforms: IBM Power9 and Intel E5. The details of the platforms are presented in Table 6.1. The IBM platform runs a Linux Red Hat kernel version 4.14, and the Intel runs a Linux Ubuntu kernel 4.4.0.

Table 6.1: Platforms used on the evaluation (sh for shared; pr for private).

Processor	Clock	Cores	HT	L1	L2	L3	RAM
IBM P9 8335-GTH	3.8 GHz	20	4	32KB pr	512KB sh	100MB sh	570GB
Intel E5-2660v3	2.60 GHz	10	2	32KB pr	256KB pr	25MB sh	62GB

The code generated on the IBM Power9 was compiled with IBM XLC (version 16.1.1; flags `-O3` and `-qHot`) and GNU GCC (version 8.2.0; flags `-O3`, `-mtune=native`, and `-ftreevectorize`). For the Intel E5, the code was compiled with ICC (version 17.0.1; flags `-O3`, `-xHost`, `-ipo`, `-ansialias`, and `-fpmodel precise`).

We compare it to the code generated by Pluto [16] (*pet* branch version 0.11.4) with no parameter tuning. Pluto is an automatic parallelization tool based on the polyhedral model. It transforms annotated affine loop nests in C programs for coarse-grained parallelism and data locality.

```

1 Search {
2   buildcmd = "make_clean;_make";
3   runcmd = "./matmul";
4 }
5 CodeReg matmul {
6   RoseLocus.Interchange(order=[0,2,1]);
7   tileI = poweroftwo(2..512);
8   tileK = poweroftwo(2..512);
9   tileJ = poweroftwo(2..512);
10  Pips.Tiling(loop="0", factor=[tileI,
11                                     tileK,
12                                     tileJ]);
13  tileI_2 = poweroftwo(2..tileI);
14  tileK_2 = poweroftwo(2..tileK);
15  tileJ_2 = poweroftwo(2..tileJ);
16  Pips.Tiling(loop="0.0.0.0",
17              factor=[tileI_2,
18                      tileK_2,
19                      tileJ_2]);
20  {
21    Pragma.OMPFor(loop="0");
22  } OR {
23    Pragma.OMPFor(loop="0",
24                  schedule=enum("static","dynamic"),
25                  chunk=integer(1..32));
26  }
27 }

```

(a)

```

1 dim=4096;
2 Search {
3   buildcmd = "make_clean_all";
4   runcmd = "./matmul";
5 }
6 CodeReg matmul {
7   perm = permutation([0,1,2], init=[0,2,1]);
8   RoseLocus.Interchange(order=perm);
9   tileI = poweroftwo(2..dim, init=512);
10  tileK = poweroftwo(2..dim, init=128);
11  tileJ = poweroftwo(2..dim, init=2048);
12  Pips.Tiling(loop="0", factor=[tileI,
13                                  tileK,
14                                  tileJ]);
15  tileI_2 = poweroftwo(2..tileI, init=256);
16  tileK_2 = poweroftwo(2..tileK, init=32);
17  tileJ_2 = poweroftwo(2..tileJ, init=32);
18  Pips.Tiling(loop="0.0.0.0",
19              factor=[tileI_2,
20                      tileK_2,
21                      tileJ_2]);
22  {
23    None;
24  } OR {
25    tileI_3 = poweroftwo(2..tileI_2);
26    tileK_3 = poweroftwo(2..tileK_2);
27    tileJ_3 = poweroftwo(2..tileJ_2);
28    Pips.Tiling(loop="0.0.0.0.0.0.0",
29                factor=[tileI_3,
30                        tileK_3,
31                        tileJ_3]);
32  }
33 }

```

(b)

Figure 6.1: Locus programs for optimizing Matrix-Matrix Multiplication (DGEMM). On the left, the search space comprises two-level tiling shapes and parameters for OpenMP parallel for. On the right, a larger search space comprises all possible loop orders, two- or three-level tiling shapes, and the search statements have the `init` argument for the initial search configuration.

## 6.2 MATRIX-MATRIX MULTIPLICATION

We present here results for the optimizations in Figure 6.1a applied to the baseline code shown in Figure 1.1a. The baseline code is a naive implementation of the double-precision matrix-matrix multiplication (DGEMM), which will be transformed by the Pluto compiler and Locus. The experiments were conducted on the Intel E5 platform. Its execution time on a single core was used to calculate the speedups. The three matrices involved have 2048 by 2048 shape. Figure 6.1 represents a search space of 34,012,224 possible variants (according to OpenTuner).

The first step of the optimization code in Figure 6.1a is to interchange the loop from *ijk* to *ikj* at Line 6 by invoking `RoseLocus.Interchange(order[0,2,1])`. The tiling happens

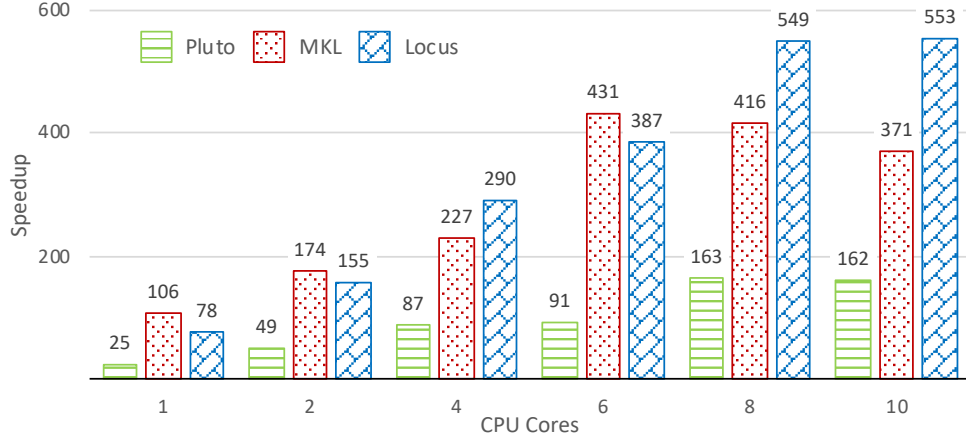


Figure 6.2: *Dgemv* speedup of Pluto, Intel MKL, and Locus on the Intel E5 platform.

on all three loops of the nest, and Lines 7-9 contains three search statements to define the tile shape of the first level of the hierarchical tiling. The shape dimensions can be any value of the power of 2 in the range from 2 to 512. `Pips.Tiling()`, at Line 10, uses the values of the three previous search statements to carry out the tiling on the three loops starting at the first loop (`loop="0"`) of the code region. In the same way, the code for the second-level tiling is at Lines 13-19. The values of the second-level tiling search statements are limited by the values selected for the first-level tiling. After each tiling, three extra loops are added to the nest. Therefore, `Pips.Tiling()` on Line 16 is carried out on the fourth, fifth, and sixth loops of the nest (`loop="0.0.0.0"` is used to determine that the tiling transformation starts on the fourth loop). After the tiling transformation being applied twice, the nest has nine loops in total. The optimization program's last step is to insert OpenMP `parallel for` before the outermost loop (`loop="0"`). We use an *OR* block to explore the space of optimization that the `parallel for` pragma provides: scheduling and chunk. There are two options represented by the OR blocks: the first, at Line 21, uses the default arguments for the `parallel for`; and the second includes a search statement for using a `static` or a `dynamic` schedule of the iterations, and another search statement for the `chunk` size.

Figure 6.2 presents results from 1 to 10 CPU cores. They are compared to code generated by Pluto (flags `--tile`, `--l2tile`, and `--parallel`) and Intel MKL version 2017.0.1. We optimized the code according to the number of threads used and executed a search process for each number of cores separately. Each Locus search was limited to 1,000 variants and took, on average, 80 minutes to complete. Pluto generated code in less than a second. The code generated by Locus using 10 cores was 553 times faster than the baseline. Intel MKL was faster than Locus when using 1, 2, and 6 cores, but slower for 4, 8, and 10 cores. On

average, Locus best candidate implementation was 3.45 times faster than the code generated by Pluto. The reason for the performance difference is not the set of transformations applied. Pluto and Locus apply the same transformations, but the empirical search can identify the best tile shapes.

### 6.3 EVALUATING HIERARCHICAL TILING

We show optimization results of a double-precision matrix-matrix multiplication as presented in Figure 1.1a using the optimization program shown in Figure 6.1b on IBM Power9 and Intel E5 platforms. The search process was conducted by OpenTuner and limited to the evaluation of 1000 variants or 5 hours. The running time of candidate implementations evaluated was capped by the elapsed time of the best one found until the candidate started its execution. This significantly reduced the total search time by limiting the bad candidates' execution that could last for hours.

The optimization program in Figure 6.1b is very similar to the one already explained in Figure 6.1a. The one used for the current evaluation (6.1b), however, includes: all permutations for reordering the original loop nest at line 7; the possibility of tiling one extra level represented by the *OR* blocks at Lines 22-32; the search statements have an `init` value that is used to generate the first candidate implementation of the search process.

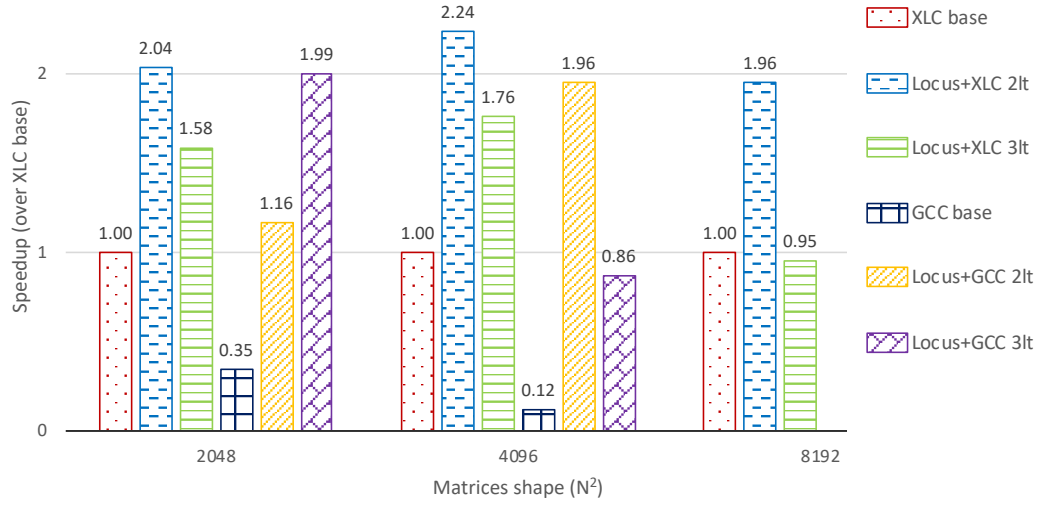
The baseline code compiled with XLC performed significantly better than when compiled with GCC for the IBM Power9 platform. The XLC flag `-qhot` requests, according to IBM manual, high-order transformations such as loop interchange, fusion, unrolling, and reduce the use of temporary arrays. Besides, by monitoring the compilation, we could see that XLC uses interprocedural analysis. As a consequence, the compilation time for XLC is at least 3-times longer than with GCC.

Moreover, the XLC compiler<sup>1</sup> detected that the baseline version was a matrix-matrix multiplication and replaced it with a function call to a hand-optimized version. The version containing the invocation to the hand-optimized code is used as a reference on the performance evaluation. The compiler, however, was unable to detect matrix multiplication in the transformed code generated by Locus, which, in turn, was able to find variants that were faster than the XLC hand-optimized version.

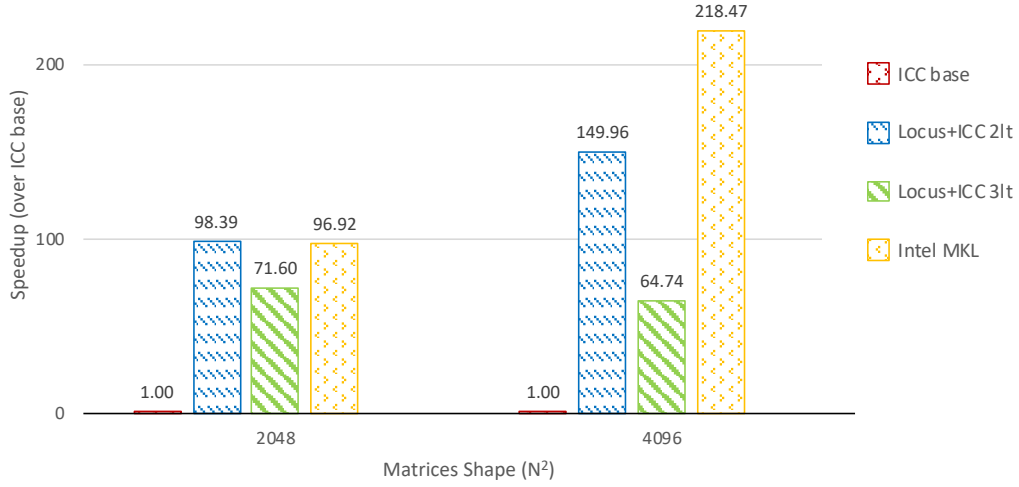
Figure 6.3 shows results for IBM Power9 on the top and Intel E5 on the bottom. On both platforms, a 2-level and 3-level hierarchical tiling has been evaluated. The results include the best variant found by Locus and the baseline version using the two compilers. For the

---

<sup>1</sup>We used GNU objdump for this analysis.



(a) IBM Power9



(b) Intel E5

Figure 6.3: Results for matrix-matrix multiplication on IBM Power9 and Intel E5. It shows the results for the baseline version and the best variant generated by Locus compiled with XLC and GCC on Power9 and ICC and MKL-library on E5. The *2lt* refers to 2-level hierarchical tiling and the *3lt* to 3-level hierarchical tiling.

matrices of 2048 by 2048 shape, the best variant generated by Locus varied according to the compiler used. For XLC, the 2-level tiling was the fastest, whereas the 3-level tiling attained the best performance for GCC. For matrices of 4096 by 4096 shape, both XLC and GCC found 2-level tiling faster. For XLC, 3-level tiling also attained performance close to that of the fastest 2-level tiling version. Locus, however, could not find a 3-level tiling variant compiled with GCC that was faster than the XLC baseline.

Table 6.2: Tiling shapes of the best variant for the matrix-matrix multiplication generated by Locus. 2-level tiling was faster than 3-level tiling for all results.

Platform	Matrices Shape	Compiler	tile I	tile K	tile J	tile I2	tile K2	tile J2
IBM P9	2048	XLC	256	128	2048	256	16	8
	4096	XLC	512	512	512	4	512	256
	8192	XLC	8192	4096	8192	8192	64	16
Intel E5	2048	ICC	512	1024	32	512	64	32
	4096	ICC	512	128	2048	256	32	32

For 8192 by 8192 matrices, due to the long execution time of most of the candidate implementations compiled with GCC, we only present results using the XLC compiler. In this case, the best performance is obtained with 2-level tiling.

Search time is strictly dominated by the time to compile and run the candidate implementations. The compilation is often fast for GCC, XLC, and ICC. As noted above, XLC is 3 times slower than GCC and ICC when it runs high-order transformations and interprocedural analysis. The variant execution, however, depends on the matrix size and can take up to 30 minutes since each variant was executed 5 times.

The search time lasted up to 5 hours for each of the experiments. While earlier experiments, which only considered 2-level tiling, only took on average 1.5 hours, the search time significantly increased after the addition of the third-level tiling. The addition of a third-level tiling within an *OR* block extended the search process because the search space in a flattened representation is not aware of the interdependent search statements. In this case, the dimensions `tileI_3`, `tileK_3`, and `tileJ_3` only matter when the second block within the *OR* block is selected; otherwise, their values are ignored in the code generation and do not affect the variant performance. However, in this case, the search module, OpenTuner, is not aware of that and may infer that those values are important. In other words, the search module processes a search space much bigger than it actually needs to explore and wastes time exploring configurations that do not improve performance.

In Table 6.2, we show the tiling shapes for the best variants for each matrix shape on the two platforms. The tiling shapes are very different from each other and result in input- and platform-specific variants to be saved in the database.

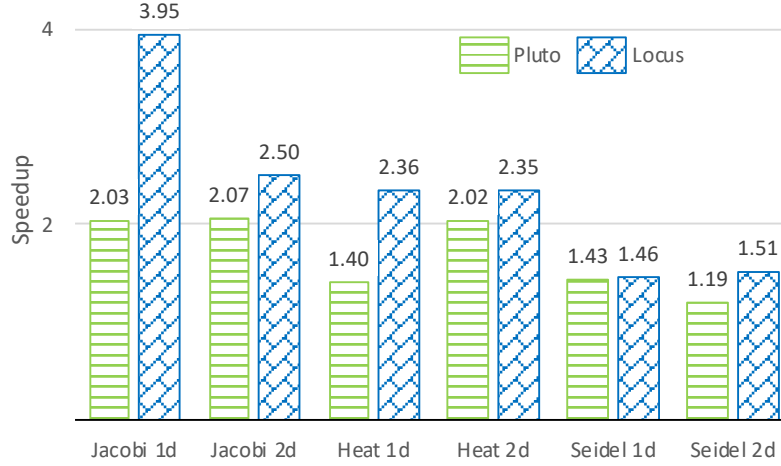


Figure 6.4: Speedup of Pluto and Locus for 1D and 2D Stencils on the Intel E5 platform.

## 6.4 1D AND 2D STENCILS

We evaluated tiling transformations using Locus on 6 stencil codes: Jacobi 1D and 2D, Heat 1D and 2D, and Seidel 1D and 2D<sup>2</sup>. The experiments were executed on the Intel E5 platform. The stencils were executed for 1,000 time steps; the dimensions of the 2D versions are 2,000 by 2,000 elements, and of the 1D are 1,600,000 elements (double-precision). As an example, Figure 6.5 presents the Heat 2D baseline version, and Figure 6.6 the optimizations applied.

We applied a skewing-1 (see Table 3.2) tiling shape using *Pips.GenericTiling*. The skewing-1 matrix for transforming 1D stencils,  $T_1$ , and for 2D stencils,  $T_2$ , are defined as follows:

$$T_1 = \begin{pmatrix} x & -x \\ 0 & x \end{pmatrix}, \quad \text{and} \quad T_2 = \begin{pmatrix} x & -x & -x \\ 0 & x & 0 \\ 0 & 0 & x \end{pmatrix}. \quad (6.1)$$

Line 6 of Figure 6.6 shows a search statement for generating candidate evaluations with different skewing factors that result in different tiling shapes. Since it represents the optimization of a 2D stencil, the variable `tmat` at Line 7 is assigned a list of lists representing the transposed matrix  $T_2$  with `skew1` variable replacing  $x$ . The `Pips.GenericTiling()` at Line 10 tiles the first loop of the code region (`loop="0"`) and uses `tmat`. `Pragma.Ivdep` and `Pragma.Vector` at Lines 11 and 12, respectively, insert pragmas before the innermost

<sup>2</sup>The stencil codes were based on Pluto's examples (*pet* branch) and can be found at <http://pluto-compiler.sourceforge.net>.

loop to provide hints to the compiler that are relevant to automatic vectorization.

```

1 #pragma @Locus loop=heat2d
2 for (t = 0; t < T; t++)
3   for (i = 1; i < N+1; i++)
4     for (j = 1; j < N+1; j++)
5       A[(t+1)%2][i][j] = 0.125 * (A[t%2][i+1][j] - 2.0 * A[t%2][i][j] + A[t%2][i-1][j])
6       + 0.125 * (A[t%2][i][j+1] - 2.0 * A[t%2][i][j] + A[t%2][i][j-1]) + A[t%2][i][j];

```

Figure 6.5: Heat 2D stencil kernel.

Pluto generated code in less than a second. Locus execution time (including the empirical search) for the stencil Heat 2D was the longest at 9 minutes. Jacobi 1D and 2D, Heat 1D, Seidel 1D and 2D lasted 3, 6, 2, 2, and 6 minutes respectively.

The Locus-generated code outperforms that of Pluto (flags `--tile` and `--pet`) as shown in Figure 6.4. Once again, the empirical search showed its importance on the process, as the set of transformations applied by both systems was the same.

## 6.5 3D STENCIL

The 3D heat baseline code is in Figure 6.7. The optimization program in Figure 6.8 tiles the accesses to the  $Y$  dimension of the  $XYZ$  input volume. It first strip-mines the  $j$  loop (responsible for traversing the  $Y$  dimension) by calling `RoseLocus.StripMine(·)`. Then, the order of the loops is changed by invoking `RoseLocus.Interchange(·)`. It moves the loop created by the strip-mine transformation to the outermost position. The strip-mine increases the reuse of the  $XY$ -plane elements as it traverses the  $Z$  dimension and improves performance when the  $XY$  plane does not fit on the last private level cache. The strip-mine is most effective when the problem is large enough that four  $XY$  (three for the input and one for the output) planes do not fit into the cache.

```

1 Search {
2   buildcmd = "make_clean;_make";
3   runcmd = "./heat-2d";
4 }
5 CodeReg heat2d{
6   skew1 = poweroftwo(16..128);
7   tmat = [[ skew1,      0,      0],
8           [-skew1, skew1,      0],
9           [-skew1,      0, skew1]];
10  Pips.GenericTiling(loop="0", factor=tmat);
11  Pragma.Ivdep(loop="0.0.0.0.0.0");
12  Pragma.Vector(loop="0.0.0.0.0.0");
13 }

```

Figure 6.6: Locus program for optimizing Heat 2D stencil.



```

1 void heat3d(double A[2][N+2][N+2][N+2])
2 {
3     int i, j, t, k;
4
5     #pragma @LOCUS loop=heat3d
6     for(t = 0; t < T-1; t++) {
7         for(i = 1; i < N+1; i++) {
8             for(j = 1; j < N+1; j++) {
9                 for(k = 1; k < N+1; k++) {
10                    A[(t+1)%2][i][j][k] = 0.125 * (A[t%2][i+1][j][k] - 2.0 * A[t%2][i][j][k] +
11                    A[t%2][i-1][j][k]) + 0.125 * (A[t%2][i][j+1][k] -
12                    2.0 * A[t%2][i][j][k] + A[t%2][i][j-1][k]) + 0.125 * (A[t%2][i][j][k-1] -
13                    2.0 * A[t%2][i][j][k] + A[t%2][i][j][k+1]) + A[t%2][i][j][k];
14                }
15            }
16        }
17    }
18 }

```

Figure 6.7: The baseline version of the finite-difference solution to the 3D heat equation in C.

```

1 dim=256;
2 Search {
3     buildcmd = "make_clean_all";
4     runcmd = "./heat3d";
5 }
6 CodeReg heat3d {
7     tileJ = poweroftwo(2..dim);
8     RoseUiuc.StripMine(loop=3, factor=tileJ);
9     RoseUiuc.Interchange(order="0,2,1,3,4");
10 }

```

Figure 6.8: Locus program for optimizing the finite-difference solution to the 3D heat equation. This program applies a strip-mine transformation on the  $j$  loop.

