Towards Transparent and Parsimonious Methods for Automatic Performance Tuning

Pedro Bruel

March 24, 2021

Contents

List of Tables

List of Figures

Part I

Historical Hardware Design Trends and Consequences for Code Optimization

Chapter 1

Introduction

Computer performance has sustained exponential increases over the last half century, despite current physical limits on hardware design. Software optimization has performed an increasingly substantial role on performance improvement over the last 15 years, requiring the exploration of larger and more complex search spaces than ever before.

Autotuning methods are one approach to tackle performance optimization of complex search spaces, enabling exploitation of existing relationships between program parameters and performance. We can derive autotuning methods from well-established statistics, although their usage is not common in or standardized for autotuning domains.

Initial work on this thesis studied the effectiveness of classical and standard search heuristics, such as Simulated Annealing, on autotuning problems. The first target autotuning domain was the set of parameters of a compiler for CUDA programs. The search heuristics for this case study were implemented using the OpenTuner framework [1], and consisted of an ensemble of search heuristics coordinated by a Multi-Armed Bandit algorithm. The autotuner searched for a set of compilation parameters that optimized 17 heterogeneous GPU kernels, from a set of approximately 10^{23} possible combinations of all parameters. With 1.5h autotuning runs we have achieved up to 4× speedup in comparison with the CUDA compiler's high-level optimizations. The compilation and execution times of programs in this autotuning domain are relatively fast, and were in the order of a few seconds to a minute. Since measurement costs are relatively small, search heuristics could find good optimizations using as many measurements as needed. A detailed description of this work is available in our paper [2] published in the *Concurrency and Computation: Practice and Experience* journal.

The next case study was developed in collaboration with *Hewlett-Packard Enterprise*, and consisted of applying the same heuristics-based autotuning approach to the configuration of parameters involved in the generation of FPGA hardware specification from source code in the C language, a process called *High-Level Synthesis* (HLS). The main difference from our work with GPU compiler parameters was the time to obtain the hardware specification, which could be in the order of hours for a single kernel.

In this more complex scenario, we achieved up to 2× improvements for different hardware metrics using conventional search algorithms. These results were obtained in a simple HLS benchmark, for which compilation times were in the order of minutes. The search space was composed of approximately 10^{123} possible configurations, which is much larger than the search space in our previous work with GPUs. Search space size and the larger measurement cost meant that we did not expect the heuristics-based approach to have the same effectiveness as in the GPU compiler case study. This work was published [3] at the 2017 *IEEE International Conference on ReConFigurable Computing and FPGAs*.

Approaches using classical machine learning and optimization techniques would not scale to industrial-level HLS, where each compilation can take hours to complete. Search space properties also increase the complexity of the problem, in particular its structure composed of binary, factorial and continuous variables with potentially complex interactions. Our results on autotuning HLS for FPGAs corroborate the conclusion that the empirical autotuning of expensive-to-evaluate functions, such as those that appear on the autotuning of HLS, require a more parsimonious and transparent approach, that can potentially be achieved using the DoE methodology. The next section describes our work on applying the DoE methodology to autotuning.

The main contribution of this thesis is a strategy to apply the Design of Experiments methodology to autotuning problems. The strategy is based on linear regression and its extensions, Analysis of Variance (ANOVA), and Optimal Design. The strategy requires the formulation of initial assumptions about the target autotuning problem, which are refined with data collected by efficiently selected experiments. The main objectives are to identify relationships between parameters, suggesting regions for further experimentation in a transparent way, that is, in a way that is supported by statistical tests of significance. The effectiveness of the proposed strategy, and its ability to explain optimizations it finds, are evaluated on autotuning problems in the source-to-source transformation domain. The initial stages of this work resulted in a publication on IEEE/ACM CCGrid [4].

Further efforts in this direction were dedicated to describe precisely what can be learned about search spaces from the application of the methodology, and to refine the differentiation of the approach for the sometimes conflicting objectives of model assessment and prediction. These discussions are presented in Chapter 12.

Because the Design of Experiments methodology requires the specification of a class of initial performance models, the methodology can sometimes achieve worse prediction capabilities when there is considerable uncertainty on initial assumptions about the underlying relationships. Chapter 13 and describe the application to autotuning of Gaussian Process Regression, an approach that trades some explanatory power for a much larger and more flexible class of underlying models. We also evaluate the performance of this approach on the source-to-source transformation problem from Chapter 12, and on larger autotuning problem on the mixed-precision quantization of *Convolutional Neural Network* (CNN) layers, in Chapter 14.

Describe thesis structure

Chapter 2

The Need for Autotuning

High Performance Computing has been a cornerstone of scientific and industrial progress for at least five decades. By paying the cost of increased complexity, software and hardware engineering advances continue to overcome several challenges on the way of the sustained performance improvements observed during the last fifty years. A consequence of this mounting complexity is that reaching the theoretical peak hardware performance for a given program requires not only expert knowledge of specific hardware architectures, but also mastery of programming models and languages for parallel and distributed computing.

If we state performance optimization problems as *search* or *learning* problems, by converting implementation and configuration choices to *parameters* which might affect performance, we can draw from and adapt proven methods from search, mathematical optimization, and statistical learning. The effectiveness of these adapted methods on performance optimization problems varies greatly, and hinges on practical and mathematical properties of the problem and the corresponding *search space*. The application of such methods to the automation of performance tuning for specific hardware, under a set of *constraints*, is named *autotuning*.

Improving performance also relies on gathering application-specific knowledge, which entails extensive experimental costs since, with the exception of linear algebra routines, theoretical peak performance is not always a reachable comparison baseline. When adapting methods for autotuning we must face challenges emerging from practical properties, such as restricted time and cost budgets, constraints on feasible parameter values, and the need to mix *categorical*, *continuous*, and *discrete* parameters. To achieve useful results we must also choose methods that make hypotheses compatible with problem search spaces, such as the existence of *discoverable*, or at least *exploitable*, relationships between parameters and performance. Choosing an autotuning method requires balancing the exploration of a problem, that is, seeking to discover and explain relationships between parameters and performance, and the exploitation of known or discovered relationships, seeking only to find the best possible performance.

Search algorithms based on machine learning heuristics are not the best candidates for autotuning domains where measurements are lengthy and costly, such as compiling industrial-level FPGA programs, because these algorithms rely on the availability of a large number of measurements. They also assume good optimizations are reachable from a starting position, and that tendencies observed locally in the search space are exploitable. These assumptions are not usually true in common autotuning domains, as shown in the work of Seymour *et al.* [5].

Autotuning search spaces also usually have non-linear constraints and undefined regions, which are also expected to decrease the effectiveness of search based on heuristics and machine learning. An additional downside to heuristics- and machine learning-based search is that, usually, optimization choices cannot be explained, and knowledge gained during optimization is not reusable. The main contribution of this thesis is to study how to overcome the reliance on these assumptions about search spaces, and the lack of explainable optimizations, from the point of view of a *Design of Experiments* (DoE), or *Experimental Design*, methodology to autotuning.

One of the first detailed descriptions and mathematical treatment of DoE was presented by Ronald Fisher [6] in his 1937 book *The Design of Experiments*, where he discussed principles of experimentation, latin square sampling and factorial designs. Later books such as the ones from Jain [7], Montgomery [8] and Box *et al.* [9] present comprehensive and detailed foundations. Techniques based on DoE are *parsimonious* because they allow decreasing the number of measurements required to determine certain relationships between parameters and metrics, and are *transparent* because parameter selections and configurations can be justified by the results of statistical tests.

In DoE terminology, a *design* is a plan for executing a series of measurements, or *experiments*, whose objective is to identify relationships between *factors* and *responses*. While factors and responses can refer to different concrete entities in other domains, in computer experiments factors can be configuration parameters for algorithms and compilers, for example, and responses can be the execution time or memory consumption of a program.

Designs can serve diverse purposes, from identifying the most significant factors for performance, to fitting analytical performance models for the response. The field of DoE encompasses the mathematical formalization of the construction of experimental designs. More practical works in the field present algorithms to generate designs with different objectives and restrictions.

The contributions of this thesis are strategies to apply to program autotuning the DoE methodology, and *Gaussian Process Regression* [10]. This thesis presents background and a high-level view of the theoretical foundations of each method, and detailed discussions of the challenges involved in specializing the general definitions of search heuristics and statistical learning methods to different autotuning problems, as well as what can be *learned* about

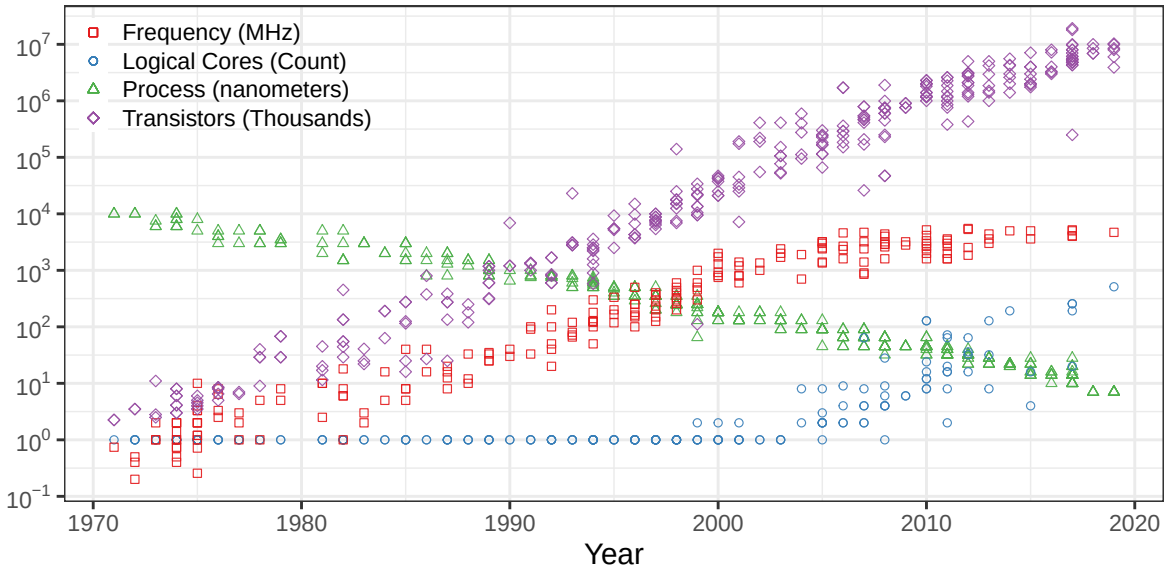

Figure 2.1: 49 years of microprocessor data, highlighting the sustained exponential increases and reductions on transistor counts and fabrication processes, the stagnation of frequency scaling around 2005, and one solution found for it, the simultaneous exponential increase on logical core count. Data from Wikipedia [13, 14]

specific autotuning search spaces, and how that acquired knowledge can be leveraged for further optimization.

This chapter aims to substantiate the claim that autotuning methods have a fundamental role to play on the future of program performance optimization, arguing that the value and the difficulty of the efforts to carefully tune software became more apparent ever since advances in hardware stopped leading to effortless performance improvements, at least from the programmer's perspective. The following sections discuss the historical context for the changes in trends on computer architecture, and characterize the search spaces found when optimizing performance on different domains.

2.1 Historical Hardware Design Trends

The physical constraints imposed by technological advances on circuit design were evident since the first vacuum tube computers that already spanned entire floors, such as the ENIAC in 1945 [11]. The practical and economical need to fit more computing power into real estate is one force for innovation in hardware design that spans its history, and is echoed in modern supercomputers, such as the *Summit* from *Oak Ridge National Laboratory* [12], which spans an entire room.

Figure 2.1 highlights the unrelenting and so far successful pursuit of smaller transistor fabrication processes, and the resulting capability to fit more computing power on a fixed chip area. This trend was already observed in integrated circuits by Gordon Moore *et al.*

in 1965 [15], who also postulated its continuity. The performance improvements produced by the design efforts to make Moore's forecast a self-fulfilling prophecy were boosted until around 2005 by the performance gained from increases in circuit frequency.

Robert Dennard *et al.* remarked in 1974 [16] that smaller transistors, in part because they generate shorter circuit delays, decrease the energy required to power a circuit and enable an increase in operation frequency without breaking power usage constraints. This scaling effect, named *Dennard's scaling*, is hindered primarily by leakage current, caused by quantum tunneling effects in small transistors. Figure 2.1 shows a marked stagnation on frequency increase after around 2005, as transistors crossed the 10^2nm fabrication process. It was expected that leakage due to tunneling would limit frequency scaling strongly, even before the transistor fabrication process reached 10nm [17].

Current hardware is now past the effects of Dennard's scaling. The increase in logical cores around 2015 can be interpreted as preparation for and mitigation of the end of frequency scaling, and ushered in an age of multicore scaling. Still, in order to meet power consumption constraints, up to half of a multicore processor could have to be powered down, at all times. This phenomenon is named *Dark Silicon* [18], and presents significant challenges to current hardware designers and programmers [19, 20, 21].

The *Top500* [22] list gathers information about commercially available supercomputers, and ranks them by performance on the *LINPACK* benchmark [23]. Figure 2.2 shows the peak theoretical performance $RPeak$, and the maximum performance achieved on the LINPACK benchmark $RMax$, in $Tflops/s$, for the top-ranked supercomputers on TOP500. Despite the smaller performance gains from hardware design that are to be expected for post-Dennard's scaling processors, the increase in computer performance has sustained an exponential climb, sustained mostly by software improvements.

Although *hardware accelerators* such as GPUs and FPGAs, have also helped to support exponential performance increases, their use is not an escape from the fundamental scaling constraints imposed by current semiconductor design. Figure 2.3 shows the increase in processor and accelerator core count on the top-ranked supercomputers on Top500. Half of the top-ranked supercomputers in the last decade had accelerator cores and, of those, all had around ten times more accelerator than processor cores. The apparent stagnation of core count in top-ranked supercomputers, even considering accelerators, highlights the crucial impact software optimization has on performance.

Advances in hardware design are currently not capable of providing performance improvements via frequency scaling without dissipating more power than the processor was designed to support, which violates power constraints and risks damaging the circuit. From the programmer's perspective, effortless performance improvements from hardware have not been expected for quite some time, and the key to sustaining historical trends in performance scaling has lied in accelerators, parallel and distributed programming libraries, and fine tun-

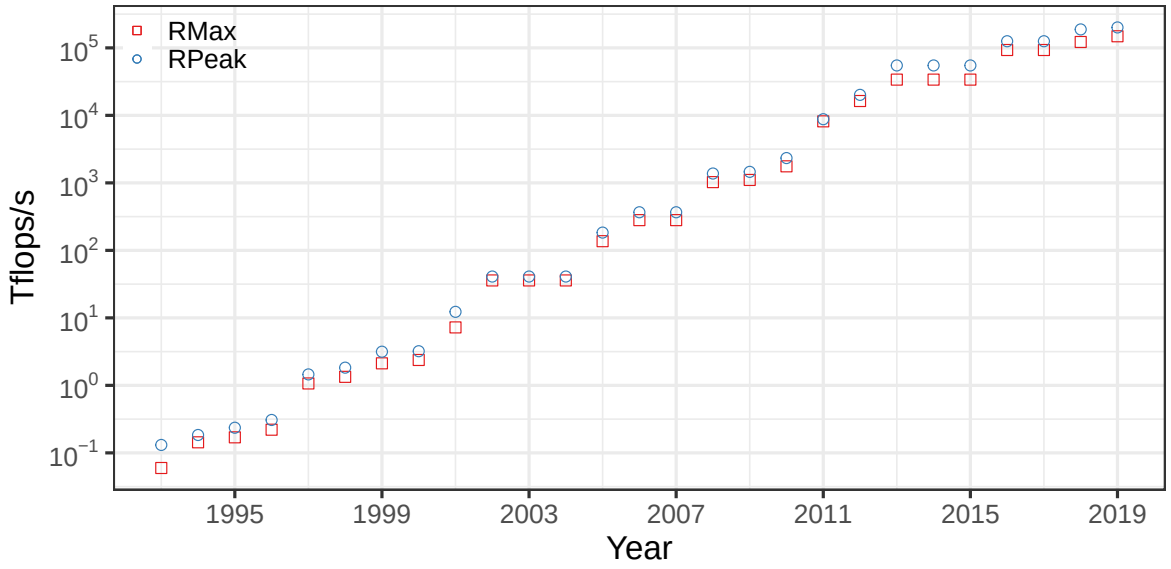

Figure 2.2: Sustained exponential increase of theoretical *RPeak* and achieved *RMax* performance for the supercomputer ranked 1st on TOP500 [22]

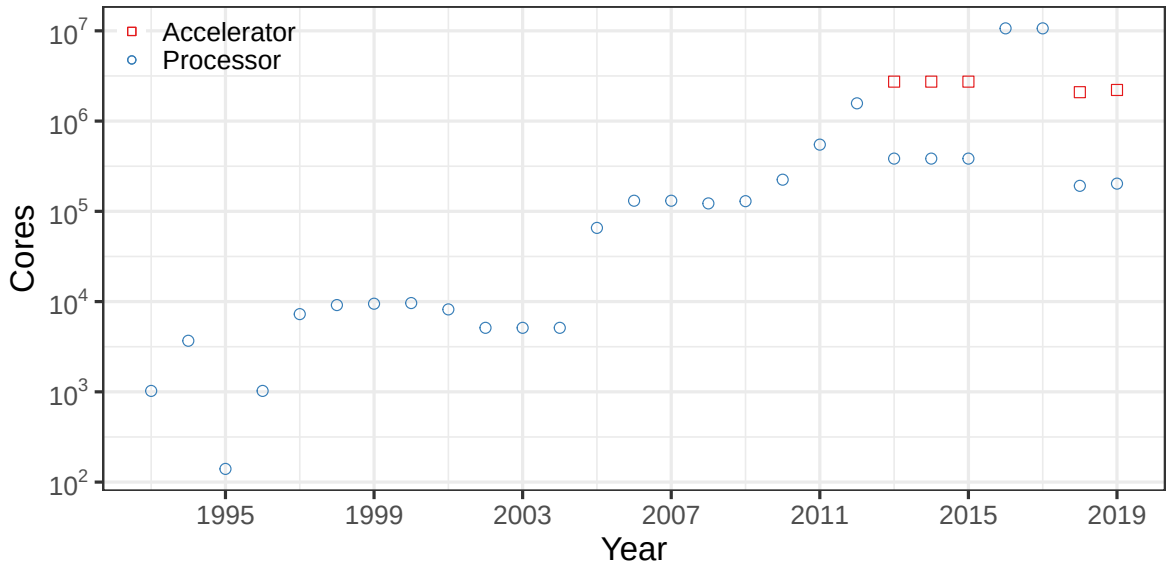

Figure 2.3: Processor and accelerator core count in supercomputers ranked 1st on TOP500 [22]. Core count trends for supercomputers are not necessarily bound to processor trends observed on Figure 2.1.

ing of several stages of the software stack, from instruction selection to the layout of neural networks.

The problem of optimizing software for performance presents its own challenges. The search spaces that emerge from autotuning problems grow quickly to a size for which it would take a prohibitive amount of time to determine the best configuration by exhaustively evaluating all possibilities. Although this means we must seek to decrease the amount of

possibilities, by restricting allowed parameter values, or dropping parameters completely, it is often unclear how to decide which parameters should be restricted or dropped. The next sections introduce a simple autotuning problem, present an overview of the magnitude of the dimension of autotuning search spaces, and briefly introduce the methods commonly used to explore search spaces, some of which are discussed in detail in Chapter II.

2.2 Consequences for Code Optimization: Autotuning Loop Nest Optimization

Algorithms for linear algebra problems are fundamental to scientific computing and statistics. Therefore, decreasing the execution time of algorithms such as general matrix multiplication (GEMM) [24], and others from the original BLAS [25], is an interesting and well motivated example, that we will use to introduce the autotuning problem.

One way to improve the performance of such linear algebra programs is to exploit cache locality by reordering and organizing loop iterations, using source code transformation methods such as loop *tiling*, or *blocking*, and *unrolling*. We will now briefly describe loop tiling and unrolling for a simple linear algebra problem. After, we will discuss an autotuning search space for blocking and unrolling applied to GEMM, and how these transformations generate a relatively large and complex search space, which we can explore using autotuning methods.

Figure 2.4 shows three versions of code in the *C* language that, given three square matrices A, B, and C, computes $C = C + A + B^\top$. The first optimization we can make is to preemptively load to cache, or *prefetch*, as many as possible of the elements we know will be needed at any given iteration, as is shown in Figure 2.4a. The shaded elements on the top row of Figure 2.5 represent the elements that could be prefetched in iterations of Figure 2.4a.

Since *C* matrices are stored in *row-major order*, each access of an element of B forces loading the next row elements, even if we explicitly prefetch a column of B. Since we are accessing B in a *column-major order*, the prefetched row elements would not be used until we reached the corresponding column. Therefore, the next column elements will have to be loaded at each iteration, considerably slowing down the computation.

We can solve this problem by reordering memory accesses to request only prefetched elements. It suffices to adequately split loop indices into blocks, as shown in Figure 2.4b. Now, memory accesses are be performed in *tiles*, as shown on the bottom row of Figure 2.5. If blocks are correctly sized to fit in cache, we can improve performance by explicitly prefetching each tile. After blocking, we can still improve performance by *unrolling* loop iterations, which forces register usage and helps the compiler to identify regions that can be *vectorized*. A conceptual implementation of loop unrolling is shown in Figure 2.4c.

Looking at the loop nest optimization problem from the autotuning perspective, the two *parameters* that emerge from the implementations are the *block size*, which controls the stride,

```
int N = 256;
float A[N][N], B[N][N], C[N][N];
int i, j;
// Initialize A, B, C
for(i = 0; i < N; i++){
  // Load line i of A to fast memory
  for(j = 0; j < N; j++){
    // Load C[i][j] to fast memory
    // Load column j of B to fast memory
    C[i][j] += A[i][j] + B[j][i];
    // Write C[i][j] to main memory
  }
}
```

(a) Regular implementation

```
int N = 256;
int B_size = 4;
int A[N][N], B[N][N], C[N][N];
int i, j, x, y;
// Initialize A, B, C
for(i = 0; i < N; i += B_size){
  for(j = 0; j < N; j += B_size){
    // Load block (i, j) of C to fast memory
    // Load block (i, j) of A to fast memory
    // Load block (j, i) of B to fast memory
    for(x = i; x < min(i + B_size, N); x++){
      for(y = j; y < min(j + B_size, N); y++){
        C[x][y] += A[x][y] + B[y][x];
      }
    }
    // Write block (i, j) of C to main memory
  }
}
```

(b) Blocked, or tiled

```
int N = 256;
int B_size = 4;
int A[N][N], B[N][N], C[N][N];
int i, j, k;
// Initialize A, B, C
for(i = 0; i < N; i += B_size){
  for(j = 0; j < N; j += B_size){
    // Load block (i, j) of C to fast memory
    // Load block (i, j) of A to fast memory
    // Load block (j, i) of B to fast memory
    C[i + 0][j + 0] += A[i + 0][j] * B[i][j + 0];
    C[i + 0][j + 1] += A[i + 0][j] * B[i][j + 1];
    // Unroll the remaining 12 iterations
    C[i + Bsize - 1][j + B_size - 2] += A[i + Bsize - 1][j] * B[i][j + B_size - 2];
    C[i + Bsize - 1][j + B_size - 1] += A[i + Bsize - 1][j] * B[i][j + B_size - 1];
    // Write block (i, j) of C to main memory
  }
}
```

(c) Tiled and unrolled

Figure 2.4: Loop nest optimizations for $C = C + A + B^\top$, in C

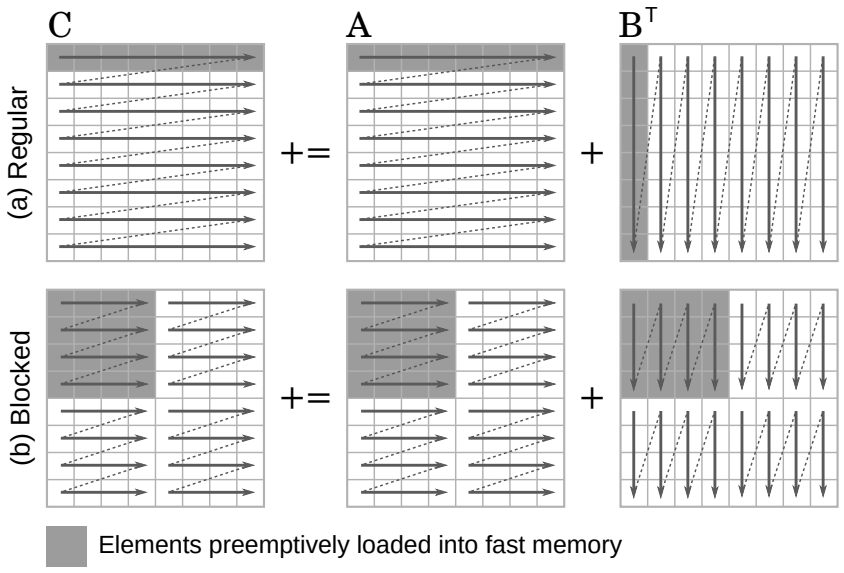

Figure 2.5: Access patterns for matrices in $C = C + A + B^\top$, with loop nest optimizations. Panel *(a)* shows the access order of a regular implementation, and panel *(b)* shows the effect of loop *tiling*, or *blocking*

and the *unrolling factor*, which controls the number of unrolled iterations. Larger block sizes are desirable, because we want to avoid extra comparisons, but blocks should be small enough

```c
int N = 256;
float A[N][N], B[N][N], C[N][N];
int i, j, k;
// Initialize A, B, C
for(i = 0; i < N; i++){
  // Load line i of A to fast memory
  for(j = 0; j < N; j++){
    // Load C[i][j] to fast memory
    // Load column j of B to fast memory
    for(k = 0; k < N; k++){
      C[i][j] += A[i][k] * B[k][j];
    }
    // Write C[i][j] to main memory
  }
}
```

(a) Regular implementation

```c
int N = 256;
int B_size = 4;
int A[N][N], B[N][N], C[N][N];
int i, j, k, x, y;
// Initialize A, B, C
for(i = 0; i < N; i += B_size){
  for(j = 0; j < N; j += B_size){
    // Load block (i, j) of C to fast memory
    for(k = 0; k < N; k++){
      // Load block (i, k) of A to fast memory
      // Load block (k, y) of B to fast memory
      for(x = i; x < min(i + B_size, N); x++){
        for(y = j; y < min(j + B_size, N); y++){
          C[x][y] += A[x][k] * B[k][y];
        }
      }
    }
    // Write block (i, j) of C to main memory
  }
}
```

(b) Blocked, or tiled

```c
int N = 256;
int B_size = 4;
int A[N][N], B[N][N], C[N][N];
int i, j, k;
// Initialize A, B, C
for(i = 0; i < N; i += B_size){
  for(j = 0; j < N; j += B_size){
    // Load block (i, j) of C to fast memory
    for(k = 0; k < N; k++){
      // Load block (i, k) of A to fast memory
      // Load block (k, y) of B to fast memory
      C[i + 0][j + 0] += A[i + 0][k] * B[k][j + 0];
      C[i + 0][j + 1] += A[i + 0][k] * B[k][j + 1];
      // Unroll the remaining 12 iterations
      C[i + Bsize - 1][j + B_size - 2] += A[i + Bsize - 1][k] * B[k][j + B_size - 2];
      C[i + Bsize - 1][j + B_size - 1] += A[i + Bsize - 1][k] * B[k][j + B_size - 1];
    }
    // Write block (i, j) of C to main memory
  }
}
```

(c) Tiled and unrolled

Figure 2.6: Loop nest optimizations for GEMM, in C

to ensure access to as few as possible out-of-cache elements. Likewise, the unrolling factor should be large, to leverage vectorization and available registers, but not so large that it forces memory to the stack.

The values of block size and unrolling factor that optimize performance will depend on the cache hierarchy, register layout, and vectorization capabilities of the target processor, but also on the memory access pattern of the target algorithm. In addition to finding the best values for each parameter independently, an autotuner must ideally aim to account for the *interactions* between parameters, that is, for the fact that the best value for each parameter might also depend on the value chosen for the other.

The next loop optimization example comes from Seymour *et al.* [5], and considers 128 blocking and unrolling values, in the interval $[0, 127]$, for the GEMM algorithm. The three panels of Figure 2.6 show conceptual implementations of loop blocking and unrolling for GEMM in *C*. A block size of zero results in the implementation from Figure 2.6a, and an unrolling factor of zero performs a single iteration per condition check.

It is straightforward to change the block size of the implementations from Figure 2.6, but the unrolling factor is not exposed as a parameter. To test different unrolling values we need

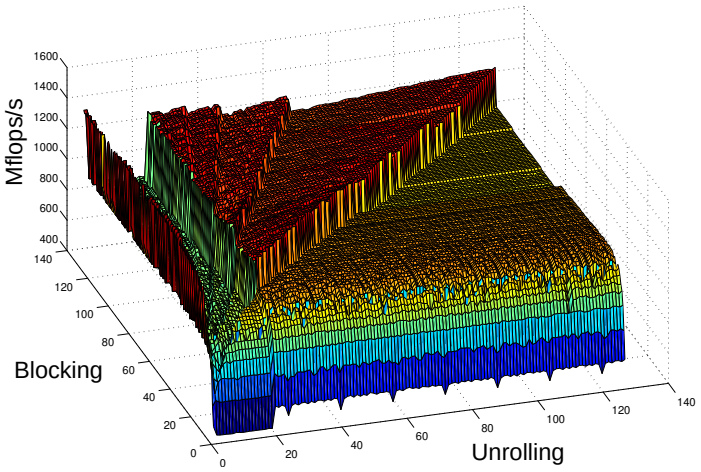

Figure 2.7: An exhaustively measured search space, defined by loop blocking and unrolling parameters, for a sequential GEMM kernel. Reproduced from Seymour *et al.* [5]

to generate new versions of the source code with different numbers of unrolled iterations. We can do that with code generators or with *source-to-source transformation* tools [26, 27, 28]. It is often necessary to modify the program we wish to optimize in order to provide a configuration interface and expose its implicit parameters. Once we are able to control the block size and the loop unrolling factor, we determine the target search space by choosing the values to be explored.

In this example, the search space is defined by the $128^2 = 16384$ possible combinations of blocking and unrolling values. The performance of each combination in the search space, shown in *Mflops/s* in Figure 2.7, was measured for a sequential GEMM implementation, using square matrices of size 400 [5]. We can represent this autotuning search space as a *3D landscape*, since we have two configurable parameters and a single target performance metric. In this setting, the objective is to find the *highest* point, since the objective is to *maximize* Mflops/s, although usually the performance metric is transformed so that the objective is its *minimization*.

On a first look, there seems to be no apparent global search space structure in the landscape on Figure 2.7, but local features jump to the eyes, such as the "valley" across all block sizes for low unrolling factors, the "ramp" across all unrolling factors for low block sizes, and the series of jagged "plateaus" across the middle regions, with ridges for identical or divisible block sizes and unrolling factors. A careful look reveals also that there is a curvature along the unrolling factor axis. Also of note is the abundance in this landscape of *local minima*, that is, points with relatively good performance, surrounded by points with worse performance. By exhaustively evaluating all possibilities, the original study determined that the best performance on this program was achieved with a block size of 80 and an unrolling factor of 2.

In this conceptual example, all $\approx 1.64 \times 10^4$ configurations were exhaustively evaluated, but it is impossible to do so in most settings where autotuning methods are useful. The

next section provides a perspective of the autotuning domains and methods employed in current research, presenting a selection of search spaces and discussing the trends that can be observed on search space size, targeted HPC domains, and chosen optimization methods.

2.3 Autotuning Approaches and Search Spaces

Autotuning methods have been used to improve performance in an increasingly large variety of domains, from the earlier applications to linear algebra subprograms, to the now ubiquitous construction and configuration of neural networks, to the configuration of the increasingly relevant tools for the re-configurable hardware of FPGAs. In this setting, it is not far-fetched to establish a link between the continued increases in performance and hardware complexity, that we discussed previously in this chapter, to the increases in dimension and size of the autotuning problems that we can now tackle.

Figure 2.8 presents search space dimension, measured as the number of parameters involved, and size, measured as the number of possible parameter combinations, for a selection of search spaces from 14 autotuning domains. Precise information about search space characterization is often missing from works on autotuning methods and applications. The characterization of most of the search spaces in Figure 2.8 was obtained directly from the text of the corresponding published paper, but for some it was necessary to extract characterizations from the available source code. Still, it was impossible to obtain detailed descriptions of search spaces for many of the published works on autotuning methods and applications, and in that way the sample shown in this section is biased, because it contains only information on works that provided it.

The left hand panel of Figure 2.8 shows search spaces with up to 60 parameters. The over-representation of search spaces for linear algebra domains in this sample stands out on the left hand panel, but the domain is not present on the remaining portion of the sample, shown on the right hand panel. The largest search spaces for which we were able to find information on published work are defined for the domains of neural network configuration, High-Level Synthesis for FPGAs, compiler parameters, and domain-specific languages.

None of the largest search spaces in this sample, that is, the ones outside the zoomed area of the left hand panel, come from works earlier than 2009. The sustained performance improvements we discussed previously have enabled and pushed autotuning research toward progressively larger problems, which has also been done to most research areas. Increased computing power has made it feasible, or at least tolerable, to apply search heuristics and statistical learning methods to find program configurations that improve performance.

It is straightforward to produce an extremely large autotuning search space. Compilers have hundreds of binary flags that can be considered for selection, generating a large set of combinations. Despite that, regarding performance improvements, it is likely that most configuration parameters will have a small impact, that is, that only a handful of parameters

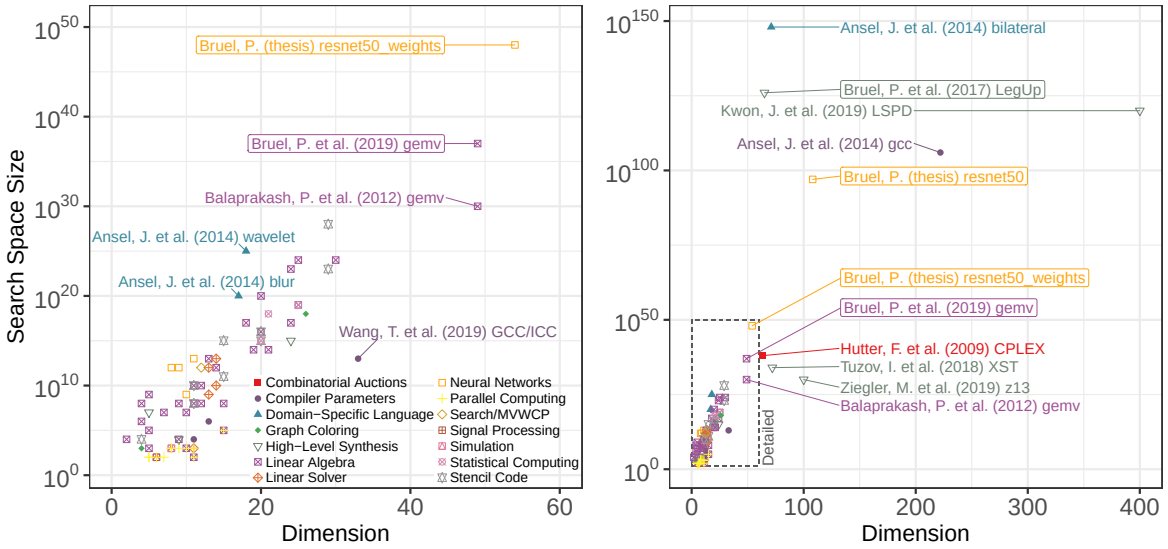

Figure 2.8: Dimension and search space size for autotuning problems from 14 domains [29, 1, 30, 31, 32, 4, 33, 2, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 5] The left panel shows a zoomed view of the right panel

are responsible for changes in performance. Search spaces are often much more restrictive than the one we discussed in Section 2.2. Autotuning problem definitions usually come with *constraints* on parameter values and limited *experimental budgets*, and *runtime* failures for some configurations are often unpredictable. In this context, finding configurations that improve performance and determining the subset of *significant parameters* are considerable challenges.