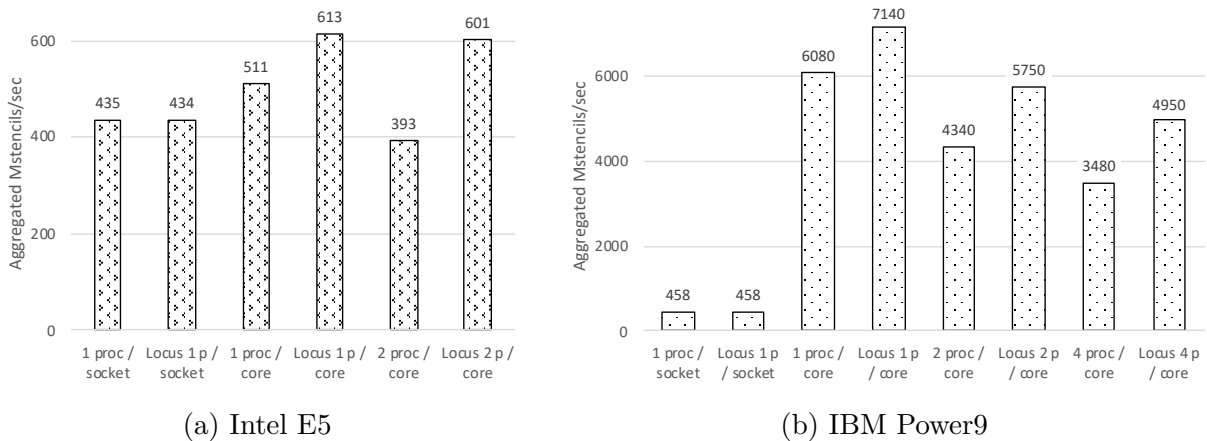


Figure 6.9: Weak scaling results for the 3D Heat stencil ( $256^3$  mesh size) on Intel E5 and IBM Power9. It shows the execution of 1, 10, and 20 processes running concurrently for Intel E5; 1, 20, 40, and 80 running concurrently on IBM Power9. Tiling appears most effective as the number of processes increases.

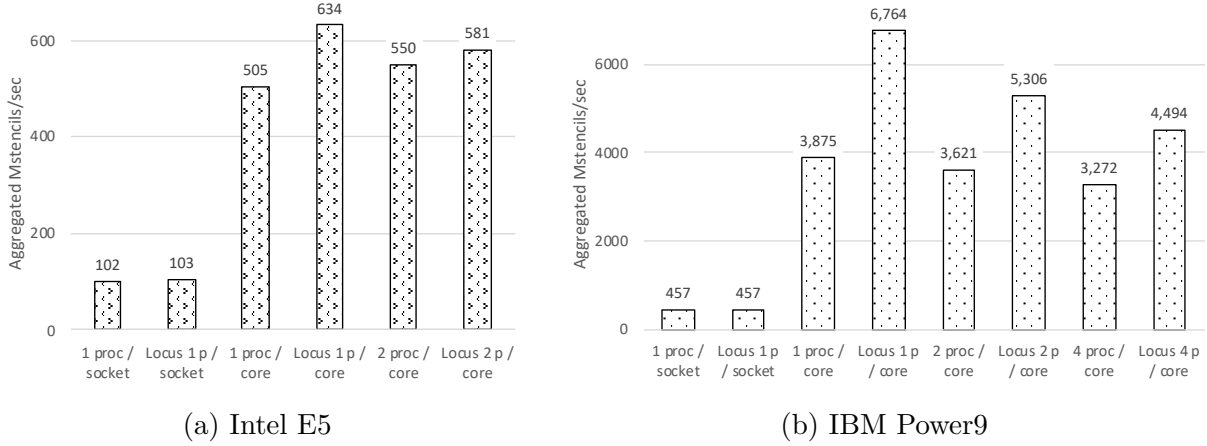


Figure 6.10: Strong scaling results for the 3D Heat stencil (1600<sup>3</sup> mesh size) on Intel E5 and IBM Power9. Only the  $Z$  dimension of the  $XYZ$  input volume is split among the processors. It shows the execution of 1, 10, and 20 processes running concurrently for Intel E5; 1, 20, 40, and 80 running concurrently on IBM Power9. Tiling appears most effective as the number of processes increases.

Figures 6.9 and 6.10 present weak scaling and strong scaling performance results comparing the baseline version and the best-tiled candidate implementation generated using Locus. The results were evaluated using 1, 10 (1 for each core), and 20 processes (2 for each core; HyperThreading is available on the processor) on Intel E5; and 1, 20, 40, and 80 processes on IBM Power9. The variants and the baseline were compiled with ICC on Intel E5 and XLC on IBM Power9. For the concurrent processes evaluation, GNU `parallel` command [60] was used to execute the same binary in parallel.

**Weak Scaling.** The stencil was executed on a 256<sup>3</sup> mesh of double-precision elements. On the Intel E5, with only one process on the socket, the tiled code does not improve in the aggregated number of millions of stencils performed per second. However, as access to the cache and memory becomes more competitive with the increasing number of processes, the tiled code performs better. The aggregated performance of the tiled code is approximately 20% higher compared to the baseline when running 10 processes (1 per core). The aggregated performance of the tiled code when using 20 processes is slightly worse than with 10 processes, which demonstrates the saturation of the memory subsystem. It is, however, 50% higher than the baseline performance. Similar conclusions can be drawn from the experiments on the IBM Power.

Despite the processes being independent of each other, the system's aggregated performance does not follow a linear increase as the number of processes increases. The memory bandwidth is a bottleneck in which optimizations such as strip-mine can help mitigate its

limitations. This contrasts with the observations in [61], for example, though those were for systems nearly a decade ago, and (except for results on the Cell processors), only one core was used in their work (see Table 2.1). However, similar to the findings in that paper and our own more recent work with the XPACC application program, it remains important to make the inner loop use stride-one indexing and be as long as possible to exploit both vectorization and memory optimizations such as prefetch.

The tile values for the best variants decreased as the number of processes increased. On Intel E5, for 10 processes, the best-found strip-mine value was 64, and for 20 processes was 16. On IBM Power9, for 20 processes, the best-found strip-mine value was 64, for 40 was 32, and for 80 was 16. As the caches are shared among more processes, the tile values have to be smaller to accommodate the increasing amount of data.

The optimal candidate implementation differed as to the concurrency in the processor increased; this information can be added to the database as a tag to the variants. Similarly, as for the matrix-matrix multiplication, the optimal code depends on the problem size.

**Strong Scaling.** Figure 6.10 presents the results for a mesh of  $1600^3$  double-precision elements. Through strip-mine, Locus could generate faster stencil code, similar to the results from the weak scaling experiments.

The best strip-mine values found on Intel E5 were 1024 for 1 process, 512 for 10 processes, and 256 for 20 processes. On IBM Power9, the best strip-mine values were 32 for 1 process, 16 for 20 processes, 8 for 40 processes, and 4 for 80 processes. Once again, as the number of processes increased, the tile size decreased. The best tiling values found on IBM Power9 were significantly smaller than the ones found on Intel E5. The gaps between the baseline and best candidate implementation were also bigger on IBM Power9.

## 6.6 FIXED VERSUS RANDOM INITIAL SEARCH CONFIGURATION

A Locus program can set the initial configuration of the search process. Figure 6.11 compares two search strategies for matrix-matrix multiplication on IBM Power9. The first search strategy, which we call *fixed*, starts with a configuration provided by the Locus program. This configuration is the one identified as the best for matrix-matrix multiplication on an Intel x86. The second search strategy uses a random configuration selected by the search module.

For the comparison, each of the two forms of search, fixed and random, is run ten times. For each run, we record the best execution time of matrix-matrix multiplication as the search progresses. The plot shows the best values obtained by each of the 10 runs at different search

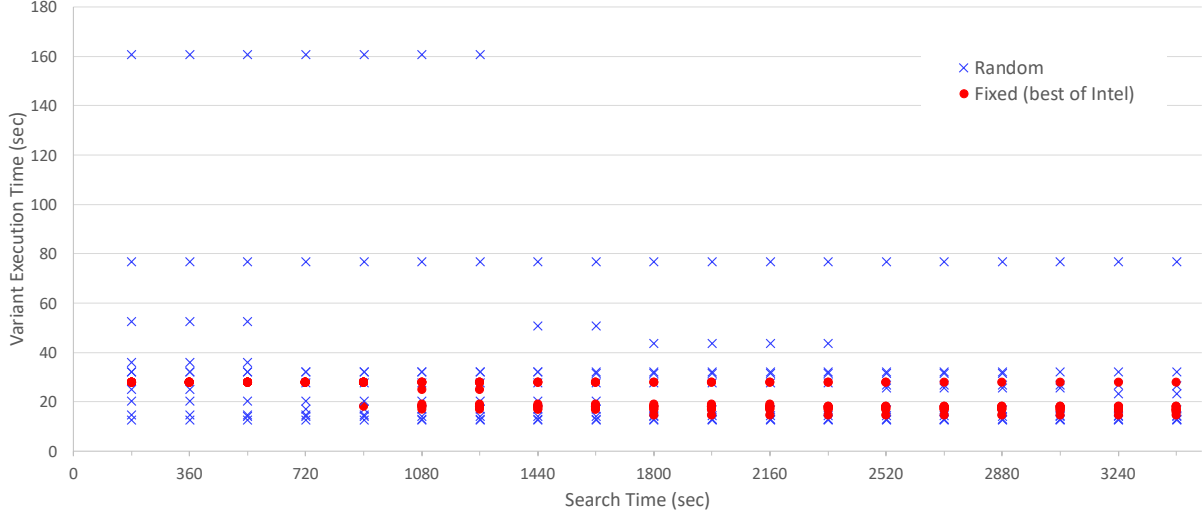


Figure 6.11: Execution time of best candidate implementation found as a function of search time for matrix-matrix multiplication on IBM Power9. Comparing random (the default) and fixed initial search configurations. The fixed one is the best found for the Intel E5 on a previous search process.

intervals. In some cases, the random search results in slightly better execution times but results in much slower values even for long searches. We conclude that although the use of an initial configuration based on results from another platform did not guarantee to find the best candidate implementation, it confined the search outcome to a more certain and narrower range (all results are below 28 seconds).

## 6.7 SUMMARY

In this chapter, we presented the use of Locus to define search spaces consisting of loop transformations for matrix-matrix multiplication and 1D, 2D, and 3D stencils.

Using the empirical search, Locus found better tile shapes that outperformed the Pluto compiler and Intel MKL library. In addition, we evaluated the use of an initial search configuration that provided narrower variability on the search process.

## CHAPTER 7: OPTIMIZING THROUGH DIVIDE AND CONQUER

Divide and conquer is a powerful strategy for solving many conceptually difficult problems. The idea relies on breaking the problem into smaller subproblems, solving them using (typically) simpler methods, and then combining the results to generate a solution to the original problem. These algorithms use recursion as their primary control structure to generate and solve the smaller subproblems. A base case is used to compute the smallest size subproblem.

They are a good match for modern parallel machines as they can benefit from high inherent parallelism and the prevalent deep memory hierarchy. The subproblems are usually independent and can be solved in parallel. The hierarchical division of the problem at multiple levels are inherently concurrent. Therefore, besides the base case, the division and combine phases of different paths on the recursion call tree can also be executed in parallel. These algorithms tend to perform well on cache-based systems because the cached-data reuse increases as the original problem is broken into subproblems small enough to fit entirely in the cache. As most of the computation takes place deep in the recursive call tree, the program's execution time is mostly spent using the cache effectively. Moreover, divide and conquer algorithms naturally work well with different memory hierarchy levels and cache sizes. They automatically adapt to the memory hierarchy and generally run well without modification in any machine. These are known as cache-oblivious algorithms [62].

In reality, however, there are some important trade-offs to exploit these divide and conquer properties fully. To be efficient, the programs must spend a significant part of the execution time in useful computation instead of dividing problems and combining results. The base case size controls the balance between the number of levels in the recursion call tree, which translates into the amount of divide and combine stages, and the amount of work per base case call. If the base case is too small, the overhead of the divide and combine phases are too high compared to the useful computation. On the other hand, if the base case is too large, the memory hierarchy is not efficiently exploited because the data is not small enough to fit in the caches. In addition, the amount of concurrency is reduced.

The simplest and most straightforward implementation reduces the problem to its minimum size, typically one element, before applying the very simple base case. Recursion, however, incurs high overhead, which often leads the programmers to unroll the recursion manually. Therefore, implementing efficient divide and conquer algorithms puts a heavy burden on the programmer [63].

Divide and conquer also provides opportunities for combining different algorithms at dif-

ferent levels of the recursion [4]. The best composition is highly dependent on the target architecture, the base case size, and the input, which results in no single, hard-coded composition optimal for all cases. Auto-tuning software has addressed these challenges, such as Atlas [5] and FFTW [64], which are, however, problem-specific and tailored to the few algorithms they support.

The complex search space found in divide and conquer algorithms puts the burden of implementing and evaluating the best optimal code on the programmer. As applications target multiple platforms, a diverse number of optimizations is necessary to approximate maximum computer power on each of them.

The search spaces for divide and conquer algorithms are often huge and virtually impossible to be exhaustively traversed. Auto-tuning has been a popular approach to find faster code with less program intervention. However, expressing these spaces concisely and efficiently is very challenging.

In the remainder of this chapter, we present ways to express the divide-and-conquer search space using Locus. We evaluated these on matrix-matrix multiplication, matrix transpose, fast Fourier Transform, symmetric eigenproblem, and sparse matrix-vector multiplication.

## 7.1 DIVIDE-AND-CONQUER SEARCH SPACE

The search space on breaking into subproblems and combining the solutions can be very large. Space grows exponentially to the number of problem dimensions that can be split up. An algorithm is cache-oblivious if no program variables depend on hardware parameters to be tuned, such as cache size or cache-line length [65]. For matrix-matrix multiplication and matrix transpose, cache-oblivious algorithms always split the largest dimension and go on until the base case size is achieved (theoretically up to a single element). In practice, a base case consisting of a single element is very inefficient. Therefore, even with a cache-oblivious strategy, searching for the best base case size and most efficient base case implementation is very important.

For some problems, though, the order in which the dimensions are divided may affect performance but incurs a much larger search space. In this more general approach, the recursion tree’s searching process can select stopping at any level for each branch, resulting in an unbalanced sub-tree. Figure 7.1 shows a balanced recursion tree representing multiple ways to subdivide a matrix transposition with two dimensions. The number of levels is the same for all branches, and the base case size on all leaves are also the same. Each node on the tree represents one possible candidate implementation to be selected during the search process.

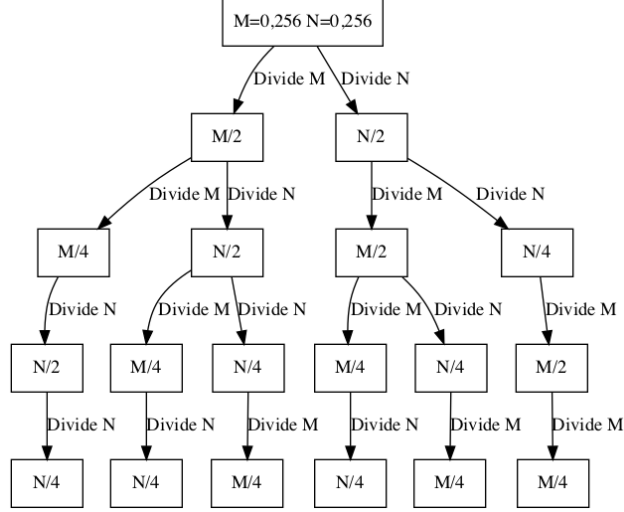


Figure 7.1: Recursive space of optimization for a transpose of a matrix  $m \times n$ ,  $m = n = 256$  and base case size minimum as 32 elements.

There are multiple ways to represent this recursive space. We devised the following classification of the search space for divide-and-conquer algorithms to capture the possibilities discussed:

- Uniform: the base case size is the only parameter of the search space. This is the cache-oblivious strategy. The order of dimensions being divided is always the same.
- Random-choice vertically: besides the base case size, the tree provides different choices vertically across levels. The order of dimensions being divided is part of the search space, but all the branches follow the same order, which generates a balanced tree.
- Random-choice horizontally: this is the most comprehensive space. It includes as parameters the base case size, different choices across levels, and horizontally between branches. In other words, it can generate unbalanced recursion trees.

We represent the random-choice strategies in Locus by combining recursion with OR blocks. We later refer to this nomenclature for the benchmarks and experiments.

The more an algorithm is subdivided, the smaller the base case size, which incurs more calls to the base case procedure. A procedure call is relatively expensive, and the amount of work performed by a procedure needs to be enough to compensate for the overhead. Compilers can inline the procedure body to avoid the call overhead. The search's goal is to find the right balance of which dimensions to split, how many levels, and base case size.

## 7.2 CODE GENERATION

The application’s source-code is annotated to define the code region to which the code generated using a divide-and-conquer strategy will be inserted. In our examples, the code region’s original code contains a single call for the base case to operate on the entire input data.

The base case implementation should be on the application’s source-code library and self-contained to independently compute a subproblem. In the Locus program, the developer needs to define an `OptSeq` to generate the code for invoking the base case each time the `OptSeq` is called by the Locus code generator. The resulting sequence of invocations to the base case is then inserted into the application’s code. The arguments about the subproblem are received by the `OptSeq` and added to the code generated. Not all the arguments of the base case are on the Locus program. Usually, only the ones related to the search space are referred.

As an example, in the case of matrix transpose, the `OptSeq transp(mS, mE, nS, nE)` (e.g., invoked at Line 15 in Figure 7.2a) is part of the `GenCode` collection. This `OptSeq` will generate the code to invoke the base case for matrix transpose. It receives only the dimensions of the submatrix: `mS` and `mE` for the first and last rows, and `nS` and `nE` for the first and last columns, respectively. The other arguments, such as the names of the variables that point to the input and output matrices, were pre-defined and are constant on the `transp(·)` implementation.

## 7.3 MATRIX TRANSPOSE

The transpose of an  $m \times n$  matrix is to compute and store  $A^T$  into an  $n \times m$  matrix  $B$ . We present three strategies for optimizing it using divide-and-conquer: cache-oblivious, random-choice vertically, and random-choice horizontally.

The system used for the experiments on matrix transpose was a processor Intel i7-3770 4-core at 3.4 GHz with 32 KB of L1 instruction and 32 KB of data cache, 256 KB of private L2 cache, and 8 MB of shared L3 cache, Ubuntu 16.04 operating system, Linux kernel 4.15.0, and 16 GB of RAM. The compilers’ versions used are GCC version 6.5, ICC 18.0.2, and Clang 8. We used OpenMP tasks to execute the base cases in parallel.

**Optimizing the Base Case.** The base case search space was to unroll for the factors  $u \in \{0, 4, 8\}$ . The evaluations were conducted with all variants using a cache-oblivious strategy and base case size to fit both matrices in the cache L2. For the two matrices to fit



```

1 mstop = nstop = 32;
2 OptSeq recCObliv(mS, mE, nS, nE)
3 {
4     mlen = mE-mS;
5     nlen = nE-nS;
6     if nlen >= mlen && nlen > nstop {
7         pivn = nS + nlen/2;
8         recCObliv(mS, mE, nS, pivn);
9         recCObliv(mS, mE, pivn, nE);
10    } elif mlen > mstop {
11        pivm = mS + mlen/2;
12        recCObliv(mS, pivm, nS, nE);
13        recCObliv(pivm, mE, nS, nE);
14    } else {
15        GenCode.Transp(mS, mE, nS, nE);
16    }
17 }
18 CodeReg transp {
19     mstop = poweroftwo(mstop..melem/2);
20     nstop = mstop;
21     recCObliv(0, melem, 0, nelem);
22 }

```

(a) Cache oblivious.

```

1 mstop = nstop = 32;
2 OptSeq recVector(mS, mE, nS, nE, vec, lvl)
3 {
4     nextlvl = lvl + 1;
5     mlen = mE-mS;
6     nlen = nE-nS;
7     if veclvl[lvl] == "splitN" {
8         pivn = nS + nlen/2;
9         recVector(mS, mE, nS, pivn, vec, nlvl);
10        recVector(mS, mE, pivn, nE, vec, nlvl);
11    } elif veclvl[lvl] == "splitM" {
12        pivm = mS + mlen/2;
13        recVector(mS, pivm, nS, nE, vec, nlvl);
14        recVector(pivm, mE, nS, nE, vec, nlvl);
15    } elif veclvl[lvl] == "leaf" {
16        GenCode.transp(mS, mE, nS, nE);
17    }
18 }
19 OptSeq recVert(mS, mE, nS, nE, vec, lvl)
20 {
21     nlvl = lvl + 1;
22     mlen = mE-mS;
23     nlen = nE-nS;
24     {
25         if nlen > nstop {
26             pivn = nS + nlen/2;
27             veclvl = veclvl + ["splitN"];
28             recVert(mS, mE, nS, pivn, vec, nlvl);
29             recVector(mS, mE, pivn, nE, vec, nlvl);
30         }
31     } OR {
32         if mlen > mstop {
33             pivm = mS + mlen/2;
34             veclvl = veclvl + ["splitM"];
35             recVert(mS, pivm, nS, nE, vec, nlvl);
36             recVector(pivm, mE, nS, nE, vec, nlvl);
37         }
38     } OR {
39         veclvl = veclvl + ["leaf"];
40         GenCode.transp(mS, mE, nS, nE);
41     }
42 }
43 CodeReg transp {
44     @pragma expandrec inline
45     recVert(0, melem, 0, nelem, [], 0);
46 }

```

(b) Random-choice vertically.