Search heuristics, such as methods based on genetic algorithms and gradient descent, are a natural way to tackle these challenges because they consist of procedures for exploiting existing and unknown relationships between parameters and performance without making or requiring *explicit hypotheses* about the problem. Despite that, most commonly used heuristics make *implicit hypotheses* about search spaces which are not always verified, such as assuming that good configurations are *reachable* from a random starting point.

Autotuning methods that make explicit hypotheses about the target program, such as methods based on Design of Experiments, require some initial knowledge, or willingness to make assumptions, about underlying relationships, and are harder to adapt to constrained scenarios, but have the potential to produce *explainable* optimizations. In general, methods based on Machine Learning have enough flexibility to perform well in complex search spaces and make few assumptions about problems, but usually provide little, if any, that can be used to explain optimization choices or derive relationships between parameters and performance.

Table 2.1 lists some autotuning systems, their target domains, and the employed method, ordered by publication date. Some systems that did not provide detailed search space descriptions and could not be included in Figure 2.8, especially some of the earlier work, provided enough information to categorize their autotuning methods. In contrast, many more recent

Table 2.1: Autotuning methods used by a sample of systems, in different domains, ordered by publishing year. Methods were classified as either Search Heuristics (SH), Machine Learning (ML), or more precisely when the originating work provided detailed information. Earlier work favored employing Search Heuristics, which are less prominent in recent work, which favors methods based on Machine Learning.

System	Domain	Method	Year
PhiPAC [46]	Linear Algebra	SH (Exhaustive)	1997
ATLAS [47]	Linear Algebra	SH (Exhaustive)	1998
FFTW [48]	Digital Signal Processing	SH (Exhaustive)	1998
Active Harmony [49]	Domain-Specific Language	SH	2002
OSKI [50]	Linear Algebra	SH	2005
Seymour, K. *et al.* [5]	Linear Algebra	SH	2008
PRO [37]	Linear Algebra	SH	2009
ParamILS [38]	Combinatorial Auctions	SH	2009
PetaBricks [28]	Domain-Specific Language	SH (Genetic Algorithm)	2009
MILEPOST GCC [51]	Compiler Parameters	ML	2011
Orio [29]	Linear Algebra	ML (Decision Trees)	2012
pOSKI [30]	Linear Algebra	SH	2012
INSIEME [52]	Compiler Parameters	SH (Genetic Algorithm)	2012
OpenTuner [1]	Compiler Parameters	SH	2014
Lgen [53]	Linear Algebra	SH	2014
OPAL [54]	Parallel Computing	SH	2014
Mametjanov, A. *et al.* [34]	High-Level Synthesis	ML (Decision Trees)	2015
CLTune [55]	Parallel Computing	SH	2015
Guerreirro, J. *et al.* [56]	Parallel Computing	SH	2015
Collective Mind [57]	Compiler Parameters	ML	2015
Abdelfattah, A. *et al.* [35]	Linear Algebra	SH (Exhaustive)	2016
TANGRAM [58]	Domain-Specific Language	SH	2016
MASE-BDI [59]	Environmental Land Change	SH	2016
Xu, C. *et al.* [36]	High-Level Synthesis	SH	2017
Apollo [60]	Parallel Computing	ML (Decision Trees)	2017
DeepHyper [32]	Neural Networks	ML (Decision Trees)	2018
Tuzov, I. *et al.* [40]	High-Level Synthesis	Design of Experiments	2018
Periscope [42]	Compiler Parameters	SH	2018
SynTunSys [41]	High-Level Synthesis	SH	2019
Kwon, J. *et al.* [43]	High-Level Synthesis	ML	2019
FuncyTuner [44]	Compiler Parameters	SH	2019
Ol'ha, J. *et al.* [45]	Parallel Computing	Sensitivity Analysis	2019
Petrovic, F. *et al.* [31]	Linear Algebra	SH	2020
Chu, Y. *et al.* [39]	Search/MVWCP	SH	2020

works, especially those using methods based on Machine Learning, did not provide specific method information. Earlier work often deals with search spaces small enough to exhaustively evaluate, and using search heuristics to optimize linear algebra programs is the most prominent category of earlier work in this sample. Later autotuning work target more varied domains, with the most prominent domains in this sample being parallel computing, compiler

parameters, and High-Level Synthesis. Systems using methods based on Machine Learning become more common on later work than systems using heuristics.

Chapter 4 provides more detailed definitions and discussions of the applicability, effectiveness, and explanatory power of stochastic autotuning methods based on search heuristics. The remainder of this chapter details the contributions of this thesis and the structure of this document.

Third Chapter, or next Section: Proposal of this thesis, list of contributions

Part II

Optimization Methods for Autotuning

Chapter 3

Notation and Search Spaces

The following chapters discuss the optimization methods that we applied to autotuning problems in different domains. Each chapter presents a group of methods, briefly discussing each method in the group and their underlying hypotheses. The objective of each chapter is to provide high-level and straightforward descriptions of optimization methods, presenting clear definitions tied to the autotuning context.

Each chapter concludes with a discussion of the applicability of each group of methods to autotuning problems. The methods we discuss have significant differences but employ the same basic concepts. We will use the same mathematical notation to discuss all methods when possible. The remainder of this chapter presents common basic concepts and the associated notation to be used in subsequent chapters, which are summarized in Table 3.1. Completing this introduction to optimization in the context of autotuning, the chapter ends with a discussion of common search space properties.

3.1 Notation

We will call *optimization* the minimization of a real-valued function with a single vector input. For a function $f : \mathcal{X} \to \mathbb{R}$, we wish to find the input vector $\mathbf{x}^* = [x_1 \ \dots \ x_p]^\top$ in the *parameter space*, or *search space* $\mathcal{X}$ for which $f(\mathbf{x}^*)$ is the smallest, compared to all other $\mathbf{x} \in \mathcal{X}$. The function f represents, in the autotuning context, the *performance metric* we wish to optimize, such as the execution time of some application, and the parameter space $\mathcal{X}$ represents the set of possible *configurations* we can explore, such as compiler flags. Therefore, we define optimization in the autotuning context as finding the configuration that minimizes the target performance metric. For the sake of simplicity, we assume we can use the opposite of the performance metrics that should be maximized.

As an example of an autotuning problem, consider optimizing the choice of flags for a compiler with $p = |\mathbf{x}|$ flags, where a configuration $\mathbf{x} = [x_1 \ \dots \ x_p]^\top$ consists of a vector of

p boolean values, denoting whether each flag $x_1, \ldots, x_p$ is turned on for the compilation of a specific application. To find the compiler configuration that generates the binary with the smallest execution time, we conduct a set of $n = |\mathbf{X}|$ *experiments*, chosen according to some criterion, generating the $n \times p$ *experimental design matrix* $\mathbf{X} = [\mathbf{x}_1 \ \ldots \ \mathbf{x}_n]^\top$. Each experiment consists of compiling the target application using the specified compiler configuration, and measuring the execution time of the resulting binary.

In this example, evaluating $f(\mathbf{x})$ involves generating the compiler configuration corresponding to the vector $\mathbf{x}$ of selected flags. This involves writing a shell command or a configuration file, running the configured compiler, checking for compilation errors, measuring the execution time of the resulting binary, and verifying the correctness of its output.

In practice, we may never be able to observe the *true value* of $f(\mathbf{x})$. In fact, empirical tests of this nature are always subject to unknown or uncontrollable effects, and to inherent imprecision in measurement. In practice, we settle for observing $y = f(\mathbf{x}) + \varepsilon$, where ε encapsulates all unknown and uncontrollable effects, as well as the measurement error. Returning to the compiler flag example, suppose that we could conduct $n = |\mathcal{X}| = 2^p$ experiments, measuring the performance of the binaries generated with all possible flag combinations. With such experimental design we would obtain the measurements

$$\mathbf{y} = [y_i = f(\mathbf{x}_i) + \varepsilon_i, \ i = 1, \ldots, 2^p]^\top. \tag{3.1}$$

The measurement y_i is an *estimate* of $f(\mathbf{x}_i)$, with error ε_i. If the error is reasonably small, an estimate of the *global optimum* $\mathbf{x}^*$ in this example is the $\mathbf{x}_i$ that produces the binary with the smallest estimated execution time y^*, the smallest $y_i \in \mathbf{y}$.

Assuming we are capable of cheaply evaluating f for a large set of experiments $\mathbf{X}$, and that we are not interested in building statistical models for the performance of our application, we can directly optimize f using stochastic descent methods, or gradient- and hessian-based methods if f is suitably convex and differentiable. These function minimization methods are discussed in Chapter 4.

If we are not capable of directly measuring f, if it is unreasonably expensive or time-consuming to do so, or if constructing statistical performance models for our application is of crucial importance, we can employ the surrogate-based methods discussed in Chapters 5, 6, and 7. These methods use different strategies to construct a *surrogate model*

$$\hat{f}_\theta : \mathcal{X} \to \mathbb{R}, \text{ with } \theta \in \Theta, \tag{3.2}$$

where θ is a parameter vector usually estimated from measurements $(\mathbf{X}, \mathbf{y})$. The function $\theta : \mathcal{X}^n \times \mathbb{R}^n \to \Theta$ represents the *estimation process*, that uses the *observations* $\mathbf{y} \in \mathbb{R}^n$ from an experimental design $\mathbf{X} = [\mathbf{x}_1 \ \ldots \ \mathbf{x}_n]^\top$. The *parameter vector* $\theta(\mathbf{X}, \mathbf{y})$ in the *parameter space* Θ will be used to compute the estimate.

The constructed surrogate model $\hat{f}_{\theta(\mathbf{X},\mathbf{y})}$ can be used as a tool to attempt to describe and optimize the underlying real function f, provided we are able to construct a useful estimate of $\theta(\mathbf{X}, \mathbf{y})$. We will discuss in Chapters 5 and 6 methods that use the m individual parameter estimates $\theta(\mathbf{X}, \mathbf{y}) = [\theta_1 \ \dots \ \theta_m]^\top$ to assess the significance of specific *factors*.

If we use the Ordinary Least Squares (OLS) estimator for θ in our compiler flag example, the parameter estimates $\theta_2, \dots, \theta_m$ could correspond to each one of the p flags. In this case, θ_1 is the estimate of the intercept of the linear model, and $m = p+1$. Optimization methods using the Bayesian inference framework, such as Gaussian Process Regression, which we discuss in Chapter 5, associate a probability distribution to $\theta(\mathbf{X}, \mathbf{y})$, which propagates with $\hat{f}_\theta$ and can be exploited in our optimization context.

Chapter 5 differentiates between parametric methods which make the hypothesis that the number $m = |\theta|$ of estimated parameters is finite and often interpretable, and nonparametric methods, which operate in parameter spaces of infinite dimension.

Table 3.1 summarizes the notation and concepts we have discussed so far, tying those concepts to the compiler flag example we used in the discussion. The notation and the basic concepts we have described in this section, although referring to abstract entities, enable a uniform discussion of different optimization methods in the next chapters.

Table 3.1: Summary of the notation, concepts, and examples discussed in this chapter, and common to the autotuning methods discussed in further chapters

Symbol	Concept	Example
$\mathcal{X}$	Search space	All possible compiler flag assignments
$\mathbf{x} = [x_1 \ \dots \ x_p]^\top \in \mathcal{X}$	Input variable	A specific flag assignment
$\mathbf{X} = [\mathbf{x}_1 \ \dots \ \mathbf{x}_n]^\top \subseteq \mathcal{X}$	Experimental design	A set of flag assignments
$p = \lvert\mathbf{x}\rvert$	Search space dimension	The number of flags to assign
$n = \lvert\mathbf{X}\rvert$	Number of experiments	Size of the set of flags to compile and measure
$f : \mathcal{X} \to \mathbb{R}$	Function to minimize	Performance metric, such as the execution time of a binary
$y_i = f(\mathbf{x}_i) + \varepsilon_i$	Observable quantity	Execution time with flags $\mathbf{x}_i$, with error ε_i
$\mathbf{y} = [y_1 \ \dots \ y_n]^\top \in \mathbb{R}^n$	Observations	List of execution times for all flags
$\mathbf{x}^* : y^* = f(\mathbf{x}^*) \leq f(\mathbf{x}), \forall \mathbf{x} \in \mathcal{X}$	Global optimum	Flag assignment with smallest execution time
Θ	Parameter space	All possible values of the coefficients of a linear model
$\theta(\mathbf{X}, \mathbf{y}) \in \Theta$	Parameter vector	OLS estimate of the coefficients of a linear model
$m = \lvert\theta\rvert$	Number of parameters	Number of OLS coefficient estimates
$\hat{f}_\theta : \mathcal{X} \to \mathbb{R}$	Surrogate for f	A linear model fit used for *predictions*
$\hat{y} = \hat{f}_\theta(\mathbf{x})$	Estimate of f at $\mathbf{x}$	A prediction of execution time for a flag assignment

Before moving on to the descriptions of derivative-based and stochastic function minimization methods and their application to autotuning problems, we discuss in the next section some of the properties of search spaces that are relevant for both autotuning and mathematical optimization.

3.2 Search Spaces

Consider a more abstract optimization problem than the compiler flag selection from the last section, consisting of finding the global optimum of the paraboloid surface defined by the

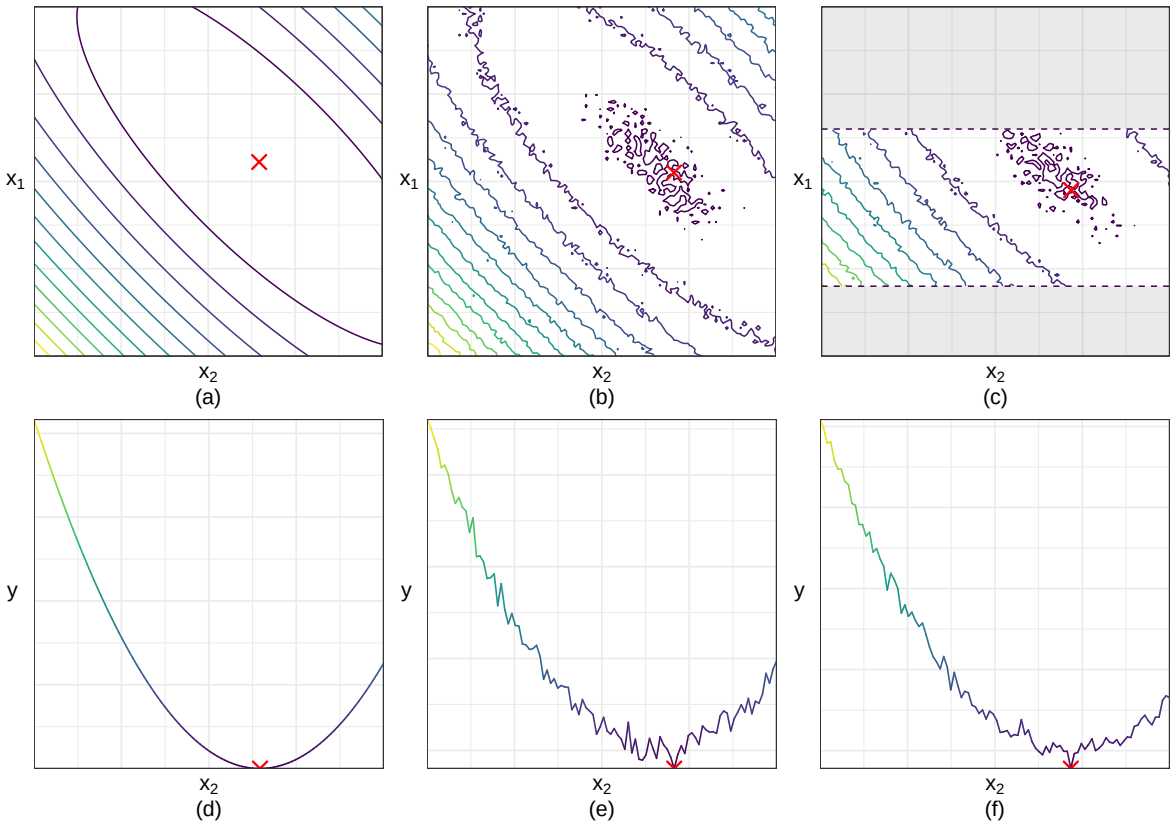

Figure 3.1: Contour plots and slices through the global optimum, marked with a ×, for search spaces defined by variations of the Booth function. Panels (a) and (d) correspond to Equation 3.3, panels (b) and (e) to Equation 3.4, and panels (c) and (f) to Equation 3.5.

Booth function,

$$y_a = f(\mathbf{x} = [x_1, x_2]^\top) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2, \;\; x_1, x_2 \in [-10, 10]. \tag{3.3}$$

In our notation, the search space for this example is $\mathcal{X} = (x_1, x_2) \in \mathbb{R}^2, \;\; x_1, x_2 \in [-10, 10]$, with global optimum $y^* = f(\mathbf{x}^* = [1, 3]) = 0$.

Besides the search space defined by the observations y_a, we will consider search spaces for two variations

$$y_b = f(\mathbf{x}) + \varepsilon, \;\; \text{with } \varepsilon \sim \mathcal{N}(0, \sigma^2), \text{ and} \tag{3.4}$$

$$y_c = f(\mathbf{x}) + \varepsilon, \;\; \text{with } \;\; x_1 \in [3, -6]. \tag{3.5}$$

The underlying objective function in this example has a closed-form expression, but in the context of our applications we consider that we can never observe the true $f(\mathbf{x})$, even in ideal experimental conditions. In that sense, the observations y_b closer to a real application, and incorporate the unknown effects and measurement errors to which the underlying objective function is subject, represented by the normally distributed random variable ε, with mean 0 and variance σ^2. There are often algorithmic, theoretical, or practical *constraints* on the allowed

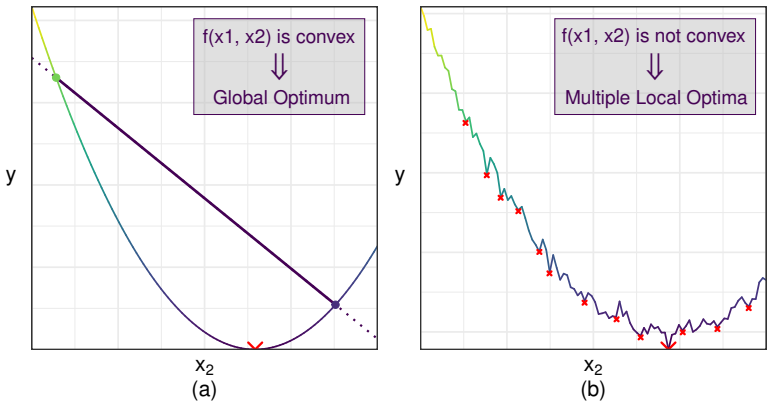

Figure 3.2: Illustrating the relationship between convexity of a function over a compact set and the presence of local minima. Panels (a) and (b) match the functions on the same panels of Figure 3.1

combinations of parameters of a given objective function. The observations y_c represent this scenario by incorporating constraints on the parameter x_1.

Panels (a), (b), and (c) of Figure 3.1 show contour plots for our search space variations in Equations 3.3, 3.4, and 3.5 respectively. The global optimum in each variation is represented by a red cross, and its location changes between scenarios because of ε. Panels (d), (e), and (f) show slices of panels (a), (b), and (c), respectively, that pass through the global optimum $\mathbf{x}^* = (x_1^*, x_2^*)$ for fixed $x_1 = x_1^*$.

The noise-free example shown in panel (a) has no local optima, and its smooth surface can be quickly navigated by the derivative-based methods discussed in Chapter 4. Such methods aim to follow the direction of *greatest descent* in the neighborhood of a given point. In a contour plot, this direction is always orthogonal to the contour lines. The panel (a) from Figure 3.2 illustrates the *convexity* of our noise-free function. Informally, a line segment connecting any two points in the graph of a convex function will not cross the graph of the function. Convexity of a function over a compact set implies the existence of a single global optimum, whereas lack of convexity implies the existence of *local optima*, as happens in our noisy functions, and is highlighted in panel (b) of Figure 3.2.

Local optima are by definition surrounded by higher values of the objective function, and can thus trap optimization methods that do not plan for such situations, such as a naive implementation of a derivative-based method. Adapting the step size of a derivative-based method is one way to deal with functions with many local optima, such as the ones on panels (b) and (c) of Figure 3.1.

Objective functions including constraints can present much harder problems to optimization methods, if certain conditions are met. For example, on panel (c) of Figure 3.1, we have constraints that cut contour lines in such a way that prevents attempts to move inside the feasible space in the direction of greatest descent. A derivative-based method would have

to drastically decrease its step size upon reaching the constraint border, and coast along the border in small steps until it finds a more appealing direction of descent.

This more abstract and simple example aimed to illustrate that it can be non-trivial to find the global optimum of the simplest of search spaces, if we consider the significant challenges introduced by unknown effects and measurement error. The additional challenges introduced by the time cost to obtain measurements, which are discussed in Chapter 6, guided the selection of the optimization methods studied in this thesis. The next chapter discusses derivative-based and stochastic methods for function minimization.

Chapter 4

Methods for Function Minimization

This chapter aims to present the intuition guiding the construction of derivative-based and stochastic methods for function minimization. We discuss the key hypotheses of these groups of methods, and for which autotuning problems they can be most effective.

We will put aside for the moment the idea of using observations to estimate a parameter vector and construct a surrogate function $\hat{f}(\mathbf{x}, \theta(\mathbf{X}, \mathbf{y}))$. This idea will return in later chapters. The methods discussed in this chapter do not construct a surrogate function, and thus attempt to directly optimize the objective function $f(\mathbf{x})$. In this sense, because they need to know how to evaluate it during optimization, these methods make the hypothesis that the objective function is known. The effectiveness of methods based on derivatives requires additional properties of $f(\mathbf{x})$ to be known or estimable, such as its first and second order derivatives, which imposes additional constraints on objective functions.

Evaluating derivatives to determine the next best step or using heuristics to explore a search space cannot be done parsimoniously, because a large number of function evaluations is required to estimate derivatives when closed-forms are unknown, and to explore a search space in the expectation of leveraging unknown structure. We will discuss Design of Experiments in Chapter 6, and present optimization strategies for situations where the cost of evaluating the objective function is prohibitive.

Methods based on derivatives can be powerful, provided their strong hypothesis are respected. We now briefly define and discuss these methods and their application to autotuning.

4.1 Methods Based on Derivatives

The derivatives of a function f at a point $\mathbf{x}$ provide information about the values of f in a neighborhood of $\mathbf{x}$. It is straightforward to construct optimization methods that use this local information, although iteratively leveraging it requires closed-form expressions for the derivatives of f, or estimates obtained by evaluating f at the neighborhoods of each point.

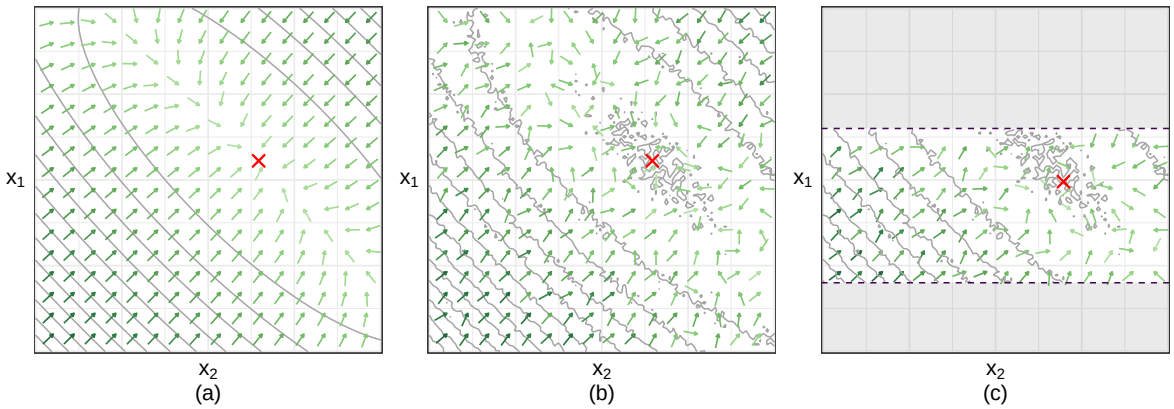

Figure 4.1: Contour plots and direction of greatest descent $-\nabla f(\mathbf{x})$, for search spaces defined by variations of the Booth function. Panels (a), (b), and (c) correspond to Equations 3.3, 3.4, and 3.5 respectively. The global optimum is marked with a ×. To aid visualization, vector magnitude was encoded by color intensity, so that darker vectors have larger magnitude. The gradient along the function's basin is near zero.

This section discusses gradient descent and Newton's method, optimization methods using first and second derivatives of f respectively, which for convex functions quickly converge to the global optimum. We will use examples with Booth's function to discuss how noise and uncertainty impact these methods.

4.1.1 Gradient Descent

The gradient $\nabla f(\mathbf{x})$ of a function $f : \mathcal{X} \to \mathbb{R}$ at point $\mathbf{x} = [x_1 \ \dots \ x_n]^\top$ is defined as

$$\nabla f(\mathbf{x} = [x_1 \ \dots \ x_n]^\top) = \left[f'_{x_1}(\mathbf{x}) \ \dots \ f'_{x_n}(\mathbf{x})\right]^\top, \tag{4.1}$$

where $f'_{x_i}(\mathbf{x})$ is the partial derivative of f at $\mathbf{x}$, with respect to variable x_i. The vector $\nabla f(\mathbf{x}_i)$ points to the direction of *greatest ascent* in which, from the perspective of $f(\mathbf{x}_i)$, the value of f increases the most.

The gradient descent method is one of the simplest ways to leverage derivative information for optimization. It consists in moving iteratively in the direction of *greatest descent*, opposite the gradient, from a starting point $\mathbf{x}_1$. If we follow the opposite of the gradient at $\mathbf{x}_1$ for additional points $\mathbf{x}_2, \dots, \mathbf{x}_n$, each iteration is written

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \alpha_k \nabla f(\mathbf{x}_{k-1}), \ k = 2, \dots, n, \tag{4.2}$$

where α_k is the step size at iteration k. The step size for each iteration can be a parameter fixed at the beginning of optimization, but the best α_k can alternatively be determined by searching along the direction of greatest descent.

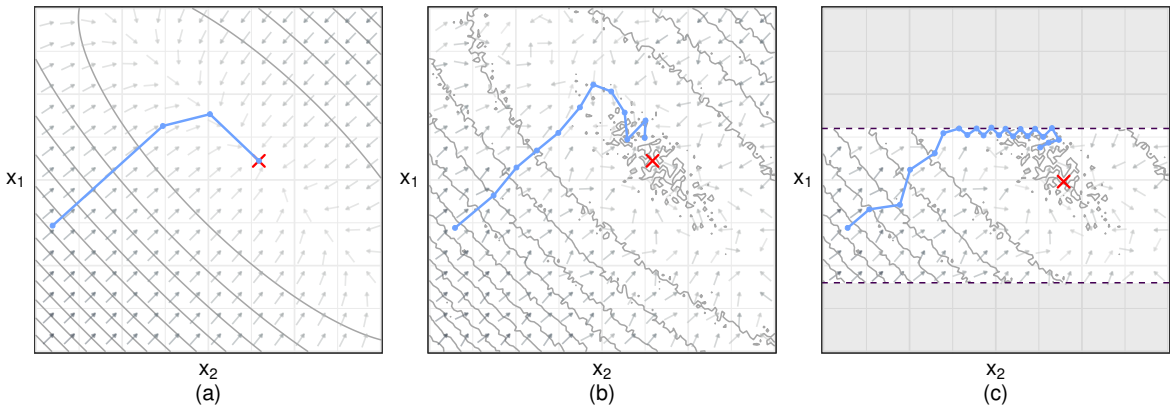

Figure 4.2: Representation of paths taken by the gradient descent method, with adaptive choice of α_k, on the search spaces defined by variations of the Booth function. Panels (a), (b), and (c) correspond to Equations 3.3, 3.4, and 3.5 respectively. Contour plots and direction of greatest descent $-\nabla f(\mathbf{x})$ are also shown, and the global optimum is marked with a ×.

Figure 4.1 shows the opposites of the gradients of the three variations of Booth's function, described by Equations 3.3, 3.4, and 3.5, in panels (a), (b), (c), respectively. Figure 4.2 shows, in equally marked panels, the paths taken by a gradient descent algorithm where α_k is chosen at each step, according to the values of f in the neighborhood x_{k-1}.

The gradient descent method iterates along the direction of greatest descent at each point and easily reaches the optimum on the search space of panel (a), unless we make an unlucky choice of α_k. The descent paths on panels (b) and (c) are not so straightforward since the several local minima, represented by the crests and loops on the contour lines, trap the descent path if α_k is not carefully chosen.

The situation is thornier in panel (c), where an unlucky choice of $\mathbf{x}_1$ or α_k throws the descent path against the top constraint border, forcing the method to zigzag along the border in short steps. This happens in this particular situation in panel (c) because all gradient information guides the descent across the constraint border, but the method cannot cross it. Gradient descent has a harder time on panels (b) and (c) even upon reaching the basin where the optimum lies, because gradient information there is also conflicting due to noise ε. Gradient descent gets stuck in our example paths, but restarting strategies picking new $\mathbf{x}_1$ or α_k could help escaping the local minima along the basin, as is shown in Figure 4.3.

4.1.2 Newton's Method

With Newton's method we can improve upon the intuition of descending along the opposite of the gradient of f by using the second partial derivatives $f''_{\mathbf{x}_{k-1}}$ to approximate f in the neighborhood of $\mathbf{x}_{k-1}$ using its second Taylor polynomial

$$f(\mathbf{x}) \approx f(\mathbf{x}_{k-1}) + \nabla f(\mathbf{x}_{k-1})^\top (\mathbf{x} - \mathbf{x}_{k-1}) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_{k-1})^\top \mathbf{H} f(\mathbf{x}_{k-1})(\mathbf{x} - \mathbf{x}_{k-1}), \tag{4.3}$$

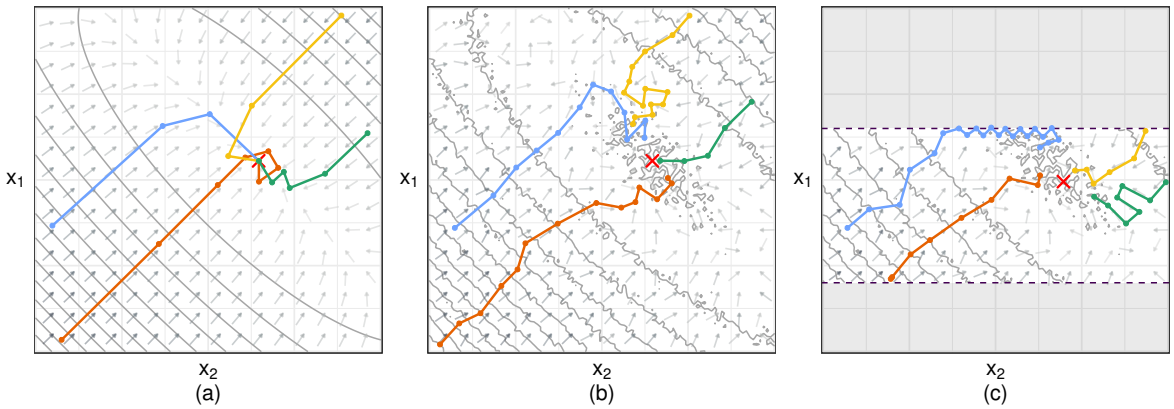

Figure 4.3: Representation of paths taken by the gradient descent method with 4 restarts, with adaptive choice of α_k, on the search spaces defined by variations of the Booth function. Panels (a), (b), and (c) correspond to Equations 3.3, 3.4, and 3.5 respectively. Contour plots and direction of greatest descent $-\nabla f(\mathbf{x})$ are also shown, and the global optimum is marked with a ×.

where $\mathbf{x}$ is in the neighborhood of $\mathbf{x}_{k-1}$, and $\mathbf{H}f(\mathbf{x}_{k-1})$ denotes the Hessian of f, a square matrix of second derivatives of f, with elements

$$(\mathbf{H}f(\mathbf{x}_{k-1}))_{i,j} = f''_{x_i,x_j}(\mathbf{x}_{k-1}). \tag{4.4}$$

We are not going to consider the approximation of f by the second Taylor polynomial to be an estimation process in the statistical sense, because it does not involve dealing with measurement or modeling error.

The second Taylor polynomial uses information about the partial derivatives of f at $\mathbf{x}_{k-1}$ to produce an approximation of f for points $\mathbf{x}$ around $\mathbf{x}_{k-1}$. If we compute the gradient of this approximation polynomial and set it to zero, we obtain the next point $\mathbf{x}_k$, as well as the iterative step of Newton's method. Starting at $\mathbf{x}_1$, for points $\mathbf{x}_2, \ldots, \mathbf{x}_n$, we have

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \mathbf{H}f(\mathbf{x}_{k-1})\nabla f(\mathbf{x}_{k-1}). \tag{4.5}$$

Provided the strong hypotheses of convexity and differentiability are respected, derivative-based methods are extremely effective. In particular, Newton's method converges to the global optimum in a single step if f is quadratic and $\mathbf{H}f$ is positive definite. This happens in the following example with Booth's function

$$y = f(\mathbf{x} = [x_1, x_2]^\top) = (x + 2y - 7)^2 + (2x + y - 5)^2, \; x_1, x_2 \in [-10, 10], \tag{4.6}$$

adapted from Kochenderfer and Wheeler [61].

If we start with $\mathbf{x}_1 = [9, 8]^\top$ and plug

$$\nabla f(\mathbf{x}_1 = [9, 8]^\top) = [10 \cdot 9 + 8 \cdot 8 - 34, 8 \cdot 9 + 10 \cdot 8 - 38]^\top = [120, 144]^\top, \tag{4.7}$$

and

$$\mathbf{H} f(\mathbf{x}_1 = [9, 8]^\top) = \begin{bmatrix} 10 & 8 \\ 8 & 10 \end{bmatrix} \tag{4.8}$$

into the Newton's method update step in Equation 4.5, we reach $\mathbf{x}^* = [1, 3]^\top$ in the next step

$$\mathbf{x}_2 = \begin{bmatrix} 9 \\ 8 \end{bmatrix} - \begin{bmatrix} 10 & 8 \\ 8 & 10 \end{bmatrix}^{-1} \begin{bmatrix} 120 \\ 144 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix} = \mathbf{x}^*. \tag{4.9}$$

Function minimization methods work very well if the objective function respects their strong hypotheses of convexity and differentiability. Unfortunately, autotuning problems seldom fulfill the conditions necessary for the application of such methods.

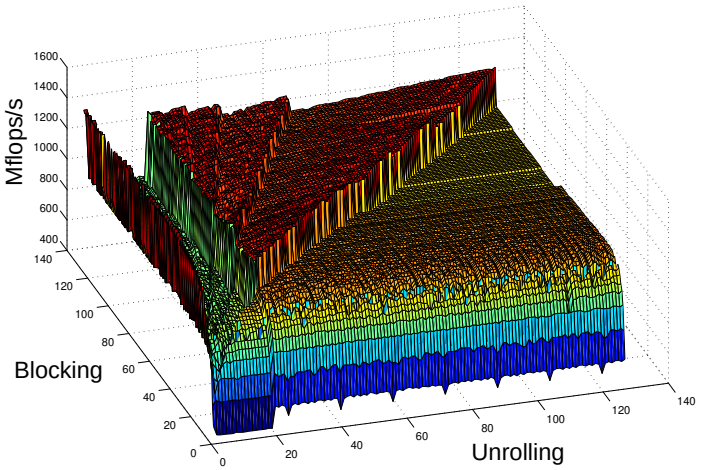

Figure 4.4: An exhaustively measured search space, defined by loop blocking and unrolling parameters, for a sequential GEMM kernel. Reproduced from Seymour *et al.* [5]

Typical autotuning search spaces, such as the one we encountered in Chapter 2, which is reproduced in Figure 4.4, clearly present considerable challenges to derivative-based methods, due to the abundance of local minima, valleys, and ridges. Since we have closed-form expression for the search space in Figure 4.4, we would have to perform a considerable number of evaluations of f in order to estimate its derivative at each step. Although gradient descent and other derivative-based methods can be effective and are historically important, measuring f is usually not cheap for autotuning search spaces. Therefore, minimizing experimentation cost is also a strong concern.

Before discussing methods to construct surrogate models in Chapter 5, and how to use such surrogates to minimize experimental cost on Chapter 6, we will relax the convexity

and differentiability requirements on objective functions and discuss stochastic methods for function minimization, and their applicability to autotuning.

4.2 Stochastic Methods

In this section we discuss some stochastic methods for function minimization that drop the convexity and differentiability requirements of gradient-based methods, becoming applicable to a wider range of autotuning problems at the cost of providing no convergence guarantees. In fact, these methods have no clearly stated hypotheses and are based on search space exploration heuristics. Often the best possible understanding of how these heuristics work comes from the intuition and motivation behind their definition, and from the analysis of empirical tests.

There are multiple ways to categorize heuristics. Our choice was to make a distinction between single-state and population-based methods. In summary, single-state methods have update rules mapping a single point $\mathbf{x}_k$ to a point $\mathbf{x}_{k+1}$, while rules for population-based methods map a population $\mathbf{P}_k = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ to a population $\mathbf{P}_{k+1}$, which may retain, combine, and modify elements from $\mathbf{P}$.

We will first discuss single-state methods, building up to Simulated Annealing from Random Walk, then we discuss Genetic Algorithms and Particle Swarm Optimization, representing widely used population-based methods.

4.2.1 Single-State Methods: Random Walk and Simulated Annealing

Random Sampling is arguably the simplest exploration heuristic, consisting of picking uncorrelated samples from a uniform distribution over the search space $\mathcal{X}$. There is no guarantee of finding local or global minima, but the chances of improving over a starting point increase if the objective function has many local minima. Despite its simplicity, Random Sampling can be surprisingly effective.

A simple way to derive a single-state heuristic from Random Sampling is to take correlated samples, so that each sample lies in a neighborhood of its predecessor, which is called Random Walk. The neighborhood $N(\mathbf{x})$ of a point is the set of points within distance d from $\mathbf{x}$. One way to define it is

$$N(\mathbf{x}) = \{\mathbf{x}_i \in \mathcal{X} : \mathbf{x}_i \neq \mathbf{x},\ \|\mathbf{x}_i - \mathbf{x}\|^2 \leq d\}. \tag{4.10}$$

As long as it is possible to compute distances between the elements of $\mathcal{X}$, we will be able to construct the neighborhood of a point and employ the stochastic methods we discuss in this section.

A random walk of length n starting at $\mathbf{x}_1$ produces a sequence where each point $\mathbf{x}_{k>1}$ is a random variable with uniform distribution over $N(\mathbf{x}_{k-1})$. There is no guarantee that we will ever find a better point with respect to the objective function than the one we started with. A straightforward extension of Random Walk is to pick at each step $k > 1$ the first point $\mathbf{x}_k$ we come across in $N(\mathbf{x}_{k-1})$ for which $f(\mathbf{x}_k) < f(\mathbf{x}_{k-1})$. This greedy strategy would require measuring the value of f for possibly many elements of $N(\mathbf{x}_{k-1})$ but it ensures that we will only move toward a better point.

If are willing to pay the cost to measure all the points in $N(\mathbf{x}_{k-1})$ we can choose the $\mathbf{x}_k^*$ that brings the best improvement, for which $f(\mathbf{x}_k^*) < f(\mathbf{x})$ for all $\mathbf{x} \in N(\mathbf{x}_{k-1})$. This best improvement strategy always moves to the best point in a neighborhood, but it can still get stuck in lock minima if the current point is already the best one in its neighborhood. Adapting the distance that defines a neighborhood can help escape local minima, but the best improvement strategy still requires measuring the entire neighborhood of a point.

Simulated Annealing is a probabilistic improvement heuristic inspired by the process of annealing, where temperature is carefully controlled to first agitate a material's crystalline structure with higher temperature, and then settle it into more desirable configurations. Adapting this idea to optimization, Simulated Annealing makes a compromise between a greedy approach to exploration and a random walk. At each step k, we pick a random uniformly distributed $\mathbf{x}_k \in N(\mathbf{x}_{k-1})$ and move to it if $f(\mathbf{x}_k) < f(\mathbf{x}_{k-1})$. In contrast to a greedy approach, we also move if $f(\mathbf{x}_k) \geq f(\mathbf{x}_{k-1})$ with probability p.

In analogy to the real annealing process, p starts high to enable exploration of the search space and then decreases to force a descent towards an optimum. All through the process, a nonzero value of p permits the heuristic to leave a local minimum. The probability p_k of moving to a worse point at iteration k is

$$p_k = exp\left(-\frac{f(\mathbf{x}_k) - f(\mathbf{x}_{k-1})}{t_k}\right), \tag{4.11}$$

where t_k is the temperature at iteration k, and can follow different decaying rates. For a starting temperature t_1 the logarithmic annealing schedule is

$$t_k = \frac{t_1}{log(k+1)}. \tag{4.12}$$

Figure 4.5 shows the paths taken by Simulated Annealing on search spaces defined by variations of Booth's function. In panel (a) we can see the effects of the annealing schedule, which enables large detours to worse points early on, but forces descent on later iterations. Since this search space has no local minima, the descent eventually reaches the optimum. In panel (b) the higher initial temperature also allows escaping the many local minima found during exploration, but as the path approaches the global optimum with the lower temperature of later iterations it gets trapped by one the local minima around the global optimum.

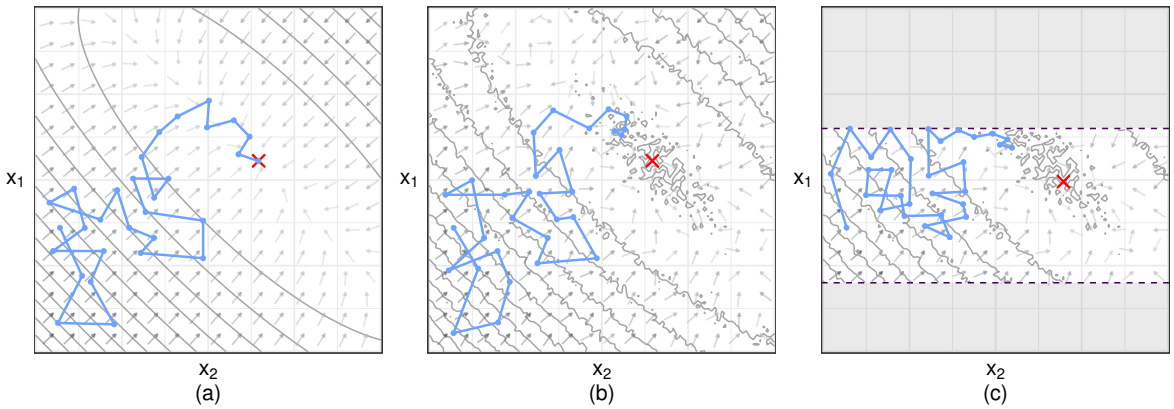

Figure 4.5: Representation of paths taken by the Simulated Annealing method on the search spaces defined by variations of the Booth function. Panels (a), (b), and (c) correspond to Equations 3.3, 3.4, and 3.5 respectively. Contour plots and direction of greatest descent are also shown, and the global optimum is marked with a ×.

Likewise, encountering a constraint border early on in panel (c) is not a challenge for Simulated Annealing, since it can bounce back toward worse points, but on later iterations the method is forced to wander around the border like gradient descent, also getting trapped at local minima.

Single-state methods for optimization provide heuristics for exploring a search space, usually based on the neighborhoods of the points that compose a path. The cost of measuring the objective function f for an entire neighborhood is prohibitive in large dimensional search spaces, but methods such as Simulated Annealing provide strategies for escaping local minima without completely evaluating a neighborhood. Despite that, the local nature of single-state methods means that the final results are heavily reliant on the starting point. Restarting and performing parallel searches are among the strategies to reduce dependence on initial choices, as is the idea of using a population of points, which we discuss next.

4.2.2 Population-Based Methods: Genetic Algorithms and Particle Swarm Optimization

Instead of progressing from a single starting point $\mathbf{x}_1$ toward a final state $\mathbf{x}_n$, we can use heuristics for moving a starting set of points, or population, $\mathbf{P}_1 = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ toward a final population $\mathbf{P}_n$. For simplicity, we keep the population size constant. The inspiration for population-based methods for function minimization comes in general from observations of processes such as evolution by natural selection and animal group behavior.

The intuition behind population-based methods is that the points in a population would provide variability that, when combined in specific ways, would eventually lead to better values of the objective function f. Genetic Algorithms, in analogy to the process of evolution by natural selection, select the fittest points in a population for generating offspring.

Figure 4.6: Some ways of producing offspring from two parents with binary chromosomes. Crossover splits parent chromosomes and combine the resulting pieces. In general, pieces from multiple splits can be combined. Mutations are introduced randomly and correspond to flipping bits on binary chromosomes.

Individuals can be selected according to multiple metrics, aiming to produce the best possible combinations in an iteration but also to maintain population variability. Since we do not know how mutations and chromosome combinations might impact the fitness of an individual, keeping worse individuals during optimization could pay off later.

The new population is generated by combining the chromosomes of each parent, using strategies that are also inspired by natural processes such as mutation and crossover. To be able to perform these operations, chromosomes must be encoded in a suitable representation. Individuals in the search space defined by compiler flags, for example, a binary array indicating whether each flag is used could be a suitable representation. Figure 4.6 shows what the mutation and crossover operations could look like on a binary encoding.

Genetic Algorithms have the potential to explore a search space more globally, simultaneously maintaining several populations distributed over a search space, and have also the potential of escaping local minima by mutation and combination of chromosomes of different individuals. Figure 4.7 shows a representation of the paths a population in a Genetic Algorithm could take while searching for the global minimum on three variations of Booth's function. Each region marked by dashed lines represents the spread of a generation on a given optimization step, and previous individuals are marked by hollow points. The final generation is marked by the filled points. In contrast to Gradient Descent and Simulated Annealing, a Genetic Algorithm do not seek to measure and minimize local properties of f, and consequently its behavior would be less impacted by the noisy scenarios on panels (b) and (c). Since the population can spread across the search space, it could still be possible to end the process with individuals in different local minima, which is represented in panel (b).

Differential Evolution presents an alternative to the strategies of mutations and crossover, represented in Figure 4.6. Consider a population $\mathbf{P}_{k-1}$, and individuals $\mathbf{x}_a, \mathbf{x}_b, \mathbf{x}_c$ uniformly distributed over the population at step $k-1$. The update step in Differential Evolution sets the parameters, or components, of an offspring $\mathbf{x}_k$ to a corresponding component of $\mathbf{x}_a$ or $\mathbf{x}_d = w \cdot (\mathbf{x}_b - \mathbf{x}_c)$, according to

$$\mathbf{x}_{k,i} = \begin{cases} \mathbf{x}_{d,i} & \text{if } i = j, \text{ or with probability } p \\ \mathbf{x}_{a,i} & \text{otherwise} \end{cases}, \tag{4.13}$$

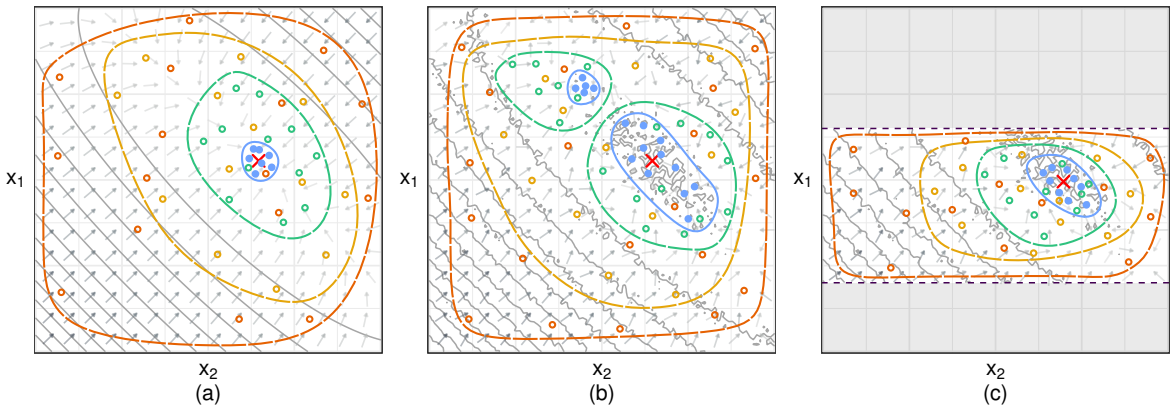

Figure 4.7: Color-coded generation snapshots of a Genetic Algorithm on the search spaces defined by variations of the Booth function. Hollow points and dashed lines mark members of previous populations and the regions they covered, while filled points and complete lines mark the final population. Panels (a), (b), and (c) correspond to Equations 3.3, 3.4, and 3.5 respectively. Contour plots and direction of greatest descent are also shown, and the global optimum is marked with a ×.

where j is a dimension favored for updating picked at random, and w is a weight given to the difference between $\mathbf{x}_b$ and $\mathbf{x}_c$. The offspring $\mathbf{x}_k$ replaces $\mathbf{x}_a$ in the population at step k if $f(\mathbf{x}_k) < f(\mathbf{x}_a)$.

A different approach to leveraging a population for optimization is to think of it as an analogy for a swarm, in a method called Particle Swarm Optimization. Each individual $\mathbf{x}_k$ in the swarm keeps track of its velocity $\mathbf{v}_k$. The velocity is updated at each step, and points to a combination of $\mathbf{x}^{best}$, the best position found by the swarm so far, and $\mathbf{x}_k^{best}$, the individual's personal best. At each step, individual $\mathbf{x}_k$ updates its position to

$$\mathbf{x}'_k = \mathbf{x}_k + \mathbf{v}_k, \tag{4.14}$$

and its velocity $\mathbf{v}_k$ according to

$$\mathbf{v}'_k = \alpha_1 \mathbf{v}_k + \alpha_2(\mathbf{x}^{best} - \mathbf{x}_k) + \alpha_3(\mathbf{x}_k^{best} - \mathbf{x}_k), \tag{4.15}$$

where α_1, α_2, and α_3 are chosen beforehand. The intuition behind this method is that the momentum of each particle toward the best points found so far would accelerate convergence and allow escaping from local minima.

In whichever way we choose to combine individuals in population-based methods, these heuristics require extensive exploration and, consequently, abundant evaluation of the objective function. By making virtually no hypotheses these methods inspired by natural processes become applicable to a much broader range of search spaces than derivative-based methods, while abdicating from convergence guarantees. Population-based methods perform a more global optimization, reliant on the initial distribution of individuals over the search space to provide variability.

4.3 Summary

The methods for function minimization we have discussed in this chapter do not strive to be parsimonious. To be effective they require many estimates of values of f and may require additional information, such as ∇f and $\mathbf{H}f$. The strong hypotheses of derivative-based methods restrict their applicability by invalidating convergence guarantees. Stochastic methods have no such guarantees to begin with, and thus require costly exploration.

Figure 4.8 shows representations of optimization trajectories for Booth's function made by Gradient Descent with restarts, on panels (a), (b), and (c), Simulated Annealing, on panels (d), (e), and (f), and a Genetic Algorithm, on panels (g), (h), and (i). Restarting optimization from a different point, seem on panels (a), (b), and (c), is a common and widely used technique to reduce reliance on starting conditions.

Despite the high cost of exploration and strong hypotheses, methods for function minimization are used for autotuning and can indeed achieve interesting results in certain problems. We will present our results with these methods on Chapters 10 and 11, where we also review their application to autotuning problems in different domains. The explorations performed by these methods are not structured in a way that favors statistical analysis. We will postpone the discussion of how to obtain well-structured experimental data until Chapter 6, and in the following chapter we will discuss learning methods that enable building surrogate models $\hat{f}$ and identifying relationships between parameters $\mathbf{X}$ in the search space $\mathcal{X}$ and observations of the objective function f.

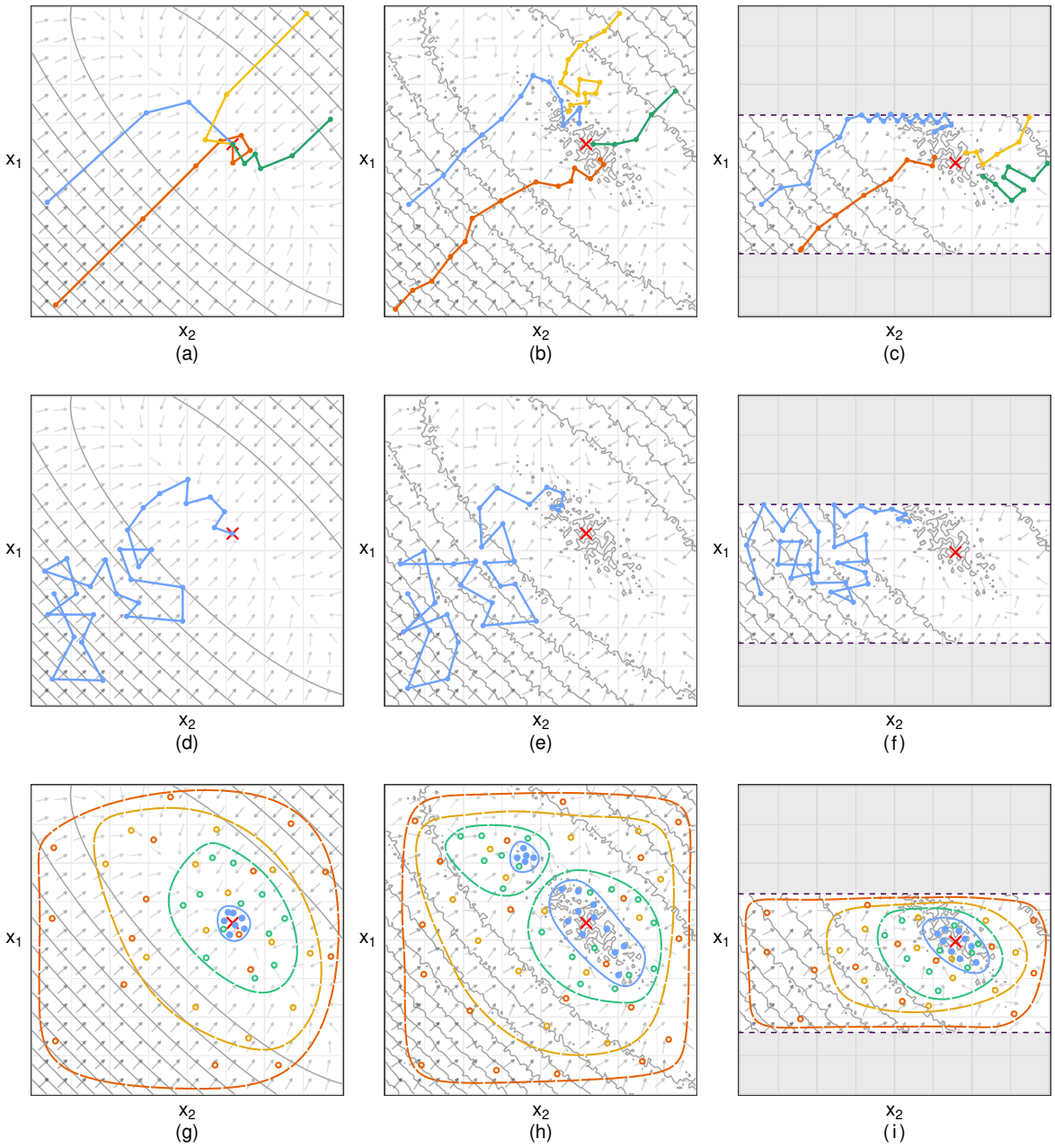

Figure 4.8: Representation of paths taken by the gradient descent method, with adaptive choice of α_k, on the search spaces defined by variations of the Booth function. Panel groups (a,d,g), (b,e,h), and (c,f,i) correspond to Equations 3.3, 3.4, and 3.5 respectively. Panel groups (a,b,c), (d,e,f), and (g,h,i) correspond to Gradient Descent with restarts, Simulated Annealing, and a Genetic Algorithm, respectively. Contour plots and direction of greatest descent $-\nabla f(\mathbf{x})$ are also shown, and the global optimum is marked with a ×.

Chapter 5

Learning: Building Surrogates

This chapter discusses two methods for building surrogate models. Linear Regression is a parametric method capable of modeling a restricted class of surrogates, for which it is relatively simple to interpret significance. Gaussian Process Regression is a flexible nonparametric method capable of modeling a large class of surrogates, for which interpreting significance is possible, but costly.

The surrogate $\hat{f}_\theta : \mathcal{X} \to \mathbb{R}$ depends on the parameter vector $\theta(\mathbf{X}, \mathbf{y})$ in the parameter space Θ constructed using the pair $(\mathbf{X}, \mathbf{y})$, where $\mathbf{X}$ is a set of experiments, and $\mathbf{y}$ the corresponding vector of observations of the objective function f. The process of using the available data to construct a surrogate model is called *learning* and involves fitting the surrogate model, which can also be called training. The class of surrogates we can fit depends on each method's definition of Θ.

In this chapter we assume that an initial pair of $(\mathbf{X}, \mathbf{y})$ is given, and that we can no longer observe $f(\mathbf{x})$ for new $\mathbf{x}$ outside of $\mathbf{X}$. In this sense, we do not know f but we can still make hypotheses about it in order to construct surrogate models. For the two methods we discuss, we present the hypotheses embedded in each parameter space Θ and how each method fits a surrogate. We also discuss strategies to evaluate the quality of fit of a surrogate, and to interpret parameter significance.

We assume for now that $\mathbf{X} \sim \text{Uniform}(\mathcal{X})$, and we discuss how we can construct better distributions of experiments over a search space on Chapter 6. We will now discuss Linear Regression, a method to build surrogates using linear models on the parameters θ.

5.1 Linear Regression

We will build a surrogate $\hat{f}_\theta$ for a function $f : \mathcal{X} \to \mathbb{R}$ using a fixed experimental design $\mathbf{X}$ and observations $\mathbf{y}$. We make the hypothesis that f is a linear model on θ, with error ε,

written

$$f(\mathbf{X}) = \mathbf{X}\theta(\mathbf{X}, \mathbf{y}) + \varepsilon. \tag{5.1}$$

The experimental design $\mathbf{X}$ is an $n \times p$ matrix of column vectors, where each element $\mathbf{x}_1, \ldots, \mathbf{x}_n$ is an experiment. We do not know how well the linear model hypothesis represents the true f, but we assume that each experiment in $\mathbf{X}$ was run, producing the response vector $\mathbf{y} = [y_1 \ \ldots \ y_n]^\top$, where each element is an observation of f subject to measurement error.

Using $\mathbf{y}$ we can construct parameter vectors θ in the parameter space

$$\Theta = \{(\theta_0, \ldots, \theta_{m-1}) : \theta_0, \ldots, \theta_{m-1} \in \mathbb{R}\}. \tag{5.2}$$

For linear models we can construct the optimal parameter vector $\hat{\theta}$ using the Ordinary Least Squares (OLS) estimator, which we discuss in the next section, and use it to write the linear model surrogate

$$\hat{f}_\theta(\mathbf{X}) = \mathbf{X}\hat{\theta}(\mathbf{X}, \mathbf{y}). \tag{5.3}$$

The vector $\hat{\mathbf{y}}$ produced by evaluating $\hat{f}_\theta(\mathbf{X})$ is called the prediction of the surrogate model for experiments $\mathbf{X}$, and the distance between $\mathbf{y}$ and $\hat{\mathbf{y}}$ can be used to optimize the parameter vector and estimate the quality of fit of the surrogate.

A key advantage of a surrogate model is that we can use $\hat{f}_\theta$ to estimate the value of f for a new design $\mathbf{X}'$ without evaluating $f(\mathbf{X}')$. For well chosen $\mathbf{X}$ and well fitted θ, these estimates can be accurate and useful. We will discuss how to choose experiments on Chapter 6, and how to evaluate the quality of fit for linear models on Section 5.1.2.

5.1.1 Fitting the Model: The Ordinary Least Squares Estimator

For simplicity, we will for now use θ to refer to $\theta(\mathbf{X}, \mathbf{y})$. The sum of the squared differences between f and $\hat{f}_\theta$ for all points in $\mathbf{X}$ is the squared model error

$$\|\varepsilon\|^2 = \|\mathbf{y} - \hat{\mathbf{y}}\|^2 = (\mathbf{y} - \mathbf{X}\theta)^\top(\mathbf{y} - \mathbf{X}\theta), \tag{5.4}$$

which is a quadratic function of θ. We can therefore differentiate it with respect to θ and set it to zero to obtain the OLS estimator

$$\hat{\theta} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}, \tag{5.5}$$

provided $\mathbf{X}^\top\mathbf{X}$ is invertible. The variance of the OLS estimator is written

$$\mathrm{Var}(\hat{\theta}) = (\mathbf{X}^\top\mathbf{X})^{-1}\sigma^2, \tag{5.6}$$

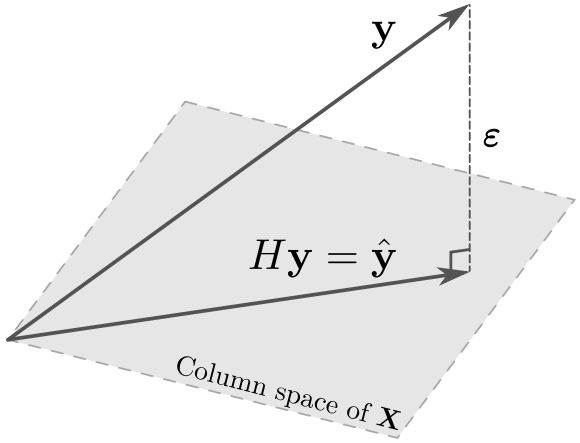

Figure 5.1: In Linear Regression, the prediction vector $\hat{\mathbf{y}}$ is the orthogonal projection of the observations vector $\mathbf{y}$ into the vector space spanned by the columns of $\mathbf{X}$.

assuming the observations $\mathbf{y}$ are uncorrelated and homoscedastic, with constant variance σ^2. Since we assume in this chapter that $\mathbf{X}$ is fixed, the variance of $\hat{\theta}$ shows explicitly how uncertainty in measurements due to error propagates to the linear model surrogate, and we can use this uncertainty to compute confidence intervals for the surrogate's predictions. We will use the fact that the variance of $\hat{\theta}$ depends on $(\mathbf{X}^\top\mathbf{X})^{-1}$ when we discuss Optimal Design on Section 6.3.

An interesting interpretation of the OLS estimator is that it approximates the observation vector $\mathbf{y}$ by its orthogonal projection into the vector space spanned by the columns of $\mathbf{X}$. Substituting $\hat{\theta}$ on the surrogate model 5.3, we obtain the projection matrix

$$H = \mathbf{X}(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top. \tag{5.7}$$

The difference between $\mathbf{y}$ and its projection $H\mathbf{y} = \hat{\mathbf{y}}$ is the model error ε, orthogonal to the column space of $\mathbf{X}$, as represented in Figure 5.1. The OLS estimator can also be derived as a special case of the Maximum Likelihood Estimator with $\varepsilon \sim \mathcal{N}(\mu, \sigma^2)$.

We assumed that the design matrix $\mathbf{X}$ was used directly in the surrogate model 5.3, but this limits the models we can represent with $\theta_{0,\dots,m-1}$ to those with $m = p$ linear terms on each factor $\mathbf{x}_{1,\dots,p}$, without an intercept term. More generally, we can have a linear model surrogate with $m \neq p$ terms. In a straightforward expansion of the linear model, we can obtain m model terms from an $n \times p$ design matrix $\mathbf{X}$ by using a basis function set $\mathcal{H} = \{h_{0,\dots,m-1} : \mathcal{X} \to \mathbb{R}\}$, generating the $n \times m$ model matrix

$$\mathcal{M}\left(\mathbf{X} = \begin{bmatrix}\mathbf{x}_1 \\ \vdots \\ \mathbf{x}_n\end{bmatrix}_{n\times p}\right) = \begin{bmatrix} h_0(\mathbf{x_1}) & \dots & h_{m-1}(\mathbf{x_1}) \\ \vdots & \ddots & \vdots \\ h_0(\mathbf{x_n}) & \dots & h_{m-1}(\mathbf{x_n}) \end{bmatrix}_{n\times m}. \tag{5.8}$$

The OLS estimator does not change if we write the linear model surrogate in Equation 5.3 with $\mathcal{M}(\mathbf{X})$. Unless we need to discuss the underlying model terms, we assume from now on that $\mathbf{X}$ represents a suitable model matrix $\mathcal{M}(\mathbf{X})$. We will now discuss how to evaluate the quality of fit of a linear model surrogate.

5.1.2 Assessing the Quality of Fit of Linear Model Surrogates

We will construct three linear model surrogates for the Booth function and use the model fits to discuss the assessment of model quality. Figure 5.2 shows the three surrogate model fits, constructed using the same design $\mathbf{X}$ with $n = 10$ random uniformly distributed measurements of the Booth function

$$f(\mathbf{x} = [x_1, x_2]^\top) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2,\ x_1, x_2 \in [-10, 10], \tag{5.9}$$

subject to measurement error ε. Panel (a) shows the surface produced by measuring f in its entire domain. The 10 random measurements composing $\mathbf{X}$ are highlighted. Panels (b), (c), and (d) show the three surrogates, which used the basis function sets

$$\mathcal{H}_{(b)} = \{\ h_0(\mathbf{x}) = 1,\ h_1(\mathbf{x}) = x_1,\ h_2(\mathbf{x}) = x_2)\ \}, \tag{5.10}$$

$$\mathcal{H}_{(c)} = h_{(b)} \cup \{\ h_3(\mathbf{x}) = x_1^2,\ h_4(\mathbf{x}) = x_2^2)\ \}, \text{and} \tag{5.11}$$

$$\mathcal{H}_{(d)} = h_{(c)} \cup \{\ h_5(\mathbf{x}) = x_1 x_2)\ \}. \tag{5.12}$$

Panels (b), (c), and (d) also show the training, testing, and true Mean Squared Error (MSE) for each surrogate on the design $\mathbf{X}$, written

$$MSE_{\hat{f}_\theta(\mathbf{X})} = \frac{1}{n}\sum_{i=1}^{n}\left((f(\mathbf{x}_i) + \varepsilon) - \hat{f}_\theta(\mathbf{x}_i)\right)^2, \tag{5.13}$$

$$MSE_{\hat{f}_\theta} = \frac{1}{|\mathcal{X}|}\sum_{i=1}^{|\mathcal{X}|}\left((f(\mathbf{x}_i) + \varepsilon) - \hat{f}_\theta(\mathbf{x}_i)\right)^2, \text{and} \tag{5.14}$$

$$MSE_f = \frac{1}{|\mathcal{X}|}\sum_{i=1}^{|\mathcal{X}|}\left(f(\mathbf{x}_i) - \hat{f}_\theta(\mathbf{x}_i)\right)^2. \tag{5.15}$$

The training error $MSE_{\hat{f}_\theta(\mathbf{X})}$ can be computed with the design points and the fitted model. In real applications we can sometimes compute the testing error $MSE_{\hat{f}_\theta}$ for the entire search space, but usually we settle for the error on a testing set distinct from $\mathbf{X}$. We can almost never compute the true error MSE_f of our surrogate, but in our toy example we can use the Booth function without the error term to get a sense of how well our models generalize. As we could expect, since the true f in our example can be represented by the basis functions on set $\mathcal{H}_{(d)}$,

the surrogate from panel (c) has the smallest MSE and the generalizes the best, although it still differs from the true f.