Figure 7.2: Locus programs for matrix transpose.

in the L2 cache, we used  $m' = n' = \left\lfloor \sqrt{\frac{\mathcal{C}}{2\mathcal{E}}} \right\rfloor$ , where  $\mathcal{C}$  is the cache's size in bytes, and  $\mathcal{E}$  is the number of bytes per element of the matrix (8 for double-precision in experiments case).

The data for all compilers and threads  $t \in \{1, 2, 3, 4\}$  are in Figure 7.3. The different compilers presented very similar behavior. The performance did improve by using more threads, but the speedup was not significant. The matrix transposition is very memory intensive, and we believe that this presented a barrier to better performance when using one thread per core.

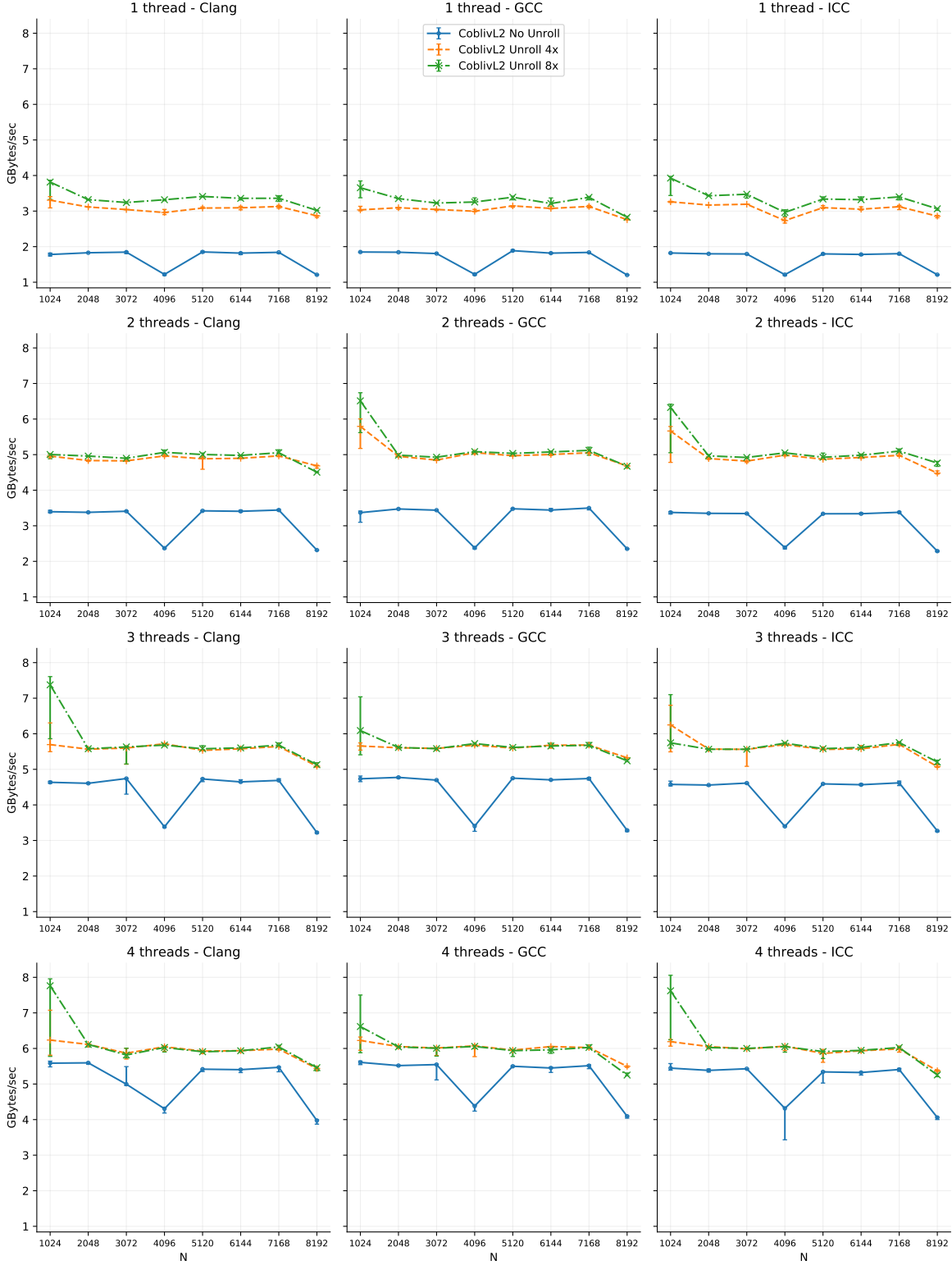


Figure 7.3: Optimizing matrix transpose's base case implementation. The unrolling factors  $u \in \{0, 4, 8\}$  were compared using a cache-oblivious strategy with values fixed to fit in the L2 cache for threads  $t \in \{1, 2, 3, 4\}$  and Clang, GCC, and ICC compilers.

**Cache Oblivious.** Optimal work and cache complexities can be obtained with a divide-and-conquer strategy [65]. If  $n \geq m$ , we partition

$$A = (A_1 \quad A_2), \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}. \quad (7.1)$$

Then,  $\text{Transpose}(A_1, B_1)$  and  $\text{Transpose}(A_2, B_2)$  are recursively executed. Conversely, if  $m > n$ , matrix  $A$  is divided horizontally, matrix  $B$  vertically, and, similarly, performs two transpositions recursively.

Figure 7.2a presents the cache-oblivious implementation using the Locus language. The cache-oblivious strategy splits the biggest dimension, as shown by the if-then-else in Lines 6-16 of the code. It also checks whether the divide has reached the *stop* value, which is the value selected for the base case’s execution. Lines 19 and 20 present the selection of the size base case (i.e., *stop*) in the search space. The same value is used for all the dimensions.

**Random-choice Vertically.** Figure 7.2b presents the random-choice vertically. In essence, this version has the freedom to choose which dimension to split at each level (vertically in the recursion tree). Differently to the cache-oblivious case, the base case size may have different values for each dimension. However, it replicates all the decisions horizontally.

The base case size (i.e., *stop*) has a lower bound. The splitting, however, can stop at any value greater than that and is dimension independent.

The optimization sequence *recVert* (Line 19) contains three OR blocks (lines 24-41) that opens up the possibilities of dividing in any dimension as it calls itself recursively. It does a depth-first traversal of the tree of possible divisions. It then saves this traversal in an array that contains a decision for each level (Lines 27, 34, and 39). The array is used to replicate the decisions into the other branches of the tree by calling the optimization sequence *recVector* (Line 2) with it as an argument.

**Random-choice Horizontally.** It is very similar to one presented for matrix-matrix multiplication in Figure 7.7b. The difference is that the matrix transpose has one less dimension to subdivide. Therefore, it has two OR blocks instead of the three shown.

Figure 7.4 shows the matrix transpose performance for matrices with 1024 by 1024 to 8192 by 8192 elements, using the three compilers, and running 1 to 4 threads. All the strategies present similar results, except the random-choice horizontally strategy that degrades performance as the matrix size increases.

Figure 7.5 presents the speedup relative to the optimized baseline for 8192 elements. The

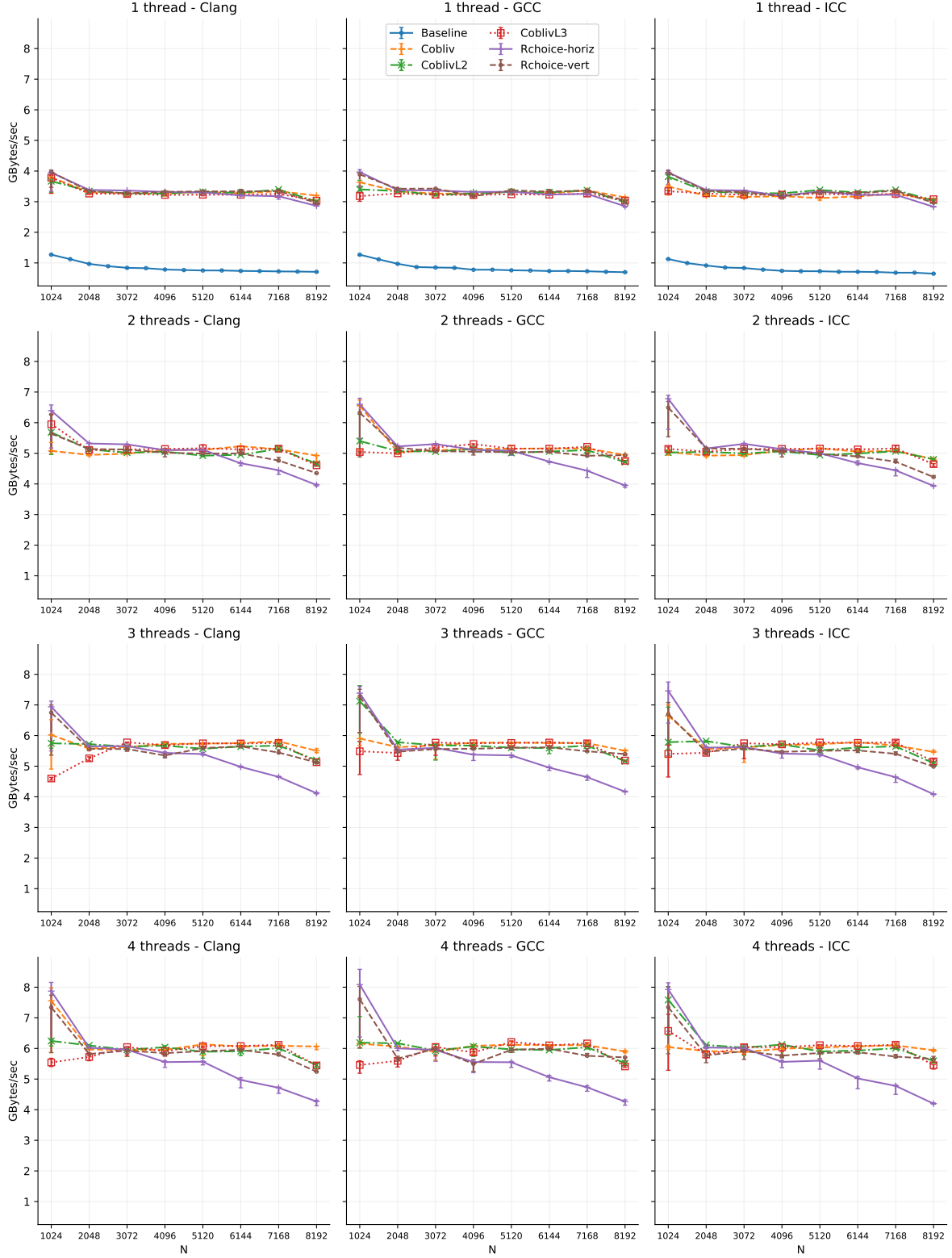


Figure 7.4: Matrix transpose results for best of baseline, best of cache-oblivious (*Cobliv*), cache-oblivious limited to L2 (*CoblivL2*) and L3 (*CoblivL3*) cache sizes, random-choice vertically (*Rchoice-vert*), and random-choice horizontally (*Rchoice-horiz*).

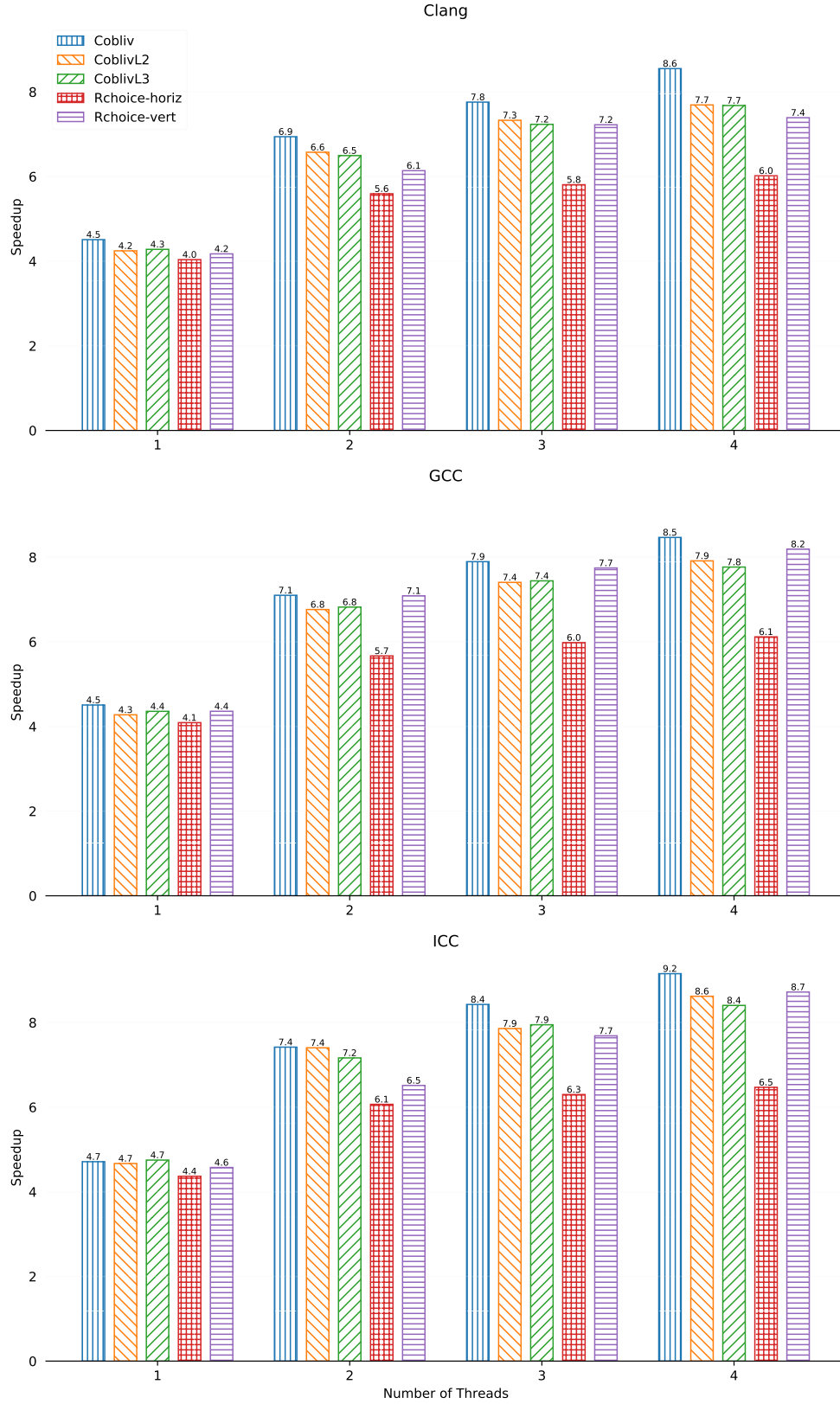


Figure 7.5: Matrix transpose speedup over optimized baseline,  $N = 8192$ .

cache-oblivious has the best results for the three compilers and 1 to 4 threads. Cache-oblivious strategies with base case sizes fixed to fit in L2 and L3 caches showed performance close to the non-fixed one and the random-choice vertically version. The random-choice horizontally results are the slowest. Although the cache-oblivious and random-choice vertically are subspaces of the random-choice horizontally strategy, the search tools could not explore the space of the random-choice horizontally efficiently and, at least, match results of the other ones under the given time limit.

## 7.4 MATRIX-MATRIX MULTIPLICATION

This section describes and analyzes three different strategies to optimize through divide-and-conquer an algorithm for multiplying an  $m \times n$  matrix by an  $n \times p$  matrix.

The system used for the experiments on matrix-matrix multiplication was an Intel E5-1630v3 4-core processor at 3.7 GHz with 32 KB of instruction and 32 KB of data L1 cache, 256 KB of private L2 cache, and 10 MB of shared L3 cache, Ubuntu 18.04 operating system, Linux kernel 4.15.0, and 32 GB of RAM. The compilers' versions used are GCC version 7.5, ICC 19.0.5, and Clang 9. We used OpenMP tasks to run the base cases in parallel.

**Optimizing the Base Case.** The reference code used for the base case is the same shown in Figure 1.1a. The loop nest was first interchanged from  $IJK$  to  $IKJ$  that we already have found in earlier experiments to improve performance significantly. The base case search space consisted of unroll-and-jam the outermost loops  $I$  and  $K$  by factors 4 and 8, and unroll the innermost loop  $J$  by factor 4.

The results for  $m = n = p = N$ , Clang, GCC, ICC compilers, and threads  $t \in \{1, 2, 3, 4\}$  are in Figure 7.6. The evaluations were conducted using a cache-oblivious strategy. For the three base case submatrices to fit in cache,  $m' = n' = p' = \left\lfloor \sqrt{\frac{\mathcal{C}}{3\mathcal{E}}} \right\rfloor$ , where  $\mathcal{C}$  is the size of the cache in bytes, and  $\mathcal{E}$  is the number of bytes per element of the matrix (8 for double-precision in experiment case). These evaluations, however, used a base case size of  $N/8$  that varied according to the matrix size. Only one configuration stood out on the results from 1 to 4 threads. For Clang and GCC, it was the one with unroll-and-jam of  $I$  and  $K$  by 4. For ICC, the best was with unroll-and-jam  $K$  by 4.

**Cache Oblivious.** According to [65], the optimal divide-and-conquer algorithm for multiplying an  $m \times n$  matrix  $A$  by an  $n \times p$  matrix  $B$  halves the largest of the three dimensions.

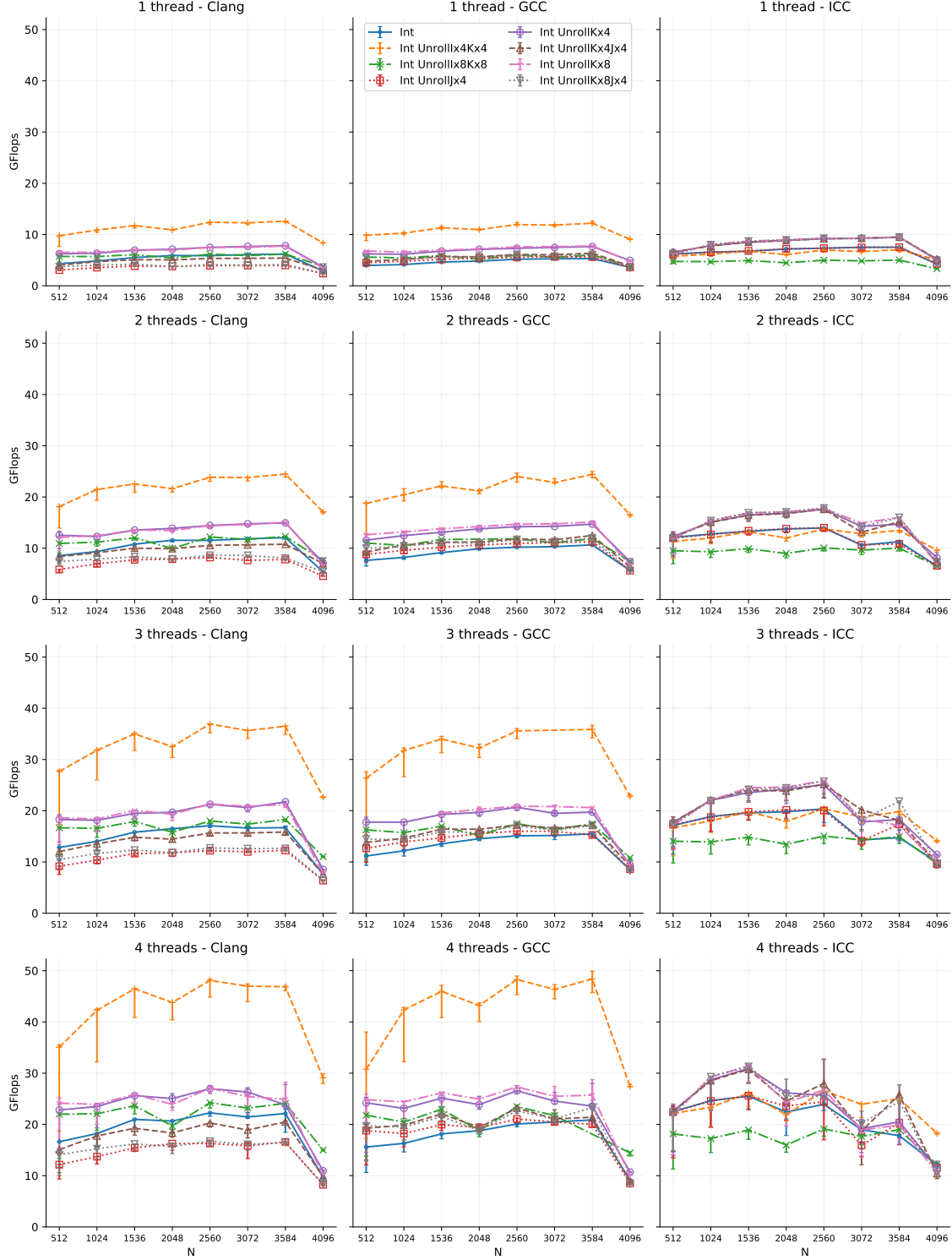


Figure 7.6: *DGEMM* evaluation for  $m = n = p = N$  using a cache-oblivious strategy with base case size fixed to 3 recursion levels on each dimension (i.e.,  $N/8$ ). The loop nest was first interchanged from  $IIK$  to  $IKJ$ , and different configurations of unroll ( $J \in \{4\}$ ) and unroll-and-jam ( $I, K \in \{4, 8\}$ ) are presented.

```

1  mstop = nstop = kstop = 32;
2  OptSeq recCObliv(mS, mE, nS, nE, kS, kE)
3  {
4      mlen = mE-mS;
5      nlen = nE-nS;
6      klen = kE-kS;
7      maxnk = nlen < klen ? klen : nlen;
8      maxmk = mlen < klen ? klen : mlen;
9      maxmn = mlen < nlen ? nlen : mlen;
10     if mlen >= maxnk && mlen > mstop {
11         "Split in M";
12         pivm = mS + mlen/2;
13         recCObliv(mS, pivm, nS, nE, kS, kE);
14         recCObliv(pivm, mE, nS, nE, kS, kE);
15     } elif nlen >= maxmk && nlen > nstop {
16         "Split in N";
17         pivn = nS + nlen/2;
18         recCObliv(mS, mE, nS, pivn, kS, kE);
19         recCObliv(mS, mE, pivn, nE, kS, kE);
20     } elif klen >= maxmn && klen > kstop {
21         "Split in K";
22         pivk = kS + klen/2;
23         recCObliv(mS, mE, nS, nE, kS, pivk);
24         recCObliv(mS, mE, nS, nE, pivk, kE);
25     } else {
26         GenCode.Gemm(mS, mE, nS, nE, kS, kE);
27     }
28 }
29 CodeReg gemm {
30     mstop = poweroftwo(mstop..melem/2);
31     nstop = mstop;
32     kstop = mstop;
33     recCObliv(0, melem, 0, nelem, 0, kelem);
34 }

1  mstop = nstop = kstop = 32;
2  OptSeq recHoriz(mS, mE, nS, nE, kS, kE) {
3      mlen = mE-mS;
4      nlen = nE-nS;
5      klen = kE-kS;
6      {
7          if mlen > mstop {
8              pivm = mS + mlen/2;
9              recHoriz(mS, pivm, nS, nE, kS, kE);
10             recHoriz(pivm, mE, nS, nE, kS, kE);
11         }
12     } OR {
13         if nlen > nstop {
14             pivn = nS + nlen/2;
15             recHoriz(mS, mE, nS, pivn, kS, kE);
16             recHoriz(mS, mE, pivn, nE, kS, kE);
17         }
18     } OR {
19         if klen > kstop {
20             pivk = kS + klen/2;
21             recHoriz(mS, mE, nS, nE, kS, pivk);
22             recHoriz(mS, mE, nS, nE, pivk, kE);
23         }
24     } OR {
25         GenCode.Gemm(mS, mE, nS, nE, kS, kE);
26     }
27 }
28 CodeReg gemm {
29     @pragma expandrec inline
30     recHoriz(0, melem, 0, nelem, 0, kelem);
31 }