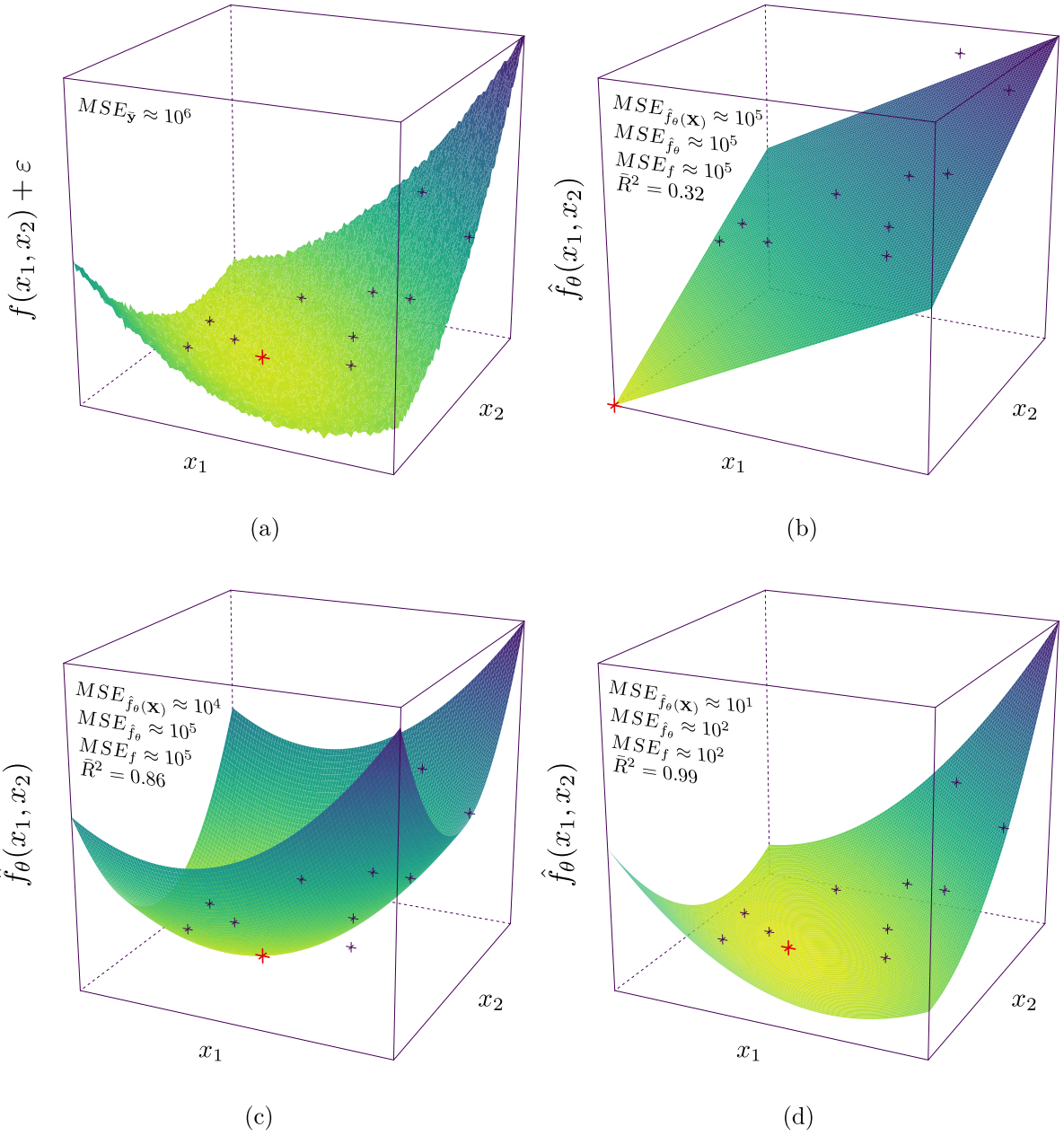

Figure 5.2: Three linear model surrogates for the Booth function, with a × marking the global optimum and best surrogate predictions. The fixed experimental design **X** used to fit all surrogates is marked by × s. Panel (a) shows noisy measurements of Booth's function, panels (b), (c), and (d) show surrogate predictions for models fit with basis functions sets from Equations 5.10, 5.11, and 5.12, respectively

We can use the mean $\bar{\mathbf{y}}$ of the observations of experiments in $\mathbf{X}$ as a surrogate, for which the MSE is written

$$MSE_{\bar{\mathbf{y}}} = \frac{1}{n}\sum_{i=1}^{n}((f(\mathbf{x}_i) + \varepsilon) - \bar{\mathbf{y}})^2, \tag{5.16}$$

shown in panel (a) from Figure 5.2. We can use this surrogate to compute the adjusted coefficient of determination

$$\bar{R}^2 = 1 - \left(\frac{n-1}{n-p-1} \cdot \frac{MSE_{\hat{f}_\theta(\mathbf{X})}}{MSE_{\bar{\mathbf{y}}}}\right), \tag{5.17}$$

which compares the squared error of a given linear model with the error of the model that predicts the mean of all observations, adjusted by the flexibility introduced by new parameters.

In general, we can measure the complexity of a surrogate model by the number of parameters that we need to estimate when fitting it to experimental data. The bias of a surrogate, for a fixed point $\mathbf{x}_0$, is the expected value of the distance between the surrogate's prediction and $f(\mathbf{x}_0)$, over surrogate fits using a large number of different experimental designs. The squared bias is written

$$\text{Bias}^2\left(\hat{f}_\theta(\mathbf{x}_0)\right) = \left(E\left[\hat{f}_\theta(\mathbf{x}_0)\right] - f(\mathbf{x}_0)\right)^2, \tag{5.18}$$

because $E[f(\mathbf{x}_0)] = f(\mathbf{x}_0)$. As we increase the flexibility of a model by adding parameters we allow the surrogate model to better approximate design points, reducing its bias. Concomitantly, we increase the number of different parameter vectors that can describe the experimental data, increasing the surrogate's variance

$$\text{Var}\left(\hat{f}_\theta(\mathbf{x}_0)\right) = E\left[\hat{f}_\theta(\mathbf{x}_0) - E\left[\hat{f}_\theta(\mathbf{x}_0)\right]\right]^2. \tag{5.19}$$

The total model error for the prediction of $\mathbf{x}_0$ still has to factor in the irreducible error ε associated with the measurements of f, and is written

$$\text{Error}\,(\mathbf{x}_0) = \text{Var}(\varepsilon) + \text{Bias}^2\left(\hat{f}_\theta(\mathbf{x}_0)\right) + \text{Var}\left(\hat{f}_\theta(\mathbf{x}_0)\right). \tag{5.20}$$

Increasing surrogate complexity reduces bias but increases variance, and this trade-off is central to selecting and assessing the quality of surrogate models, however we define the parameter space Θ.

5.1.3 Inference: Interpreting Significance with ANOVA

The variance of the OLS estimator enables us to compute confidence intervals and p-values for the effects of each model term, or factor, for a single surrogate model fit. In a frequentist interpretation a 95% confidence interval for the estimate of a mean of a factor's effect is

interpreted as the interval that would contain 95% of our estimates, were we to repeat the estimation multiple times. The p-value of a factor effect's estimate is interpreted in frequentist inference as the probability of observing an effect at least as large as what was observed, if the factor's true effect is zero. When they can be computed, confidence intervals are in general more useful than p-values for judging the accuracy of a factor effect's estimate, because they are explicitly defined in the context of the magnitude of that estimate.

Analysis of Variance (ANOVA) is a more refined statistical tool for significance testing, able to estimate relative factor significance. The steps of an ANOVA test are grouping the observations $\mathbf{y}$ by factors and factor levels, computing separate group means, and testing the significance of the differences between group means with an F-test.

The ANOVA test can be understood as a special case of the linear model we have discussed in this section, in which case its formal hypotheses are the same as the linear model's, that is, that the observations $\mathbf{y}$ are uncorrelated, the residuals are normally distributed, and the variances of each group are homoscedastic.

Running an ANOVA test as a special case of the linear model consists of running F-tests for multiple models $\mathbf{y} = \mathbf{X}_i\theta$, with specially constructed model matrices of indicator variables for group and group interaction membership. A detailed description of ANOVA in relation to linear models can be found Chapter 6 of Dobson *et al.* [62], among other reference texts [63, 64].

5.1.4 Linear Models: Interpretable but Biased Surrogates

Linear regression can be successful in learning and interpreting relationships between factors and an objective function f. If the underlying functions have complex structure that cannot be sufficiently well represented by a finite number of parameters, linear models might not be useful beyond identifying the strongest factor effects. The following section will discuss Gaussian Process Regression, a nonparametric approach that fits a model by conditioning a probability distribution over functions.

5.2 Gaussian Process Regression

Similarly to the surrogate model we built using linear models, in this section we will fit a surrogate $\hat{f}_\theta : \mathcal{X} \to \mathbb{R}$ to a fixed experimental design $\mathbf{X}$ and observations $\mathbf{y}$. We make the hypothesis that the observations $y_1, \ldots, y_n$ are normally distributed. This is a reasonable hypothesis to make, especially after our discussion on linear models, where our hypothesis over f was

$$f(\mathbf{X}) = \mathbf{X}\theta + \varepsilon, \tag{5.21}$$

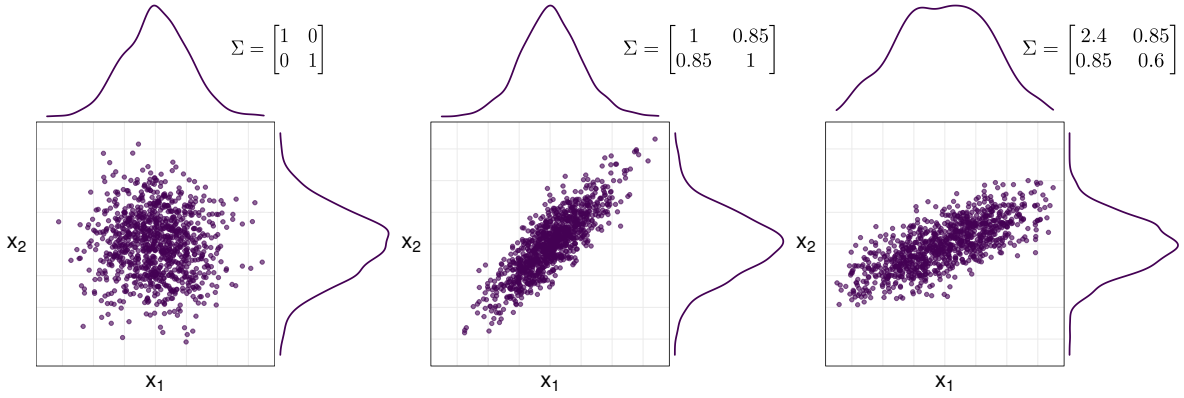

Figure 5.3: Effects of three covariance matrices on a multivariate normal distribution with mean vector $\boldsymbol{\mu} = [0\ 0]^\top$ and covariance matrix $\boldsymbol{\Sigma}$ as shown on the upper right corner of each plot

with $\varepsilon \sim \mathcal{N}(0, \sigma^2)$, which makes $f(\mathbf{X})$ a multivariate random variable, written

$$f(\mathbf{X}) \sim \mathcal{N}(\mathbf{X}\theta, \sigma^2\mathbf{I}), \tag{5.22}$$

where $\mathbf{I}$ is the $n \times p$ identity matrix.

Another way to state this key idea is to say that we make the hypothesis that f is a Gaussian Process, that is, that it belongs to the class of models containing all the functions whose values on any set of experiments $\mathbf{X}$ can be represented by a single sample of a multivariate normal distribution, with dimension $n = |\mathbf{X}|$. We can write this hypothesis as a prior probability distribution

$$f(\mathbf{X}) = [y_1\ \ldots\ y_n]^\top \sim \mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0), \tag{5.23}$$

where the mean vector $\boldsymbol{\mu}_0$ is usually a vector of n zeros and the covariance matrix $\boldsymbol{\Sigma}_0$ is computed using a kernel and depends on $\mathbf{X}$.

This prior is more general than the linear model hypothesis in Equation 5.22, but it generalizes even further. In Chapter *6* of *Gaussian Processes for Machine Learning* [10], Rasmussen and Williams describe relationships and equivalences of Gaussian Processes to other methods such as Splines, Support Vector Machines, and OLS estimation. They also describe how Neural Networks can be represented by specific Gaussian Processes in Section *4.2.3*.

In contrast to the linear model, the definition and exploration of a parameter space Θ for a Gaussian Process surrogate is done indirectly, by controlling kernel parameters and noise amplitudes. A first approach to define the parameter space could be explicitly searching for the covariance matrix and mean vector that best fit the data, where the size of the parameter space would increase as more training data becomes available. For a given design $\mathbf{X}$ of size n,

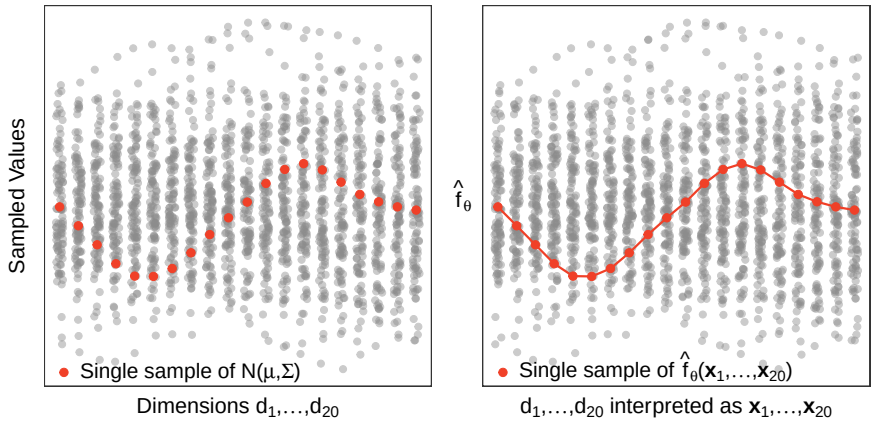

Figure 5.4: Reinterpreting the unrolled dimensions of 100 samples of a 20 dimension multivariate normal, on the left panel, to obtain 100 samples of functions evaluated on 20 different input points, on the right panel

this parameter space would be

$$\Theta_0 = \left\{ \left(\mu_{\mathbf{X},\mathbf{y}},\ \Sigma_{\mathbf{X}}\right),\ \mu_{\mathbf{X},\mathbf{y}} \in \mathbb{R}^n,\ \Sigma_{\mathbf{X}} \in \mathbb{R}^{n\times n} \right\}. \tag{5.24}$$

Note that the mean vector depends on the observations $\mathbf{y}$, while the covariance matrix depends only on $\mathbf{X}$. In a multivariate Gaussian, the diagonal of the covariance matrix contains the variances associated with its dimensions, and the off-diagonal elements contain the covariances between pairs of dimensions. Figure 5.3 shows the impact of three covariance matrices, which must always be symmetric and positive semi-definite.

Instead of looking for the best parameter vector in Θ_0, we will condition the prior Gaussian distribution to the observed data, obtaining a posterior distribution. The conditioned distribution representing our surrogate model is also a multivariate normal written

$$\hat{f}_\theta(\mathbf{X}) \sim f(\mathbf{X}) \mid \mathbf{X}, \mathbf{y}. \tag{5.25}$$

We will see how to compute this posterior in the next section. Note that it would require an infinite parameter vector to fit a Gaussian Process for all points in a search space $\mathcal{X}$ consisting of a single real number. In this sense, Gaussian Process Regression is a nonparametric method.

To build an intuitive understanding of Gaussian Processes it can be helpful to think of them as a reinterpretation of the dimensions of a sample of a multivariate normal distribution. If we take 100 samples of a 20-dimension Gaussian and unroll each dimension into a single axis, we end up with the left panel of Figure 5.4. Each column of points contains the 100 values on each dimension of our samples, and each dimension is correlated according to the distribution's covariance matrix. The values for one arbitrary sample are marked in larger red dots.

Table 5.1: Expressions for the covariance functions, or kernels, shown in Figure 5.5. The variables v and l are kernel parameters that can themselves be estimated. The Matérn kernel depends on the gamma function Γ and on the Bessel function of the second kind K_v. We refer the reader to Chapter 4 of Rasmussen and Williams [10] for detailed definitions and discussions

Kernel	Expressions
Exponential	$k(\mathbf{x}, \mathbf{x}') = exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|}{l}\right)$
Squared Exponential	$k(\mathbf{x}, \mathbf{x}') = exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2l^2}\right)$
Matérn	$k(\mathbf{x}, \mathbf{x}') = \frac{1}{2^{v-1}\Gamma(v)}\left(\frac{\sqrt{2v}}{l}\|\mathbf{x} - \mathbf{x}'\|\right)^v K_v\left(\frac{\sqrt{2v}}{l}\|\mathbf{x} - \mathbf{x}'\|\right)$

The right panel of Figure 5.4 shows the same data in the left panel but we now interpret the values of each dimension d_i, for all 100 samples, as a distribution of values for the surrogate $\hat{f}_\theta$ evaluated at point $\mathbf{x}_i$. Each $\hat{f}_\theta(\mathbf{x}_i)$ is correlated to other values of the surrogate function, with covariance given by the distribution's covariance matrix. In this example we used the Matérn kernel, discussed in the next section, to compute the covariance matrix. This prior over functions directly estimates a mean and its associated variance, for each value of the surrogate. We will now discuss how to fit a Gaussian Process to observed data, generating predictions of means and variances conditioned to observations.

5.2.1 Fitting the Model: Posterior Distributions over Functions

Before fitting our surrogate, we must compute the covariance matrix of the prior Gaussian distribution from Equation 5.23 using a covariance function, or kernel, $K : \mathcal{X}^2 \to \mathbb{R}$. For any pair $(\mathbf{x}, \mathbf{x}') \in \mathcal{X}^2$, the kernel determines how strong the covariance between $\hat{f}_\theta(\mathbf{x})$ and $\hat{f}_\theta(\mathbf{x}')$ should be, based on the distance $\|\mathbf{x} - \mathbf{x}'\|$. Figure 5.5 shows four exponential kernels, also called radial basis functions, whose formulas are shown in Table 5.1.

The actual parameter space Θ over which optimization is performed when fitting a Gaussian Process surrogate is composed by the parameters of the chosen covariance kernel, called hyperparameters. Typical hyperparameters are the Section *5.1* of Rasmussen and Williams [10], . Explicitly listing the hyperparameters for a Gaussian Process fit using the exponential kernel from Table 5.1 would result in

$$k(\mathbf{x}, \mathbf{x}') = \sigma_f^2 exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|}{l}\right), \tag{5.26}$$

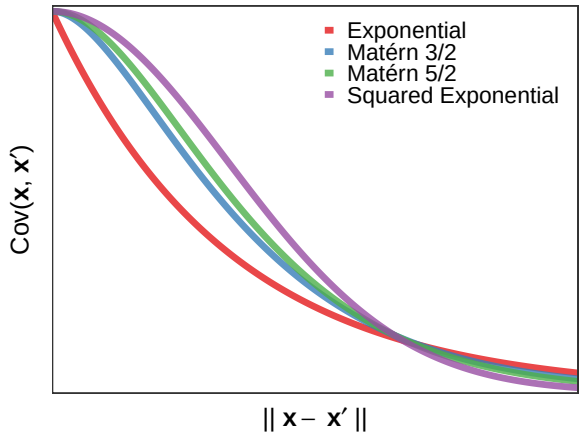

Figure 5.5: Covariance of points $(\mathbf{x}, \mathbf{x}')$ according to four covariance functions based on the distance $\|\mathbf{x} - \mathbf{x}'\|$. Expressions for each kernel are shown in Table 5.1, and Matérn kernels use parameters $v_1, v_2 = \{\frac{3}{2}, \frac{5}{2}\}$

and the parameter space Θ would be defined as

$$\Theta = \left\{ \sigma_f, l \in \mathbb{R} \right\}. \tag{5.27}$$

We can then use the conditioned posterior distribution, discussed below, to determine the best specific hyperparameter values by minimizing the cross-validated mean squared error, for example, which we discuss in Section 5.2.2, or by maximizing the posterior likelihood.

The posterior distribution $\hat{f}_\theta$ is computed by conditioning the prior distribution from Equation 5.23 to observed data $(\mathbf{X}_k, \mathbf{y}_k)$, obtaining the distribution in Equation 5.25, which is also a Gaussian distribution, and can be written

$$\hat{f}_\theta(\mathbf{X}_k) \sim \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k), \tag{5.28}$$

where the conditioned vector mean, of size $|\mathbf{X}_k|$, is

$$\boldsymbol{\mu}_k = \mathbf{K}(\mathbf{X}_k, \mathbf{X}_{k-1})\, \mathbf{K}(\mathbf{X}_{k-1}, \mathbf{X}_{k-1})^{-1}\, \mathbf{y}_{k-1}^\top, \tag{5.29}$$

and the conditioned $|\mathbf{X}_k| \times |\mathbf{X}_k|$ covariance matrix is written

$$\boldsymbol{\Sigma}_k = \mathbf{K}(\mathbf{X}_k, \mathbf{X}_k) \; - \; \mathbf{K}(\mathbf{X}_k, \mathbf{X}_{k-1})\, \mathbf{K}(\mathbf{X}_{k-1}, \mathbf{X}_{k-1})^{-1}\, \mathbf{K}(\mathbf{X}_{k-1}, \mathbf{X}_k). \tag{5.30}$$

The function $\mathbf{K}$ in Equations 5.29 and 5.30 produces the covariance matrices corresponding to applying one of the covariance kernels $k(\mathbf{x}, \mathbf{x}')$ from Table 5.1 to all pairs of points on the input designs for $\mathbf{K}$. The complete text of Rasmussen and Williams [10], the Chapter 6 of Bishop [65], and the Chapter 15 of Kochenderfer and Wheeler [61] are among the texts that present detailed discussions and derivations of important properties of Gaussian Process Regression.

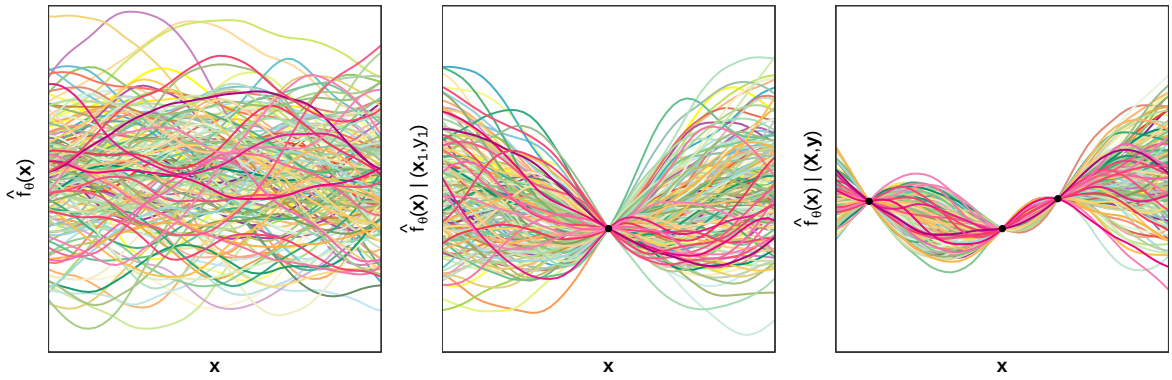

Figure 5.6: Fitting a Gaussian Process to three noise-free observations. The left panel shows 300 samples from a Gaussian prior, using the Matérn kernel to compute the covariance matrix. The center and right panels show 300 samples from the posterior distributions conditioned by one, then two more, successive noise-free observations

The left panel of Figure 5.6 shows 300 sampled functions from the prior in Equation 5.23, for a 50-dimension Gaussian distribution, with covariance matrix $\boldsymbol{\Sigma}_0$ given by the Matérn kernel from Table 5.1, with $v = \frac{5}{2}$. The starting mean vector $\boldsymbol{\mu}_0$ is zero. We see that, for all radial basis functions shown in Figure 5.5, the covariance between inputs decreases as the distance between inputs increases, approaching zero. Using different kernels we can control properties of sampled functions, such as smoothness and periodicity. The center panel of Figure 5.6 shows samples from the conditioned posterior after a single observation $(\mathbf{x}, \mathbf{y})$, computed using Equations 5.28, 5.29, and 5.30. The right panel shows samples from the posterior after observing two additional observations. Note that all 300 functions sampled from the conditioned distributions pass exactly through the observations on the center and rightmost panels.

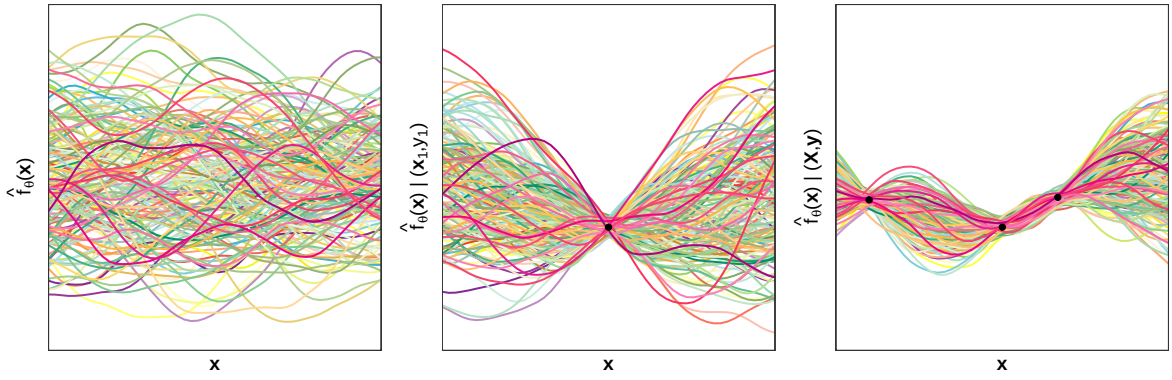

Figure 5.7: Fitting a Gaussian Process to three noisy observations, in the same conditions and with the same panel structure in Figure 5.6

Although in specific autotuning applications involving deterministic or extremely fast processes we can observe consistent measurements and produce useful models with fits that assume noiseless measurements, such as in Figure 5.6, we can also produce a Gaussian

Process fit that incorporates the uncertainty from noisy data. Figure 5.7 shows 300 prior samples, in the left panel, and results of conditioning the prior distribution to one and then three observations, under the assumption that the underlying objective function is subject to measurement error $\varepsilon \sim \mathcal{N}(0, \sigma^2)$, as described in Equation 5.21. In this scenario, we write the mean vector and covariance matrix for Equation 5.28 as

$$\boldsymbol{\mu}_k = \mathbf{K}(\mathbf{X}_k, \mathbf{X}_{k-1}) \left(\mathbf{K}(\mathbf{X}_{k-1}, \mathbf{X}_{k-1})^{-1} + \sigma^2 \mathbf{I}\right) \mathbf{y}_{k-1}^{\top}, \tag{5.31}$$

and

$$\boldsymbol{\Sigma}_k = \mathbf{K}(\mathbf{X}_k, \mathbf{X}_k) \; - \; \mathbf{K}(\mathbf{X}_k, \mathbf{X}_{k-1}) \left(\mathbf{K}(\mathbf{X}_{k-1}, \mathbf{X}_{k-1})^{-1} + \sigma^2 \mathbf{I}\right) \mathbf{K}(\mathbf{X}_{k-1}, \mathbf{X}_k). \tag{5.32}$$

Before discussing quality of fit assessment metrics for Gaussian Process surrogates, we will discuss how to incorporate hypotheses over the search space to fitted surrogates using basis functions, which we described when we discussed linear regression. Trend functions can be added to the surrogate's predicted mean to leverage underlying trends in data, that should be followed on prediction regions far from measurements.

Figure 5.8 shows Gaussian Process surrogates fitted with and without model trends. Black circles represent six measurements of a single-input function, and each line represents a model trend. We are considering noise-free measurements in these fits, so all surrogates agree on the predictions at the measurements. We can see that predictions for input between or sufficiently far from measurements present stronger influence from the underlying trend. A trend can help leverage previous knowledge of the relationships between factors and the values of the objective function, especially when the cost of measuring a single point is expensive, or when new experiments cannot be performed. However, if we can choose which measurements to perform we can use space-filling designs, which we discuss on Chapter 6, to decrease or sometimes remove the need for a model trend.

The fits and predictions shown in Figure 5.8 were computed using the *DiceKriging R* package [66]. We refer the reader to the detailed descriptions of trend functions and their application to Gaussian Process Regression presented in Section *2.1* of the package's accompanying paper [67].

As we did for the linear model surrogates discussed earlier in this chapter, in the next section we will assess the quality of fit of Gaussian Process surrogates, and quantify their prediction error on a testing set outside of the measurements used for fitting.

5.2.2 Assessing the Quality of Fit of Gaussian Process Surrogates

We will construct three Gaussian Process surrogates for the Booth function and use the model fits to discuss the assessment of model quality. Figure 5.9 shows the three surrogate fits, constructed using the same design $\mathbf{X}$ with $n = 10$ random uniformly distributed measurements

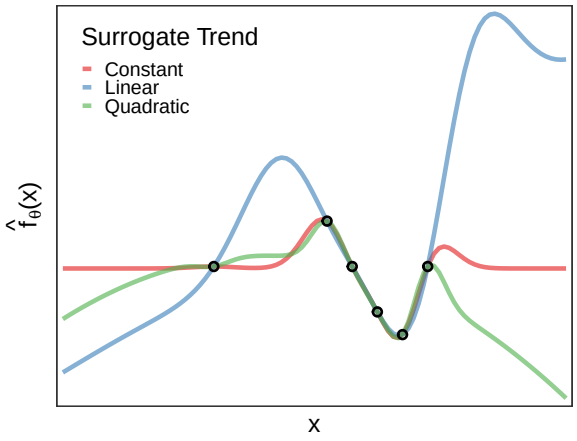

Figure 5.8: Gaussian Process surrogates using three different model trends, fit to six noise-free observations of a single-input objective function, marked with black circles

of the Booth function

$$f(\mathbf{x} = [x_1, x_2]^\top) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2, \; x_1, x_2 \in [-10, 10], \tag{5.33}$$

subject to measurement error ε. Panel (a) shows the surface produced by measuring f in its entire domain and the MSE of the constant mean predictor $MSE_{\bar{\mathbf{y}}}$. The 10 random measurements in $\mathbf{X}$ are highlighted. Panels (b), (c), and (d) show the three surrogates, which used model trends using the basis function sets described by Equations 5.10, 5.11, and 5.12. Panels (b), (c), and (d) show the testing $MSE_{\hat{f}_\theta}$ and true MSE_f prediction errors for each surrogate, described by Equations 5.14 and 5.15.

We have assumed noisy measurements when fitting the Gaussian Process surrogates in this example. In this setting, we could compute the training prediction error $MSE_{\hat{f}_\theta(\mathbf{X})}$ as described by Equation 5.13, but this would not work for a surrogate assuming noise-free measurements. Since all surrogates would interpolate the observations, the prediction error for points in the training set would always be zero. Still, analogously, in our example we also controlled the variances representing the measurement errors, as described by Equations 5.31 and 5.32.

The strategy we used for computing test prediction error for linear models would not be accurate for this example, and we used a Leave-One-Out (LOO) cross-validation strategy to compute the testing error $MSE^{LOO}_{\hat{f}_\theta}(\mathbf{X})$. The strategy consists of computing the mean of all $MSE_{\hat{f}_\theta(\mathbf{X})}$ for different surrogates, with the same model trend, fit to the testing sets generated by removing one distinct training point at a time, for all training points. This strategy also works to compute the testing prediction error for noise-free Gaussian Process fits.

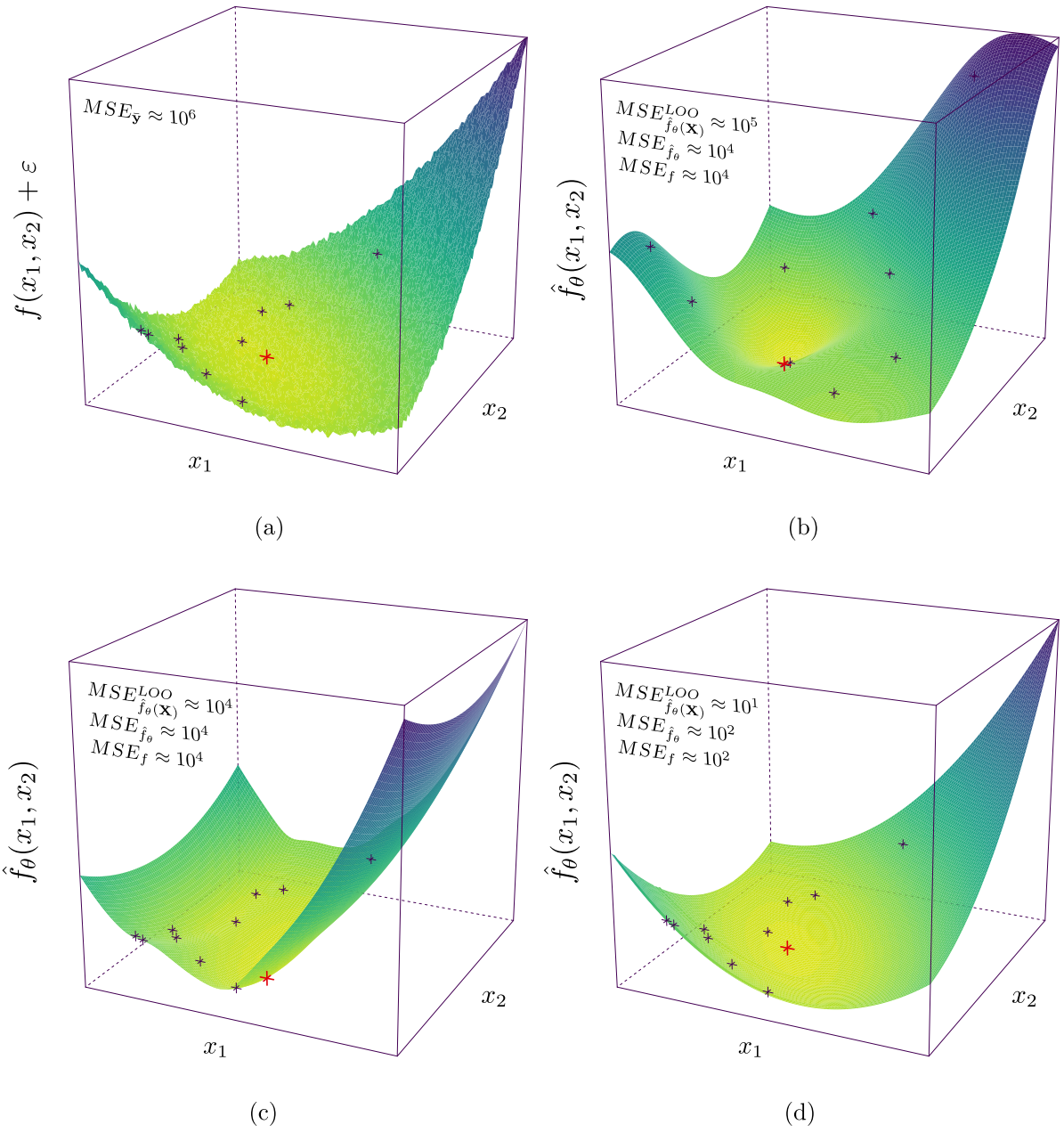

Figure 5.9: Three Gaussian Process Regression surrogates with noisy fits for the Booth function, with a × marking the global optimum and best surrogate predictions. The fixed experimental design **X** used to fit all surrogates is marked by × s. Panel (a) shows noisy measurements of Booth's function, panels (b), (c), and (d) show surrogate predictions for models fit with linear, quadratic, and quadratic plus interactions trends, respectively

5.2.3 Inference: Interpreting Significance with Sobol Indices

Sobol indices [68] are a variance-based global sensitivity analysis method, which computes the relative importance of the input factors $\mathbf{x}_1, \ldots, \mathbf{x}_p$ of an objective function f by decomposing the variance observed in measurements of f. The first-order Sobol index S_i for each factor $\mathbf{x}_i$ represents the normalized variance of the expected value of f given a fixed factor value, written

$$S_i = \frac{V_{\mathbf{x}_i}\left(E_{\mathbf{X}\setminus\mathbf{x}_i}\left(f(\mathbf{X}) \mid \mathbf{x}_i\right)\right)}{V\left(f(\mathbf{X})\right)}. \tag{5.34}$$

The variance of a function can be further decomposed to produce $2^p - 1$ indexes that measure the sensitivity of factor interactions of up to the p^{th} level. Additionally, we can aggregate the effects of the interactions of a factor and compute total-effect indices, in order to decrease the total number of indices to compute.

Sobol sensitivity indices can be estimated using $N(p + 2)$ evaluations of the objective function, where N controls the accuracy of the estimate and is typically in the order of thousands. The Monte Carlo method described in Section *4.1.6* of Saltelli *et al.* [69] determines how to construct the experiment matrices containing all samples, and computes the sensitivity indices using the estimators described in the book and later reviewed and summarized by the authors [70].

5.2.4 Gaussian Processes: Flexible but Hard to Interpret Surrogates

A couple of paragraphs to wrap the GPR section

5.3 Summary

A couple of paragraphs to Wrap up the Learning/Surrogates Chapter

Sketch:

- Two big classes of models. Generality vs. interpretability. Yet everything we mentioned assumes X is given, sampled from observations. In our contects, We can choose which X to test, either to test the model, or to improve its quality, or to find a "good" value in our space $\mathcal{X}$.

Chapter 6

Design of Experiments

TODO: Describe Full Factorial Designs

TODO: Fill in the sections sketched below

In this chapter we first present the assumptions of a traditional DoE methodology using an example of *2-level screening designs*, an efficient way to identify main effects. We then discuss techniques for the construction of efficient designs for factors with arbitrary numbers and types of levels, and present *D-Optimal* designs, the method used in this our approach.

6.1 2-Level Factorial Designs

6.2 Screening Designs

- Super efficient but very limited

6.2.1 Plackett-Burman Designs

Paley Construction of Hadamard Matrices

- Describe algorithm

6.2.2 An Example of Screening with Plackett-Burman Designs

TODO: Adapt this section from the CCGRID paper. It does not make sense to reference steps from the DLMT method here.

Screening designs identify parsimoniously the main effects of 2-level factors in the initial stages of studying a problem. While interactions are not considered at this stage, identifying

main effects early enables focusing on a smaller set of factors on subsequent experiments. A specially efficient design construction technique for screening designs was presented by Plackett and Burman [71] in 1946, and is available in the `FrF2` package [72] of the `R` language [73].

Despite having strong restrictions on the number of factors supported, Plackett-Burman designs enable the identification of main effects of n factors with $n + 1$ experiments. Factors may have many levels, but Plackett-Burman designs can only be constructed for 2-level factors. Therefore, before constructing a Plackett-Burman design we must identify *high* and *low* levels for each factor.

Assuming a linear relationship between factors and the response is fundamental for running ANOVA tests using a Plackett-Burman design. Consider the linear relationship

$$\mathbf{Y} = \boldsymbol{\beta}\mathbf{X} + \varepsilon, \tag{6.1}$$

where ε is the error term, $\mathbf{Y}$ is the observed response, $\mathbf{X} = \{1, x_1, \ldots, x_n\}$ is the set of n 2-level factors, and $\boldsymbol{\beta} = \{\beta_0, \ldots, \beta_n\}$ is the set with the *intercept* β_0 and the corresponding *model coefficients*. ANOVA tests can rigorously compute the significance of each factor, we can think of that intuitively by noting that less significant factors will have corresponding values in $\boldsymbol{\beta}$ close to zero.

The next example illustrates the screening methodology. Suppose we wish to minimize a performance metric Y of a problem with factors $x_1, \ldots, x_8$ assuming values in $\{-1, -0.8, -0.6, \ldots, 0.6, 0.8, 1\}$. Each $y_i \in Y$ is defined as

$$\begin{aligned} y_i = &- 1.5x_1 + 1.3x_3 + 3.1x_5 + \\ &- 1.4x_7 + 1.35x_8^2 + 1.6x_3x_5 + \varepsilon. \end{aligned} \tag{6.2}$$

Suppose that, for the purpose of this example, the computation is done by a very expensive black-box procedure. Note that factors $\{x_2, x_4, x_6\}$ have no contribution to the response, and we can think of the error term ε as representing not only noise, but our uncertainty regarding the model. Higher amplitudes of ε might make isolating factors with low significance harder to justify.

Table 6.1: A Plackett-Burman design for 7 2-level factors, where low and high levels are represented by −1 and 1, respectively

Run	A	B	C	D	E	F	G
1	1	-1	1	-1	-1	1	1
2	1	1	1	-1	1	-1	-1
3	-1	1	-1	-1	1	1	1
4	-1	1	1	1	-1	1	-1
5	1	-1	-1	1	1	1	-1
6	1	1	-1	1	-1	-1	1
7	-1	-1	1	1	1	-1	1
8	-1	-1	-1	-1	-1	-1	-1

To study this problem efficiently we decided to construct a Plackett-Burman design, which minimizes the experiments needed to identify significant factors. The analysis of this design will enable decreasing the dimension of the problem. Table 6.2 presents the Plackett-Burman design we generated. It contains high and low values, chosen to be -1 and 1, for the factors $x_1, \ldots, x_8$, and the observed response $\mathbf{Y}$. It is a required step to add the 3 "dummy" factors $d_1, \ldots, d_3$ to complete the 12 columns needed to construct a Plackett-Burman design for 8 factors [71].

So far, we have performed steps 1, 2, and 3 from Figure 12.5. We use our initial assumption in Equation (6.1) to identify the most significant factors by performing an ANOVA test, which is step 5 from Figure 12.5. The results are shown in Table 6.3, where the *significance* of each factor is interpreted from the F-test and Pr(> F) values. Table 6.3 uses "$*$", as is convention in the R language, to represent the significance values for each factor.

We see on Table 6.3 that factors $\{x_3, x_5, x_7, x_8\}$ have at least one "$*$" of significance. For the purpose of this example, this is sufficient reason to include them in our linear model for the next step. We decide as well to discard factors $\{x_2, x_4, x_6\}$ from our model, due to their low significance. We see that factor x_1 has a significance mark of "$\cdot$", but comparing F-test and Pr(> F) values we decide that they are fairly smaller than the values of factors that had no significance, and we keep this factor.

Moving forward to steps 6, 7, and 8 in Figure 12.5, we will build a linear model using factors $\{x_1, x_3, x_5, x_7, x_8\}$, fit the model using the values of Y we obtained when running our design, and use model coefficients to predict the levels of each factor that minimize the real response. We can do that because these factors are numerical, even though only discrete values are allowed.

We now proceed to the prediction step, where we wish to identify the levels of factors $\{x_1, x_3, x_5, x_7, x_8\}$ that minimize our fitted model, without running any new experiments. This can be done by, for example, using a gradient descent algorithm or finding the point where the derivative of the function given by the linear regression equals to zero.

Table 6.2: Randomized Plackett-Burman design for factors $x_1, \ldots, x_8$, using 12 experiments and "dummy" factors $d_1, \ldots, d_3$, and computed response $\mathbf{Y}$

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	d_1	d_2	d_3	Y
1	-1	1	1	1	-1	-1	-1	1	-1	1	13.74
-1	1	-1	1	1	-1	1	1	1	-1	-1	10.19
-1	1	1	-1	1	1	1	-1	-1	-1	1	9.22
1	1	-1	1	1	1	-1	-1	-1	1	-1	7.64
1	1	1	-1	-1	-1	1	-1	1	1	-1	8.63
-1	1	1	1	-1	-1	-1	1	-1	1	1	11.53
-1	-1	-1	1	-1	1	1	-1	1	1	1	2.09
1	1	-1	-1	-1	1	-1	1	1	-1	1	9.02
1	-1	-1	-1	1	-1	1	1	-1	1	1	10.68
1	-1	1	1	-1	1	1	1	-1	-1	-1	11.23
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	5.33
-1	-1	1	-1	1	1	-1	1	1	1	-1	14.79

Table 6.3: Shortened ANOVA table for the fit of the naive model, with significance intervals from the R language

	F value	Pr(> F)	Signif.
x_1	8.382	0.063	·
x_2	0.370	0.586	
x_3	80.902	0.003	**
x_4	0.215	0.675	
x_5	46.848	0.006	**
x_6	5.154	0.108	
x_7	13.831	0.034	*
x_8	59.768	0.004	**

Table 6.4 compares the prediction for Y from our linear model with the selected factors $\{x_1, x_3, x_5, x_7, x_8\}$ with the actual global minimum Y for this problem. Note that factors $\{x_2, x_4, x_6\}$ are included for the global minimum. This happens here because of the error term ε, but could also be interpreted as due to model uncertainty.

Using 12 measurements and a simple linear model, the predicted best value of Y was around 10× larger than the global optimum. Note that the model predicted the correct levels for x_3 and x_5, and almost for x_7. The linear model predicted wrong levels for x_1, perhaps due to this factor's interaction with x_3, and for x_8. Arguably, it would be impossible to predict the correct level for x_8 using this linear model, since a quadratic term composes the true formula of Y. As we showed in Figure 6.1, a D-Optimal design using a linear model could detect the significance of a quadratic term, but the resulting regression will often lead to the wrong level.

We can improve upon this result if we introduce some information about the problem and use a more flexible design construction technique. Next, we will discuss the construction of efficient designs using problem-specific formulas and continue the optimization of our example.

6.3 Optimal Design

- Extensive theory development
  - General equivalence of design criteria
- A "flexible" screening: allows to include non-linear terms, interactions, etc. if needed.

Table 6.4: Comparison of the response Y predicted by the linear model and the true global minimum. Factors used in the model are bolded

	$\boldsymbol{x_1}$	x_2	$\boldsymbol{x_3}$	x_4	$\boldsymbol{x_5}$	x_6	$\boldsymbol{x_7}$	$\boldsymbol{x_8}$	Y
Lin.	-1.0	–	-1.0	–	-1.0	–	1.0	-1.0	-1.046
Min.	1.0	-0.2	-1.0	0.6	-1.0	0.4	0.8	0.0	-9.934

- Awesome but if a parameter was not included in the model or if the model is too simple (e.g. only comprised a linear term where a quadratic one would have been needed, or an important interaction was not included), we won't be able to detect it (lack of fit).

6.3.1 The KL-Exchange Algorithm

- Add pseudocode, or schematic figure
- Fedorov's algorithm is a special case of this

6.3.2 An Example with D-Optimal Designs

TODO: Adapt this section from the CCGRID paper. It does not make sense to reference steps from the DLMT method here.

The application of DoE to autotuning problems requires design construction techniques that support factors of arbitrary types and number of levels. Autotuning problems typically combine factors such as binary flags, integer and floating point numerical values, and unordered enumerations of abstract values. Because Plackett-Burman designs only support 2-level factors, we had to restrict factor levels to interval extremities in our example. We have seen that this restriction makes it difficult to measure the significance of quadratic terms. We will now show how to optimize our example further by using *D-Optimal designs*, which increase the number of levels we can efficiently screen for and enables detecting the significance of more complex terms.

To construct a D-Optimal design it is necessary to choose an initial model, which can be done based on previous experiments or on expert knowledge of the problem. Once a model is selected, algorithmic construction is performed by searching for the set of experiments that minimizes *D-Optimality*, a measure of the *variance* of the *estimators* for the *regression coefficients* associated with the selected model. This search is usually done by swapping experiments from the current candidate design with experiments from a pool of possible experiments, according to certain rules, until some stopping criterion is met. In the example in this section and in our approach we use Fedorov's algorithm [74] for constructing D-Optimal designs, implemented in `R` in the `AlgDesign` package [75].

In our example, suppose that in addition to using our previous screening results we decide to hire an expert in our problem's domain. The expert confirms our initial assumptions that the factor x_1 should be included in our model since it is usually significant for this kind of problem and has a strong interaction with factor x_3. She also mentions we should replace the linear term for x_8 by a quadratic term for this factor.

Using our previous screening and the domain knowledge provided by our expert, we choose a new performance model and use it to construct a D-Optimal design using Fedorov's algorithm. Since we need enough degrees of freedom to fit our model, we construct the

Table 6.5: D-Optimal design constructed for the factors $\{x_1, x_3, x_5, x_7, x_8\}$ and computed response Y

x_1	x_3	x_5	x_7	x_8	Y
-1.0	-1.0	-1.0	-1.0	-1.0	2.455
-1.0	1.0	1.0	-1.0	-1.0	6.992
1.0	-1.0	-1.0	1.0	-1.0	-7.776
1.0	1.0	1.0	1.0	-1.0	4.163
1.0	1.0	-1.0	-1.0	0.0	0.862
-1.0	1.0	1.0	-1.0	0.0	5.703
1.0	-1.0	-1.0	1.0	0.0	-9.019
-1.0	-1.0	1.0	1.0	0.0	2.653
-1.0	-1.0	-1.0	-1.0	1.0	1.951
1.0	-1.0	1.0	-1.0	1.0	0.446
-1.0	1.0	-1.0	1.0	1.0	-2.383
1.0	1.0	1.0	1.0	1.0	4.423

design with 12 experiments shown in Table 6.5. Note that the design includes levels -1, 0, and 1 for factor x_8. The design will sample from different regions of the search space due to the quadratic term, as was shown in Figure 6.1.

We now fit this model using the results of the experiments in our design. Table 6.6 shows the model fit table and compares the estimated and real model coefficients. This example illustrates that the DoE approach can achieve close model estimations using few resources, provided the approach is able to use user input to identify significant factors, and knowledge about the problem domain to tweak the model.

Table 6.7 compares the global minimum of this example with the predictions made by our initial linear model from the screening step, and our improved model. Using screening, D-Optimal designs, and domain knowledge, we found an optimization within 10% of the global optimum while computing Y only 24 times. We were able to do that by first reducing the problem's dimension when we eliminated insignificant factors in the screening step. We then constructed a more careful exploration of this new problem subspace, aided by domain knowledge provided by an expert. Note that we could have reused some of the 12 experiments from the previous step to reduce the size of the new design further.

Table 6.6: Correct model fit comparing real and estimated coefficients, with significance intervals from the R language

	Real	Estimated	t value	Pr(> |t|)	Signif.
Intercept	0.000	0.050	0.305	0.776	
x_1	-1.500	-1.452	-14.542	0.000	***
x_3	1.300	1.527	15.292	0.000	***
x_5	3.100	2.682	26.857	0.000	***
x_7	-1.400	-1.712	-17.141	0.000	***
x_8	0.000	-0.175	-1.516	0.204	
x_8^2	1.350	1.234	6.180	0.003	**
x_1x_3	1.600	1.879	19.955	0.000	***

Table 6.7: Comparison of the response Y predicted by our models and the true global minimum. Factors used in the models are bolded

	$\boldsymbol{x_1}$	x_2	$\boldsymbol{x_3}$	x_4	$\boldsymbol{x_5}$	x_6	$\boldsymbol{x_7}$	$\boldsymbol{x_8}$	Y
Quad.	1.0	–	-1.0	–	-1.0	–	1.0	0.0	-9.019
Lin.	-1.0	–	-1.0	–	-1.0	–	1.0	-1.0	-1.046
Min.	1.0	-0.2	-1.0	0.6	-1.0	0.4	0.8	0.0	-9.934

We are able to explain the performance improvements we obtained in each step of the process, because we finish steps with a performance model and a performance prediction. Each factor is included or removed using information obtained in statistical tests, or expert knowledge. If we need to optimize this problem again, for a different architecture or with larger input, we could start exploring the search space with a less naive model. We could also continue the optimization of this problem by exploring levels of factors $\{x_2, x_4, x_6\}$. The significance of these factors could now be detectable by ANOVA tests since the other factors are now fixed. If we still cannot identify any significant factor, it might be advisable to spend the remaining budget using another exploration strategy such as uniform random or latin hypercube sampling.

The process of screening for factor significance using ANOVA and fitting a new model using acquired knowledge is equivalent to steps 5, 6, and 7 in Figure 12.5. In the next section we evaluate the performance of our DoE approach in two scenarios.

6.4 Space-Filling Designs

- Not very efficient for parameter estimate but good to evaluate the lack of fit.
- Also good for variance minimization in GP.
- Quasi-random Sequences
  - Sobol Sequences

6.5 Comparing Sampling Strategies

TODO: Adapt this section from the CCGRID paper.

Figure 6.1 shows the contour of a search space defined by a function of the form $z = x^2 + y^2 + \varepsilon$, where ε is a local perturbation, and the exploration of that search space by six different strategies. In such a simple search space, even a uniform random sample can find points close to the optimum, despite not exploiting geometry. A Latin Hypercube [76] sampling strategy covers the search space more evenly, but still does not exploit the space's geometry. Strategies based on neighborhood exploration such as simulated annealing and

gradient descent can exploit local structures, but may get trapped in local minima. Their performance is strongly dependent on search starting point. These strategies do not leverage global search space structure, or provide exploitable knowledge after optimization.

Measurement of the kernels optimized on the performance evaluations in Section Performance Evaluation can exceed 20 minutes, including the time of code transformation, compilation, and execution. Measurements in other problem domains can take much longer to complete. This strengthens the motivation to consider search space exploration strategies capable of operating under tight budget constraints. These strategies have been developed and improved by statisticians for a long time, and can be grouped under the DoE term.

The D-Optimal sampling strategies shown on the two rightmost bottom panels of Figure 6.1 are based on the DoE methodology, and leverage previous knowledge about search spaces for an efficient exploration. These strategies provide transparent analyses that enable focusing on interesting subspaces. In the next section we describe our approach to autotuning based on the DoE methodology.

6.6 Summary

- Designs to obtain good-quality parameter estimates
  - Screening and D-opt for LM
  - SFD for GP
- Designs to test the model quality (lack of fit)
  - SFD fo LM
- If model based, parameter significance and estimation can be used to reduce dimension and guide the optimization.
- With this DoE approach, we have a clear separation between the sampling phase and the interpretation phase. But what if no parameter really appears significant anymore?

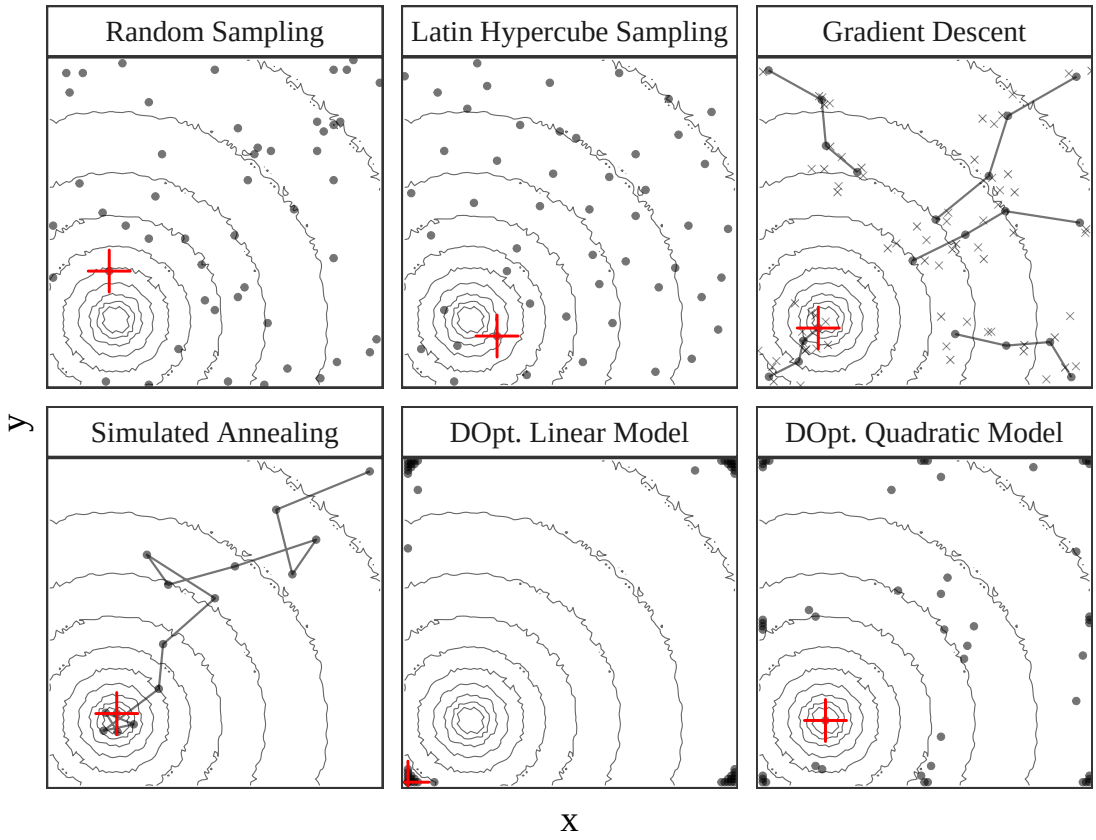

Figure 6.1: Exploration of the search space, using a fixed budget of 50 points. The red "+" represents the best point found by each strategy, and "×"s denote neighborhood exploration

Chapter 7

Online Learning

TODO: Fill in the sketched section below

7.1 Independent Bandits

- simple (discrete choice, optimize $regret = \sum_t R_t$)
- UCB

7.2 Expected Improvement for Gaussian Process Regression

- (continuous choice, optimize $EI = \max_t R_t$)
- Also mention GP-UCB and contrast with EGO. There are also variants for Linear Model (LinUCB)

7.3 Reinforcement Learning

- Unsupervised learning
- Actor-critic model
- Main motivation for adding this section is to contextualize results with GPR and Neural Nets. Could move to last Application chapter

7.4 Summary

- These methods seamlessly mix exploration and exploitation but the overall objective function is generally the regret, which makes sense for a self-optimizing system (e.g. facebook) but not in an autotuning context (where EI is more meaningful)

Chapter 8

Towards Transparent and Parsimonious Methods for Automatic Performance Tuning

TODO: Revisit semi-conclusions of each chapter, wrap up this part, hook to Applications part

- use glassbox (DoE based) approach to perform the optimization, always try to interpret the results
- 2 big methods based on different exploration/exploitation strategy:
  1. Evaluate parameter significance and reduce dimension (two phases, iterative)
  2. Expected Improvement (first a general exploration phase with a SFD, then seamlessly mix exploration and exploitation)

  Possibly combinations of both approaches could be used back and forth depending on the specific information we learn on the use case.
- In this thesis, we evaluate these DoE-based approaches for several autotuning use cases and try to compare them with approaches that had beed previously proposed for these use cases.

TODO: Adapt this paragraph

The effectiveness of stochastic descent methods on autotuning can be limited [5, 77, 78], between other factors, by underlying hypotheses about the search space, such as the reachability of the global optimum and the smoothness of search space surfaces, which are frequently not respected. The derivation of relationships between parameters and performance from

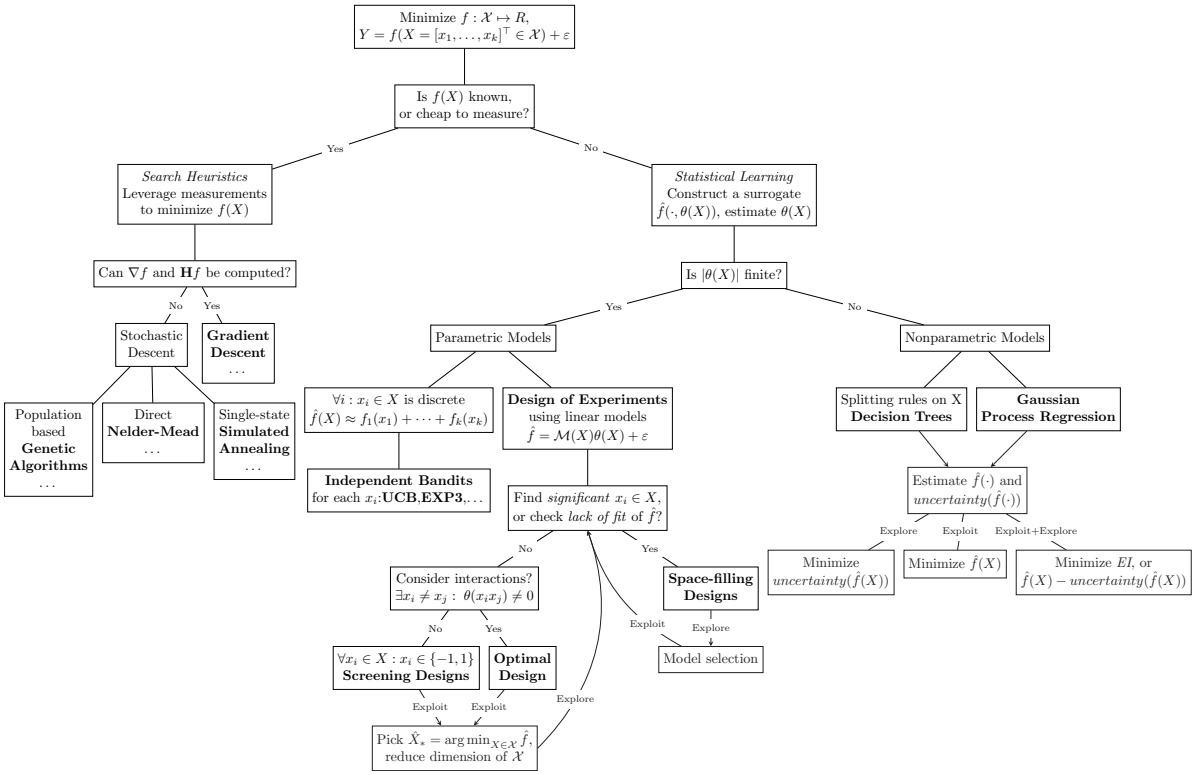

Figure 8.1: Some autotuning methods

search heuristic optimizations is greatly hindered, if not rendered impossible, by the biased way these methods explore parameters. Some parametric learning methods, such as Design of Experiments, are not widely applied to autotuning. These methods perform structured parameter exploration, and can be used to build and validate performance models, generating transparent and cost-effective optimizations [34, 4]. Other methods from the parametric family are more widely used, such as Bandit Algorithms [36]. Nonparametric learning methods, such as Decision Trees [79] and Gaussian Process Regression [80], are able to reduce model bias greatly, at the expense of increased prediction variance. Figure 8.1 categorizes some autotuning methods according to some of the key hypotheses and branching questions underlying each method.

Part III

Applications

Chapter 9

Research Methodology: Efforts for Reproducible Science

Sketch:

- Tools and such
- Explain difficulty of finding a needle in a haystack:
  - how to know whether we found the optimal value ?
  - how to know how far we are from the optimal value ?
  - how to know whether there is anything to find ?
  - how to know whether the geometry hypothesis we make are sound in an unknown space ?
  - ...

9.1 Introduction

9.2 Computational Documents with Emacs and Org mode

9.3 Versioning for Code, Text, and Data

Chapter 10

Stochastic Methods and Screening for CUDA Kernels

A Graphics Processing Unit (GPU) is a parallel computing coprocessor specialized in accelerating vector operations such as graphics rendering. General Purpose computing on GPUs (GPGPU) depends on accessible programming interfaces for languages such as C and Python that enable the use of GPUs in different parallel computing domains. The Compute Unified Device Architecture (CUDA) is a GPGPU toolchain introduced by the NVIDIA corporation.

In this study we measured the performance improvements obtained with an ensemble of stochastic methods for function minimization applied to the parameters of the CUDA compiler. We implemented an autotuner using the OpenTuner framework [1] and used it to search for the compilation parameters that optimize the performance of 17 heterogeneous GPU kernels, 12 of which are from the Rodinia Benchmark Suite [81]. We used 3 different NVIDIA GPUs in the experiments, the Tesla K40, the GTX 980, and the GTX 750. We published the results of this study, which we summarize in this chapter, at the *Concurrency and Computation: Practice and Experience* journal [2].

The optimizations we found often beat compiler high-level optimization options, such as the *-O2* flag. The autotuner found compilation options that produced speedups, in relation to *-O2*, of around 2 times for the {Gaussian Elimination} problem from the Rodinia Benchmark Suite, 2 times for *Heart Wall* problem, also from Rodinia, and 4 times for one of the matrix multiplication algorithms we implemented. We observed that the compilation parameters that optimize an algorithm for a given GPU architecture will not always achieve the same performance in different hardware, which was a reasonable expectation.

Figure 10.1 shows a representation of the autotuner implemented in this study, and illustrates the time scale of our experiments. The autotuner, in light blue, receives as input a GPU kernel, a target GPU, input data for the kernel, and a search space composed of NVCC flags. After running an ensemble of stochastic search methods, the autotuner outputs a flag

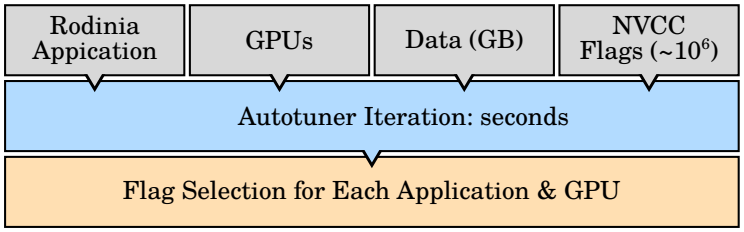

Figure 10.1: Autotuner representation and time scale of the experiments

selection. The compilation of the kernels we targeted in this study takes few seconds, so it was possible to test thousands of flag combinations per hour using a sequential autotuner.

The rest of this chapter is organized as follows. Section 10.1 discusses NVIDIA GPU architecture and programming toolchain, and presents related work on GPU performance tuning and modeling. Section 10.2 presents the search space of CUDA parameters and discusses the autotuner we implemented. Section 10.3 presents GPUs and the kernels whose compilation we attempted to optimize. Section 10.4 presents and discusses the performance improvements achieved by the autotuner, and attempts to identify significant flags using clustering. Section 10.5 was written at a later moment, after studying Experimental Design methods, and revisits this problem with a screening experiment. Finally, Section 10.6 summarizes the discussion and presents perspectives for future work.

10.1 General-Purpose Computing on NVIDIA GPUs

10.1.1 NVIDIA GPU Micro-Architecture

NVIDIA GPU architectures have multiple asynchronous and parallel Streaming Multiprocessors (SMs) which contain Scalar Processors (SPs), Special Function Units (SFUs) and load/store units. Each group of 32 parallel threads scheduled by and SM, or *warp*, is able to read from memory concurrently. NVIDIA architectures vary in a large number of features, such as number of cores, registers, SFUs, load/store units, on-chip and cache memory sizes, processor clock frequency, memory bandwidth, unified memory spaces and dynamic kernel launches. Those differences are summarized in the Compute Capability (C.C.) of an NVIDIA GPU.

The hierarchical memory of an NVIDIA GPU contains global and shared portions. Global memory is big, off-chip, has a high latency and can be accessed by all threads of the kernel. Shared memory is small, on-chip, has a low-latency and can be accessed only by threads in a same SM. Each SM has its own shared L1 cache, and new architectures have coherent global L2 caches. Optimizing thread accesses to different memory levels is essential to achieve good performance.

10.1.2 Compute Unified Device Architecture (CUDA)

The CUDA programming model and platform enables the use of NVIDIA GPUs for scientific and general purpose computation. A single *main* thread runs in the CPU, launching and managing computations on the GPU. Data for the computations has to be transferred from the main memory to the GPU's memory. Multiple computations launched by the main thread, or *kernels*, can run asynchronously and concurrently. If the threads from a same warp must execute different instructions the CUDA compiler must generate code to branch the execution correctly, making the program lose performance due to this *warp divergence*.

The CUDA language extends C and provides a multi-step compiler, called *NVCC*, that translates CUDA code to Parallel Thread Execution code (PTX). *NVCC* uses the host's C++ compiler in several compilation steps, and also to generate code to be executed in the host. The final binary generated by *NVCC* contains code for the GPU and the host. When PTX code is loaded by a kernel at runtime, it is compiled to binary code by the host's device driver. This binary code can be executed in the device's processing cores, and is architecture-specific. The target architecture can be specified using *NVCC* parameters.

10.1.3 GPU Performance Models and Autotuning

The accuracy of a GPU performance model is subject to low level elements such as instruction pipeline usage and small cache hierarchies. A GPU's performance approaches its peak when the instruction pipeline is saturated, but becomes unpredictable when the pipeline is under-utilized [82, 83]. Considering the effects of small cache hierarchies [84, 85] and memory-access divergence [86, 87] is also critical to a GPU performance model.

Guo and Wang [88] introduce a framework for autotuning the number of threads and the sizes of blocks and warps used by the CUDA compiler for sparse matrix and vector multiplication GPU applications. Li *et al.* [89] discuss the performance of autotuning techniques in highly-tuned GPU General Matrix to Matrix Multiplication (GEMMs) routines, highlighting the difficulty in developing optimized code for new GPU architectures. Grauer-Gray *et al.* [90] autotune an optimization space of GPU kernels focusing on tiling, loop permutation, unrolling, and parallelization. Chaparala *et al.* [91] autotune GPU-accelerated Quadratic Assignment Problem solvers. Sedaghati *et al.* [92] build a decision model for the selection of sparse matrix representations in GPUs.

10.2 Autotuner and Search Space for the NVCC Compiler

This section discusses our autotuner implementation and the search space of NVCC compiler parameters.

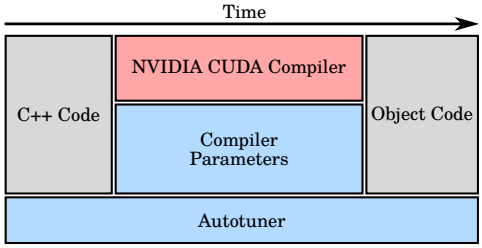

Figure 10.2: Simplified view of NVCC compilation

10.2.1 An Autotuner for CUDA Parameters using OpenTuner

The autotuner was implemented using the OpenTuner framework [1], and used a multi-armed bandit algorithm that aims to maximize the successes across the autotuning process. A success is defined as finding a program configuration that improves upon the best performance found so far. The stochastic method which finds such a configuration gets its score increased for that tuning session. A bandit algorithm, called *MABAUC* in the paper, for Multi-Armed Bandit Area Under the Curve, considers a sliding window covering a given number of measurements to compute the scores of methods in an ensemble, and to decide which method, or "arm", to play next. The ensemble of methods we used was composed by implementations of the Nelder-Mead algorithm and three variations of genetic algorithms. Figure 10.2 shows a simplified version of the steps necessary to generate the object code that will be measured later. The code for our autotuner and all the experiments and results is available [93] under the GNU General Public License.