```

(a) Cache oblivious.

(b) Random-choice horizontally.

Figure 7.7: Locus programs for matrix-matrix multiplication.

It recurs according to one of the following three cases:

$$AB = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}, \quad (7.2)$$

$$AB = (A_1 \ A_2) \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2, \quad (7.3)$$

$$AB = A (B_1 \ B_2) = (AB_1 + AB_2). \quad (7.4)$$

In case 7.2, we have  $m \geq \max\{n, p\}$ , and  $A$  is split horizontally, both halves are multiplied by matrix  $B$ . In the case 7.3, we have  $n \geq \max\{m, p\}$ , both matrices are split, and the two halves multiplied. In the case 7.4, we have  $p \geq \max\{m, n\}$ , and matrix  $B$  is split vertically, and each half is multiplied by  $A$ .

The base case for the *original* cache-oblivious occur when  $m = n = p = 1$ . That is



when two elements are multiplied and added to the result matrix. In our implementation, however, the base case size is selected by the empirical process. In Figure 7.7a, there is the cache-oblivious implementation using Locus language. The cache-oblivious strategy splits the biggest dimension, as shown in Lines 12, 17, and 22 of the code. It also checks whether the base case size has reached the *stop* value, which is the value selected for the execution of the base case. Lines 30, 31, and 32 present the selection of the size base case (*stop* variable) in the search space. The same value is used for all the dimensions.

**Random-choice Vertically.** The code is similar to the one shown in Figure 7.2b. The difference is that the matrix-matrix multiplication contains one extra dimension to subdivide.

**Random-choice Horizontally.** Figure 7.7b presents the random-choice horizontally implementation using Locus language. It extends the number of possibilities compared to the random-choice vertically by allowing independent decisions for each recursive subdivision.

Figure 7.8 shows the performance for matrices with 512 by 512 to 4096 by 4096 elements, using the three compilers, and running 1 to 4 threads. The cache-oblivious strategy is the best one as the matrices size and the number of threads increase. However, the random-choice vertically strategy outperformed the cache-oblivious for 4096 by 4096 matrices with GCC and Clang and executing 4 threads.

Figure 7.9 presents the speedup over optimized reference for Clang and GCC compilers, 1 to 4 threads, and matrices with 4096 by 4096 elements. Cache-oblivious and random-choice vertically showed the best results. The cache-oblivious fixed to the base case fit in the L3 cache was significantly slower than the non-fixed cache-oblivious results. Similar to the results with the matrix transpose, the random-choice horizontally results were meager. Although the cache-oblivious and random-choice vertically are subspaces of the random-choice horizontally strategy, the search tools could not explore the space of the random-choice horizontally efficiently and, at least, match results of the other ones under the given time limit.

## 7.5 FAST FOURIER TRANSFORM

The fast Fourier transform (FFT) is an efficient way to convert a signal from its original domain (time or space) to the frequency and vice-versa. It is extensively used in digital signal processing, and its applications are numerous, including data compression, image and sound analysis, and telecommunications.

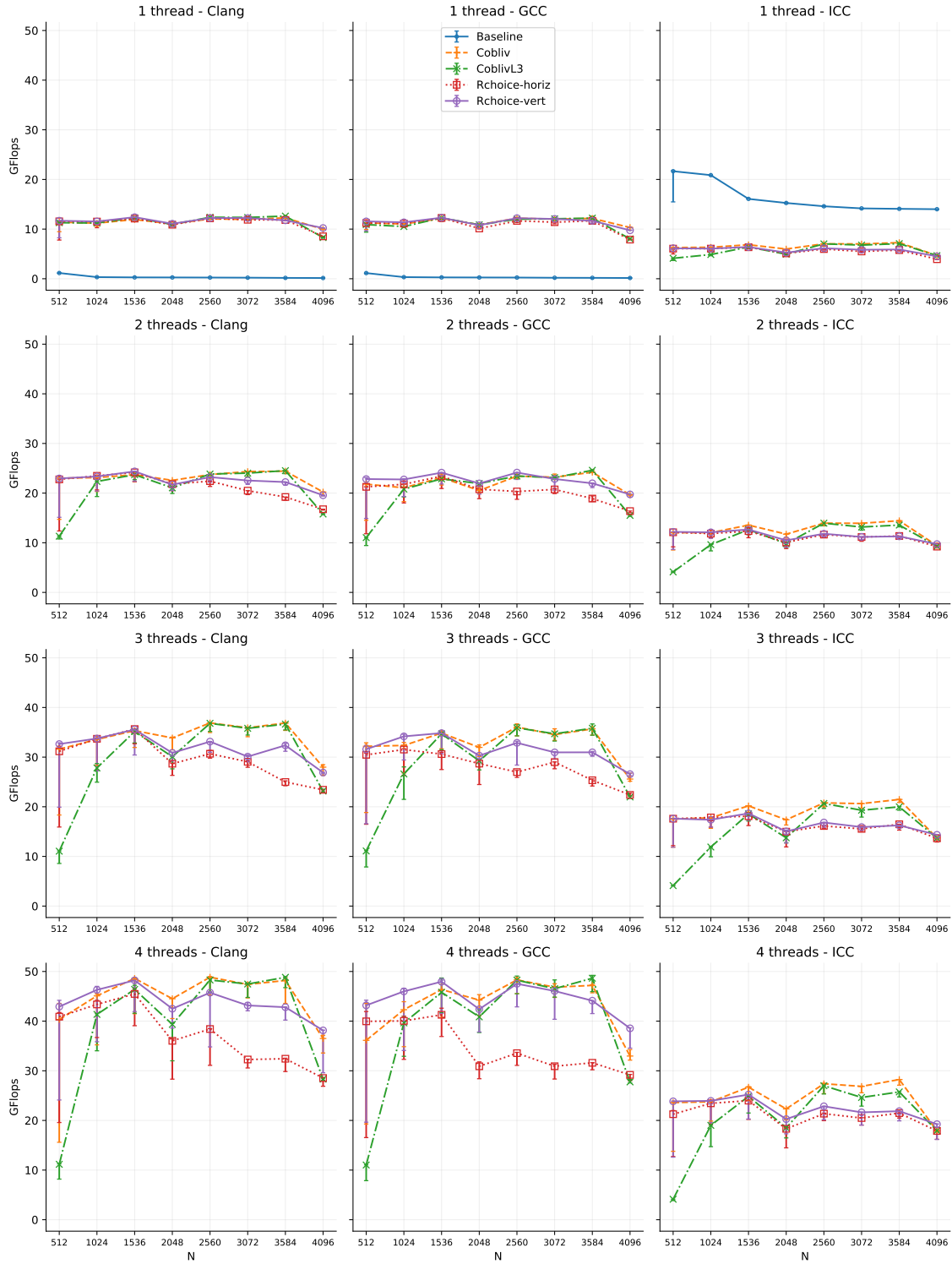


Figure 7.8: *DGEMM* results for best of reference, cache-oblivious (*Cobliv*), cache-oblivious limited to L3 cache size (*CoblivL3*), random-choice vertically (*Rchoice-vert*), and random-choice horizontally (*Rchoice-horiz*).

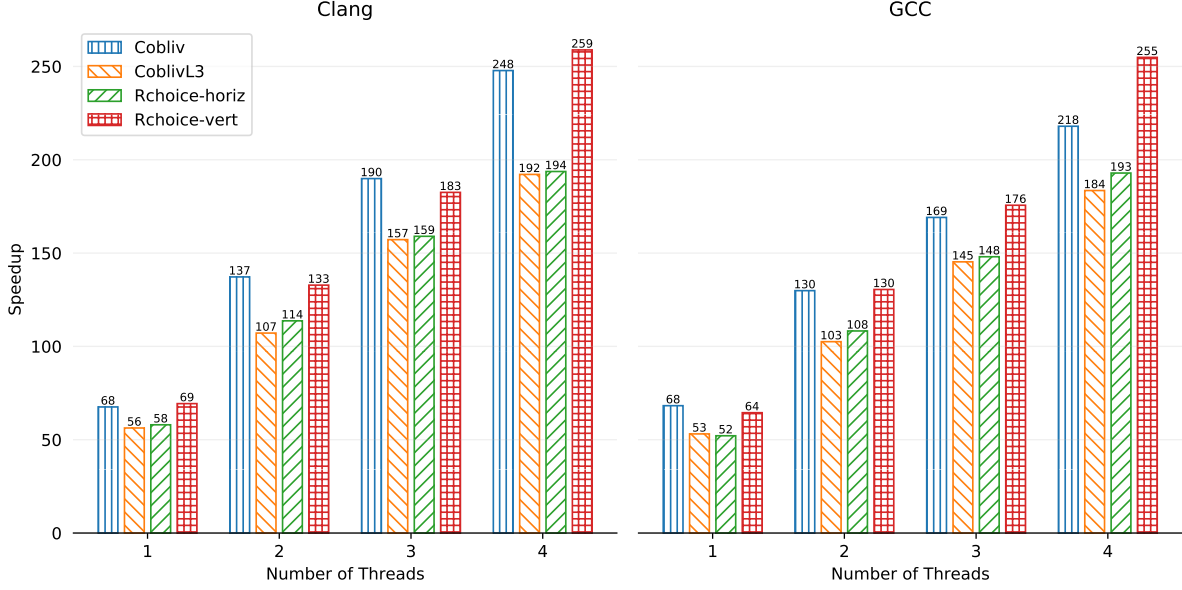


Figure 7.9: *DGEMM* speedup over the optimized reference,  $N = 4096$ .

FFTW [64, 66] is a widely used library for applications that require FFT. It can compute a discrete Fourier transform (DFT) in problems with  $n$  dimensions, arbitrary size, and both real and complex data. It includes a collection of algorithms structured as a library of *codelets*. A codelet is a C subroutine that computes a Fourier transform of a fixed size. The codelets are produced automatically by the FFTW special-purpose compiler. A composition of codelets is called a *plan*. The precise plan used to compute the FFT depends on the input dimensions and on which codelets happen to run faster on the underlying system. The plan is devised automatically. The user, however, can select which heuristic to use for finding the fastest plan. The longer the planning time, the more likely it is to produce a faster plan. The FFTW *planner* accepts four modes:

- *Estimate*: instead of actual measurements of different composition of codelets, a simple heuristic is used to select a plan as fast as possible;
- *Measure*: find an optimized plan by actually computing several FFTs and measuring their execution time. This is the default;
- *Patient*: similar to *Measure*, but considers a wider range of algorithms and often produces a faster plan, but at the expense of several times longer planning time;
- *Exhaustive*: similar to *Patient*, but considers an even wider range of algorithms, including many that authors would assume are unlikely to be fast. Planning time is substantially larger.

```

1  OptSeq recdft(n, r, vecrnk) {
2      if n > stop {
3          {
4              Nvecrnk = vecrnk + 1;
5              if vecrnk > 1 {
6                  GenCode.ProlVT();
7                  Nvecrnk = vecrnk;
8              }
9              {
10                 newr = enum(2,4,8,16,32,64);
11                 newm = n/newr;
12                 if newm > 1 {
13                     GenCode.ProlDftCt();
14                     recdftw(newm, newr, Nvecrnk);
15                     recdft(newm, newr, Nvecrnk);
16                     GenCode.EpilDftCt();
17                 }
18             }
19             if vecrnk > 1 {
20                 GenCode.EpilVT();
21             }
22         } OR {
23             # Direct solvers
24             if vecrnk > 1 {
25                 GenCode.ProlVT();
26             }
27             if n == 2 {
28                 GenCode.N1fv_2() OR
29                 GenCode.N2fv_2();
30             } elif n == 4 {
31                 GenCode.N1fv_4() OR
32                 GenCode.N2fv_4();
33             } elif n == 8 {
34                 GenCode.N1fv_8() OR
35                 GenCode.N2fv_8();
36             } elif n == 16 {
37                 GenCode.N1fv_16() OR
38                 GenCode.N2fv_16();
39             } elif n == 32 {
40                 GenCode.N1fv_32() OR
41                 GenCode.N2fv_32();
42             } elif n == 64 {
43                 GenCode.N1fv_64() OR
44                 GenCode.N2fv_64();
45             } elif n == 128 {
46                 GenCode.N1fv_128();
47             } else {
48                 GenCode.Iterative();
49             }
50             if vecrnk > 1 {
51                 GenCode.EpilVT();
52             }
53         }
54     }
55 }
56 CodeReg fft {
57     GenCode.ProlAiplan();
58     @pragma expandrec inline
59     recdft(nelem, 1, 0);
60     GenCode.EpilAiplan();
61 }

```

```

1  OptSeq recdftw(n, r, vecrnk) {
2      {
3          if r == 2 {
4              GenCode.T1fv_4() OR
5              GenCode.T2fv_4();
6          } elif r == 4 {
7              GenCode.T1fv_4() OR
8              GenCode.T2fv_4() OR
9              GenCode.T3fv_4();
10         } elif r == 8 {
11             GenCode.T1fv_8() OR
12             GenCode.T2fv_8() OR
13             GenCode.T3fv_8();
14         } elif r == 16 {
15             GenCode.T1fv_16() OR
16             GenCode.T2fv_16() OR
17             GenCode.T3fv_16();
18         } elif r == 32 {
19             GenCode.T1fv_32() OR
20             GenCode.T2fv_32() OR
21             GenCode.T3fv_32();
22         } elif r == 64 {
23             GenCode.T1fv_64() OR
24             GenCode.T2fv_64();
25         } else {
26             GenCode.Iterative();
27         }
28     } OR {
29         GenCode.ProlDftw-Generic();
30         recdft(n, r, vecrnk);
31         GenCode.EpilDftw-Generic();
32     }
33 }

```

Figure 7.10: Locus program for fast Fourier transform.

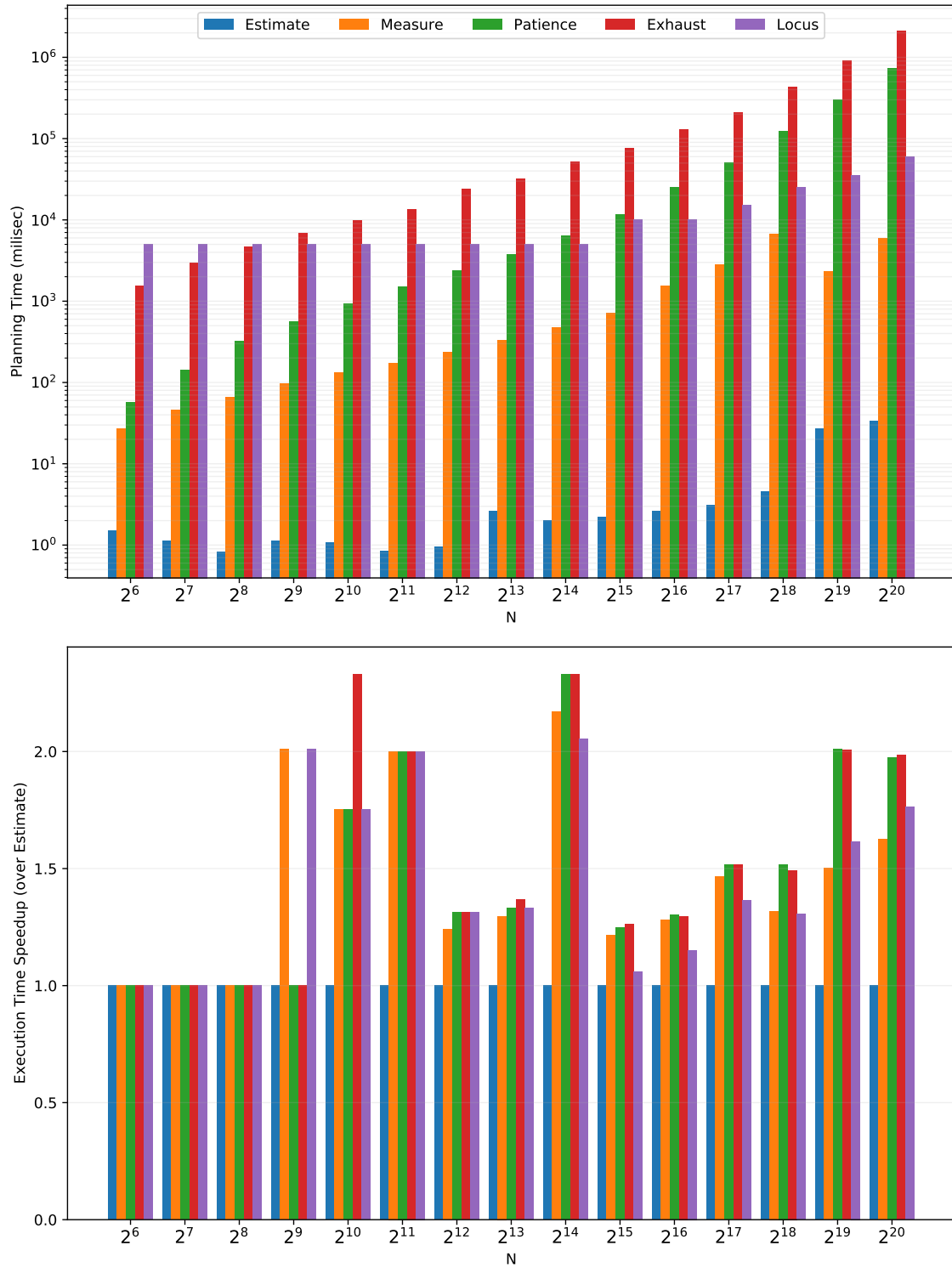


Figure 7.11: FFT Planning time (top) and execution time speedup over the best plan found by FFTW Estimate (bottom) of an array of  $N$  complex double-precision elements.

A discrete Fourier transform in FFTW is expressed in terms of structures of I/O tensors, which are, in turn, described in terms of I/O dimensions. An I/O dimension  $d$  is a tuple  $d = (n, \iota, o)$ , where  $n$  is a non-negative integer called length,  $\iota$  is an integer called the input stride, and  $o$  is an integer called the output stride. An I/O tensor  $t = \{d_1, d_2, \dots, d_\rho\}$  is a set of I/O dimensions. The non-negative integer  $\rho = |t|$  is called the rank of the I/O tensor. A DFT problem  $dft(N, V, I, O)$  consists of two I/O tensors  $N$  and  $V$ , and two pointers  $I$  and  $O$ .

The FFTW planner explores a space of valid plans for a given problem selecting the fastest one using different heuristics. Many plans exist to solve a given DFT problem. FFTW takes three steps to solve a general DFT problem: reduce a problem of arbitrary vector rank to a vector rank-0; reduces multi-dimensional DFT to a sequence of rank-1 problems (i.e., one-dimensional DFTs); solve the rank-1, vector rank-0 problem by some DFT algorithm such as Cooley-Tukey.