10.2.2 Search Space for CUDA Parameters

Table 10.1 presents a subset of the CUDA configuration parameters [94] used in this study. Parameters can target different compilation steps, namely the *PTX* optimizing assembler, the *NVLINK* linker, and the *NVCC* compiler. We compared the performance of programs generated by tuned parameters with the standard compiler optimizations *opt-level* equal to *0*, *1*, *2*, *3*. Different optimization levels could also be selected during tuning. We did not use compiler options that target the host linker or the library manager since they do not affect performance. The size of the search space defined by all possible combinations of flags is in the order of 10^6, making hand-optimization or exhaustive searches unwieldy.

10.3 Target GPUs and Kernels

This section presents the GPU testbed, the algorithm benchmark, the autotuner implementation and its search space.

Table 10.1: Description of flags in the search space

Flag	Description
`no-align-double`	Specifies that `malign-double` should not be passed as a compiler argument on 32-bit platforms. **Step**: NVCC
`use_fast_math`	Uses the fast math library, implies `ftz=true`, `prec-div=false`, `prec-sqrt=false` and `fmad=true`. **Step**: NVCC
`gpu-architecture`	Specifies the NVIDIA virtual GPU architecture for which the CUDA input files must be compiled. **Step**: NVCC **Values**: `sm_20`, `sm_21`, `sm_30`, `sm_32`, `sm_35`, `sm_50`, `sm_52`
`relocatable-device-code`	Enables the generation of relocatable device code. If disabled, executable device code is generated. Relocatable device code must be linked before it can be executed. **Step**: NVCC
`ftz`	Controls single-precision denormals support. `ftz=true` flushes denormal values to zero and `ftz=false` preserves denormal values. **Step**: NVCC
`prec-div`	Controls single-precision floating-point division and reciprocals. `prec-div=true` enables the IEEE round-to-nearest mode and `prec-div=false` enables the fast approximation mode. **Step**: NVCC
`prec-sqrt`	Controls single-precision floating-point squre root. `prec-sqrt=true` enables the IEEE round-to-nearest mode and `prec-sqrt=false` enables the fast approximation mode. **Step**: NVCC
`def-load-cache`	Default cache modifier on global/generic load. **Step**: PTX **Values**: `ca`, `cg`, `cv`, `cs`
`opt-level`	Specifies high-level optimizations. **Step**: PTX **Values**: `0 - 3`
`fmad`	Enables the contraction of floating-point multiplies and adds/subtracts into floating-point multiply-add operations (FMAD, FFMA, or DFMA). **Step**: PTX
`allow-expensive-optimizations`	Enables the compiler to perform expensive optimizations using maximum available resources (memory and compile-time). If unspecified, default behavior is to enable this feature for optimization level ⩾O2. **Step**: PTX
`maxrregcount`	Specifies the maximum number of registers that GPU functions can use. **Step**: PTX **Values**: `16 - 64`
`preserve-relocs`	Makes the `PTX` assembler generate relocatable references for variables and preserve relocations generated for them in the linked executable. **Step**: NVLINK

10.3.1 Target GPU Architectures

To be able to show that different GPUs require different options to improve performance, and that it is possible to achieve speedups in different hardware, we wanted to tune our benchmark for different NVIDIA architectures. Table 10.2 summarizes the hardware characteristics of the three GPUs.

10.3.2 Benchmark of CUDA Algorithms

We composed a benchmark with 17 heterogeneous GPU kernels. The benchmark contains 4 optimization strategies for *matrix multiplication* counted as a single kernel, 1 vector addition problem, 1 solution for the *maximum sub-array problem* [95], 2 *sorting* algorithms and 12 kernels from the Rodinia Benchmark Suite [81].

Table 10.2: Hardware specifications of the target GPU architectures

Model	C.C.	Global Memory	Bus	Bandwidth	L2	Cores/SM	Clock
Tesla-K40	3.5	12 GB	384-bit	276.5 GB/s	1.5 MB	2880/15	745 Mhz
GTX-750	5.0	1 GB	128-bit	86.4 GB/s	2 MB	512/4	1110 Mhz
GTX-980	5.2	4 GB	256-bit	224.3 GB/s	2 MB	2048/16	1216 Mhz

Table 10.3 shows the Rodinia kernels contained in our benchmark and the corresponding three-letter code. The other kernels in the benchmark were the following CUDA implementations:

- Matrix multiplications using:
  - Global memory with non-coalesced accesses (MMU)
  - Global memory with coalesced accesses (MMG)
  - Shared memory with non-coalesced accesses to global memory (MSU)
  - Shared memory with coalesced accesses to global memory (MMS)
- Simple vector addition algorithm (VAD)
- Solution for the Maximum Sub-Array Problem (MSA) [97, 95]
- Quicksort (QKS) and Bitonicsort (BTN)

10.4 Performance Improvements and Parameter Clustering Attempt

This section presents the speedups achieved for all algorithms in the benchmark, highlights the most significant speedups, and discusses the performance and accuracy of the autotuner.

Table 10.3: Rodinia [81] kernels used in the experiments

Kernel	Berkeley Dwarf[96]	Domain
B+Tree (BPT)	Graph Traversal	Search
Back Propagation (BCK)	Unstructured Grid	Pattern Recognition
Breadth-First Search (BFS)	Graph Traversal	Graph Algorithms
Gaussian Elimination (GAU)	Dense Linear Algebra	Linear Algebra
Heart Wall (HWL)	Structured Grid	Medical Imaging
Hot Spot (HOT)	Structured Grid	Physics Simulation
K-Means (KMN)	Dense Linear Algebra	Data Mining
LavaMD (LMD)	N-Body	Molecular Dynamics
LU Decomposition (LUD)	Dense Linear Algebra	Linear Algebra
Myocyte (MYO)	Structured Grid	Biological Simulation
Needleman-Wunsch (NDL)	Dynamic Programming	Bioinformatics
Path Finder (PTF)	Dynamic Programming	Grid Traversal

10.4.1 Performance Improvements

Figure 10.3 compares the means of 30 measurements of the tuned results for each kernel to the high-level optimization *opt-level* set to 2. We see that that the autotuned solution for the Heart Wall problem (HWL) in the *Tesla K40* achieved over 2 times speedup in comparison with the high-level CUDA optimizations. The tuned kernel for the *GTX-980* reached almost 2.5 times speedup for Gaussian Elimination (GAU). We also found smaller speedups for most kernels, such as a 10% speedup for Path Finder (PTF) in *Tesla K40*, and around 10% speedup for Myocyte (MYO) on *GTX-750*.

Figure 10.4 summarizes the results for the other CUDA kernels we implemented. The autotuner did not improve upon the high-level optimizations for BTN, VAD, and QKS in any of the GPUs in the testbed, but it found solutions that achieved speedups for at least one GPU for the other kernels.

We were not able to determine the hardware characteristics that impacted performance. We believe that the Maxwell GPUs, the *GTX-980* and *GTX-750*, had differing results from the *Tesla K40* because they are consumer grade GPUs, producing less precise results, and configured with different default optimizations. The similarities between the compute capabilities of the GTX GPUs could also explain the observed differences from the K40. The overall greater computing power of the *GTX-980* could explain its differing results from the *GTX-750*, since the GTX 980 has a greater number of Cores, SMs/Cores, Bandwidth, Bus, Clock and Global Memory.

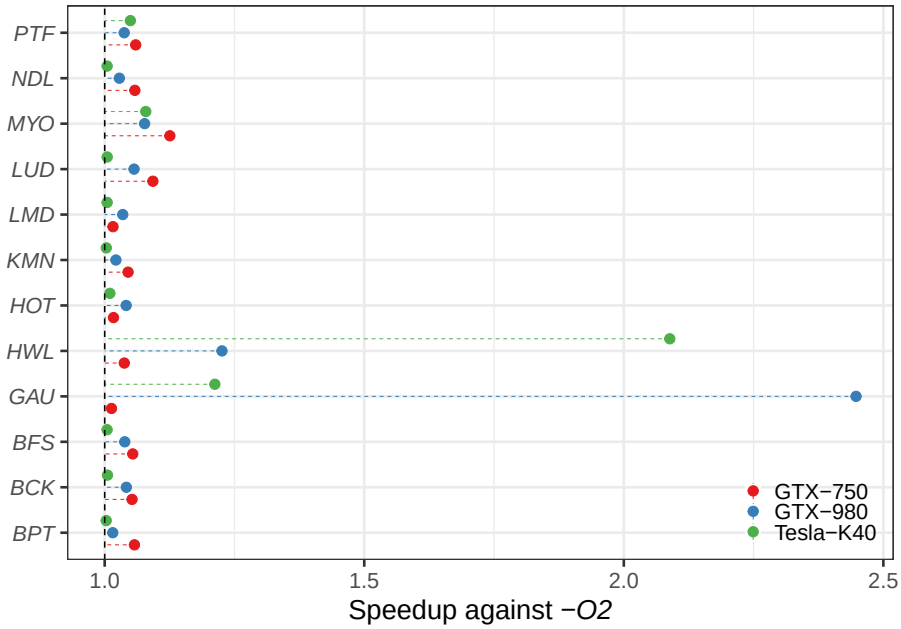

Figure 10.3: Mean speedup over 30 repetitions of the tuned solutions for Rodinia kernels versus the *opt-level* = 2 baseline

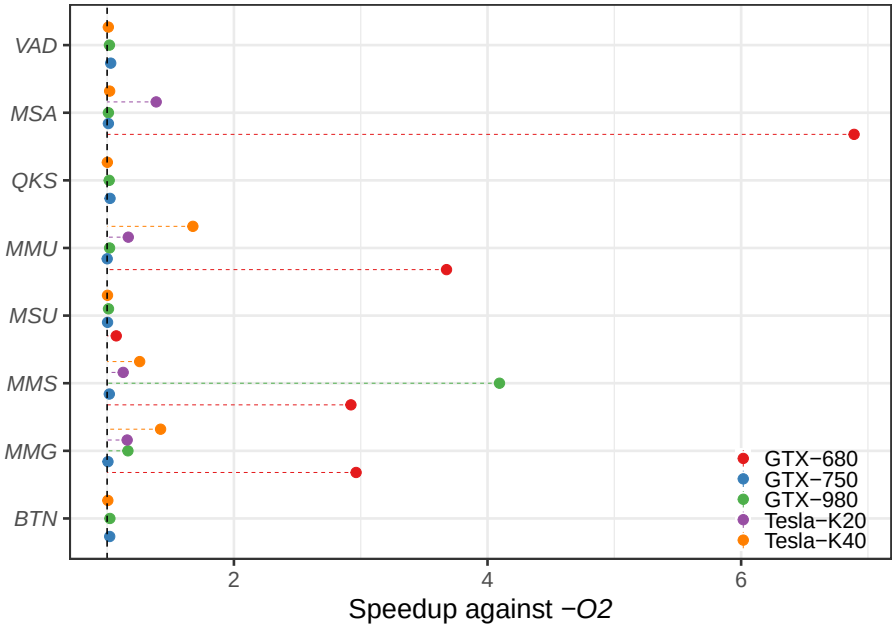

Figure 10.4: Mean speedup over 30 repetitions of the tuned solutions for kernels we implemented versus the *opt-level* = 2 baseline

10.4.2 CUDA Compiler Autotuner Performance

This section presents an assessment of the autotuner's performance. Figures 10.5 and 10.6 present the speedup of the best solution found across tuning time. Points in each graph represent the performance, in the y-axis, of the best configuration found at the corresponding tuning time, shown in the x-axis. The leftmost point in each graph represents the performance of a configuration chosen at random by the autotuner. Each subsequent point represents the performance of the best configuration found across optimization. Note that the autotuner is able to quickly improve upon the initial random configuration, but the rate of improvement also decays quickly. The duration of all tuning runs was two hours, or 7200 seconds. The rightmost point in each graph represents the performance of the last improving configuration found by the autotuner.

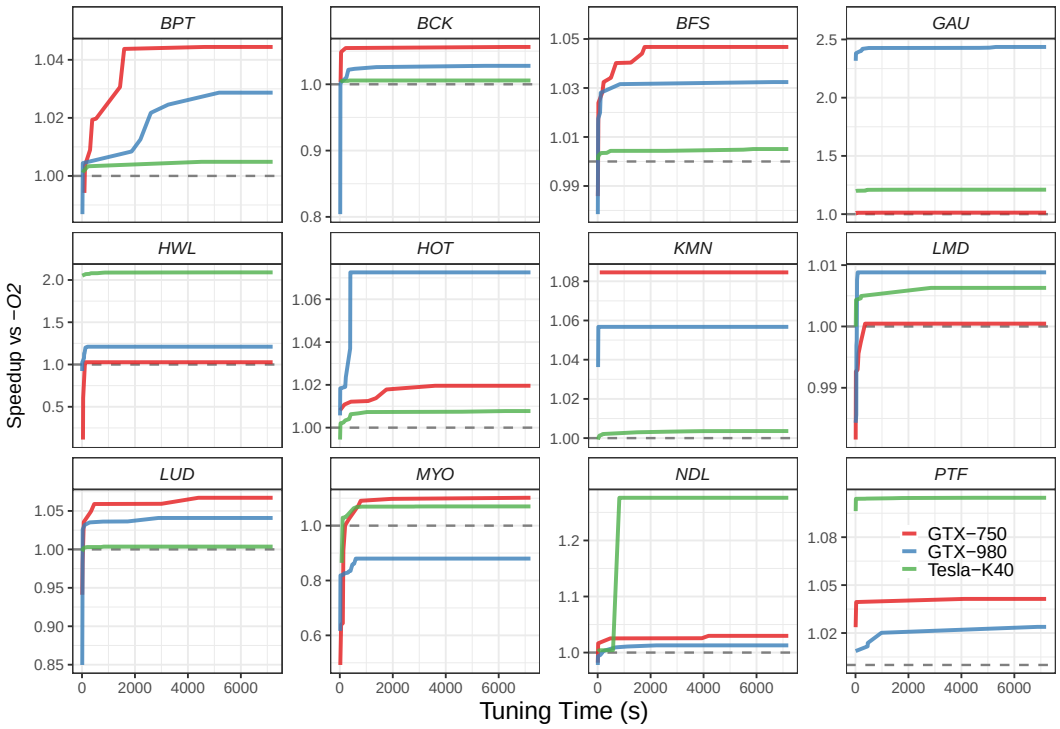

Figure 10.5: Mean speedup over 30 repetitions of the tuned solutions for Rodinia kernels versus the *opt-level* = 2 baseline, across two hours of tuning. Notice the difference in the *y*-axis scales for each panel

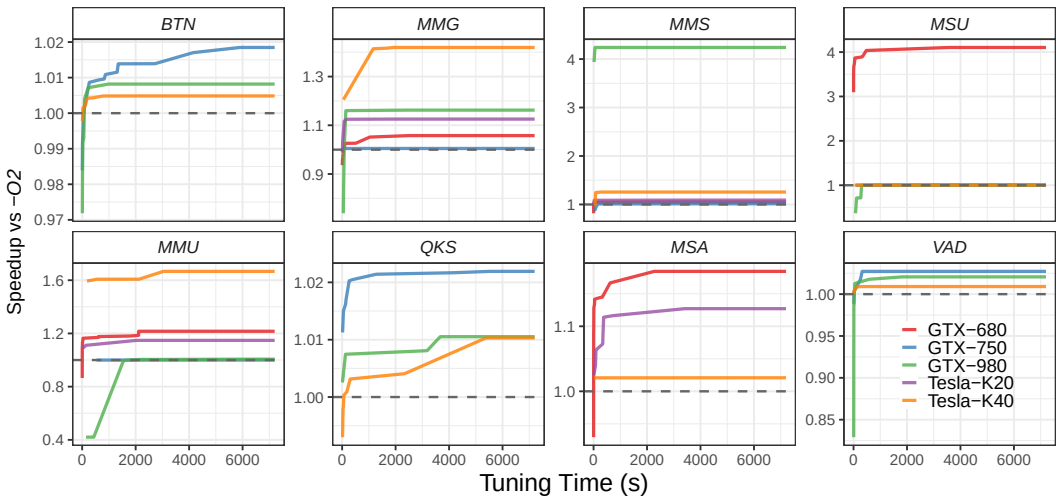

Figure 10.6: Mean speedup over 30 repetitions of the tuned solutions for the kernels we implemented versus the *opt-level* = 2 baseline, across two hours of tuning. Notice the difference in the *y*-axis scales for each panel

Flag	Cluster 0 (17%)	Cluster 1 (83%)
no-align-double	on	on
use_fast_math	on	on
preserve-relocs	off	off
relocatable-device-code	true	false
ftz	true	true
prec-div	true	false
prec-sqrt	true	true
fmad	false	false
allow-expensive-optimizations	false	true
gpu-architecture	sm_20	sm_50
def-load-cache	cv	ca
opt-level	1	3
maxrregcount	42	44.6

Table 10.4: Parameter clusters for all Rodinia problems in the GTX 750

10.4.3 Clustering Parameters found by Stochastic Experiments

We attempted to associate compilation parameters to kernels and GPUs using WEKA clustering algorithms [98]. Although we could not find significant relations for most kernels we detected that the *ftz=true* in MMS and the Compute Capabilities 3.0, 5.0 and 5.2 in GAU caused the speedups observed in the GTX 980 for these kernels. Table 10.4 shows clusters obtained with the K-means WEKA algorithm for autotuned parameter sets for the Rodinia Benchmark in the GTX 750. Unlike most clusters found for all GPUs and problems, these clusters did not contain an equal number of instances. The difficulty of finding associations between compiler optimizations, kernels and GPUs justifies the autotuning of compiler parameters in the general case.

10.5 Assessing Parameter Significance with Screening

This study was done in 2015, the first year of this thesis. Since at that time we did not have experience with the methods for statistical modeling which we later studied, and since the measurement time for each kernel was sufficiently small, we did not deal with the problem of building a performance model for the CUDA compiler in this study, and originally attempted to find good compiler configurations using stochastic methods for function minimization. Since then, we performed a new experiment using screening and targeting the same problem, in an attempt to identify the flags responsible for any improvements we observe. In this section, before closing the chapter, we will present and discuss this new screening experiment.

The data in this section has not yet been published at the time of the writing of this thesis. We designed and ran screening experiments for the CUDA compiler, adding the parameters described in Table 10.5 to the search space defined in Table 10.1. We no longer had access to the GPUs from the testbed described in the previous section, and we targeted the two GPUs described in Table 10.6.

Table 10.5: Flags added to the search space from Table 10.1

Flag	Description
`force-store-cache`	Force specified cache modifier on global/generic store. **Step**: PTX **Values**: `cs`, `cg`
`force-load-cache`	Replaces `def-load-cache`, force specified cache modifier on global/-generic load. **Step**: PTX **Values**: `cs`, `cg`
`optimize`	Specificy optimization level for host code. **Step**: NVCC **Values**: `2`, `3`

For this experiment we picked *Heart Wall* (HWL), *Gaussian Elimination* (GAU), and *Needleman-Wunsch* (NDL), the three Rodinia [81] kernels for which we found the largest speedups in the last study. Table 10.3 gives more details on each kernel. We constructed a Plackett-Burman design for the 15 parameters in the search space, defining a minimum and maximum value for each parameter. For categorical parameters we fixed two arbitrary levels, for the optimization level parameters we picked the equivalents of *-O2* and *-O3* for both host and device code, and for the parameter that controls the number of available registers, we picked the largest number and the midpoint.

Table 10.6: Hardware specifications of the GPU architectures targeted in the screening experiments

Model	C.C.	Global Memory	Bus	Bandwidth	L2	Cores	Clock
Quadro M1200	5.0	4 GB	128-bit	80.19 GB/s	2 MB	640	1148 Mhz
Titan X	5.2	12 GB	384-bit	336.5 GB/s	3 MB	3072	1075 Mhz

The Plackett-Burman design for 15 2-level factors contains 20 experiments, and we performed 20 repetitions of each parameter assignment for each of the three kernels. The top and bottom panels of Figure 10.7 show for each target GPU the means and 95% confidence intervals for the coefficients of a linear model fit to the screening measurements, which represent the main effect estimates for each factor.

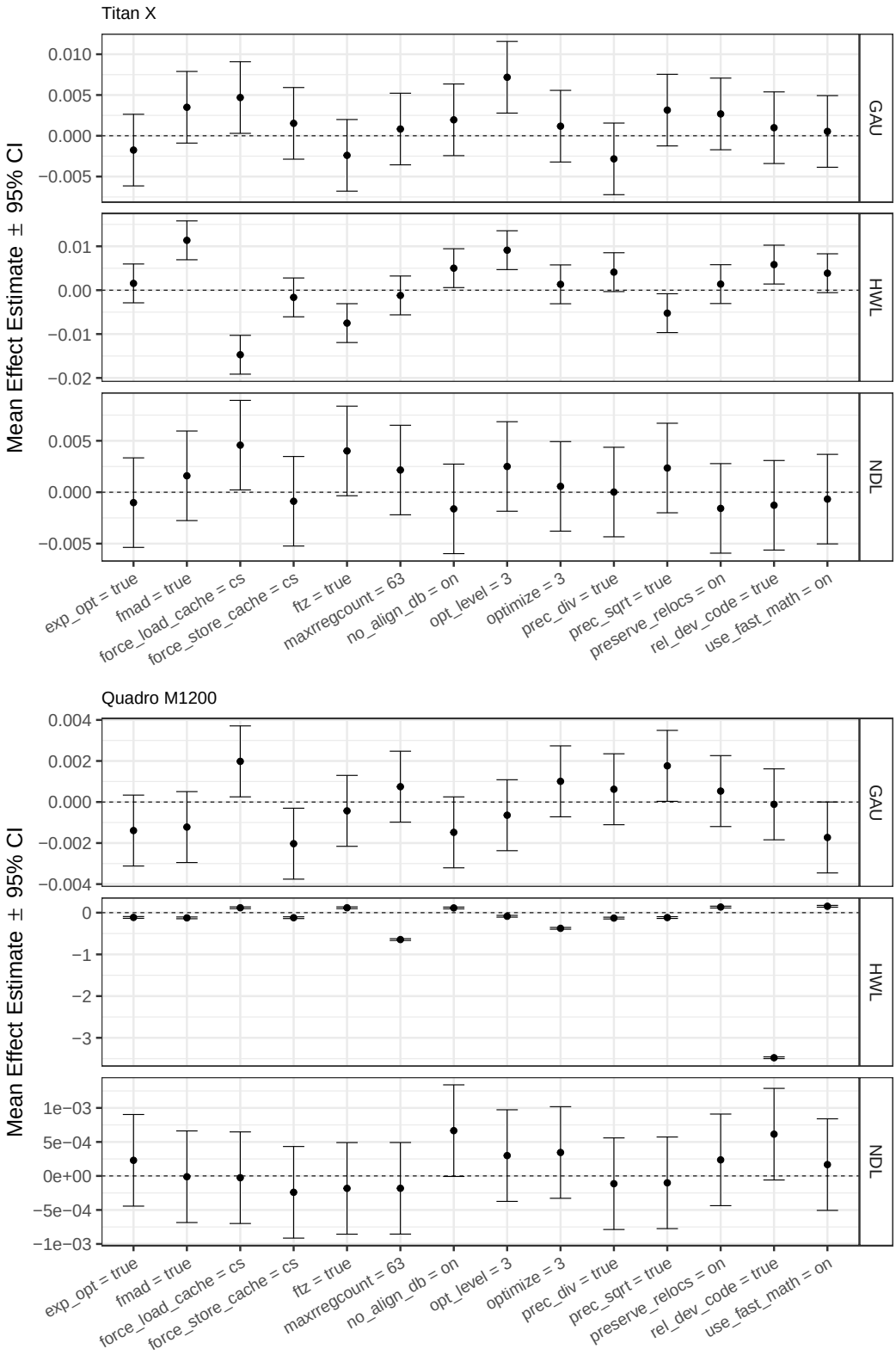

Figure 10.7: Main effects estimates and 95% confidence intervals for the *high level* of the 15 factors in the Plackett-Burman design, for target kernels measured on the Titan X and Quadro M1200 GPUs

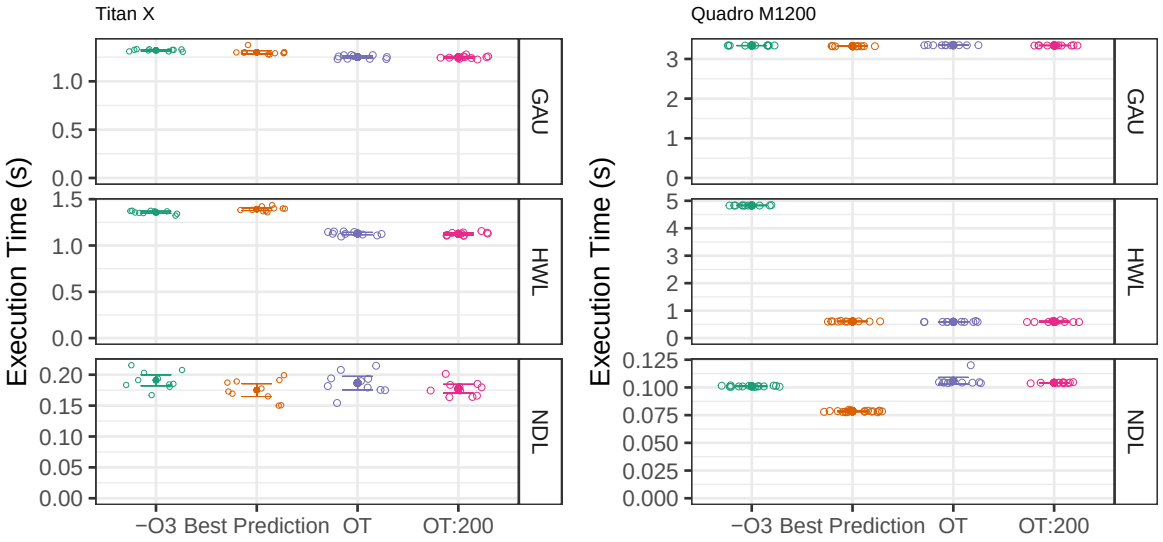

Figure 10.8: Execution time of baseline and model-predicted NVCC flag configurations for three Rodinia kernels on the Titan X and Quadro M1200 GPUs. Circles mark each of the 20 measurements of each flag configuration, filled dots and whiskers mark the mean and 95% confidence intervals

Note that because the Plackett-Burman design matrix is orthogonal the confidence intervals for the effect mean estimates all have the same size. In opposition to the exploration of stochastic methods, this well-balanced design allows the detection of factor main effects, which are mostly small, except in the HWL kernel on the Quadro GPU, where two factors present large effects. The linear models we built for each kernel using screening results do not account for interactions between factors, and thus their predictions favor turning on the factors whose mean estimates are negative and have confidence intervals that do not cross zero.

We generated a large set of points in the CUDA parameters search space and picked the points for which the models predict the smallest execution time. Figure 10.8 shows the performances of the points picked by the models and the baseline for comparison, for each Rodinia kernel. The baseline for comparison consisted of using only the PTX-stage *opt-level=3* high-level compiler optimization. There was no significant difference in performance between second and third levels of *opt-level* in these experiments.

The left panel shows results for the Titan X GPU, where we found small speedups for the GAU and NDL kernels, but small slowdowns for HWL. The right panel shows results for the Quadro M1200 GPU, where we found statistically significant but small speedups for GAU and NDL, and 8 times speedup for the HWL kernel. The speedups obtained using screening are highlighted in Figure 10.9.

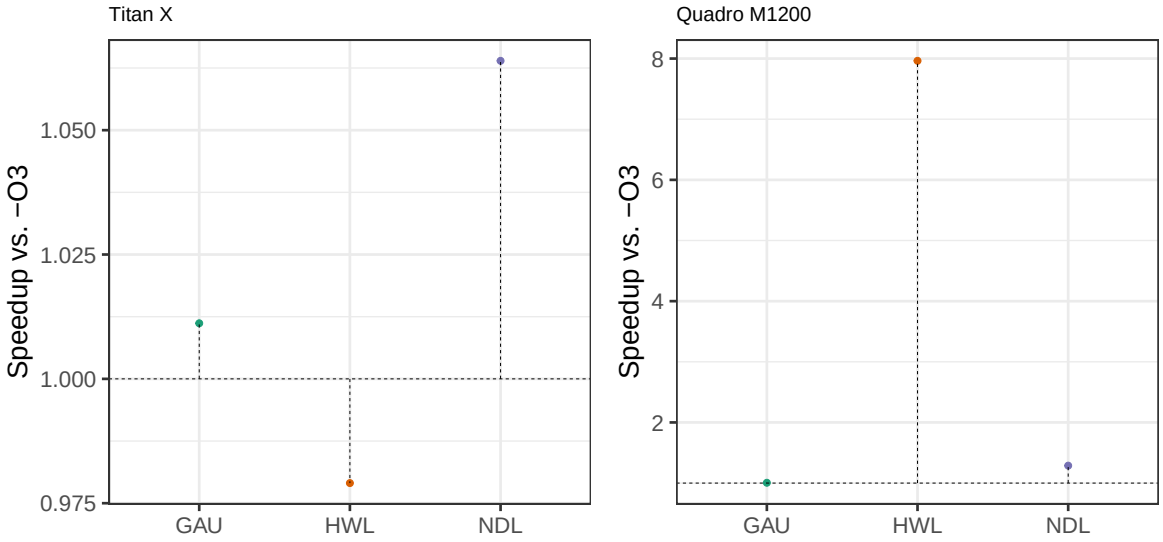

Figure 10.9: Mean speedups found using the predictions of the screening model, for three Rodinia kernels on the Titan X and Quadro M1200 GPUs. The horizontal dashed lines mark the PTX-stage `opt-level=3` comparison baseline

10.6 Summary

This study provides evidence that it is possible to improve the performance of GPU kernels with by selecting CUDA compiler parameters and flags. The inconclusive clustering attempts for the data obtained from heuristic and stochastic explorations of the search spaces emphasize the importance of well-designed experiment. Later experiments with screening reproduced the expressive speedups found for some kernels in different GPUs, while simultaneously allowing the identification of the flags responsible for performance improvements. Future work in this direction will explore the impact of adding host compiler optimizations and kernel-specific parameters. In this larger search space it would be more interesting to employ the more flexible Optimal Design methods we discuss in Chapters 12 and 13.

Chapter 11

Stochastic Methods for High-Level Synthesis FPGA Kernels

A Field-Programmable Gate Array (FPGA) is a reprogrammable circuit that enables writing specialized hardware specifications for a variety of applications without changing the underlying chip. In recent years High Performance Computing tasks that require low power and latency have increasingly switched to FPGAs. Hardware specifications are written using low-level languages such as Verilog and VHDL, creating a challenge for software engineers to leverage FPGA capabilities. Developing strategies to decrease the effort required to program FPGAs is becoming more relevant with the overspread adaptation of FPGAs for data centers [99, 100, 101, 102, 103, 104], with direct vendor backing [105, 106].

Essential support for software engineers can be provided by High-Level Synthesis (HLS), where hardware descriptions are generated from high-level code. HLS compilers intend to lower the complexity of hardware design and have become increasingly valuable as part of the FPGA design workflow, with support from vendor HLS tools [107, 108] for C/C++ and OpenCL. The benefits of higher-level abstractions often come with the cost of decreased performance, making FPGAs less viable as accelerators. Thus, optimizing HLS still requires domain expertise and exhaustive or manual exploration of design spaces and configurations.

High-Level Synthesis is a challenging problem, and a common strategy for its solution involves the divide-and-conquer approach [109]. The most important sub-problems to solve are *scheduling*, where operations are assigned to specific clock cycles, and *binding*, where operations are assigned to specific hardware functional units, which can be shared between operations. LegUp [110] was an initially open-source HLS tool, that has since gone paid and closed-source [111], and was implemented as a compiler pass for the LLVM Compiler Infrastructure [112]. LegUp receives code in LLVM's intermediate representation as input and produces as output a hardware description in Verilog, exposing configuration parameters of its HLS process, which are set with a configuration file.

This chapter presents our implementation of an autotuner for LegUp HLS parameters, also using the OpenTuner framework, which we published at ReConFig in 2017 [3]. The autotuner we implemented in this study targeted 8 hardware metrics obtained from Altera Quartus, for applications of the CHStone HLS benchmark suite [113] targeting the Intel StratixV FPGA. The program whose configurations are explored in this study is the LegUp HLS compiler, and the search space is composed of approximately 10^{126} possible combinations of HLS parameters.

One of the obstacles we faced was the impossibility of making accurate predictions for hardware metrics from a set of HLS parameters, or even for the generated Verilog, requiring that we run the autotuner using the metrics reported by the lengthier process of complete hardware synthesis instead. To mitigate the extra time cost, we implemented a virtualized deployment workflow which enabled launching several distributed tuning runs at the same time. We present data showing that the autotuner using stochastic methods from OpenTuner found optimized HLS parameters for CHStone applications that decreased the *Weighted Normalized Sum* (*WNS*) of hardware metrics by up to 21.5%, in relation to the default LegUp configuration.

It takes a considerable time to synthesize hardware from specifications in hardware description languages, ranging from minutes to hours. The process of generating those specifications from high-level C code is much faster in comparison, taking only seconds. Section 11.2 describes in more detail the High-Level Synthesis process.

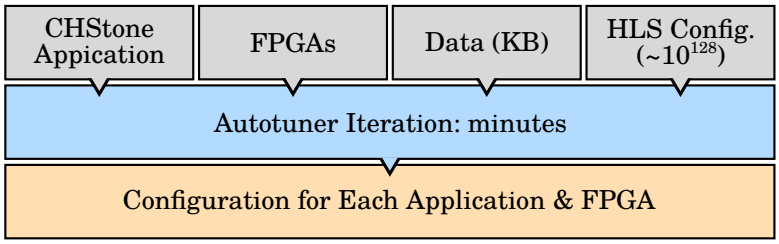

Figure 11.1: Autotuner representation and time scale of the experiments

Figure 11.1 shows a representation of our autotuner for this experiment, and highlights the time scales involved. The autotuner, represented by the light blue box, receives as input a CHStone kernel, a target FPGA, input data for the kernel in the order of Kilobytes and a search space composed of LegUp's HLS parameters. The autotuner outputs an HLS configuration for each kernel and FPGA.

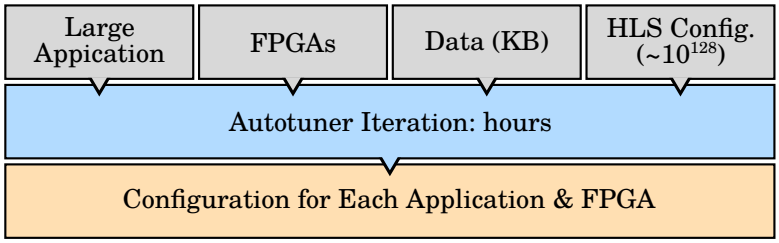

Figure 11.2: Autotuner representation and time scale of more complex FPGA applications

The kernels from CHStone presented further in Table 11.3 are very simple benchmark kernels, and generating hardware for them takes time in the order of minutes. Generating hardware for more complex FPGA applications can take several hours. Figure 11.2 shows a representation and time scale for an autotuner that targets complex FPGA applications. Each iteration now takes several hours, and limits the configurations that can be tested per hour. We worked around this limitation in this experiment by implementing a virtualized autotuner using Docker containers, which enabled parallel measurements of different configurations.