Rank-1 DFT problems denote ordinary Fourier transformations. FFTW solves most of the rank-1 problems through direct and Cooley-Tukey plans. Other kinds of plans exist and are applied to certain special cases.

*Direct* plans are provided for small rank-1 problems. These plans solve the problem directly in a single call. This applies to problems  $dft(\{(n, \iota, o)\}, V, I, O)$ , where  $|V| \leq 1$  and  $n \in \{2, \dots, 16, 32, 64\}$ . These plans operate by calling a C codelet specialized for one particular size.

Cooley-Tukey plans implement a radix- $r$  Cooley-Tukey algorithm for problems of the form  $dft(\{(n, \iota, o)\}, V, I, O)$  where  $n = rm$ . FFTW generates a plan for each suitable value of  $r$ , in addition to a direct plan. It implements two of the many known Cooley-Tukey algorithms, distinguished mainly by whether the codelets multiply their inputs or outputs by twiddle factors.

A *decimation in time* (DIT) plan solves  $dft(\{(m, r \cdot \iota, o)\}, V \cup \{r, \iota, m \cdot o\}, I, O)$ , multiplies the output array  $O$  by the twiddle factors, and finally solves  $dft(\{(r, m \cdot o, m \cdot o)\}, V \cup \{m, o, o\}, O, O)$ . For performance, the last two steps are fused in a twiddle codelet, which multiplies its input by the *twiddle* factors and performs a DFT of size  $r$ , operating in-place on  $O$ . FFTW contains one such codelet for each  $n \in \{2, \dots, 16, 32, 64\}$ .

A *decimation in frequency* (DIF) plan operates backward with respect to a DIT plan. The plan solves  $dft(\{r, m \cdot \iota, m \cdot \iota\}, V \cup \{m, \iota, \iota\}, I, I)$ , multiplies the input array  $I$  by the twiddle factors, and finally solves  $dft(\{m, \iota, r \cdot o\}, V \cup \{r, m \cdot \iota, o\}, I, O)$ . The first two steps are fused in a single codelet. The DIF destroys the input array, and FFTW generates them only if  $I = O$ .

The heuristic used to select the solvers is pretty much a closed box. It is possible to

include new solver implementations, but the planner heuristic is very hard to modify.

The advantage of exposing this search space is that we can use different search techniques to find the fastest sequence of solvers for a particular DFT problem. Performance expert knowledge can be included to reduce the planning time. The space can be segmented according to the characteristics of the target platform and input problem.

Using Locus, the search space can be programmed and explored by a diverse set of search techniques. Figure 7.10 presents the FFTW search space in Locus notation. The space presented works for rank-1, power of 2 length problems in which all the problems are reduced to as discussed. It generates only DIT plans but could be easily extended to include DIF plans as well. The FFTW interface was modified to accept the invocation of any solver externally, including the ones relying on recursion.

The experimental results are shown in Figure 7.11. They were executed on the same system used in Section 7.3. The top chart presents the planning time. The lower chart presents the execution time of the best candidates generated during the planning time. It is worth pointing out that the planning time of the FFTW is done by manipulating a dependence graph between the already-compiled solvers. In our Locus approach, each candidate had to be generated into a temporary file, compiled, and executed. Despite the extra overheads that Locus had for generating and compiling each candidate, the planning times were relatively similar. The best candidates found also had similar execution times. Our prototype did not include all the solvers included in the FFTW. Some of the ones missing are especially efficient for bigger input sizes. The inclusion of these solvers should close the gap between the best candidates found by Locus and the ones used by the FFTW.

## 7.6 SYMMETRIC EIGENPROBLEM

Eigenvalues and eigenvectors of symmetric matrices have a broad range of applications in many areas, such as search engines, population studies, quantum mechanics, and aeronautics. Let  $A$  be a real symmetric or complex Hermitian  $n \times n$  matrix. If  $Ax = \lambda x$ , the scalar  $\lambda$  is called an eigenvalue and a nonzero column vector  $x$  the respective eigenvector.

We examine 3 direct methods for the symmetric eigenproblem: Tridiagonal QR iteration, divide-and-conquer, and Bisection and inverse iteration. All the algorithms assume that  $A = A^\top$  and the orthogonal matrix  $Q$  has been found so that  $A = QTQ^\top$  and  $T$  is tridiagonal. The eigenvalues of  $A$  and  $T$  are the same, and the eigenvectors of  $A$  are obtained multiplying  $Q$  by the eigenvectors of  $T$  [67].

The Tridiagonal QR iteration finds all eigenvalues of  $T$  in 2 QR steps per eigenvalue on average, resulting in  $O(n^2)$  operations. However, to find all the eigenvectors as well, the QR

```

1 stop = 16;
2 OptSeq recCObliv(mS, mE)
3 {
4   len = mE-mS;
5   if len > stop {
6     piv = mS + len/2;
7     GenCode.DCProlog(mS, mE, elem);
8     recCObliv(mS, piv);
9     recCObliv(piv, mE);
10    GenCode.DCEpilog(mS, mE, elem);
11  } else {
12    {
13      GenCode.QRTD(mS, mE, elem);
14    } OR {
15      GenCode.Bisect(mS, mE, elem);
16    }
17  }
18 }
19 CodeReg eig {
20   stop = poweroftwo(stop..elem/2);
21   recCObliv(0, elem);
22 }

```

(a) Cache oblivious.

```

1 stop = 16;
2 OptSeq recEigen(mS, mE)
3 {
4   len = mE-mS;
5   {
6     piv = mS + len/2;
7     if len > stop {
8       GenCode.DCProlog(mS, mE, elem);
9       recEigen(mS, piv);
10      recEigen(piv, mE);
11      GenCode.DCEpilog(mS, mE, elem);
12    }
13  } OR {
14    GenCode.QRTD(mS, mE, elem);
15  } OR {
16    GenCode.Bisect(mS, mE, elem);
17  }
18 }
19 CodeReg eig {
20   @pragma expandrec inline
21   recEigen(0, elem);
22 }

```

(b) Random-choice horizontally.

Figure 7.12: Locus programs for symmetric eigenproblem.

iteration takes  $O(n^3)$  operations. The divide-and-conquer approach computes the eigenvalues and eigenvectors of  $T$  by composing the results of computing the ones from the two diagonal sub-matrices  $T_1$  and  $T_2$  of  $T = \begin{pmatrix} T_1 & 0 \\ 0 & T_2 \end{pmatrix}$ . This approach can be applied recursively and also results in  $O(n^3)$  complexity.

The Bisection and inverse iteration may be used to find a subset of the eigenvalues of a symmetric tridiagonal matrix. It requires  $O(kn)$  operations, where  $k$  is the number of eigenvalues desired. The inverse iteration is used to find the corresponding eigenvectors. In the best case, the inverse costs  $O(nk)$ , in the worse,  $O(nk^2)$  operations. Finding all eigenvalues and eigenvectors results in  $O(n^3)$  complexity. This algorithm is based on a simple formula to count the number of eigenvalues less than a given value. Bisection and inverse iteration are embarrassingly parallel since each eigenvalue and eigenvector may be found independently of the others.

The code generated by Locus for these three primary algorithms uses LAPACK routines [68]. The Locus program, shown in Figure 7.12, automates the choices of these three algorithms and computes the eigenvalues and eigenvectors of the input matrix. From the three choices, two are non-recursive. The non-recursive ones are QR iterations and Bisection by inverse iteration. At the recursive call, Locus will decide whether to continue making recursive calls or switch to one of the non-recursive methods. By doing this, Locus is also deciding the best base case size, which is where the recursion stops.

Despite looking very similar, the codes in Figures 7.12a and 7.12b have a fundamental



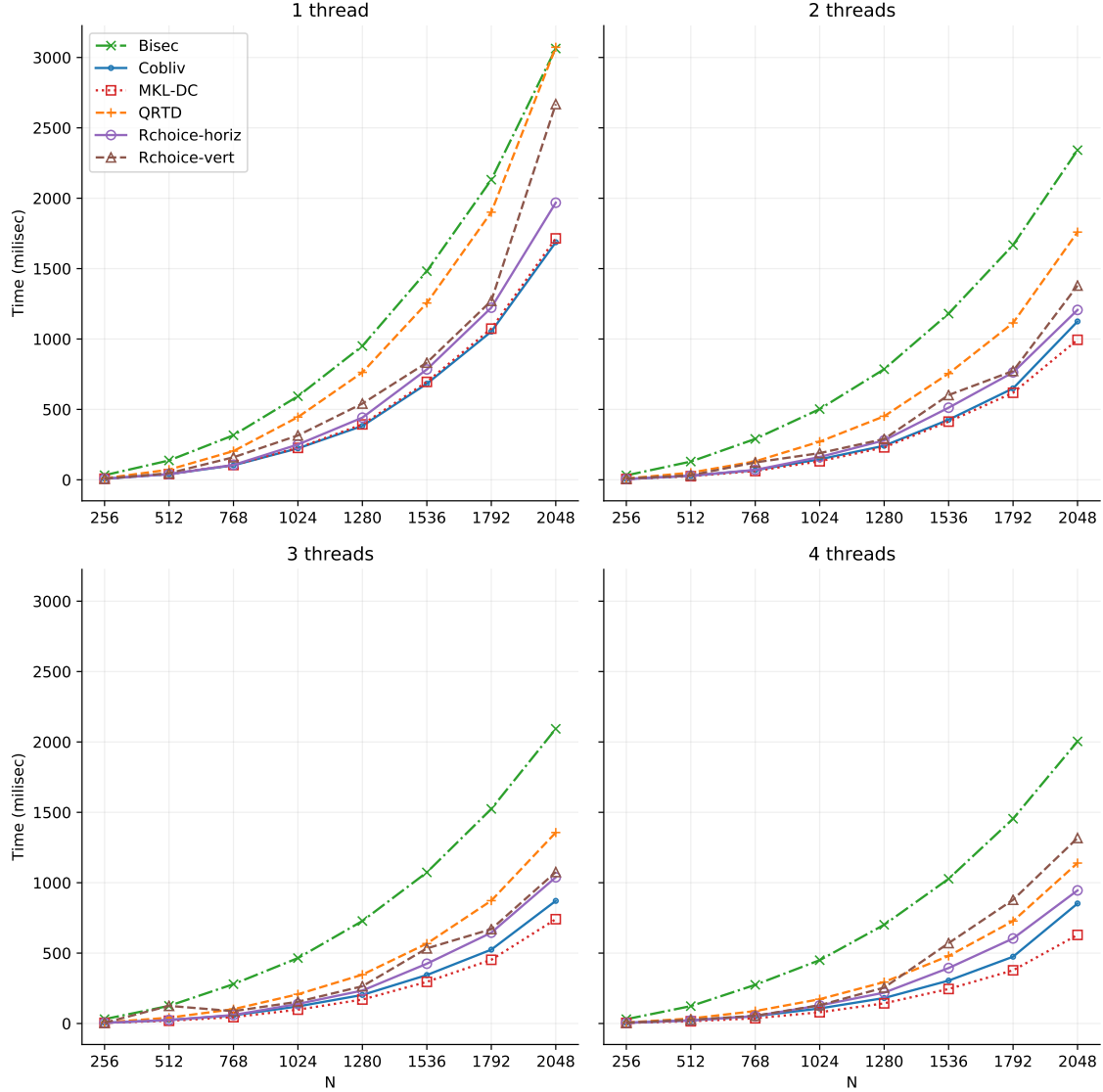


Figure 7.13: Symmetric eigenproblem performance.

difference. In 7.12a, the search space is restricted to finding the best stop value and which algorithm to use on all subproblems, whereas in 7.12b, it allows subproblems to stop at different levels, base cases of different sizes, and different algorithms. The latter is a much larger space and requires expanding the recursion tree before the search starts, as explained in Section 5.4.2.

We evaluated these methods using the LAPACK routines implemented by the Intel MKL library (version 20190005) running on the same system configuration used in Section 7.4. In Figure 7.13, we have the standalone results for tridiagonal QR iteration (*QRTD*), Bisection and inverse iteration (*Bisec*) along with Intel MKL's own hand-optimized divide-and-conquer (*MKL-DC*). We also present the results attained by Locus with cache-oblivious (*Cobliv*),

random-choice vertically (*Rchoice-horiz*), and random-choice horizontally (*Rchoice-vert*). The experiments were executed in parallel using OpenMP tasks. The results promote the good performance of the divide-and-conquer approach. Locus cache-oblivious’ performance attained performance very close to the MKL’s. The random-choice versions, however, achieved performance worse than the cache-oblivious. Their search space was more complex and challenging to be explored by the search modules.

## 7.7 SPARSE MATRIX-VECTOR MULTIPLICATION

Sparse matrix kernels are frequently computational bottlenecks in diverse applications in science and engineering. Nonetheless, these kernels perform poorly on most modern processors due to a low compute-to-memory ratio and irregular memory access patterns. Attaining near-peak performance from them on modern cache-based superscalar machines is difficult because of the complexity of cached-based memory systems. Moreover, performance is highly dependent on the non-zero structure of the matrix.

Many sparse matrices from applications have a natural block structure that can be exploited by storing the matrix as a collection of blocks. However, there are cases in which an “obvious” block structure is not always the best [69].

Consider the sparse matrix-vector multiplication (SpMV) operation  $y \leftarrow y + Ax$ , where  $A$  is an  $m \times n$  sparse matrix. In SpMV, the elements of  $A$  are accessed sequentially but not reused. The elements of  $y$  are also accessed sequentially but are reused for each non-zero in the row of  $A$ . The access to  $x$  is irregular as it depends on the column indices of non-zero elements of  $A$ .

Register reuse of  $y$  and  $A$  cannot be improved, but access to  $x$  may be improved if there are elements in  $A$  that are in the same column and nearby one another so that an element of  $x$  may be saved in a register. In this case, the use of a blocked format for the sparse matrix may improve locality.

### 7.7.1 Sparse matrix formats

A wide variety of formats are in use, each tailored to a particular application and matrix. Besides, several of the formats were specifically created to benefit from vector architectures. Common-used formats include coordinate (COO), compressed sparse stripe (CSR, CSC), jagged diagonal (JAD), variable block row (VBR), and blocked compressed sparse stripe (BSR). We focus our discussion on the compressed sparse stripe (CSR, CSC) and blocked compressed stripes formats (BSR) that are present in our evaluation.

**Compressed stripe storage.** This class includes the compressed sparse row (CSR) format (and compressed sparse column format - CSC). CSR can be viewed as a collection of sparse vectors, allowing random access to entire rows and efficient enumeration of non-zeros within each row. The compressed stripe formats tend to capture the irregular structure of sparse matrices well but suit poorly vector architectures.

In CSR, a sparse vector represents each row. The CSR uses three arrays:

- a single value array `val` stores all sparse row vector values in order;
- an array of integer, `ind`, stores the column indices of each value of the `val` array;
- the `ptr` array contains the indices at which new rows start in `val`.

Algorithm 7.1 presents a sparse matrix-vector multiply for a matrix using the CSR format.

---

**Algorithm 7.1:** SpMV for a matrix using CSR format.

---

**input** : Array  $x$  representing vector  $x$ , and arrays `val`, `ptr`, `ind` representing matrix  $A$  in CSR.  
**output:** Array  $y$  representing vector  $y$ .  
1 **foreach** *row*  $i$  **do**  
2     **for**  $l \leftarrow \text{ptr}[i]$  **to**  $\text{ptr}[i+1] - 1$  **do**  
3          $y[i] \leftarrow y[i] + \text{val}[l] \cdot x[\text{ind}[l]]$   
4     **end**  
5 **end**

---

**Blocked compressed stripe storage.** The common occurrence of dense block structure in sparse matrices led to the design of blocked compressed sparse stripe formats. The block compressed sparse format is the same as CSR, but instead of each non-zero, it stores an  $r \times c$  dense block. The case of  $r = c = 1$  is exactly the same as CSR.

We focus on the block compressed sparse row (BSR) format. The  $r \times c$  BSR generalizes CSR: Matrix  $m$ -by- $n$   $A$  is divided into  $\lceil \frac{m}{r} \rceil$  block rows, and each block row is stored as a sequence of dense  $r \times c$  blocks. The values of  $A$  are stored in an array (`val`) of  $K_{rc} \cdot r \cdot c$  elements, where  $K_{rc}$  is the number of non-zero blocks. The blocks are stored consecutively by row, and each block may be stored in any format used for dense matrices (e.g., row- or column-major). The starting column index of each block is stored in array `ind`, and array `ptr` contains the indices of the row at which each block starts. Filling the blocks with zero elements may be required as they are treated as a full dense block.

Blockings are not unique depend on the convention selected by libraries and systems on how to align the blocks. Intel MKL aligns blocks so that the first column  $j$  of each block

is multiple of  $c$ , i.e.,  $j \bmod c = 0$ . Other libraries use a greedy approach that scans each block row column-by-column, starting a new  $r \times c$  block upon encountering the first column containing a non-zero.

Algorithm 7.2 presents a sparse matrix-vector multiply for a matrix using the BSR format assuming  $r$  divides  $m$  and  $c$  divides  $n$ .

---

**Algorithm 7.2:** SpMV for a matrix using BSR format.

---

**input** : Array  $x$  representing vector  $x$ , and arrays  $val$ ,  $ptr$ ,  $ind$  representing matrix  $A$  in BSR.  
**output:** Array  $y$  representing vector  $y$ .

- 1 *A closed range of integers from  $a$  to  $b$  inclusive is denoted as  $[a : b]$ .*
- 2 **foreach** block row  $I$  **do**
- 3      $i_0 \leftarrow I \cdot r$
- 4      $y' \leftarrow y[i_0 : (i_0 + r - 1)]$
- 5     **for**  $L \leftarrow ptr[I]$  **to**  $ptr[I+1] - 1$  **do**
- 6          $j_0 \leftarrow ind[L] \cdot c$
- 7          $\hat{x} \leftarrow x[j_0 : (j_0 + c - 1)]$
- 8          $\hat{A} \leftarrow val[(L \cdot r \cdot c) : ((L + 1) \cdot r \cdot c - 1)]$
- 9          $y' \leftarrow y' + \hat{A} \cdot \hat{x}$
- 10    **end**
- 11     $y[i_0 : (i_0 + r - 1)] \leftarrow y'$
- 12 **end**

---

This blocked format has the advantage of reducing the amount of memory required to store indices for the matrices since a single index is stored per block. Because  $r$  and  $c$  are fixed, the innermost loop can be fully unrolled.

### 7.7.2 Results

By analogy to tiling in the dense case, finding the best-blocking shape is also a difficult aspect to optimize sparse kernels. We used Locus to search for the composition of blocking sizes that resulted in the best performance. The sparse matrix is recursively divided and different blocking sizes evaluated independently in the search space defined using the random-choice horizontally strategy.

The Locus program is shown in Figure 7.14. The Locus expands the recursion tree, and, at each level, it is possible to divide the matrix in the dimension  $m$ , in the dimension  $n$ , or generate the base case's invocation. The base case invocation has six possibilities of blocking sizes (CSR, 2-by-2, 3-by-3, 4-by-4, 5-by-5, or 6-by-6 BSR).

```

1 mstop = melem/4;
2 nstop = nelem/4;
3 OptSeq recHoriz(mS, mE, nS, nE) {
4     mlen = mE-mS;
5     nlen = nE-nS;
6     {
7         if mlen > mstop {
8             pivm = mS + mlen/2;
9             recHoriz(mS, pivm, nS, nE);
10            recHoriz(pivm, mE, nS, nE);
11        }
12    } OR {
13        if nlen > nstop {
14            pivn = nS + nlen/2;
15            recHoriz(mS, mE, nS, pivn);
16            recHoriz(mS, mE, pivn, nE);
17        }
18    } OR {
19        {
20            GenCode.SpmvCSR(mS, mE, nS, nE);
21        } OR {
22            blk = enum(2,3,4,5,6);
23            GenCode.SpmvBSR(mS, mE, nS, nE,
24                           blk);
25        }
26    }
27 }
28 CodeReg spmv {
29     @pragma expandrec inline
30     recHoriz(0, melem, 0, nelem);
31 }

```

Figure 7.14: Locus program for SpMV using the random-choice horizontally strategy.

We evaluated the sparse matrices shown in Table 7.1. They were taken from the SuiteSparse Matrix Collection [70]<sup>1</sup>. We used the Intel MKL (version 20190005) implementations of the CSR and BSR formats and the SpMV operation for the base case. The experiments were executed on the same system configuration used in Section 7.4. Because Intel MKL aligns blocks so that the first column  $j$  of each block is multiple of  $c$ , the sparse matrices dimensions were extended to be multiple of all possible blocking sizes.

Table 7.1: Sparse Matrices evaluated from the SuiteSparse Matrix Collection.