The rest of this chapter is organized as follows. Section 11.1 discusses the background on HLS tools and autotuning. Section 11.2 presents the search space defined by LegUp HLS parameters and the autotuner we implemented using OpenTuner. Section 11.3 introduces the optimization scenarios balancing different hardware metrics we targeted in this study, discusses the HLS kernels we optimized, and presents the settings in which the experiments were performed. Section 11.4 presents and discusses the results on each scenario. Finally, Section 11.5 summarizes the chapter and discusses future work.

11.1 Autotuning High-Level Synthesis for FPGAs

In this section we discuss background work related to HLS tools, and autotuning for FPGAs.

11.1.1 Tools for HLS

Various research and vendor tools for High-Level Synthesis have been developed [107, 108]. Villareal *et al.* [114] implemented extensions to the Riverside Optimizing Compiler for Configurable Circuits (ROCCC), which also uses the LLVM compiler infrastructure, to add support for generating VHDL from C code. Implemented within GCC, GAUT [115] is an open-source HLS tool for generating VHDL from C/C++ code. Other HLS tools such as Mitrion [116], Impulse [117] and Handel [118] also generate hardware descriptions from C code. We refer the reader to the survey from Nane *et al.* [119] for a comprehensive analysis of recent approaches to HLS.

11.1.2 Autotuning for FPGAs

Recent work studies autotuning approaches for FPGA compilation. Xu et al. [36] uses distributed OpenTuner instances to optimize the compilation flow from hardware description to bitstream. They optimize configuration parameters from the Verilog-to-Routing (VTR) toolflow [120] and target frequency, wall-clock time and logic utilization. Huang *et al.* [121] study the effect of LLVM pass ordering and application in LegUp's HLS process, demonstrating the complexity of the search space and the difficulty of its exhaustive exploration. They exhaustively explore a subset of LLVM passes and target logic utilization, execution cycles,

frequency, and wall-clock time. Mametjanov {et al.} [34] propose a machine-learning-based approach to tune design parameters for performance and power consumption. Nabi and Vanderbauwhede [122] present a model for performance and resource utilization for designs based on an intermediate representation.

11.2 Autotuner and Search Space for the LegUp HLS Compiler

This section describes our autotuner implementation, the LegUp HLS parameters selected for tuning, and the autotuning metrics used to measure the quality of HLS configurations.

11.2.1 Autotuner

We implemented our autotuner with OpenTuner [1], using ensembles of search techniques to find an optimized selection of LegUp [123] HLS parameters, according to our cost function, for 11 of the CHStone [113] kernels.

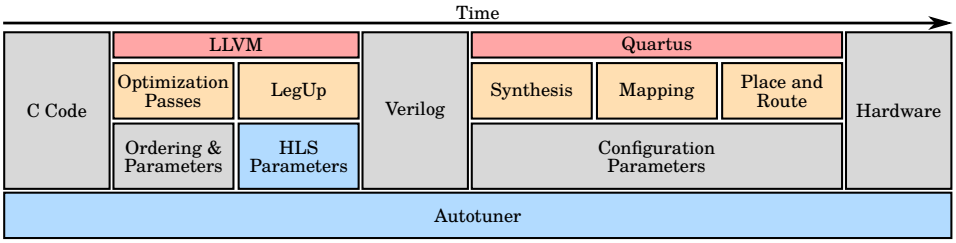

Figure 11.3: High-Level Synthesis compilation process. The autotuner search space at the HLS stage is highlighted in blue

Figure 11.3 shows the steps to generate a hardware description from C code. It also shows the Quartus steps to generate bitstreams from hardware descriptions and to obtain the hardware metrics we targeted. Our autotuner used LegUp's HLS parameters as the search space, but it completed the hardware generation process to obtain metrics from Quartus, as represented by the blue boxes in Figure 11.3.

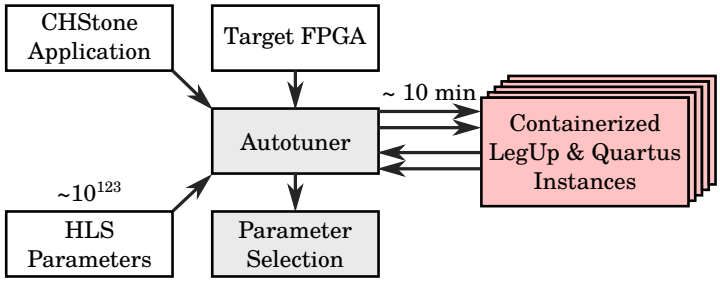

Figure 11.4: Autotuner Setup

Figure 11.4 shows our setup using Docker containers running LegUp and Quartus. This virtualization setup enabled portable dependency management and can be used to run exper-

iments in distributed environments. The arrows coming from the autotuner to the containers represent the flow of new configurations generated by search techniques, and the arrows coming from the containers to the autotuner represent the flow of measurements for a set of parameters. For CHStone kernels measurements take approximately 10 minutes to complete, and the majority of this time is spent in Quartus's synthesis, mapping, and place and route steps.

11.2.2 High-Level Synthesis Parameters

We selected an extensive set of LegUp High-Level Synthesis parameters, shown partially in Table 11.1. Each parameter in the first two rows of Table 11.1 has an 8, 16, 32 and 64 bit variant. *Operation Latency* parameters define the number of clock cycles required to complete a given operation when compiled with LegUp. *Zero-latency* operations can be performed in a single clock cycle. *Resource Constraint* parameters define the number of times a given operation can be performed in a clock cycle. *Boolean or Multi-Valued* parameters are used to set various advanced configurations. For example, the $enable_{patternsharing}$ parameter can be set to enable resource sharing for patterns of computational operators, as is described by Hadjis *et al.* [124]. For a complete list and description of each parameter, please refer to LegUp's official documentation [125].

Type	Parameters
Operation Latency	**altfp_**[divide, truncate, fptosi, add, subtract, multiply, extend, sitofp], **unsigned_**[multiply, divide, add, modulus], **signed_**[modulus, divide, multiply, add, **comp_**[o, u]], [local_mem, mem]**_dual_port**, **reg**
Resource Constraint	**signed_**[divide, multiply, modulus, add], **altfp_**[multiply, add, subtract, divide], **unsigned_**[modulus, multiply, add, divide], [shared_mem, mem]**_dual_port**
Boolean or Multi-value	**pattern_share_**[add, shift, sub, bitops], **sdc_**[multipump, no_chaining, priority], **pipeline_**[resource_sharing, all], **ps_**[min_size, min_width, max_size, bit_diff_threshold], **mb_**[minimize_hw, max_back_passes], **no_roms**, **multiplier_no_chain**, **dont_chain_get_elem_ptr**, **clock_period**, **no_loop_pipelining**, **incremental_sdc**, **disable_reg_sharing**, **set_combine_basicblock**, **enable_pattern_sharing**, **multipumping**, **dual_port_binding**, **modulo_scheduler**, **explicit_lpm_mults**

Table 11.1: Subset of All Autotuned LegUP HLS Parameters

11.2.3 HLS Autotuning Metrics

To obtain values for hardware metrics we needed to perform the synthesis, mapping and place and route steps. We used Quartus to do so, and selected 8 hardware metrics reported by Quartus to compose our cost or fitness function. From the fitter summary we obtained 6 metrics. *Logic Utilization* (*LUT*) measures the number of logic elements and is composed of Adaptive Look-Up Table (ALUTs), memory ALUTs, logic registers or dedicated logic registers. The *Registers* (*Regs.*), *Virtual Pins* (*Pins*), *Block Memory Bits* (*Blocks*), *RAM Blocks* (*BRAM*) and *DSP Blocks* (*DSP*) metrics measure the usage of the resources indicated by their names.

From the timing analysis we obtained the *Cycles* and *FMax* metrics, used to compute the *Wall-Clock Time* metric. This metric composed the cost function, but *Cycles* and *FMax* were not individually used. We chose to do that because all of our hardware metrics needed to be minimized except for *FMax*, and computing *Wall-Clock Time* instead solved that restriction. The *Wall-Clock Time* wct is computed by $wct = Cycles \times (\alpha/FMax)$, where $\alpha = 10^6$ because *FMax* is reported in MHz.

The objective function used by the autotuner to evaluate sets of HLS parameters is written

$$f(\mathbf{m}, \mathbf{w}) = \frac{\sum_{\substack{m \in \mathbf{m} \\ w \in \mathbf{w}}} w\left(\frac{m}{m^0}\right)}{\sum_{w \in \mathbf{w}} w}, \tag{11.1}$$

and computes a *Weighted Normalized Sum* (*WNS*) of the measured metrics $m \in \mathbf{m}$, where $\mathbf{m}$ is a vector with measurements of the 8 hardware metrics described previously. Each weight $w \in \mathbf{w}$ corresponds to one of the scenarios in Table 11.2. A value is computed for each metric m in relation to an initial default value m^0, measured for each metric in a given kernel, using LegUp's default configuration. For a given set measurement vector $\mathbf{x}$, $f(\mathbf{x}, \mathbf{w}) = 1.0$ means that there was no improvement relative to the default HLS configuration.

11.3 Target Optimization Scenarios and HLS Kernels

This section describes the optimization scenarios, the CHStone kernels, and the experimental settings.

11.3.1 Optimization Scenarios

Table 11.2 shows the assigned weights in our 4 optimization scenarios. The *Area*-targeting scenario assigns low weights to wall-clock time metrics. The *Performance and Latency* scenario assigns high weights to wall-clock time metrics and also to the number of registers used. The *Performance* scenario assigns low weights to area metrics and cycles, assigning a high weight only to frequency. The *Balanced* scenario assigns the same weight to every metric. The weights assigned to the metrics that do not appear on Table 11.2 are always 1. The weights are integers and powers of 2.

We compared results when starting from a *Default* configuration with the results when starting at a *Random* set of parameters. The default configuration for the StratixV was provided by LegUp and the comparison was performed in the *Balanced* optimization scenario.

Table 11.2: Weights for Optimization Scenarios
(*High* = 8, *Medium* = 4, *Low* = 2)

Metric	*Area*	*Perf. and Lat*	*Performance*	*Balanced*
LUT	High	Low	Low	Medium
Registers	High	High	Medium	Medium
BRAMs	High	Low	Low	Medium
DSPs	High	Low	Low	Medium
FMax	Low	High	High	Medium
Cycles	Low	High	Low	Medium

11.3.2 Kernels

To test and validate our autotuner we used 11 kernels from the CHStone HLS benchmark suite [113]. CHStone kernels are implemented in the C language and contain inputs and previously computed outputs, allowing for correctness checks to be performed for all kernels.

Table 11.3: Autotuned CHStone Kernels

Kernel	Short Description
blowfish	Symmetric-key block cypher
aes	Advanced Encryption Algorithm (AES)
adpcm	Adaptive Differential Pulse Code Modulation dec. and enc.
sha	Secure Hash Algorithm (SHA)
motion	Motion vector decoding from MPEG-2
mips	Simplified MIPS processor
gsm	Predictive coding analysis of systems for mobile comms.
dfsin	Sine function for double-precision floating-point numbers
dfmul	Double-precision floating-point multiplication
dfdiv	Double-precision floating-point division
dfadd	Double-precision floating-point addition

Table 11.3 provides short descriptions of the 11 CHStone kernels we used. We were not able to compile the *jpeg* CHStone kernel, so did not use it. All experiments targeted the *Intel StratixV 5SGXEA7N2F45C2* FPGA.

11.3.3 Experiments

We performed 10 tuning runs of 1.5h for each kernel. Section 11.4 presents the mean relative improvements for each kernel and individual metric. The code needed to run the experiments and generate the figures, as well as the implementation of the autotuner and all data we generated, is open and hosted at GitHub [126].

The experimental settings included Docker for virtualization and reproducibility, *LegUp v4.0*, *Quartus Prime Standard Edition v16.0*, and *CHStone* [113]. All experiments were performed on a machine with two *Intel Xeon CPU E5-2699 v3* with 18 *x86_64* cores each, and 503GB of

RAM. The instructions and the code to reproduce the software experimental environment are open and hosted at GitHub [127].

11.4 Performance Improvements using Stochastic Methods

This section presents summaries of the results from 10 autotuning runs of 1.5*h* in the scenarios from Table 11.2. Results are presented in *heatmaps* where each row has one of the 11 CHStone kernels in Table 11.3 and each column has one of the 8 hardware metrics and their *Weighted Normalized Sum* (*WNS*) as described in Section 11.2.3.

Cells on heatmaps show the ratio of tuned to initial values of a hardware metric in a CHStone kernel, averaged over 10 autotuning runs. The objective of the autotuner is to minimize all hardware metrics, except for *FMax*, whose inverse is minimized. Cell values less than 1.0 always mark an improvement on a given metric. Darker blue squares in the following heatmaps mark improvements, and darker red squares mark worse values in relation to the starting point.

	LUTs	Pins	BRAM	Regs	Blocks	Cycles	DSP	FMax
aes	0.79	0.91	1.00	0.56	1.00	0.47	1.00	1.12
adpcm	0.68	1.13	1.00	0.54	1.00	0.56	0.60	0.98
sha	1.00	1.03	1.00	0.82	1.00	0.55	1.00	0.89
motion	1.02	1.00	0.60	0.85	1.00	0.57	1.00	0.94
mips	1.00	0.93	1.00	0.44	1.00	0.45	0.98	1.24
gsm	0.83	1.17	1.00	0.48	1.00	0.56	0.52	0.99
dfsin	0.79	0.97	1.00	0.61	1.00	0.60	1.49	1.53
dfmul	1.00	1.06	0.90	0.47	0.90	0.47	1.31	1.10
dfdiv	0.83	1.07	0.80	0.73	0.80	0.65	1.32	1.49
dfadd	1.00	0.94	1.00	0.82	1.00	0.71	1.00	0.93
blowfish	–	–	–	–	–	–	–	–

Figure 11.5: Comparison of the absolute values for Random and Default starting points in the Balanced scenario

Figure 11.5 compares the ratios of absolute values for each hardware metric for *Default* and *Random* starts, in the *Balanced* scenario. Cell values less than 1.0 mean that the *Default* start achieved smaller absolute values that the *Random* start. Cells with "–" mean that the *Default* start could not find a set of HLS parameters that produced valid output during any of the 1.5*h* tuning runs. The *Default* start found better values for most metrics.

The *Random* start found better values for *DSP*, *Pins* and *FMax* for some kernels. For example, it found values 49% smaller, 6% smaller and 53% larger for *DSP*, *Pins* and *FMax*, respectively, for the *dfsin* kernel. The *Default* start found better values for *Regs* and *Cycles* for all kernels. For example, it found values 53% smaller for *Regs* and *Cycles* for the *dfmul* kernel, and 56% and 55% smaller for *Regs* and *Cycles*, respectively, for the *mips* kernel.

The *Random* start found worst values in most cases because of the size of the search space and the stochastic nature of the explorations performed by OpenTuner methods. In such scenarios, knowing and leveraging a reasonably good starting configuration for a given

kernel is extremely important. The remaining results in this Section used one of the *Default* starting configurations provided by LegUp, specifically targeted to Stratix V boards.

Figure 11.6 shows the results for the *Balanced* scenario. These results are the baseline for evaluating the autotuner in other scenarios, since all metrics had the same weight. The optimization target was the *Weighted Normalized Sum* (*WNS*) of hardware metrics, but we were also interested in changes in other metrics, as their relative weights changed. In the *Balanced* scenario we expected to see smaller improvements of *WNS* due to the competition of concurrent improvements on every metric.

The autotuner found values of *WNS* 16% smaller for the *adpcm* and *dfdiv* kernels, and 15% smaller for *dfmul*. Even for the *Balanced* scenario it is possible to see that some metrics decreased while others decreased consistently over the 10 tuning runs. *FMax* and *DSP* had the larger improvements for most kernels, for example, 51% greater *FMax* in *adpcm* and 69% smaller *DSP* in *dfmul*. *Cycles*, *Regs* and *Pins* had the worst results in this scenario, with 34% larger *Cycles* in *dfdiv*, 15% larger *Regs* in *dfdiv* and 17% larger *Pins* in *gsm*. Other metrics had smaller improvements or no improvements at all in most kernels.

	WNS	LUTs	Pins	BRAM	Regs	Blocks	Cycles	DSP	FMax
aes	0.94	0.90	1.00	1.00	0.97	1.00	0.98	1.00	0.76
adpcm	0.84	0.73	1.13	1.00	0.91	1.00	1.05	0.63	0.49
sha	0.98	1.00	1.03	1.00	0.96	1.00	0.86	1.00	0.97
motion	0.98	0.97	1.00	1.00	0.99	1.00	0.95	1.00	0.95
mips	0.95	1.00	1.07	1.00	0.90	1.00	1.01	0.80	0.89
gsm	0.95	0.95	1.17	1.00	0.99	1.00	0.95	0.49	1.08
dfsin	0.93	1.03	1.03	1.00	1.05	1.00	1.07	0.65	0.73
dfmul	0.85	1.00	1.13	0.90	0.88	0.90	1.06	0.31	0.76
dfdiv	0.84	1.00	1.07	0.80	1.15	0.80	1.34	0.41	0.57
dfadd	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.97
blowfish	0.99	1.00	1.00	1.00	0.98	1.00	0.98	1.00	0.99

Figure 11.6: Relative improvement for all metrics in the *Balanced* scenario

	WNS	LUTs	Pins	BRAM	Regs	Blocks	Cycles	DSP	FMax
aes	0.96	0.92	1.00	1.00	0.93	1.00	0.97	1.00	0.86
adpcm	0.83	0.78	1.11	1.00	0.96	1.00	1.04	0.63	0.52
sha	0.96	1.00	1.06	1.00	0.84	1.00	0.83	1.00	1.08
motion	0.97	0.94	1.00	1.00	0.97	1.00	0.92	1.00	0.95
mips	0.91	1.00	1.06	1.00	0.89	1.00	1.00	0.72	0.84
gsm	0.89	0.83	1.06	1.00	0.86	1.00	0.89	0.82	1.14
dfsin	0.94	1.00	1.06	1.00	1.00	1.00	1.02	0.76	0.80
dfmul	0.82	1.00	1.06	1.00	0.90	1.00	1.21	0.27	0.86
dfdiv	0.77	1.00	1.22	0.67	1.03	0.67	1.62	0.28	0.77
dfadd	0.99	1.00	1.11	1.00	0.94	1.00	1.00	1.00	1.04
blowfish	0.99	1.00	1.00	1.00	0.97	1.00	0.91	1.00	1.01

Figure 11.7: Relative improvement for all metrics in the *Area* scenario

Figure 11.7 shows the results for the *Area* scenario. We believe that the greater coherence of optimization objectives is responsible for the greater improvements of *WNS* in the following scenarios. The autotuner found values of *WNS* 23% smaller for *dfdiv*, 18% smaller for *dfmul*, and smaller values overall in comparison with the *Balanced* scenario. Regarding individual metrics, the values for *FMax* were worse overall, with 14% smaller *FMax* in *gsm* and 62% greater *Cycles*, for example. As expected for this scenario, metrics related to area had better improvements than in the *Balanced* scenario, with 73% and 72% smaller *DSP* for *dfmul* and *dfdiv* respectively, 33% smaller *Blocks* and *BRAM* in *dfdiv* and smaller values overall for *Regs* and *LUTs*.

Figure 11.8 shows the results for the *Performance* scenario. The autotuner found values of *WNS* 23% smaller for *dfmul*, 19% smaller for *dfdiv*, and smaller values overall than in the *Balanced* scenario. *FMax* was the only metric with a *High* weight in this scenario, so most metrics had improvements close overall to the *Balanced* scenario. The values for *FMax* were

best overall, with better improvements in most kernels. For example, 41%, 30%, 44% and 37% greater *FMax* in *dfdiv*, *dfmul*, *dfsin* and *aes* respectively.

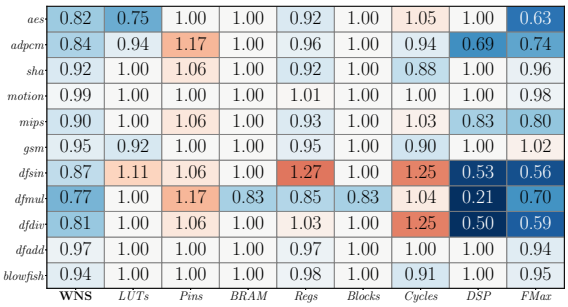

Figure 11.8: Relative improvement for all metrics in the *Performance* scenario

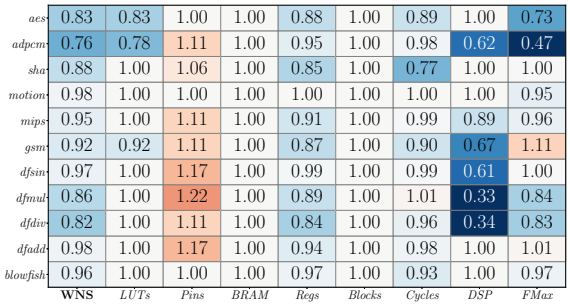

Figure 11.9: Relative improvement for all metrics in the *Performance and Latency* scenario

Figure 11.9 shows the results for the *Performance and Latency* scenario. The autotuner found values of *WNS* 24% smaller for *adpcm*, 18% smaller for *dfdiv*, and smaller values overall than in the *Balanced* scenario. *Regs*, *Cycles* and *FMax* had higher weights in this scenario, and also better improvements overall. For example, 16% and 15% smaller *Regs* in *dfdiv* and *sha* respectively, 23% and 11% smaller *Cycles* in *sha* and *aes* respectively, and 53% greater *FMax* in *adpcm*. Although *FMax* had the worst improvements in relation to the *Balanced* scenario, the *Wall-Clock Time* was still decreased by the smaller values of *Cycles*.

Figure 11.10 summarizes the average improvements on *WNS* in the 4 scenarios over 10 runs. Only the *Weighted Normalized Sum* of metrics directly guided optimization. With the exception of *dfadd* in the *Balanced* scenario, the autotuner decreased *WNS* for all kernels in all scenarios by 10% on average, and up to 24% for *adpcm* in the *Performance and Latency* scenario. The figure also shows the average decreases for each scenario.

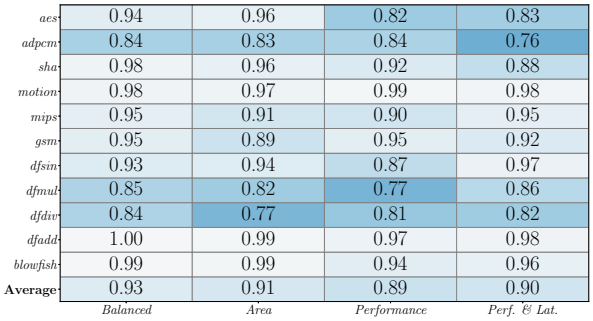

Figure 11.10: Relative improvement for **WNS** in all scenarios

11.5 Summary

This study highlights the importance of starting positions for an autotuner based on stochastic methods. This becomes more relevant as the size of the search space and the number

of targeted metrics increase. The flexibility of our virtualized approach is evidenced by the results for different optimization scenarios. Improvements in *WNS* increased when higher weights were assigned to metrics that express a coherent objective such as area, performance and latency. The improvements of metrics related to those objectives also increased.

Kernels with large measurement time still presented a challenge for the autotuning approach we used in this problem, even with distributed measurements, because the stochastic methods implemented in OpenTuner reach better results if longer explorations of the search space are performed. An Experimental Design approach such as the one we later developed and applied to other problems, as described in Chapter 13, would be ideal for this type of problem. Since the HLS toolchain used in this study became proprietary software and FPGA programming is still a very closed domain, requiring paid licenses and still expensive hardware, we did not have the opportunity to revisit this study with the Experimental Design methods we later studied.

Future work in this direction will study the impact of different starting points on the final tuned values in each optimization scenario, for example we could start tuning for *Performance* at the best autotuned *Area* value. We expected that starting positions tailored for each target kernel will enable the autotuner to find better *WNS* values faster. We will also apply this autotuning methodology to HLS tools that enable a fast prediction of metric values. These tools will enable the exploration of the trade-off between prediction accuracy and the time to measure an HLS configuration.

Chapter 12

Stochastic Methods, Experimental Design, and Gaussian Process Regression for a Laplacian GPU Kernel

This chapter introduces our Experimental Design approach to autotuning, and presents an application were we continued the exploration, started by Masnada [128], of performance optimization methods for the search space defined by the parameters of a Laplacian GPU kernel. The kernel was implemented using BOAST [26], a framework that enables writing and optimizing HPC applications using metaprogramming, and is publicly hosted on GitHub [129]. We targeted the *Nvidia K40c* GPU, and the objective function we minimized was the time to compute each pixel. We evaluated the performance of nine optimization methods for this kernel, including sampling strategies, stochastic methods, variations on linear regression, our Experimental Design approach, and Gaussian Process Regression. Our transparent and parsimonious Experimental Design approach achieved the most consistent results across the methods we tested, which motivated searching for a more comprehensive set of applications in which we could evaluate the performance of our approach. This more comprehensive study is described in Chapter 13. The initial steps of the studies presented in this chapter and the next were published together in the CCGRID conference [4].

The remainder of this chapter is organized as follows. Section 12.1 presents the target kernel and the associated search space exposed by the BOAST implementation. Section 12.2 presents our adaptation of a sequential approach to autotuning, starting with modeling assumptions about the search space and iteratively refining models based on ANOVA tests. Section 12.3 discusses our initial modeling hypotheses. Section 12.4 looks at a single run of our approach, linking the factor levels that were chosen at each step to ANOVA p-values, and comparing to the levels of the global minimum. Section 12.5 evaluates the performance of each method. Finally, Section 12.6 the discussion and concludes the chapter.

Figure 12.1: Edge-detection effect of a *Laplacian of Gaussian* filter

12.1 The Laplacian Kernel

The Laplacian Δf of a function f with inputs x and y is written

$$\Delta f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}, \tag{12.1}$$

and can be interpreted as providing a measure of how much the values of f in a neighborhood of a point (x, y) deviate, on average, from the value of $f(x, y)$. The discrete Laplacian operator is commonly used in image processing algorithms such as edge detection, where it is typically composed with smoothing filters. In the discrete setting the Laplacian of an image can be computed by a convolution filter, where a single kernel encodes the smoothing and Laplacian filters. Figure 12.1 shows the effect of applying a *Laplacian of Gaussian* convolution filter to detect edges in a picture.

12.1.1 BOAST Code and the OpenCL Kernel

Our Experimental Design approach can be applied to any autotuning domain that expresses optimization as a search problem, but the performance evaluations we present in this thesis were obtained in the domain of source-to-source transformation. Several frameworks, compilers, and autotuners provide tools to generate and optimize architecture-specific code [27, 26, 37, 130, 28]. We used BOAST [26] to generate code for GPUs by generating an OpenCL Laplacian kernel, optimizing parameters controlling vectorization, loop transformations, and data structure size and copying.

Figure 12.1.1 shows a Laplacian kernel naively written in C. In this code, the RGB channels of an image are stored a single *int* array, representing a color image. A three by three convolution kernel is applied to each color channel separately.

```
void kernel(int width, int height, uint8_t *source, uint8_t *destination){
    int i, j, d, ii, jj, tmp;
    for (j = 1; j < height - 1; j++) {
        for (i = 1; i < width - 1; i++) {
            // Process RGB components separately
            for (d = 0; d < 3; d ++) {
                tmp = 0;
                for(jj = -1; jj < 1; jj++) {
                    for(ii = -1; ii < 1; ii++) {
                        if(ii == 0 && jj == 0) {
                            tmp += 9 * source[d + (3 * (i + (width * j)))];
                        } else {
                            tmp -= source[d + (3 * ((i + ii) + (width * (j + jj))))];
                        }
                    }
                }
                // Clamp to valid pixel values
                destination[d + (3 * (i + (width * j)))] = tmp < 0 ? 0 :
                    (tmp > 255 ? 255 : tmp);
            }
        }
    }
}
```

Figure 12.2: A CPU Laplacian kernel written in C

For comparison, the BOAST Ruby script for generating a parameterized Laplacian kernel is partially shown in Figure 12.1.1. A sample of the resulting OpenCL kernel is shown in Figure 12.1.1. The difference in complexity between the naive implementation in Figure 12.1.1 and the parameterized generated version helps motivate the usage of automated optimization methods for autotuning.

```
def laplacian(options)
  default_options = {:x_component_number => 1,
                     :vector_length => 1,
                     :y_component_number => 1,
                     :temporary_size => 2,
                     :vector_recompute => false,
                     :load_overlap => false}
  # [...] (Omitted)
  # Begining of the generated procedure, based on the baseline
  p = Procedure("kernel", [psrc, pdst, width, height]) {
    # [...] (Omitted)
    # Using vector_length to compute the number of vectors
    vector_number = (x_component_number.to_f/vector_length).ceil
    total_x_size = vector_recompute ? vector_number *
                                      vector_length : x_component_number
    # [...] (Omitted)
    # Using y_component_number to compute offsets
    y_offset = y_component_number + 1
    # Controlling load_overlap
    if not load_overlap then
      total_load_window = total_x_size + 6
      tempload = []
      ranges = split_in_ranges(total_load_window, vector_length)
      ranges.each { |r|
        tempload.push( Int("tempload#{r.begin}_#{r.end}", :size => 1,
                           :vector_length => (r.end - r.begin + 1), :signed => false) )
      }
      decl *(tempload)
    else
      tempnn = (0..2).collect { |v_i|
        (0...vector_number).collect { |x_i|
          (0...(y_component_number+2)).collect { |y_i|
            Int("temp#{x_i}#{v_i}#{y_i}", :size => 1,
                :vector_length => vector_length, :signed => false)
          }
        }
      }
      decl *(tempnn.flatten)
    end
    # [...] (Omitted)
    # Using temporary_size to allocate data
    tempcnn = (0..2).collect { |v_i|
      (0...vector_number).collect { |x_i|
        (0...(y_component_number+2)).collect { |y_i|
          Int("tempc#{x_i}#{v_i}#{y_i}", :size => temporary_size,
              :vector_length => vector_length)
        }
      }
    }
    decl *(tempcnn.flatten)
    # [...] (Omitted)
    # Generate kernel and clamp to valid pixel values
    (0...vector_number).each { |v_i|
      (0...y_component_number).each { |y_i|
        pr rescnn[v_i][y_i] === - tempcnn[0][v_i][y_i] -
                                  tempcnn[1][v_i][y_i] -
                                  tempcnn[2][v_i][y_i] -
                                  tempcnn[0][v_i][y_i + 1] +
                                  tempcnn[1][v_i][y_i + 1] *
                                  "(#{temp_type})9" -
                                  tempcnn[2][v_i][y_i + 1] -
                                  tempcnn[0][v_i][y_i + 2] -
                                  tempcnn[1][v_i][y_i + 2] -
                                  tempcnn[2][v_i][y_i + 2]
        pr resnn[v_i][y_i] === clamp(rescnn[v_i][y_i],
                                     "(#{temp_type})0",
                                     "(#{temp_type})255",
                                     :returns => rescnn[v_i][y_i])
      }
    }
    # [...] (Omitted)
  }
  # Variable p contains the complete generated OpenCL code
  pr p
  k.procedure = p
  return k
end
```

Figure 12.3: Excerpts of the BOAST code, in Ruby, that generates the Laplacian OpenCL kernel. The code is publicly hosted at GitHub [131]

```
__kernel void kernel(const __global uchar * psrc, __global uchar * pdst,
                     const int width, const int height){
    int y, x, w;
    x = (get_global_id(0)) * (1);
    y = (get_global_id(1)) * (3);
    w = (width) * (3);
    x = (x < 3 ? 3 : (x > w - (11) ? w - (11) : x));
    y = (y < 1 ? 1 : (y > height - (4) ? height - (4) : y));
    uchar8 tempload0_7, tempload8_11, tempload12_13,
           res00, res01, res02, tempc000;
    int8 tempc001, tempc002, tempc003, tempc004, tempc010, tempc011, tempc012,
         tempc013, tempc014, tempc020, tempc021, tempc022, tempc023, tempc024,
         resc00, resc01, resc02;
    tempload0_7 = vload8(0, &psrc[x + -3 + (w) * (y + -1)]);
    tempload8_11 = vload4(0, &psrc[x + 5 + (w) * (y + -1)]);
    tempload12_13 = vload2(0, &psrc[x + 9 + (w) * (y + -1)]);
    tempc000 = convert_int8( (uchar8)(tempload0_7.s01234567) );
    tempc010 = convert_int8( (uchar8)(tempload0_7.s345,tempload0_7.s67,tempload8_11.s012) );
    tempc020 = convert_int8( (uchar8)(tempload0_7.s67,tempload8_11.s0123,
                                      tempload12_13.s01) );
    tempload0_7 = vload8(0, &psrc[x + -3 + (w) * (y + 0)]);
    tempload8_11 = vload4(0, &psrc[x + 5 + (w) * (y + 0)]);
    tempload12_13 = vload2(0, &psrc[x + 9 + (w) * (y + 0)]);
    tempc001 = convert_int8( (uchar8)(tempload0_7.s01234567) );
    tempc011 = convert_int8( (uchar8)(tempload0_7.s345,tempload0_7.s67,tempload8_11.s012) );
    tempc021 = convert_int8( (uchar8)(tempload0_7.s67,tempload8_11.s0123,
                                      tempload12_13.s01) );
    tempload0_7 = vload8(0, &psrc[x + -3 + (w) * (y + 1)]);
    tempload8_11 = vload4(0, &psrc[x + 5 + (w) * (y + 1)]);
    tempload12_13 = vload2(0, &psrc[x + 9 + (w) * (y + 1)]);
    tempc002 = convert_int8( (uchar8)(tempload0_7.s01234567) );
    tempc012 = convert_int8( (uchar8)(tempload0_7.s345,tempload0_7.s67,tempload8_11.s012) );
    tempc022 = convert_int8( (uchar8)(tempload0_7.s67,tempload8_11.s0123,
                                      tempload12_13.s01) );
    tempload0_7 = vload8(0, &psrc[x + -3 + (w) * (y + 2)]);
    tempload8_11 = vload4(0, &psrc[x + 5 + (w) * (y + 2)]);
    tempload12_13 = vload2(0, &psrc[x + 9 + (w) * (y + 2)]);
    tempc003 = convert_int8( (uchar8)(tempload0_7.s01234567) );
    tempc013 = convert_int8( (uchar8)(tempload0_7.s345,tempload0_7.s67,
                                      tempload8_11.s012) );
    tempc023 = convert_int8( (uchar8)(tempload0_7.s67,tempload8_11.s0123,
                                      tempload12_13.s01) );
    tempload0_7 = vload8(0, &psrc[x + -3 + (w) * (y + 3)]);
    tempload8_11 = vload4(0, &psrc[x + 5 + (w) * (y + 3)]);
    tempload12_13 = vload2(0, &psrc[x + 9 + (w) * (y + 3)]);
    tempc004 = convert_int8( (uchar8)(tempload0_7.s01234567) );
    tempc014 = convert_int8( (uchar8)(tempload0_7.s345,tempload0_7.s67,tempload8_11.s012) );
    tempc024 = convert_int8( (uchar8)(tempload0_7.s67,tempload8_11.s0123,
                                      tempload12_13.s01) );
    // Kernel computation
    resc00 =  -(tempc000) - (tempc010) - (tempc020) - (tempc001) + (tempc011) *
        ((int)9) - (tempc021) - (tempc002) - (tempc012) - (tempc022);
    res00 = convert_uchar8( clamp(resc00, (int)0, (int)255) );
    resc01 =  -(tempc001) - (tempc011) - (tempc021) - (tempc002) + (tempc012) *
        ((int)9) - (tempc022) - (tempc003) - (tempc013) - (tempc023);
    res01 = convert_uchar8( clamp(resc01, (int)0, (int)255) );
    resc02 =  -(tempc002) - (tempc012) - (tempc022) - (tempc003) + (tempc013) *
        ((int)9) - (tempc023) - (tempc004) - (tempc014) - (tempc024);
    res02 = convert_uchar8( clamp(resc02, (int)0, (int)255) );
    vstore8( res00, 0, &pdst[x + 0 + (w) * (y + 0)] );
    vstore8( res01, 0, &pdst[x + 0 + (w) * (y + 1)] );
    vstore8( res02, 0, &pdst[x + 0 + (w) * (y + 2)] );
}
```

Figure 12.4: A sample OpenCL Laplacian kernel generated by BOAST

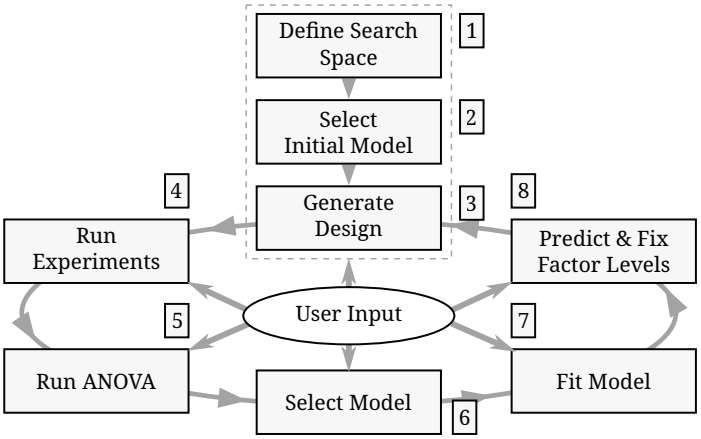

Figure 12.5: Overview of the ED approach to autotuning we implemented

12.2 A Transparent and Parsimonious Approach to Autotuning using Optimal Design

In this section we discuss in detail our iterative, transparent, and parsimonious Experimental Design approach to autotuning. Figure 12.5 presents an overview. In step 1 we define the factors and levels that compose the search space of the target problem, in step 2 we select an initial performance model, and in step 3 we generate an experimental design. We run the experiments in step 4 and then, as we discuss in the next section, we identify significant factors with an ANOVA test in step 5. This enables selecting and fitting a new performance model in steps 6 and 7. The new model is used in step 8 for predicting levels for each significant factor. We then go back to step 3, generating a new design for the new problem subspace with the remaining factors. Informed decisions made by the user at each step guide the outcome of each iteration.