	Name	Application Area	Dimension	Non-zeros	% of Non-zeros
1	bcsstk35	Automobile frame	30237×30237	1450163	0.16
2	crystk02	FEM Crystal	13965×13965	968583	0.50
3	net100	Optimization	29920×29920	1031560	0.11
4	olafu	Accuracy problem	16146×16146	515651	0.20

<sup>1</sup><https://sparse.tamu.edu/about>

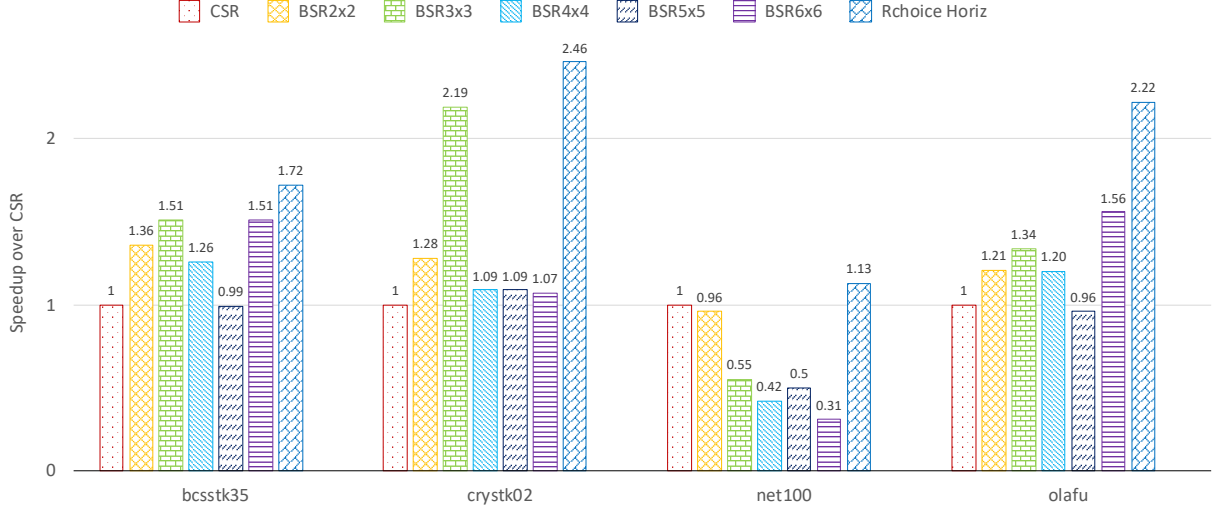


Figure 7.15: Sparse Matrix-Vector multiplication results using different sparse formats for the matrix.

Figure 7.15 presents the speedup over the CSR format. Each matrix had a different blocking size as the one that resulted in the best performance. The random-choice horizontally strategy further improved the performance by using different blocking shapes across the sub-matrices. It resulted in performance improvement of 14%, 12%, 13%, and 42% for the matrices bcsstk35, crystk02, net100, and olafu, respectively.

The blockings that attained the best performance for each matrix are presented below. For instance, the maximum performance attained for crystk02 (Equation 7.6) was due to dividing twice on each dimension, and the use of formats BSR 3x3, BSR 6x6, BSR 5x5, BSR 3x3 on sub-matrices 00, 01, 10, 11, respectively.

$$\text{bcsstk35} = \begin{bmatrix} \text{BSR } 6 \times 6 & \text{BSR } 6 \times 6 & \text{BSR } 2 \times 2 \\ \text{BSR } 4 \times 4 & \text{BSR } 6 \times 6 & \text{BSR } 3 \times 3 \\ \text{BSR } 5 \times 5 & \text{BSR } 6 \times 6 & \text{BSR } 3 \times 3 \end{bmatrix} \quad (7.5)$$

$$\text{crystk02} = \begin{bmatrix} \text{BSR } 3 \times 3 & \text{BSR } 2 \times 2 \\ \text{BSR } 5 \times 5 & \text{BSR } 3 \times 3 \end{bmatrix} \quad (7.6)$$

$$\text{net100} = \begin{bmatrix} \text{BSR } 2 \times 2 & \text{CSR} \\ \text{CSR} & \text{BSR } 3 \times 3 \end{bmatrix} \quad (7.7)$$

$$\text{olafu} = \begin{bmatrix} BSR\ 6\times 6 & BSR\ 4\times 4 \\ BSR\ 4\times 4 & BSR\ 6\times 6 \end{bmatrix} \quad (7.8)$$

## 7.8 SUMMARY

This chapter presented the optimization of matrix transposition, matrix-matrix multiplication, fast Fourier transform, symmetric eigenproblem, and sparse matrix-vector multiplication.

We implemented in Locus three strategies: cache-oblivious, random-choice vertically, and random-choice-horizontally strategies. Each of these strategies creates a different search space for optimizing through divide and conquer.

The experimental results showed that the cache-oblivious strategy combined with empirical search was the best for benchmarks operating on dense data structures. The random-choice horizontally, however, showed its value when optimizing the SpMV operation due to the irregular structure of the sparse matrices.

## CHAPTER 8: OTHER EVALUATIONS

In this chapter, we evaluate a miscellaneous collection of applications that we optimized using Locus: Kripke is a mini-app comprised of 5 kernels that are optimized independently according to the input data layout; we used Locus as a compiler to optimize 856 loops extracted from 16 relevant benchmarks with a generic optimization sequence; evaluated search modules using a complex search space; searched for the best way to execute in parallel and using GPUs a large, complex multi-physics application called Plascom2. In the remainder of this chapter, these applications and their respective search spaces are further discussed.

### 8.1 KRIPKE

Kripke [71] is a deterministic particle transport code and a proxy-app for the Ardra project developed at LLNL. It supports the storage of angular fluxes using a three-dimensional array indexed by direction (D), group (G), and zone (Z). It has 5 kernels: *LTimes*, *LPlusTimes*, *Scattering*, *Source*, and *Sweep*. Their implementation contains 6 hand-optimized versions of each kernel, one for each data layout. Each layout corresponds to a different linearization of the 3D arrays according to one of the 6 permutations of D, G, and Z.

```
1  #pragma @Locus loop=Scattering
2  for(int nm = 0; nm < num_moments; ++nm)
3      for(int g = 0; g < num_groups; ++g)
4          for(int gp = 0; gp < num_groups; ++gp)
5              for(int zone = 0; zone < num_zones; ++zone)
6                  for(int mix = zones_mixed[zone]; mix < zones_mixed[zone] + num_mixed[zone]; ++mix)
7                      {
8                          int material = mixed_material[mix];
9                          double fraction = mixed_fraction[mix];
10                         int n = moment_to_coeff[nm];
11
12                         #####
13                         # Address calculation to be included here.
14                         #####
15
16                         *phi_out += *sigs * *phi * fraction;
17                      }
```

Figure 8.1: Kripke’s Scattering kernel.

Using Locus, we can create a more compact representation of all versions of Kripke. Our representation contains only one skeleton for each kernel plus the code for each of the six address computations, one for each data layout. We then use transformation modules to generate versions that achieve a performance comparable to that of the hand-optimized ones.

Figure 8.1 contains the code representing our version of one of the five kernels, *Scattering*,



```

1  datalayout=enum("DZG","DGZ","GDZ","GZD","ZDG","ZGD");
2  CodeReg Scattering {
3      if (datalayout == "DGZ") {
4          omploop="0.0.0.0";
5      } elif (datalayout == "GDZ") {
6          looporder=[1,2,0,3,4];
7          omploop="0.0.0.0";
8      } elif (datalayout == "GZD") {
9          looporder=[1,2,3,4,0];
10         omploop="0.0.0";
11     } elif (datalayout == "ZGD") {
12         looporder=[3,4,1,2,0];
13         omploop="0";
14     } elif (datalayout == "ZDG") {
15         looporder=[3,4,0,1,2];
16         omploop="0";
17     } elif (datalayout == "DZG") {
18         looporder=[0,3,4,1,2];
19         omploop="0.0";
20     }
21     sourcepath="scatter_"+datalayout+".txt";
22     BuiltIn.AltDesc(stmt="0.0.0.0.0.3", source=sourcepath);
23     RoseLocus.Interchange(order=looporder);
24     RoseLocus.LICM();
25     RoseLocus.ScalarRepl();
26     Pragma.OMPFor(loop=omplloop);
27 }

```

Figure 8.2: Locus program for optimizing Kripke’s Scattering kernel.

and Figure 8.2 contains the Locus program to create the six versions of the kernel. These will differ in the evaluation of the addresses used by the expression  $(\ast\text{phi\_out} += \ast\text{sigs} \ast \ast\text{phi} \ast \text{fraction})$  at the end of the loop in Figure 8.1.

The main structure of the other four kernels (*LTimes*, *LPlusTimes*, *Source*, and *Sweep*) is very close to the one shown for *Scattering*, and the Locus program for each of these kernels is similar to that in Figure 8.2.

The Locus program specifies that, for *Scattering*, each of the six data layouts (*DZG*, *DGZ*, *GZD*, *ZDG*, and *ZGD*) will be used to select one of six different files (*scatter\_DZG.txt*, *scatter\_DGZ.txt*, ..., and *scatter\_ZGD.txt*). Each of these files contains one way of computing the addresses, depending on the layout. Different versions of *Scattering* are created by inserting the content of each of these files into the innermost loop of the kernel using the module `BuiltIn.AltDesc`. The specific place of insertion is after the third statement of the innermost loop, which is identified as 0.0.0.0.0.3. Then:

- the loop nest order (`looporder`) is altered by the module `RoseLocus.Interchange`;
- loop invariant code motion is applied by the transformation module `RoseLocus.LICM` to move each part of the computation to the most efficient location within the loop nest;

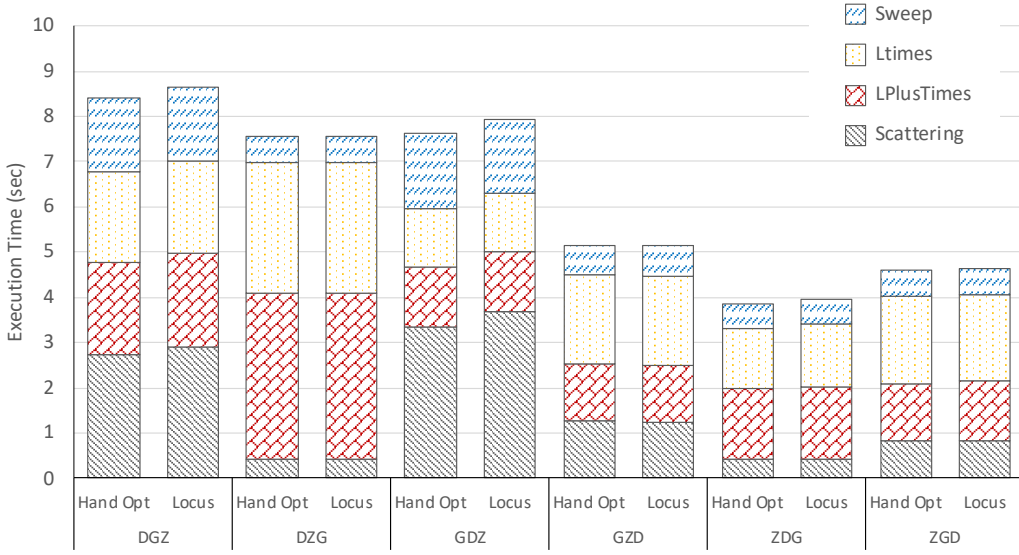


Figure 8.3: Kripke execution time comparing hand-optimized versions and using Locus for 6 different data layouts.

- scalar replacement [26] is performed to improve register usage by the module `RoseLocus.ScalarRepl`;
- the loop to be parallelized (`omploop`) is annotated with OpenMP directives by the module `Pragma.OMPFor`.

The data layout was the only search variable defined. Other possibilities, however, could have been added to the search space. For instance, we used the loop nest orders identical to the hand-optimized versions, but it would be straightforward to explore different orders for each data layout.

Performance attained using Locus is very close to that of the hand-optimized kernels, as presented in Figure 8.3. However, our proposed version is simpler and easier to maintain because it contains a single version of each kernel for all data layouts.

## 8.2 OPTIMIZATION OF ARBITRARY LOOP NESTS

We have shown performance improvements of optimizations specific to source codes known beforehand. In this experiment, we used a generic Locus program to optimize arbitrary loop nests, whose structure and characteristics were not known in advance. Gong et al. [91, 92] developed a loop nest extractor and transformed the extracted loops with subsets of the following two sequences:

Table 8.1: The benchmarks used for loop extraction, the number of loop nests selected, and variants assessed.

Benchmark	Number of loop nests	Variants assessed
ALPBench [72]	13	39
ASC Sequoia [73]	1	3
Cortextsuite [74]	47	1,297
FreeBench [75]	30	431
Parallel Research Kernels [76]	37	1,055
Livermore Loops [77]	11	121
MediaBench [78]	39	159
Netlib [79]	18	260
NAS Parallel Benchmarks [80]	208	23,384
Polybench [81]	93	7,582
Scimark2 [82]	4	83
SPEC2000 [83]	71	2,228
SPEC2006 [84]	50	216
Extended TSVC [85]	156	6,943
Libraries [86, 87, 88, 89]	61	1,966
Neural Network Kernels [90]	17	132
Total	856	45,899

1. interchange  $\rightarrow$  unroll-and-jam  $\rightarrow$  distribution  $\rightarrow$  unrolling;
2. interchange  $\rightarrow$  tiling  $\rightarrow$  distribution  $\rightarrow$  unrolling.

We represented these optimization sequences and their subsets in a Locus program, shown in Figure 8.4. The optimizations have a data dependence check, and only valid transformations were used to generate variants. The first step is to check if the data dependencies can be computed for the loop nest (if not, only unrolling is applied). The next step is to check whether the loop is perfectly nested. Then, the loop nest depth is obtained. These data are used to define which transformations are going to be applied next. Interchange is only applied to perfectly nested loops with nest depth greater than 1. Tiling is only applied to perfectly nested loops and unroll-and-jam only on nests with a depth greater than 1. Along with distribution, these are only applied if the data dependencies can be computed. Unrolling is always applied at the end of the process.

An *OR* construct is used to express the choice among tiling, unroll-and-jam, or none before distribution. It is also important to note the *\** on the distribution, which denotes that it is optional.

Table 8.1 presents the number of the loop nests extracted and the number of variants assessed for each benchmark. In total, 3,146 loop nests were extracted, and we selected 856 whose executions are longer than 10,000 CPU cycles. The search for each loop nest was

```

1 Search {
2   buildcmd = "make_clean;_make_LOOPEXTRACTED";
3   runcmd = "LOOPEXTRACTED_../input_10";
4 }
5 CodeReg scop {
6   @pragma query
7   perfect = BuiltIn.IsPerfectLoopNest();
8   @pragma query
9   depth = BuiltIn.LoopNestDepth();
10  @pragma query
11  depavail = RoseLocus.IsDepAvailable() {
12
13    if (depavail) {
14      if (perfect && depth > 1) {
15        permorder = permutation(seq(0,depth));
16        RoseLocus.Interchange(order=permorder);
17      }
18      {
19        if (perfect) {
20          indexT1 = integer(1..depth);
21          T1fac = poweroftwo(2..32);
22          RoseLocus.Tiling(loop=indexT1, factor=T1fac);
23        }
24      } OR {
25        if (depth > 1) {
26          indexUAJ = integer(1..depth-1);
27          UAJfac = poweroftwo(2..4);
28          RoseLocus.UnrollAndJam(loop=indexUAJ,
29                                factor=UAJfac);
30        }
31      } OR {
32        None; # No tiling, interchange, or unroll and jam.
33      }
34      innerloops = BuiltIn.ListInnerLoops();
35      *RoseLocus.Distribute(loop=innerloops);
36    }
37    innerloops = BuiltIn.ListInnerLoops();
38    RoseLocus.Unroll(loop=innerloops,
39                     factor=poweroftwo(2..8));
40 }

```

Figure 8.4: Locus program for optimizing arbitrary loop nests.

limited to 500 variants. In total, 45,899 variants were evaluated. We ran Pluto on the same set of selected loop nests.

Our work was able to reproduce the performance results presented by Gong et al. However, the Locus program required 37 lines while the implementation by Gong et al. for this purpose was approximately 1200 lines long. Besides, in their approach, the implementations of the two optimization sequences, their subsets, and parameters were hard-coded and cumbersome to modify. With the Locus approach, modifying and experimenting with new optimization sequences becomes trivial.

In this experiment, the variants generated by Locus and the code generated by Pluto (flags `--tile`, `--prevector`, and `--unroll`) were compiled with GCC 6.3.0 (flags `-O3`, and `-ftreevectorize`).

```

1 CodeReg Loop {
2   perm = permutation([0,1,2]);
3   RoseLocus.Interchange(order=perm);
4   tileI = tileK = tileJ = 512;
5   @pragma expandloop # init=2,3
6   for (i = 0; i < integer(1..3); i+= 1) {
7     tileI = poweroftwo(2..tileI);
8     tileK = poweroftwo(2..tileK);
9     tileJ = poweroftwo(2..tileJ);
10    Pips.Tiling(loop=idx, factor=[tileI,
11                                tileK,
12                                tileJ]);
13    idx += ".0.0.0";
14  }
15 }

```

Figure 8.5: Optimizing a loop for finding the number of levels for tiling.

On average, the best variant generated by Locus achieved a speedup of 1.15 while Pluto achieved 1.05. Locus could transform 822 loop nests out of the ones selected and Pluto 397. Pluto transformed a smaller number of loop nests because it is based on the polyhedral model. This model is only applicable to loop nests in which the data access functions and the loop bounds are affine combinations of the enclosing loop variables and parameters. Locus achieved speedups higher than 1.05 for 360 loops nests and Pluto for 170. Out of these 170 optimized by both tools, Locus generated faster code than Pluto on 129 of them.

### 8.3 COMPARING SEARCH MODULES

Considering that OpenTuner and HyperOpt represent the search space differently (OpenTuner uses a flat representation and HyperOpt does it hierarchically), we would like to evaluate their performance with a hierarchical space as input. For that purpose, we used the search for the best number of levels of hierarchical tiling on matrix-matrix multiplication. Figure 8.5 presents the Locus program used for optimizing source-code in Figure 1.1a. The *expandloop* pass described in Section 5.4.1 was used to expand the loop from Figure 8.5 into a hierarchical space, whose number of levels is defined according to the maximum number of loop iterations (based on `i < integer(1..3)` on Line 6).

The Figure 8.6 presents two representations of the space in Locus syntax after the pass was executed: one for using 2-level tiling as the initial search configuration, and one for using 3-level tiling as the initial search configuration. The Locus system uses the first block of each OR-block level for the initial search configuration. Hence, the initial search configuration for both cases executes the blocks starting at Lines 7, 16, and 25. However, the block starting at

```

1 CodeReg Loop {
2   perm = permutation([0,1,2]);
3   RoseLocus.Interchange(order=perm);
4   tileI = tileK = tileJ = 512;
5   @pragma expandloop init=2
6   {
7     tileI_1 = poweroftwo(2..tileI);
8     tileK_1 = poweroftwo(2..tileK);
9     tileJ_1 = poweroftwo(2..tileJ);
10    Pips.Tiling(loop=idx,
11               factor=[tileI_1,
12                      tileK_1,
13                      tileJ_1]);
14    idx += ".0.0.0";
15    {
16      tileI_2 = poweroftwo(2..tileI_1);
17      tileK_2 = poweroftwo(2..tileK_1);
18      tileJ_2 = poweroftwo(2..tileJ_1);
19      Pips.Tiling(loop=idx,
20                 factor=[tileI_2,
21                        tileK_2,
22                        tileJ_2]);
23      idx += ".0.0.0";
24      {
25        None;
26      } OR {
27        tileI_3 = poweroftwo(2..tileI_2);
28        tileK_3 = poweroftwo(2..tileK_2);
29        tileJ_3 = poweroftwo(2..tileJ_2);
30        Pips.Tiling(loop=idx,
31                   factor=[tileI_3,
32                          tileK_3,
33                          tileJ_3]);
34        idx += ".0.0.0";
35      }
36    } OR {
37      None;
38    }
39  }
40 }

```

(a)

```

1 CodeReg Loop {
2   perm = permutation([0,1,2]);
3   RoseLocus.Interchange(order=perm);
4   tileI = tileK = tileJ = 512;
5   @pragma expandloop init=3
6   {
7     tileI_1 = poweroftwo(2..tileI);
8     tileK_1 = poweroftwo(2..tileK);
9     tileJ_1 = poweroftwo(2..tileJ);
10    Pips.Tiling(loop=idx,
11               factor=[tileI_1,
12                      tileK_1,
13                      tileJ_1]);
14    idx += ".0.0.0";
15    {
16      tileI_2 = poweroftwo(2..tileI_1);
17      tileK_2 = poweroftwo(2..tileK_1);
18      tileJ_2 = poweroftwo(2..tileJ_1);
19      Pips.Tiling(loop=idx,
20                 factor=[tileI_2,
21                        tileK_2,
22                        tileJ_2]);
23      idx += ".0.0.0";
24      {
25        tileI_3 = poweroftwo(2..tileI_2);
26        tileK_3 = poweroftwo(2..tileK_2);
27        tileJ_3 = poweroftwo(2..tileJ_2);
28        Pips.Tiling(loop=idx,
29                   factor=[tileI_3,
30                          tileK_3,
31                          tileJ_3]);
32        idx += ".0.0.0";
33      } OR {
34        None;
35      }
36    } OR {
37      None;
38    }
39  }
40 }

```

(b)

Figure 8.6: Locus programs resulting from the *expandloop* pass executed on the loop from Figure 8.5 using two different *init* arguments (Line 5). The main difference between these two programs is on Line 25. The program’s initial configuration on the left (a) uses 2-level tiling, whereas the initial configuration of the program on the right (b) uses 3-level tiling. The Locus system uses the first block of each OR-block level as the initial configuration. Hence, the initial configuration for both cases executes blocks starting at Lines 7, 16, and 25. For (a), Line 25 is a *None*, but for (b), Line 25 starts the block responsible for the third-level tiling.

Line 25 of the 2-level tiling initial configuration contains just the *None* placeholder. On the other hand, for the 3-level tiling initial configuration, the block starting at Line 25 includes the statements to carry out the third-level tiling.

In theory, the OpenTuner space representation of this example is much larger than the HyperOpt one. Opentuner representation forms a Cartesian product of all search variables.

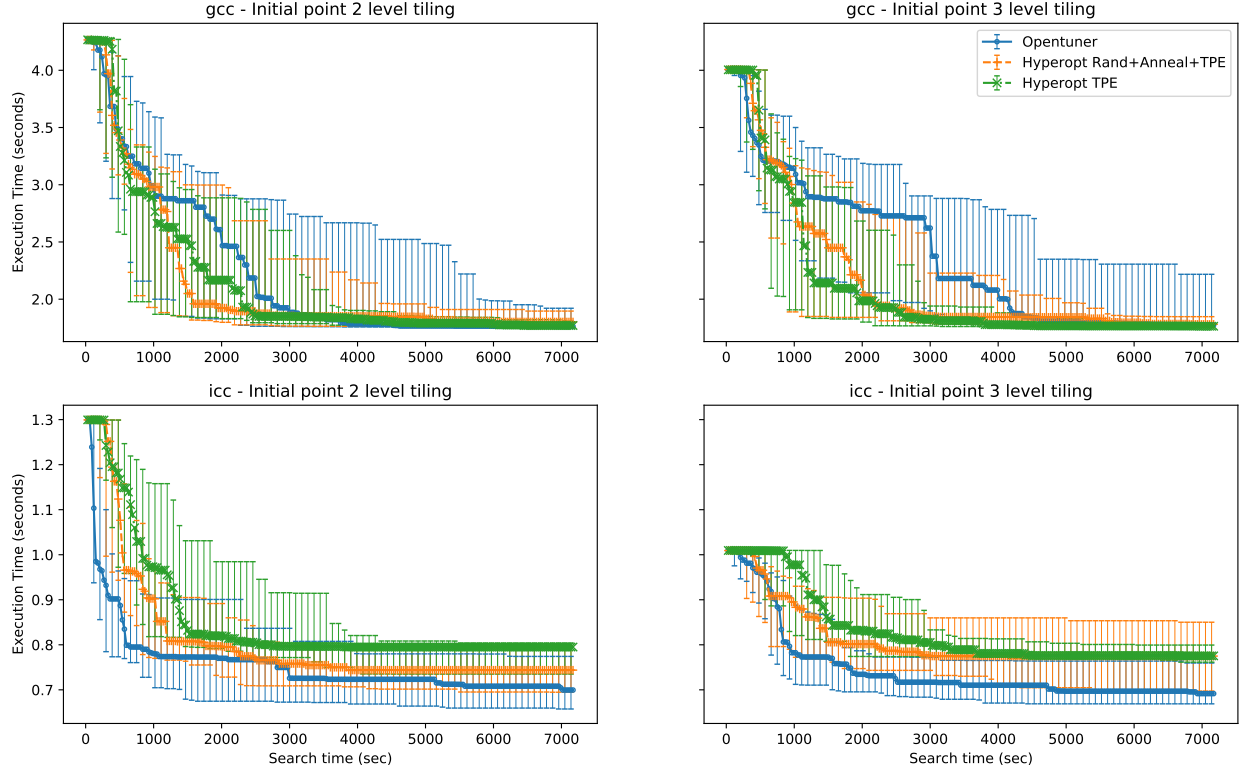


Figure 8.7: Evaluating different backends. Each search backend was executed 30 times, presenting 20 and 80 percentile. The search space was matrix-matrix multiplication using two compilers and initializing the search with a 2 or 3 level tiling.

Hyperopt, on the other hand, can segment the space into independent subspaces. The program in Figure 8.6a presents a hierarchical search space that can be segmented into independent subspaces. For instance, when HyperOpt decides to evaluate 1-level tiling by selecting the second option of the first OR Block (Line 37), the search variables of the 2-level tiling (Lines 16-18) and 3-level tiling (Lines 27-29) are not part of the search.

Figure 8.7 presents the results using two different compilers, ICC and GCC, and using 2-level or 3-level tiling as the initial search configuration. Each search was executed 30 times. The figure presents the 20 and 80 percentile of them. HyperOpt was evaluated using TPE and a mix of 25% for random, 25% for TPE, and 50% for simulated annealing. The selected mix composition was the best found after the evaluation of multiple ones. Our initial hypothesis was that HyperOpt would perform better because it can represent search spaces hierarchically. Hyperopt indeed converged faster than OpenTuner when using the GCC for compiling the candidates, especially with 3-level tiling as the initial search configuration. The experiments with ICC, nonetheless, showed OpenTuner converging faster than HyperOpt. Besides, the performance of the candidates with ICC was much faster than the GCC ones.

Although the input space is more suitable for HyperOpt, OpenTuner has shown good results and eventually matched the ones achieved by HyperOpt. We believe that OpenTuner’s meta-technique strategy can effectively use multiple techniques and achieve good results in such complex search spaces.

## 8.4 PLASCOM2

Plascom2 is a large, complex multi-physics application developed as part of the Center for the Exascale Simulation of Plasma-Coupled Combustion (XPACC) [46]. It is built up from modular, reusable components designed to use modern, heterogeneous architectures efficiently. The parallel execution model used by Plascom2 is based on hybrid MPI+OpenMP. Most of the Plascom2 application is written in C++, but the computational kernels are written in Fortran 90. These kernels consist of pointwise operations and stencils.

Plascom2 follows the development model in which the baseline version contains an easy-to-maintain code with no architecture- or compiler-specific optimizations. Automatic code transformation tools and runtime systems are then used to make it run faster, adapting accordingly to different target architectures.

Locus can optimize Plascom2 through loop transformations to the computation kernels and stencils, and enabling different runtime systems. The initial step, and not less challenging, was to define the baseline version out of a code that had already evolved into an optimized version. Subsequently, all the hand-optimized code regions were included in the Locus program as a preferred version. These preferred versions are loaded into the application at build-time. Below, we present the use of code transformations on top of these preferred versions. Locus also allows us to enable and compare different tools for parallel execution and the use of accelerators.

We evaluate the performance of two computation kernels: ZAXPY and ZXDOTY. They were evaluated on a single-node IBM Power9 system with two CPUs (40 cores and 160 threads) and four Nvidia Volta V100 GPUs.

### 8.4.1 ZAXPY

ZAXPY performs the operation  $Z = \alpha X + Y$ , where  $Z$ ,  $X$ , and  $Y$  are vectors and,  $\alpha$  is a scalar. Figure 8.8 shows the code with the *pragma* generated by Hydra [93, 94] to execute it on the GPU. Hydra is also responsible for the data movement required to offload ZAXPY execution.



```

1 !$omp target teams distribute parallel do simd
2 DO I = iStart, iEnd
3     Z(I) = Y(I) + a*X(I)
4 END DO

```

Figure 8.8: ZAXPY source-code with OpenMP *pragma* for offloading to accelerators.

The `omp target teams distribute parallel do` indicates to the compiler that the following section of code should be compiled for host and accelerator. It also indicates to the OpenMP runtime to transfer to the device the input data structures used in this section and transfer back to the host the output data structures. If no device is available, the code is executed solely on the host. This allows a single version of the code to execute in various devices. It distributes the iterations of the loop among the threads (teams). The `simd` is a hint for the compiler to use vector instructions.

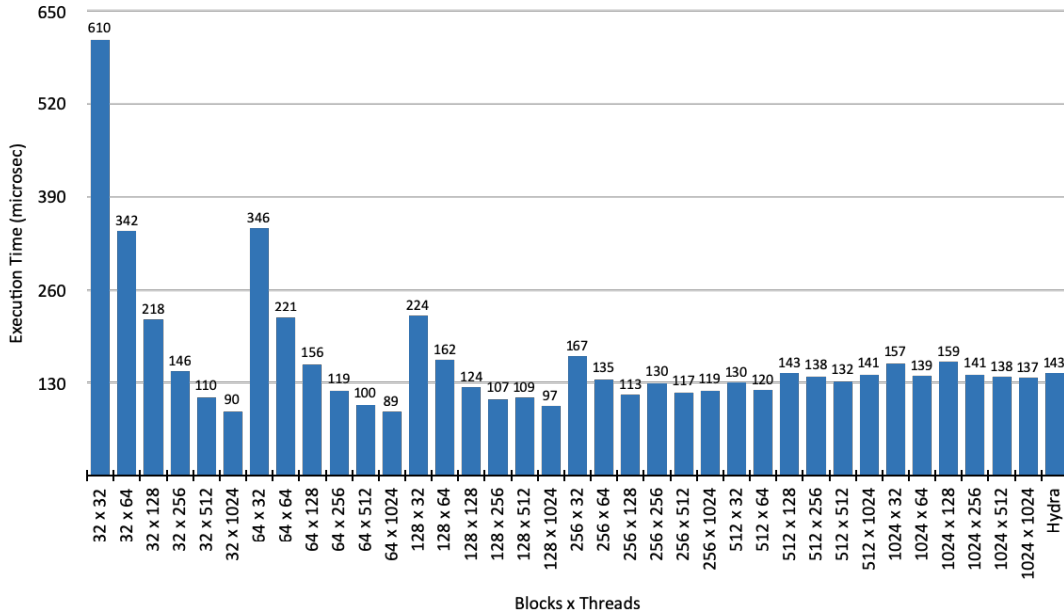


Figure 8.9: Performance of ZAXPY in Plascom2. Tangram kernels tuned by varying the number of blocks and the number of threads per block. Hydra uses offloading through OpenMP.

Tangram [95] is a high-level programming framework for synthesizing efficient GPU code. It provides composable code blocks that can be synthesized to provide different GPU architectures with algorithmic choices. Locus is able to generate code that either uses OpenMP offloading via Hydra or CUDA kernels synthesized by Tangram. The two solutions, however, require further tuning parameters to achieve optimal performance.

Figure 8.9 presents the performance of the kernel generated by Tangram as a function of

```

1  !$Locus loop=zxdoty
2  DO iComp = 0, numComponents-1
3    DO K = kStart, kEnd
4      DO J = jStart, jEnd
5        DO I = iStart, iEnd
6          zIdx = (K-1) * nPlane
7          yzIdx = zIdx + (J-1) * xSize
8          cOffset = iComp * numPoints
9          bIdx = yzIdx + I
10         Z(bIdx) += X(bIdx + cOffset) * Y(bIdx + cOffset)
11       END DO
12     END DO
13   END DO
14 END DO

```

Figure 8.10: ZXDOTY source-code.

the number of blocks and threads per block and the offloading using OpenMP (Hydra). The performance of Tangram kernels varies greatly according to the configuration of blocks of threads and the number of threads per block. It outperforms the OpenMP offloading as the number of blocks and threads per blocks increases better utilizing the GPU resources.

#### 8.4.2 ZXDOTY

The operation performed by ZXDOTY is

$$Z(k, j, i) = \sum_{c=1}^{numcomp} X_c(k, j, i) * Y_c(k, j, i) \quad (8.1)$$

where  $Z$ ,  $X$ , and  $Y$  are 3D data volumes indexed by  $k$ ,  $j$ , and  $i$ . The values of  $X$  and  $Y$  are multiplied and accumulated over all the components.

The code for ZXDOTY is presented in Figure 8.10. The iterations of the loops indexed by variables  $K$ ,  $J$ , and  $I$  can be executed in parallel. The  $iComp$  loop, however, cannot, as there is a reduction on  $Z$ . Therefore, the parallelization of ZXDOTY must be mindful of the position of the  $iComp$  loop. In general, the more work there is to provide to the threads, the more efficient the parallelization will be. A loop nest can be collapsed to increase the number of iterations partitioned across the threads by reducing the granularity of work done by each thread. The iterations of the  $K$ ,  $J$ , and  $I$  loops can be collapsed into a single, large loop to improve the scalability of ZXDOTY. The  $iComp$  loop, then, must be in the outermost or innermost position, which may improve data locality.

Locus generated all variants automatically by using loop interchange to change the position of the  $iComp$  loop. OpenMP carried out the collapse of the loops.

Figure 8.11 presents the results for parallelizing the ZXDOTY with the  $iComp$  loop in

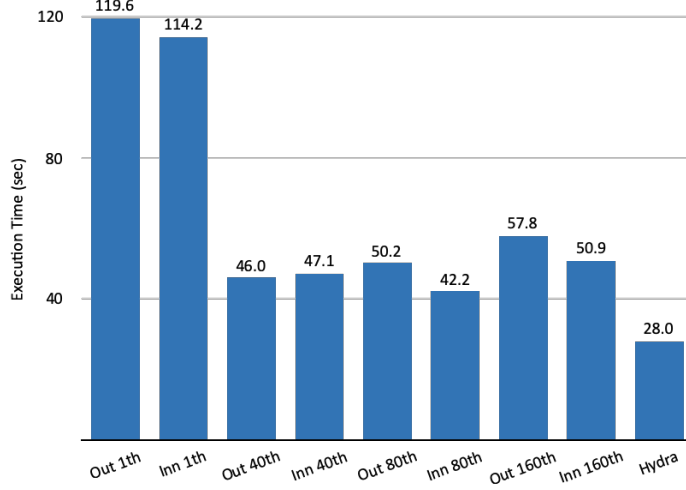


Figure 8.11: Performance of ZXDOTY from Plascom2 as a function of the *iComp* loop position and the number of threads running in the CPU, and Hydra in the GPU. *Out* is for the *iComp* loop in the outermost position, *Inn* for in the innermost.

outermost or innermost position, collapsing loops  $K$ ,  $J$ , and  $I$ , number of threads  $n \in \{1, 40, 80, 160\}$  running in the CPU, and Hydra running in the GPU. The input dimensions are  $Z = X = Y = 701$  elements, and  $numberofcomponents = 3$ . Hydra offloads to the GPU using `omp target teams distribute parallel do simd collapse(3)`. The results were best for the *iComp* loop in the innermost position and running 80 threads (2 per core). The performance of Hydra on the GPU was still  $1.5\times$  faster.

## 8.5 SUMMARY

The optimization process complexity depends on many factors, such as the kind of application, the number of code regions, and the architecture's features. In this chapter, we evaluated applications that challenged Locus in many ways. Locus was able to match the 5 hand-optimized for 6 data layouts Kripke kernels, improve the performance of hundreds of loops with the same optimization program, and optimize for GPUs kernels of a large, complex multi-physics application. Locus was able to show that the system and the language are robust to optimize a diverse collection of applications.

## CHAPTER 9: CONCLUSIONS

This dissertation presented Locus, a new system, and a language for optimizing complex, long-lived applications for different platforms. As architecture-specific optimizations are required to harness performance available on current machines, the Locus system can integrate different modules and combine expert knowledge with automated transformations to assist performance experts and code developers in the performance optimization process.

We describe the requirements that we believe are necessary for making automatic performance tuning widely adopted. We present the design and implementation of a system that fulfills these requirements. It includes a domain-specific language that is able to represent complex collections of transformations, an interface to integrate external modules, and a database to manage platform-specific efficient code. The database allows the system users to access optimized code without having to install the code generation toolset. In all, the system presents an approach for performance portability.

Locus allows the integration of complex search spaces with the programming of optimization sequences. The optimization sequences can produce complex, hard to read programs, but thanks to Locus, the programmer does not have to deal with these complex versions, which are generated automatically. The optimization of Locus programs by our compiler removes from the optimization sequences unnecessary search statements that represent search variables on the search space and make the exploration harder. Interdependent search variables are common in search spaces for program optimization. They are, however, hard to represent concisely, requiring a verbose, hard to maintain description. We developed compiler passes to generate complex spaces containing interdependent search variables automatically. The compiler passes are also used to ensure that these spaces only include useful search statements.

The experiments showed that it is possible to obtain higher speedups over a reference version by using empirical search compared to conventional restructurings. The results include  $553\times$  on matrix-matrix multiplication and up to  $4\times$  on stencil computations over original, unmodified versions.

We presented evaluations that used the system to generate optimized code for two different platforms. Locus generated a matrix-matrix multiplication code that outperformed the IBM XLC internal hand-optimized version by  $2\times$  on the Power 9 processors. On Intel CPUs, Locus was able to generate code with performance comparable to that of Intel MKL's, which is hand-optimized and platform-specific. The 3D heat stencil optimized by Locus was up to 75% more efficient than the baseline version. We also showed the benefits of using a fixed

configuration based on results from a different platform as the initial search configuration, which confined the search outcome to a narrower range.

The performance attained varied significantly depending on the platform and the input dimensions and showed the value of having a database to save efficient platform-specific code for each code region.

The divide-and-conquer strategy is a good match for modern parallel machines because of their inherent parallelism and ability to benefit from the prevalent deep memory hierarchy. There are, nonetheless, important trade-offs to fully exploit the divide-and-conquer strategy. We used Locus to create search spaces using three different strategies to determine the best shapes for the base case and the best way of subdividing the problem that attained the best performance. We used Locus to optimize matrix transpose, matrix-matrix multiplication, fast Fourier transform, symmetric eigenproblem, and sparse matrix-vector multiplication using divide and conquer. The results showed that the empirical search was important to improve performance by generating faster base cases and finding the best splitting. The experimental results showed that combining a cache-oblivious strategy and search space for the base case size was the best for benchmarks operating on dense data structures. The random-choice horizontally, however, despite being a large and complicated space that resulted in no performance improvement for benchmarks operating on dense data structures, showed its value when optimizing the SpMV operation by improving performance up to 42% due to the irregular structure of the sparse matrices.

Locus was also able to optimize large, complex applications. The Kripke transport code's baseline version consists of 6 versions for each of the 5 kernels, each version hand-optimized for a specific input data layout. Using Locus, it was possible to match Kripke's hand-optimized performance for all data layouts using a single optimization sequence. We created a generic optimization sequence to be applied to a collection of 856 loops extracted from 16 benchmarks, which resulted in good performance improvements. The Plascom2 multi-physics application was optimized to find the best way to use a multi-core CPU and GPU, along with exploring different modules and their solutions. Tangram, Hydra, and OpenMP provided an interesting search space that improved the performance of Plascom2 by approximately  $4.3\times$  on ZAXPY and ZXDOTY kernels.

Locus showed that the definition of optimization sequences combined with search spaces separated from the application's code works. We improved the performance of a diverse set of applications and believe it benefits developers in the complicated performance optimization process.

## 9.1 FUTURE DIRECTIONS

We envision a variety of ways in which to extend the work in this dissertation. The optimization of other domains and improvements to the search process are discussed next.

### 9.1.1 Beyond Source-to-source Transformations

Locus could be extended to optimize code using other transformations than source-to-source. Despite the high speedups achieved by source-to-source transformations, optimizations in lower representations may further improve the performance of key applications.

Optimizations in lower representations closer to the architecture instruction set would require new ways to define code regions. The use of labels and the arbitrary granularity that we have in the source-code level may not be possible on lower-level representations.

Superoptimization [96, 97] is a program optimization technique to search for a correct and optimal machine-specific sequence of instructions. The combination of heuristics and search in this area have presented interesting results. For instance, Stoke [98] is a superoptimizer that formulates the loop-free binary superoptimization task as a search problem. The search space explored by Stoke could be defined in the Locus language and explored by other search techniques in addition to Markov Chain Monte Carlo that Stoke uses.

The idea of a full compiler using the Locus approach has been introduced by the *optimizing arbitrary loop nests* shown in Section 8.2. We believe that the Locus approach could be used to implement a full compiler. The system and the language could be extended to invoke and explore search spaces of lower-level compiler passes. In theory, the optimization of code on lower-level representations is possible as long as the result of a transformation is accepted by the subsequent one.

Locus would require the compiler passes to be developed in a modular way so they can be invoked independently of each other. Locus accepts the definition of nested code regions. Thus, passes that comprise larger code regions, or even the entire source-code, could be pre-specified by the Locus system and the selection of the passes to be used on those code regions customized by the developer.

To determine, based on the unit of optimization of each compiler pass, the places of the code where to apply each pass is very challenging on lower-level representations. The lower the representation level, the more complicated it is to determine the exact portion of the code to be optimized by a specific compiler pass. Besides, keeping track (e.g., through labels or annotations) of the unit of optimization across representation levels is a big undertaking. Therefore, we believe that in lower representation, the optimizations would be carried out

on coarse-grained regions of code (e.g., at function level).

The heuristics used by compiler passes are often presented as a black-box. We believe that combining heuristics with a search-based approach would help generate faster code. Exposing such search spaces would allow their exploration by different search techniques. The compiler infrastructure should strive to open up more the search space of its passes and allow the combination of search spaces along with heuristics to generate fast, platform-, and domain-specific code.

### 9.1.2 Optimizing Domain-specific Languages

Programmer productivity has dramatically increased by the use of high-level (e.g., Python, Ruby) and domain-specific languages (DSL) (e.g., Halide, TensorFlow) as they raise the level of abstraction and increase software reuse. These languages, however, still require a compiler stack to generate code that runs efficiently on the target architectures (e.g., CPUs, GPUs, FPGAs). The construction of a compiler stack and optimization passes to harness the potential performance of each target architecture for each DSL is an immense undertaking.

DSLs have been designed to decouple algorithms from their schedules. Changing the schedule enables a single algorithm to achieve high performance on diversity machines. Halide [10] and Tiramisu [99] separate what you want to compute (algorithm) from how you want to compute it (schedule), including choices about memory locality, redundant computation, and parallelism.

While they make it easy to try different schedules, writing schedules that achieve high performance is hard. It requires expertise in hardware architecture and optimization, but the space of possible schedules is large, and their performance can be difficult to model.

The Locus system can be expanded to generate high-performance schedules automatically. In a similar fashion to what was done for optimizing arbitrary loop nests (Section 8.2), an optimization sequence combined with a search space could generate schedules tailored to an algorithm and a target platform.

This approach would make fast, efficient code accessible to relatively untrained audiences, ease the burden of developing energy-efficient applications/architectures, and help developers devote more time to new interesting features and less time dealing with the intricacies of each new architecture.

In summary, languages that are highly productive, but domain-specific are becoming the norm. The programs using these languages require complex scheduling of operations to result in efficient code for an increasing variety of architectures. Locus could help to lower the barrier of achieving high performance on applications developed using DSLs.

### 9.1.3 Better Control of the Search Process

Locus has allowed the integration of the definition of a search space with the programming of optimizations sequences. The search techniques, nonetheless, have shown difficulty in exploring large, hierarchical search spaces.

Ideally, the Locus system would identify such spaces and adapt them to semantically equivalent ones that are more amenable to the search techniques being used. For instance, the search space could be divided, and multiple independent searches carried out.

Another important challenge is to define when to stop the search process. We believe that Locus could help users by automatically defining when to stop based on previous searches saved on the database, on a pre-selected model (e.g., Roofline model [100]), on analyzing the progress of the current search over time, or on a combination of all of them.

Selecting the initial candidates for the search process may result in a faster convergence to the best result. The initial candidates can be selected from a database of previously evaluated candidates or specified by the search statements in the Locus program. The search process could be further *guided* by giving bias to the options of each search statement. Currently, each possibility of a search statement follows a uniform probability distribution. The language could be extended to enable the use of different probability distributions for each search statement.

### 9.1.4 New Target Architectures

New architectures are a natural direction for future research and development. Although the experiments included CPUs and GPUs, Locus could be used to optimize other architectures, such as FPGAs and domain-specific accelerators.

Domain-specific hardware accelerators remain one of the few paths to continue to increase the performance and efficiency of computing hardware with the end of Dennard Scaling and Moore’s Law. As these domains greatly vary, the likely result is, however, the development of a multitude of different architectures and instruction sets that add more burden to the compiler-stack development [101]. Besides, accelerators that include special instructions and special engines are becoming the norm. The HMMA (half-precision matrix multiply-accumulate) is one example of specialized instruction added to a general-purpose processor. It is part of the NVIDIA Volta V100 GPU [102] and multiplies two  $4 \times 4$  half-precision (16-bit) floating-point matrices accumulating the results in a  $4 \times 4$  single-precision (32-bit) floating-point matrix. The development of instruction, engine, and hardware specialization makes the optimization of applications even more complicated, and the separation of



algorithm specification from the platform-specific optimizations very important.

The optimization of applications for distributed-memory systems is another target to which Locus could be applied. Data partitioning and process placement can become a hard problem to solve, especially for current large and heterogeneous machines. This area is particularly challenging due to the difficulties of characterizing dynamic workloads and modeling interactions of many complex system components.

## REFERENCES

- [1] C. Chen, J. Chame, and M. Hall, “CHiLL: A framework for composing high-level loop transformations,” University of Utah, Tech. Rep., 2008.
- [2] A. Hartono, B. Norris, and P. Sadayappan, “Annotation-based empirical performance tuning using Orio,” in *IPDPS’09*, 2009.
- [3] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. J. Garzarán, D. Padua, and K. Pingali, “A language for the compact representation of multiple program versions,” in *Languages and Compilers for Parallel Computing*, E. Ayguadé, G. Baumgartner, J. Ramanujam, and P. Sadayappan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 136–151.
- [4] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, “PetaBricks: A language and compiler for algorithmic choice,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, Jun 2009. [Online]. Available: <http://groups.csail.mit.edu/commit/papers/2009/ansel-pldi09.pdf>
- [5] R. C. Whaley and J. J. Dongarra, “Automatically tuned linear algebra software,” in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, ser. SC ’98. Washington, DC, USA: IEEE Computer Society, 1998. [Online]. Available: <http://dl.acm.org/citation.cfm?id=509058.509096> pp. 1–27.
- [6] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, “SPIRAL: Code generation for DSP transforms,” *Proceedings of the IEEE, special issue on “Program Generation, Optimization, and Adaptation”*, vol. 93, no. 2, pp. 232–275, 2005.
- [7] S. J. Pennycook, J. D. Sewall, and V. W. Lee, “A Metric for Performance Portability,” *arXiv e-prints*, Nov. 2016. [Online]. Available: <https://arxiv.org/abs/1611.07409>
- [8] M. Wolfe, “Compilers and more: What makes performance portable?” April 2016, [Online; posted April 19, 2016]. [Online]. Available: <https://www.hpcwire.com/2016/04/19/compilers-makes-performance-portable>
- [9] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach, “Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance OpenCL code,” in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2015. New York, NY, USA: ACM, 2015. [Online]. Available: <http://doi.acm.org/10.1145/2784731.2784754> pp. 205–217.
- [10] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, “Halide: Decoupling algorithms from schedules for high-performance image processing,” *Commun. ACM*, vol. 61, no. 1, pp. 106–115, Dec. 2017. [Online]. Available: <http://doi.acm.org/10.1145/3150211>

- [11] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, “The Pochoir stencil compiler,” in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA ’11. New York, NY, USA: ACM, 2011. [Online]. Available: <http://doi.acm.org/10.1145/1989493.1989508> pp. 117–128.
- [12] M. Frigo, “A fast Fourier transform compiler,” in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, ser. PLDI ’99. New York, NY, USA: ACM, 1999. [Online]. Available: <http://doi.acm.org/10.1145/301618.301661> pp. 169–180.
- [13] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, “Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology,” in *Proceedings of the 11th International Conference on Supercomputing*, ser. ICS ’97. New York, NY, USA: ACM, 1997. [Online]. Available: <http://doi.acm.org/10.1145/263580.263662> pp. 340–347.
- [14] R. Vuduc, J. W. Demmel, and K. A. Yelick, “OSKI: A library of automatically tuned sparse matrix kernels,” *Journal of Physics: Conference Series*, vol. 16, no. 1, p. 521, 2005. [Online]. Available: <http://stacks.iop.org/1742-6596/16/i=1/a=071>
- [15] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan, “POET: Parameterized optimizations for empirical tuning,” in *2007 IEEE International Parallel and Distributed Processing Symposium*, March 2007, pp. 1–8.
- [16] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, “A practical automatic polyhedral parallelizer and locality optimizer,” in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’08. New York, NY, USA: ACM, 2008. [Online]. Available: <http://doi.acm.org/10.1145/1375581.1375595> pp. 101–113.
- [17] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, “Sequoia: Programming the memory hierarchy,” in *SC 2006 Conference, Proceedings of the ACM/IEEE*, Nov 2006, pp. 4–4.
- [18] R. D. Hornung and J. A. Keasler, “The RAJA portability layer: Overview and status,” Lawrence Livermore National Lab., Tech. Rep. LLNL-TR-661403, 9 2014.
- [19] H. C. Edwards, C. R. Trott, and D. Sunderland, “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0743731514001257>
- [20] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro, “Nitro: A framework for adaptive code variant tuning,” in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, May 2014, pp. 501–512.

- [21] I.-H. Chung and J. K. Hollingsworth, “Using information from prior runs to improve automated tuning systems,” in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, ser. SC ’04. Washington, DC, USA: IEEE Computer Society, 2004. [Online]. Available: <https://doi.org/10.1109/SC.2004.65> pp. 30–.
- [22] A. Tiwari, J. K. Hollingsworth, C. Chen, M. Hall, C. Liao, D. J. Quinlan, and J. Chame, “Auto-tuning full applications: A case study,” *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 3, pp. 286–294, Aug. 2011. [Online]. Available: <http://dx.doi.org/10.1177/1094342011414744>
- [23] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O’Reilly, and S. Amarasinghe, “OpenTuner: An extensible framework for program autotuning,” in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT ’14. New York, NY, USA: ACM, 2014. [Online]. Available: <http://doi.acm.org/10.1145/2628071.2628092> pp. 303–316.
- [24] S. Goedecker and A. Hoisie, *Performance Optimization of Numerically Intensive Codes*. Society for Industrial and Applied Mathematics, 2001. [Online]. Available: <http://epubs.siam.org/doi/abs/10.1137/1.9780898718218>
- [25] A. Aho, *Compilers: Principles, Techniques, & Tools*, ser. Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007. [Online]. Available: [https://books.google.co.in/books?id=dIU\\_AQAAIAAJ](https://books.google.co.in/books?id=dIU_AQAAIAAJ)
- [26] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.
- [27] A. Venkat, M. Hall, and M. Strout, “Loop and data transformations for sparse matrix code,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’15. New York, NY, USA: ACM, 2015. [Online]. Available: <http://doi.acm.org/10.1145/2737924.2738003> pp. 521–532.
- [28] M. S. Lam and M. E. Wolf, “A data locality optimizing algorithm,” *SIGPLAN Not.*, vol. 39, no. 4, pp. 442–459, Apr. 2004. [Online]. Available: <http://doi.acm.org/10.1145/989393.989437>
- [29] N. Manjikian and T. S. Abdelrahman, “Fusion of loops for parallelism and locality,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 2, pp. 193–209, Feb 1997.
- [30] 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 *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010, pp. 1–12.

- [31] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, “Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures,” in *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2008, pp. 1–12.
- [32] D. A. Menascé, “Software, performance, or engineering?” in *Proceedings of the 3rd International Workshop on Software and Performance*, ser. WOSP ’02. New York, NY, USA: ACM, 2002. [Online]. Available: <http://doi.acm.org/10.1145/584369.584407> pp. 239–242.
- [33] V. R. Basili, J. C. Carver, D. Cruzes, L. M. Hochstein, J. K. Hollingsworth, F. Shull, and M. V. Zelkowitz, “Understanding the high-performance-computing community: A software engineer’s perspective,” *IEEE Software*, vol. 25, no. 4, pp. 29–36, 2008.
- [34] C. Țăpuș, I.-H. Chung, and J. K. Hollingsworth, “Active Harmony: Towards automated performance tuning,” in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, ser. SC ’02. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002. [Online]. Available: <http://dl.acm.org/citation.cfm?id=762761.762771> pp. 1–11.
- [35] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu, “A comparison of empirical and model-driven optimization,” in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI ’03. New York, NY, USA: ACM, 2003. [Online]. Available: <http://doi.acm.org/10.1145/781131.781140> pp. 63–76.
- [36] S. Hong and H. Kim, “An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness,” in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA ’09. New York, NY, USA: ACM, 2009. [Online]. Available: <http://doi.acm.org/10.1145/1555754.1555775> pp. 152–163.
- [37] M. J. Wolfe, *High Performance Compilers for Parallel Computing*, C. Shanklin and L. Ortega, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.
- [38] J. Lee, H. Kim, and R. Vuduc, “When prefetching works, when it doesn’t, and why,” *ACM Trans. Archit. Code Optim.*, vol. 9, no. 1, Mar. 2012. [Online]. Available: <https://doi.org/10.1145/2133382.2133384>
- [39] G. J. Narlikar and G. E. Blelloch, “Pthreads for dynamic and irregular parallelism,” in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, ser. SC ’98. Washington, DC, USA: IEEE Computer Society, 1998. [Online]. Available: <http://dl.acm.org/citation.cfm?id=509058.509089> pp. 1–16.

- [40] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, “Cilk: An efficient multithreaded runtime system,” in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95. New York, NY, USA: ACM, 1995. [Online]. Available: <http://doi.acm.org/10.1145/209936.209958> pp. 207–216.
- [41] E. Chong and S. Zak, *An Introduction to Optimization*, ser. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley, 2004.
- [42] R. Bellman, *Adaptive Control Processes: A Guided Tour*, ser. Princeton Legacy Library. Princeton, New Jersey, USA: Princeton University Press, 1961. [Online]. Available: <https://press.princeton.edu/books/paperback/9780691625850/adaptive-control-processes>
- [43] J. Bergstra, D. Yamins, and D. D. Cox, “Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures,” in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ser. ICML'13. Atlanta, GA, USA: JMLR.org, 2013, p. I-115–I-123.
- [44] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, “Taking the human out of the loop: A review of Bayesian optimization,” *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2016.
- [45] J. Bergstra and Y. Bengio, “Algorithms for hyper-parameter optimization,” in *In NIPS*, 2011, pp. 2546–2554.
- [46] “Center for the Exascale Simulation of Plasma-Coupled Combustion web page,” 2020, <http://xpacc.illinois.edu>. [Online]. Available: <http://xpacc.illinois.edu>
- [47] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, “PETSc Web page,” <http://www.mcs.anl.gov/petsc>, 2018. [Online]. Available: <http://www.mcs.anl.gov/petsc>
- [48] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, “PETSc users manual,” Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.10, 2018. [Online]. Available: <http://www.mcs.anl.gov/petsc>
- [49] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, “Efficient management of parallelism in object oriented numerical software libraries,” in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.

- [50] T. S. F. X. Teixeira, C. Ancourt, D. Padua, and W. Gropp, “Locus: A system and a language for program optimization,” in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. Washington, DC, USA: IEEE Press, 2019, p. 217–228.
- [51] T. S. F. X. Teixeira, W. Gropp, and D. Padua, “Managing code transformations for better performance portability,” *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1290–1306, 2019. [Online]. Available: <https://doi.org/10.1177/1094342019865606>
- [52] M. D. Smith, “Overcoming the challenges to feedback-directed optimization (keynote talk),” in *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, ser. DYNAMO ’00. New York, NY, USA: Association for Computing Machinery, 2000. [Online]. Available: <https://doi.org/10.1145/351397.351408> p. 1–11.
- [53] R. Keryell, C. Ancourt, F. Coelho, F. Irigoin, and P. Jouvelot, “Pips: a workbench for building interprocedural parallelizers, compilers and optimizers,” MINES ParisTech, Tech. Rep., 06 1996.
- [54] X. Zhou, M. J. Garzarán, and D. A. Padua, “Optimal parallelogram selection for hierarchical tiling,” *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 58:1–58:23, Jan. 2015. [Online]. Available: <http://doi.acm.org/10.1145/2687414>
- [55] U. Bondhugula, V. Bandishti, and I. Pananilath, “Diamond tiling: Tiling techniques to maximize parallelism for stencil computations,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1285–1298, 2017.
- [56] MINES ParisTech, “Pips project documentation,” 2020. [Online]. Available: <https://pips4u.org/doc>
- [57] J. Lidman, D. J. Quinlan, C. Liao, and S. A. McKee, “ROSE::FTTransform - a source-to-source translation framework for exascale fault-tolerance research,” in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*, June 2012, pp. 1–6.
- [58] L. Dagum and R. Menon, “OpenMP: an industry standard API for shared-memory programming,” *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, Jan 1998.
- [59] J. Bergstra, D. Yamins, and D. D. Cox, “Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures,” in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ser. ICML’13. JMLR.org, 2013. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3042817.3042832> pp. I-115–I-123.



- [60] O. Tange, “GNU parallel - the command-line power tool,” *login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, Feb. 2011. [Online]. Available: <http://www.gnu.org/s/parallel>
- [61] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, “Optimization and performance modeling of stencil computations on modern microprocessors,” *SIAM Rev.*, vol. 51, no. 1, pp. 129–159, Feb. 2009. [Online]. Available: <http://dx.doi.org/10.1137/070693199>
- [62] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, “Cache-oblivious algorithms,” *ACM Trans. Algorithms*, vol. 8, no. 1, Jan. 2012. [Online]. Available: <https://doi.org/10.1145/2071379.2071383>
- [63] R. Rugina and M. C. Rinard, “Recursion unrolling for divide and conquer programs,” in *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, ser. LCPC ’00. Berlin, Heidelberg: Springer-Verlag, 2000, p. 34–48.
- [64] M. Frigo and S. G. Johnson, “FFTW: an adaptive software architecture for the FFT,” in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP ’98 (Cat. No.98CH36181)*, vol. 3. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1381–1384 vol.3.
- [65] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, “Cache-oblivious algorithms,” in *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. New York, NY: IEEE Computer Society Press, 1999, pp. 285–297.
- [66] M. Frigo and S. G. Johnson, “The design and implementation of FFTW3,” *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.
- [67] J. W. Demmel, *Applied Numerical Linear Algebra*. Washington, DC, USA: Society for Industrial and Applied Mathematics, 1997. [Online]. Available: <https://epubs.siam.org/doi/abs/10.1137/1.9781611971446>
- [68] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users’ Guide*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.
- [69] R. W. Vuduc, “Automatic performance tuning of sparse matrix kernels,” Ph.D. dissertation, University of California, Berkeley, 2003.
- [70] T. A. Davis and Y. Hu, “The University of Florida sparse matrix collection,” *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011. [Online]. Available: <https://doi.org/10.1145/2049662.2049663>
- [71] A. J. Kunem, P. Brown, T. S. Bailey, and P. G. Maginot, “Kripke: 3D Sn deterministic particle transport code,” <https://computation.llnl.gov/projects/co-design/kripke>, 2014.



- [72] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes, “The ALPBench benchmark suite for complex multimedia applications,” in *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, Oct 2005, pp. 34–45.
- [73] LLNL, “ASC Sequoia benchmark,” <https://asc.llnl.gov/sequoia/benchmarks/>, 2008.
- [74] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. B. Taylor, “CortexSuite: A synthetic brain benchmark suite,” in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2014, pp. 76–79.
- [75] P. Rundberg and F. Warg, “The FreeBench v1.0 benchmark suite,” <http://www.freebench.org>, 2002.
- [76] R. F. V. der Wijngaart and T. G. Mattson, “The parallel research kernels,” in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2014, pp. 1–6.
- [77] T. Peters, “Livermore loops coded in C,” <http://www.netlib.org/benchmark/livermorec>, 1992.
- [78] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf, “MediaBench II video: Expediting the next generation of video systems research,” *Microprocessors and Microsystems*, vol. 33, no. 4, pp. 301 – 318, 2009, media and Stream Processing. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S014193310900026X>
- [79] J. Dongarra, G. H. Golub, E. Grosse, C. Moler, and K. Moore, “Netlib and NA-Net: Building a scientific computing community,” *IEEE Annals of the History of Computing*, vol. 30, no. 2, pp. 30–41, April 2008.
- [80] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, “The NAS parallel benchmarks — summary and preliminary results,” in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing ’91. New York, NY, USA: ACM, 1991. [Online]. Available: <http://doi.acm.org/10.1145/125826.125925> pp. 158–165.
- [81] L.-N. Pouchet, “Polybench: The polyhedral benchmark suite.” <http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/>, 2012.
- [82] B. M. Roldan Pozo, “SciMark 2.0,” <http://math.nist.gov/scimark2>, 2004.
- [83] J. L. Henning, “SPEC CPU2000: measuring CPU performance in the new millennium,” *Computer*, vol. 33, no. 7, pp. 28–35, July 2000.
- [84] J. L. Henning, “SPEC CPU2006 benchmark descriptions,” *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006. [Online]. Available: <http://doi.acm.org/10.1145/1186736.1186737>

- [85] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua, “An evaluation of vectorizing compilers,” in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’11. Washington, DC, USA: IEEE Computer Society, 2011. [Online]. Available: <http://dx.doi.org/10.1109/PACT.2011.68> pp. 372–382.
- [86] GAP, “Groups, algorithms, programming - a system for computational discrete algebra.” [www.gap-system.org](http://www.gap-system.org), 2007.
- [87] LAME, “LAME MP3 encoder.” [lame.sourceforge.net](http://lame.sourceforge.net), 2017.
- [88] Mozilla, “Mozilla JPEG encoder project.” [github.com/mozilla/mozjpeg](https://github.com/mozilla/mozjpeg), 2017.
- [89] TwoLAME, “TwoLAME - MPEG audio layer 2 encoder,” [github.com/mozilla/mozjpeg](https://github.com/mozilla/mozjpeg), 2017.
- [90] J. Redmon, “Darknet: Open source neural networks in C,” <http://pjreddie.com/darknet/>, 2013–2016.
- [91] Z. Gong, Z. Chen, J. Szaday, D. Wond, Z. Sura, N. Watkinson, S. Maleki, D. Padua, A. Veidenbaum, A. Nicolau, and J. Torrellas, “An empirical study of the effect of source-level loop transformations on compiler stability,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2018.
- [92] Z. Chen, Z. Gong, J. J. Szaday, D. C. Wong, D. Padua, A. Nicolau, A. V. Veidenbaum, N. Watkinson, Z. Sura, S. Maleki, J. Torrellas, and G. DeJong, “Lore: A loop repository for the evaluation of compilers,” in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2017, pp. 219–228.
- [93] M. Diener, L. V. Kale, and D. J. Bodony, “Heterogeneous computing with OpenMP and Hydra,” *Concurrency and Computation: Practice and Experience*, vol. n/a, no. n/a, p. e5728, 2020. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5728>
- [94] M. Diener, D. J. Bodony, and L. Kale, “Accelerating scientific applications on heterogeneous systems with HybridOMP,” in *High Performance Computing for Computational Science – VECPAR 2018*, H. Senger, O. Marques, R. Garcia, T. Pinheiro de Brito, R. Iope, S. Stanzani, and V. Gil-Costa, Eds. Cham: Springer International Publishing, 2019, pp. 174–187.
- [95] S. G. De Gonzalo, S. Huang, J. Gómez-Luna, S. Hammond, O. Mutlu, and W.-m. Hwu, “Automatic generation of warp-level primitives and atomic instructions for fast and portable parallel reduction on GPUs,” in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. IEEE Press, 2019, p. 73–84.

- [96] H. Massalin, “Superoptimizer: A look at the smallest program,” in *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS II. Washington, DC, USA: IEEE Computer Society Press, 1987. [Online]. Available: <https://doi.org/10.1145/36206.36194> p. 122–126.
- [97] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati, “Scaling up superoptimization,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: <https://doi.org/10.1145/2872362.2872387> p. 297–310.
- [98] E. Schkufza, R. Sharma, and A. Aiken, “Stochastic superoptimization,” in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: <https://doi.org/10.1145/2451116.2451150> p. 305–316.
- [99] R. Baghdadi, J. Ray, M. B. Romdhane, E. D. Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, “Tiramisu: A polyhedral compiler for expressing fast and portable code,” in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019, pp. 193–205.
- [100] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, no. 4, p. 65–76, Apr. 2009. [Online]. Available: <https://doi.org/10.1145/1498765.1498785>
- [101] J. L. Hennessy and D. A. Patterson, “A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development,” [https://iscaconf.org/isca2018/turing\\_lecture.html](https://iscaconf.org/isca2018/turing_lecture.html), 2018.
- [102] J. Choquette, O. Giroux, and D. Foley, “Volta: Performance and programmability,” *IEEE Micro*, vol. 38, no. 2, pp. 42–52, 2018.