Step 1 of our approach is to define target factors and which of their levels are worth exploring. Then, the user must select an initial performance model in step 2. Compilers typically expose many 2-level factors in the form of configuration flags, and the performance model for a single flag can only be a linear term, since there are only 2 values to measure. Interactions between flags and numerical factors such as block sizes in CUDA programs or loop unrolling amounts are also common. Deciding which levels to include for these kinds of factors requires more careful analysis. For example, if we suspect the performance model has a quadratic term for a certain factor, the design should include at least three factor levels. The ordering between the levels of other compiler parameters, such as *-O(0,1,2,3)*, is not obviously translated to a number. Factors like these are named *categorical*, and must be treated differently when constructing designs in step 3 and analyzing results in step 5.

After the design is constructed in step 3, we run each selected experiment in step 4. This step can run in parallel since experiments are independent. Not all target programs run successfully in their entire input range, making runtime failures common in this step. The user

can decide whether to construct a new design using the successfully completed experiments or to continue to the analysis step if enough experiments succeed.

After running the ANOVA test in step 5, the user should apply domain knowledge to analyze the ANOVA table and determine which factors are significant. Certain factors might not appear significant and should not be included in the regression model. Selecting the model after the ANOVA test in step 6 also benefits from domain knowledge.

A central assumption of ANOVA is the *homoscedasticity* of the response, which can be interpreted as requiring the observed error on measurements to be independent of factor levels and of the number of measurements. Fortunately, there are statistical tests and corrections for lack of homoscedasticity. Our approach uses the homoscedasticity check and correction by power transformations from the `car` package [132] of the `R` language.

We fit the selected model to our design's data in step 7, and use the fitted model in step 8 to find levels that minimize the response. The choice of the method used to find these levels depends on factor types and on the complexity of the model and search space. If factors have discrete levels, neighborhood exploration might be needed to find levels that minimize the response around predicted levels. Constraints might put predicted levels on an undefined or invalid region on the search space. This presents challenge, because the borders of valid regions would have to be explored.

In step 8 we also fix factor levels to those predicted to achieve best performance. The user can also decide the level of trust placed on the prediction at this step, by keeping other levels available. In step 8 we perform a reduction of problem dimension by eliminating factors and decreasing the size of the search space. If we identified significant parameters correctly, we will have restricted further search to better regions.

In this chapter and in Chapter 13 we evaluate the performance of this Experimental Design approach to autotuning in different applications. The next section discusses the implementation of a Laplacian kernel for GPUs and the resulting search space.

12.3 Building a Performance Model

This section describes the search space exposed by the BOAST Laplacian kernel and our initial modeling hypotheses.

12.3.1 Search Space

The factors and levels defining the search space are listed in Table 12.1. The complete search space contains 1.9×10^5 configurations, but removing invalid configurations, or configurations that fail at runtime, yields a search space with 2.3×10^4 configurations. The valid search space

Table 12.1: Parameters of the Laplacian Kernel

Factor	Levels	Short Description
vector_length	$2^0, \ldots, 2^4$	Size of vectors
load_overlap	*true*, *false*	Load overlaps in vectorization
temporary_size	$2, 4$	Byte size of temporary data
elements_number	$1, \ldots, 24$	Size of equal data splits
y_component_number	$1, \ldots, 6$	Loop tile size
threads_number	$2^5, \ldots, 2^{10}$	Size of thread groups
lws_y	$2^0, \ldots, 2^{10}$	Block size in y dimension

took 154 hours to be completely evaluated on an *Intel Xeon E5-2630v2* CPU, with *gcc* version *4.8.3* and *Nvidia* driver version *340.32*.

12.3.2 Modeling the Impact of Each Factor

This section briefly describes each factor and our modeling hypotheses regarding their impact on performance.

Linear Terms

The effects of the following factors were modeled with linear terms, whether because they are categorical binary factors or to attempt to exploit the simplest relationship under uncertainty of the effects.

- The *vector_length* factor controls the vector size used during computation. Architectures supporting vectorization provide speedups by saving instruction decoding time, but vectors must be properly sized. Although *NVIDIA* GPUs did not support vectorization at the time of this study, vectorization can still impact performance via cache effects
- The *load_overlap* factor is binary, and its effect is consequently modeled with a linear term. The parameter encodes the choice of whether to save memory load instructions by overlapping vectors in memory
- The *lws_y* factor controls the size of a thread block, in the y-axis. We did not have assumptions about the behavior of this factor, so we attempted a linear term

Linear plus Inverse Terms

The effects of the following factors were modeled with a linear term plus an inverse term. In our initial model we assumed that these factors had a linear effect on performance, but also that a more complex relationship existed due to different kinds of overhead. In all cases, we modeled the expected overhead as an inverse term.

- The *elements_number* factor controls the size of image portion that will be processed by each thread, and using smaller portions implies using more threads. As more threads are used we expect an improvement on performance, but also an overhead due to extra memory loads and to the cost of managing threads
- The *y_component_number* factor controls tiling on the y-axis of the target image, inside each thread. The optimal tiling size is a compromise between fitting a tile in cache and providing enough prefetched memory to not slow computation down
- The *threads_number* parameter controls the size of an OpenCL thread work group. Threads in the same group share data and can be scheduled together. Using more smaller groups can improve performance from better scheduling, but imply in more management overhead

The Complete Initial Model

Putting together the terms for all factors, the complete initial model for the time to compute a single pixel is written

$$\begin{aligned} time\_per\_pixel \sim & y\_component\_number + \frac{1}{y\_component\_number} + \\ & temporary\_size + vector\_length + load\_overlap + \\ & lws\_y + \frac{1}{lws\_y} + elements\_number + \frac{1}{elements\_number} + \\ & threads\_number + \frac{1}{threads\_number}, \end{aligned} \tag{12.2}$$

where coefficients were omitted. This model was pruned at each iteration of our method, fixing significant factors to their best predicted level.

12.4 Looking at a Single DLMT Run

This section presents a more detailed look at the iterative process performed during the optimization of the Laplacian kernel by our method. Table 12.2 shows the ANOVA tests performed at each step in Figure 12.6, highlighting the factors identified to be significant at each step, which were subsequently fixed to their best predicted levels. Note that the size of the model being tested decreases at each step, as factors are fixed. These progressive restrictions to slices of the search space enable the ANOVA test to detect differences in means that were previously unclear.

Factors were chosen based on ANOVA tests where factors with p-values below a threshold of $p < 0.05$ were considered significant. The threshold for significance must be adjusted to better fit the target problem, and could be dispensed with altogether, according to the

Table 12.2: ANOVA tests at each step. Red lines mark model terms that were considered significant, with $p < 0.05$, and fixed in a given step

Step	Term	Sum Sq.	F-value	p(>F)
1st	*y_component_number*	2.1×10^{-18}	7.3×10^{-1}	4.1×10^{-1}
	1/y_component_number	4.4×10^{-18}	1.6×10^{0}	2.4×10^{-1}
	vector_length	1.3×10^{-17}	4.4×10^{0}	4.7×10^{-2}
	lws_y	6.9×10^{-17}	2.4×10^{1}	3.5×10^{-4}
	1/lws_y	1.8×10^{-17}	6.2×10^{0}	2.8×10^{-2}
	load_overlap	9.1×10^{-20}	3.2×10^{-2}	8.6×10^{-1}
	temporary_size	7.1×10^{-18}	2.5×10^{0}	1.4×10^{-1}
	elements_number	3.1×10^{-19}	1.1×10^{-1}	7.5×10^{-1}
	1/elements_number	1.3×10^{-18}	4.4×10^{-1}	5.2×10^{-1}
	threads_number	7.2×10^{-18}	2.5×10^{0}	1.4×10^{-1}
	1/threads_number	4.3×10^{-18}	1.5×10^{0}	2.4×10^{-1}
2nd	*y_component_number*	1.2×10^{-19}	2.1×10^{1}	1.4×10^{-3}
	1/y_component_number	1.4×10^{-20}	2.4×10^{0}	1.5×10^{-1}
	load_overlap	4.1×10^{-21}	7.3×10^{-1}	4.1×10^{-1}
	temporary_size	1.4×10^{-21}	2.6×10^{-1}	6.2×10^{-1}
	elements_number	6.0×10^{-22}	1.1×10^{-1}	7.5×10^{-1}
	1/elements_number	2.7×10^{-21}	4.8×10^{-1}	5.0×10^{-1}
	threads_number	7.2×10^{-21}	1.3×10^{0}	2.9×10^{-1}
	1/threads_number	2.9×10^{-20}	5.1×10^{0}	4.0×10^{-2}
3rd	*load_overlap*	7.4×10^{-25}	3.8×10^{0}	1.1×10^{-1}
	temporary_size	1.1×10^{-22}	5.7×10^{2}	2.4×10^{-1}
	elements_number	9.3×10^{-22}	4.7×10^{3}	1.2×10^{-8}
	1/elements_number	3.1×10^{-22}	1.6×10^{3}	1.9×10^{-7}

desired degree of automation and trust in the initial modeling assumptions. All experiments with this method presented in this thesis were completely automated using fixed significance thresholds.

Figure 12.6 shows the factors that were eliminated in each of the 3 steps performed by DLMT, and compares each factor's predicted best level with the level for that factor in the global optimum.

We see that the chose factor level matches the level on the global optimum in all factor fixing steps, except for the number of threads. This seems to have a small effect in the final results, but could be fixed in this specific case by dropping the inverse term, since the level on the global optimum is the highest one possible. This is an example of how this transparent approach can help learning about the problem begin optimized by challenging initial modeling assumptions. In this case, perhaps thread group management is a larger overhead than expected, and it is better to have larger groups.

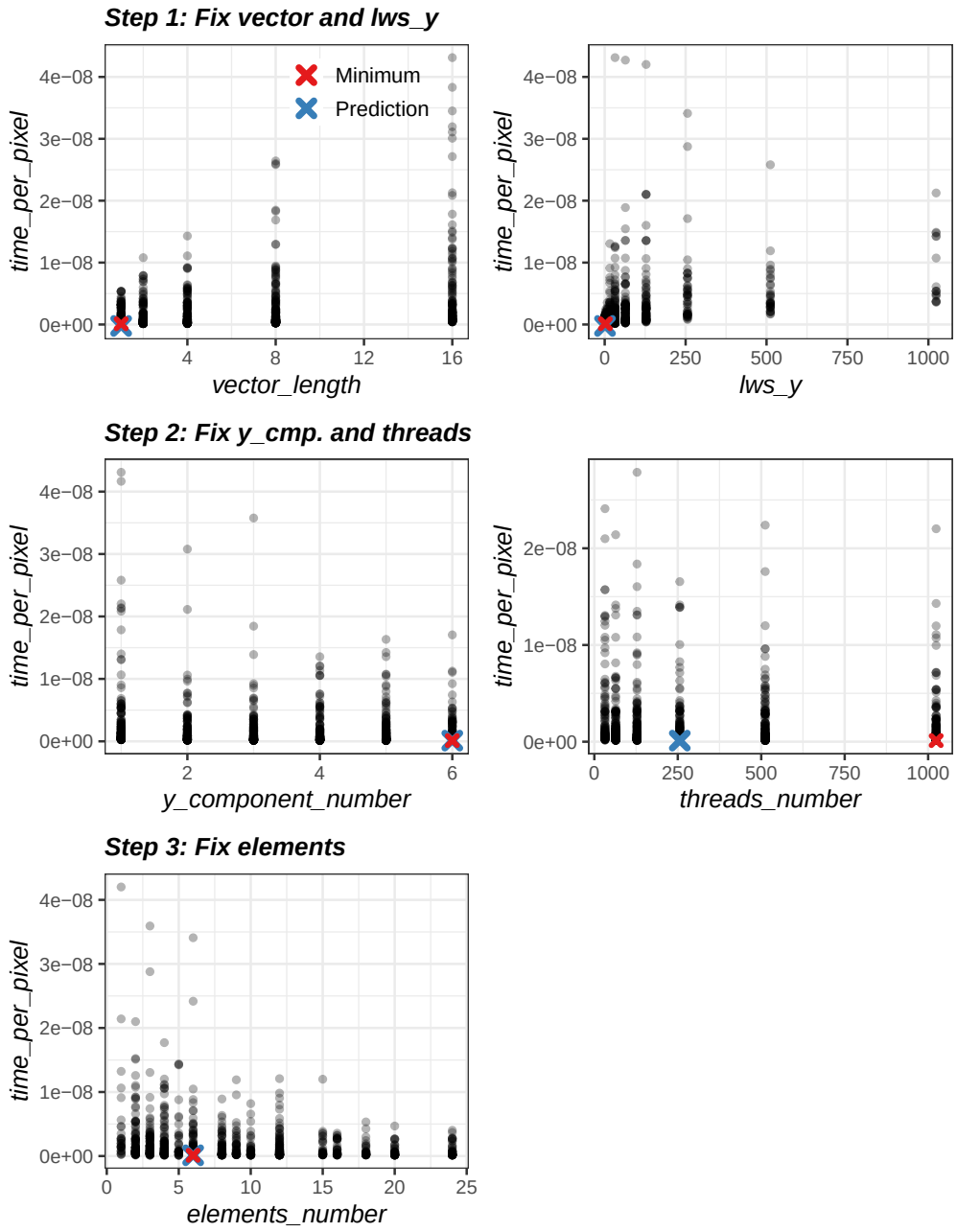

Figure 12.6: Overview of a single DLMT run. Each panel shows a view of the completely evaluated search space, from the perspective of one of the factors eliminated in a given step. A × marks the predicted best level of each factor, and a × marks the factor level on the global optimum.

12.5 Evaluation of Optimization Methods

This section presents a performance evaluation of the approach described in Section 12.2, comparing this approach to eight other optimization methods. The performance model described in the previous section was used by the Iterative Linear Model (LM) and the Quantile Regression (QR) methods, as well as by our approach, named *D-Optimal Designs, with Linear Model and heteroscedasticity correction Transform* (DLMT). The LM method is almost identical to our approach, described in Chapter 13, but it uses a fixed-size random sample of the search space instead of generating D-Optimal designs. Likewise, the QR method used random samples and Quantile Regression, in an attempt to decrease the impact of noisy outliers in the final fit. Additionally, we compared the performance of our approach with nine other methods, namely uniform Random Sampling (RS), Latin Hypercube Sampling (LHS), Greedy Search (GS), Greedy Search with Restart (GSR), and a Genetic Algorithm (GA). At a later date, after the CCGRID publication, we added to this performance comparison an implementation of Gaussian Process Regression with Expected Improvement acquisition function (GPR). Each method performed at most 125 measurements over 1000 repetitions, without user intervention.

Since we measured the entire valid search space for this kernel, we could use the *slowdown* relative to the global optimum to compare the performances of each method. Table 12.3 shows the mean, minimum, and maximum slowdowns in comparison to the global optimum for each method. It also shows the mean and maximum budget used by each method. Figure 12.7 presents histograms with the count of the slowdowns found by each of the 1000 repetitions. Arrows point the maximum slowdown found by each method. Note that maximum slowdown of the GS method was left out of range to help the comparison between the other method.

All methods performed relatively well in this kernel, with only GS not being able to find slowdowns smaller than 4 times in some of the runs. As expected, other search algorithms had results similar to RS. LM was able to find slowdowns close to the global optimum on most runs, but some of the runs could not find slowdowns smaller than 4 times. Our approach reached a slowdown of 1% from the global optimum in all of the 1000 runs while using at most fewer than half of the allotted budget.

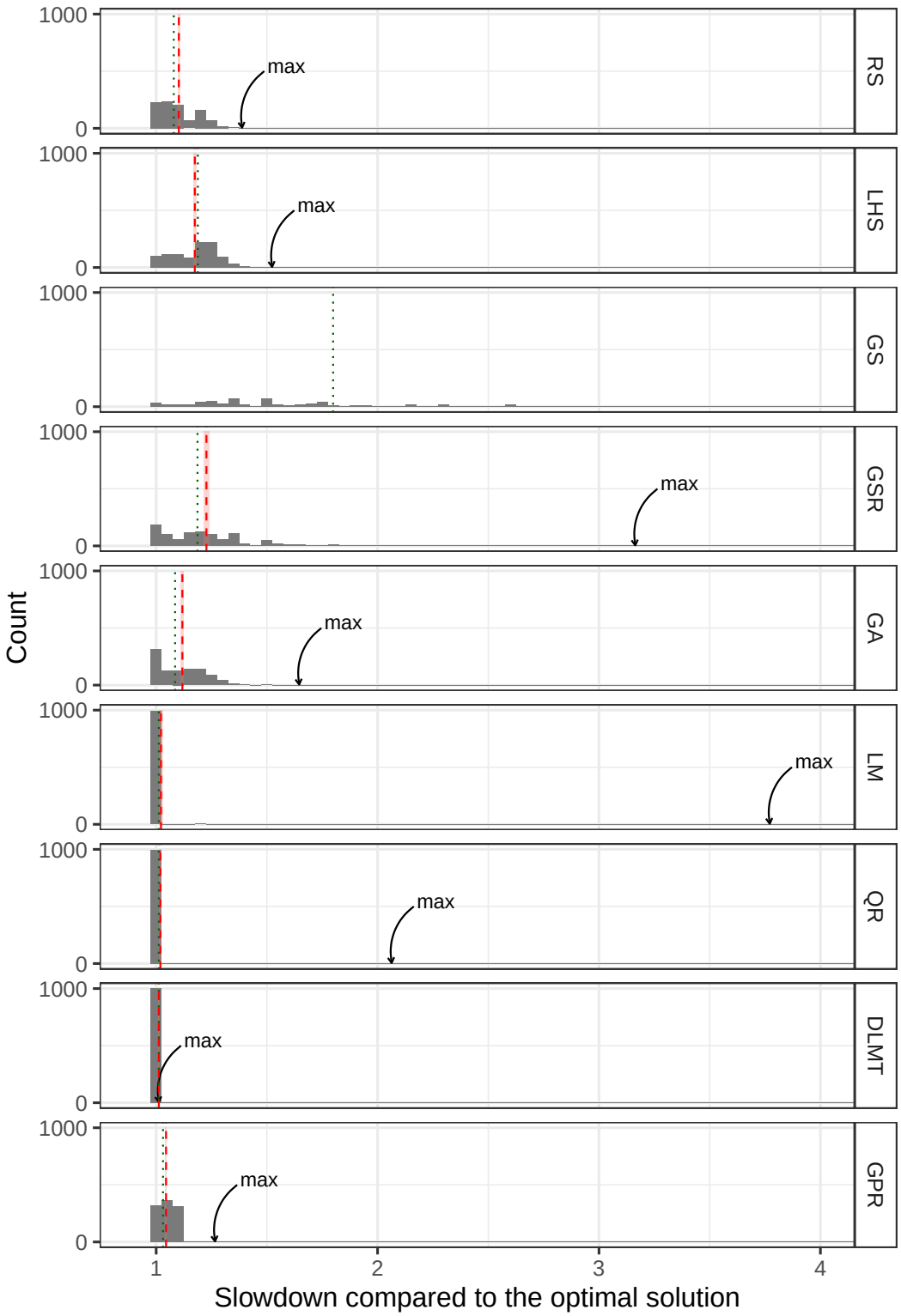

Figure 12.7: Distribution of slowdowns in relation to the global optimum for 7 optimization methods on the Laplacian Kernel, using a budget of 125 points over 1000 repetitions

We implemented a simple approach for the prediction step in this problem, choosing the best value of our fitted models on the complete set of valid level combinations. This was possible for this problem since all valid combinations were known. For problems were the search space is too large to be generated, we would have to either adapt this step and run the prediction on a sample.

This kernel provided ideal conditions for using our approach, where the performance model is approximately known and the complete valid search space is small enough to be used for prediction. The global optimum also appears to not be isolated in a region of points with bad performance, since our approach was able to exploit search space geometry. In Chapter 13 we present a more comprehensive performance evaluation of our approach in a larger benchmark suite.

12.6 Summary

This chapter described a transparent and parsimonious approach to autotuning, based on D-Optimal Designs, ANOVA tests, and Linear Regression. We dealt with the heteroscedasticity that could violate modeling assumptions by using standard detection tests and transformation procedures. This approach produced optimizations consistently within 1% of the global optimum of an OpenCL Laplacian kernel run in an *NVIDIA* GPU, using half of the allotted measurement budget for optimization.

We demonstrated how the choices made during optimization are completely transparent, in such a way that it is straightforward to envision how users could interfere with and learn from the optimization process by, for example, selecting specific factors that should be treated differently, and changing initial modeling assumptions based on the results produced.

The development, testing, and tweaking of this approach relied heavily on the availability of a completely evaluated search space, with known global optimum. Having access to this sort of search space is not possible in real applications, as we will see in the next two

Table 12.3: Slowdown and budget used by 7 optimization methods on the Laplacian Kernel, using a budget of 125 points with 1000 repetitions

	Slowdown			Budget	
Method	Mean	Min.	Max.	Mean	Max.
Random Sampling (RS)	1.10	1.00	1.39	120.00	120
Latin Hypercube Sampling (LHS)	1.17	1.00	1.52	98.92	125
Greedy Search (GS)	6.46	1.00	124.76	22.17	106
Greedy Search w. Restart (GSR)	1.23	1.00	3.16	120.00	120
Genetic Algorithm (GA)	1.12	1.00	1.65	120.00	120
Linear Model (LM)	1.02	1.01	3.77	119.00	119
Quantile Regression (QR)	1.02	1.01	2.06	119.00	119
D-Opt., Linear Model w. Transform (DLMT)	1.01	1.01	1.01	54.84	56
Gaussian Process Regression w. EI (GPR)	1.04	1.01	1.27	71.45	120

chapters, but it is also unfortunately uncommon in the autotuning literature. We hope the studies presented in this thesis, and the one in this chapter in particular, help arguing for and justifying the elaboration and evaluation of well-defined and completely-evaluated search spaces for autotuning, composed of high-interest computing kernels, that could be used for the development and comparison of optimization methods for autotuning.

Chapter 13

Experimental Design and Gaussian Process Regression for CPU Kernels

This chapter presents an application of Optimal Design and Gaussian Process Regression to the optimization of source-to-source transformation kernels from the SPAPT [29] benchmark suite.

The chapter is organized as follows. Section 13.1 discusses our choice of using D-Optimal designs, considering its applicability to autotuning among other Experimental Design methods. Section 13.2 presents the CPU kernels to which we applied our approach, and Section 13.3 discusses the results. The work presented up to Section 13.3 was published at CCGrid [4] in 2019. The studies we performed later were not yet published at the writing of this thesis. In Section 13.4 we explore the optimization of one of the target kernels and identify the factors and levels that where responsible for the observed performance improvements. In Section 13.5 we discuss the application of Gaussian Process Regression with Expected Improvement to the same kernel. Finally, Section 13.6 summarizes the chapter and discusses future work.

13.1 Choosing an Experimental Design Method for Autotuning

Our application of the ED methodology requires support for factors of different types and numbers of levels, such as binary flags, integer and floating point numerical values and enumerations of categorical values. We also need designs that minimize the number of experiments needed for identifying the most relevant factors of a problem, since at this moment we are not interested in a precise analytical model. The design construction techniques that fit these requirements are limited. The first ED approach we studied was *screening* which, despite being extremely parsimonious, does not have enough flexibility to explore the search spaces which we wished to study.

Despite being parsimonious, screening designs are extremely restrictive Algorithms for constructing Optimal Designs can adapt to our requirements, being able to mix factors of different types and optimizing a starting designs. Before settling on D-Optimal designs, which we use in this study, we explored other design construction techniques such as extensions to Plackett-Burman [71] designs that use random factor levels, the *contractive replacement* technique presented by Addelman-Kempthorne [133], and the *direct generation* algorithm by Grömping and Fontana [134]. These techniques have strong requirements on design size and level mixing, so we opted for a more flexible technique that would enable exploring a more comprehensive class of autotuning problems.

The modified screening method provides a strategy to use factors with more than two levels by, instead of encoding fixed high and low levels, sampling two levels uniformly, each time the factor is included in a Plackett-Burman design. This approach has the advantage of a small design size with good main effect estimation capability, but it is still not capable of estimating interactions.

The *contractive replacement* method starts with a large 2-level design and generates mixed-level designs by re-encoding and replacing pairs of columns with a new column for a multi-level factor. The contractive replacement method presented by Addelman-Kempthorne [133] is a strategy of this kind. In addition to small design size and good estimation capability, their method maintains the orthogonality of starting designs, although it places strong requirements on initial designs, such that only orthogonal 2-level matrices can be contracted with their method.

The *direct generation* algorithm introduced by Grömping and Fontana [134] enables the direct generation of multi-level designs that minimize the *Generalized Minimum Aberration* [135] optimality criterion by solving mixed integer problems. We did not pursue this method because of the limitations it imposes on the size and the shape of the designs that can be generated, and also because it relied on proprietary MIP solvers that we did not have access to.

Weighting flexibility, effectiveness, parsimony, and the cost and availability of algorithmic construction, we picked *D-Optimal Designs* among the methods that fulfilled the requirements for application to autotuning problems. In particular, we used the KL-exchange algorithm [136], which enables mixing categorical and numerical factors in the same design, while biasing sampling according to the performance model we wish to explore. This enables the exploitation of global search space structures, if we use the right model. We can safely optimize for the D optimality criterion among other criteria without loosing quality of designs [137].

Table 13.1: Kernels from the SPAPT benchmark used in this evaluation

Kernel	Operation	Factors	Size
`atax`	Matrix transp. & vector mult.	18	2.6×10^{16}
`dgemv3`	Scalar, vector & matrix mult.	49	3.8×10^{36}
`gemver`	Vector mult. & matrix add.	24	2.6×10^{22}
`gesummv`	Scalar, vector, & matrix mult.	11	5.3×10^{9}
`hessian`	Hessian computation	9	3.7×10^{7}
`mm`	Matrix multiplication	13	1.2×10^{12}
`mvt`	Matrix vector product & transp.	12	1.1×10^{9}
`tensor`	Tensor matrix mult.	20	1.2×10^{19}
`trmm`	Triangular matrix operations	25	3.7×10^{23}
`bicg`	Subkernel of BiCGStab	13	3.2×10^{11}
`lu`	LU decomposition	14	9.6×10^{12}
`adi`	Matrix sub., mult., & div.	20	6.0×10^{15}
`jacobi`	1-D Jacobi computation	11	5.3×10^{9}
`seidel`	Matrix factorization	15	1.3×10^{14}
`stencil3d`	3-D stencil computation	29	9.7×10^{27}
`correlation`	Correlation computation	21	4.5×10^{17}

13.2 The SPAPT Benchmark Suite

The *Search Problems in Automatic Performance Tuning* (SPAPT) [29] benchmark suite provides parametrized CPU kernels from different HPC domains. The kernels shown in Table 13.1 are implemented using the code annotation and transformation tools provided by Orio [27]. Search space sizes are larger than in the Laplacian Kernel example. Kernel factors are either integers in an interval, such as loop unrolling and register tiling amounts, or binary flags that control parallelization and vectorization.

We used the Random Sampling (RS) implementation available in Orio and integrated an implementation of our approach (DLMT) to the system. We omitted the other Orio algorithms because other studies using SPAPT kernels [77, 78] showed that their performance is similar to RS regarding budget usage. The global minima are not known for any of the problems, and search spaces are too large to allow complete measurements.

13.3 Performance Improvements using Optimal Design

We used the performance of each SPAPT kernel compiled with the *gcc -O3* flag, with no code transformations, as a *baseline* for computing the *speedups* achieved by each strategy. We performed 10 autotuning repetitions for each kernel using RS and DLMT, using a budget of *at most* 400 measurements. DLMT was allowed to perform only 4 of the iterations shown in Figure 12.5. Experiments were performed using Grid5000 [138], on *Debian Jessie*, using an *Intel Xeon E5-2630v3* CPU and *gcc* version *6.3.0*.

The time to measure each kernel varied from a few seconds to up to 20 minutes. In testing, some transformations caused the compiler to enter an internal optimization process that did not stop for over 12 hours. We did not study why these cases delayed for so long,

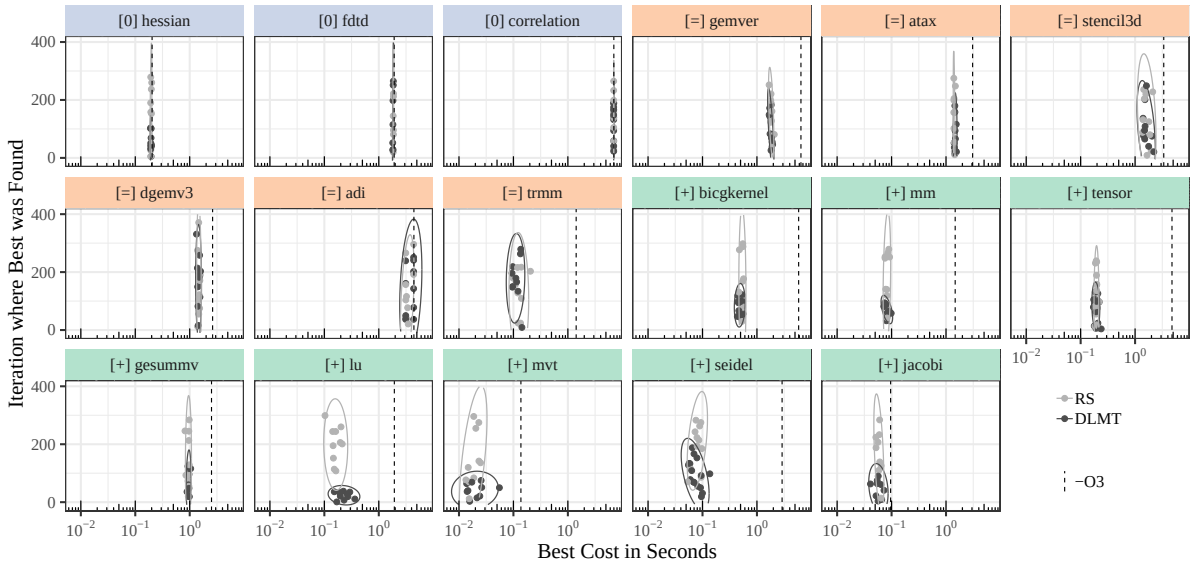

Figure 13.1: Cost of best points found on each run, and the iteration where they were found. RS and DLMT found no speedups with similar budgets for kernels marked with "[0]" and *blue* headers, and similar speedups with similar budgets for kernels marked with "[=]" and *orange* headers. DLMT found similar speedups using smaller budgets for kernels marked with "[+]" *green* headers. Ellipses delimit an estimate of where 95% of the underlying distribution lies

and implemented an execution timeout of 20 minutes, considering cases that took longer than that to compile to be runtime failures.

We automated factor elimination based on ANOVA tests so that a comprehensive evaluation could be performed. We also did not tailor initial performance models, which were the same for all kernels. Initial models had a linear term for each factor with two or more levels, plus quadratic and cubic terms for factors with sufficient levels. Although automation and identical initial models might have limited the improvements at each step of our application, our results show that it still succeeded in decreasing the budget needed to find significant speedups for some kernels.

Figure 13.1 presents the *speedup* found by each run of RS and DLMT, plotted against the algorithm *iteration* where that speedup was found. We divided the kernels into 3 groups according to the results. The group where no algorithm found any speedups contains 3 kernels and is marked with "[0]" and *blue* headers. The group where both algorithms found similar speedups, in similar iterations, contains 6 kernels and is marked with "[=]" and *orange* headers. The group where DLMT found similar speedups using a significantly smaller budget than RS contains 8 kernels and is marked with "[+]" and *green* headers. Ellipses delimit an estimate of where 95% of the underlying distribution lies, and a dashed line marks the *-03* baseline. In comparison to RS, our approach significantly decreased the average number of iterations needed to find speedups for the 8 kernels in the green group.

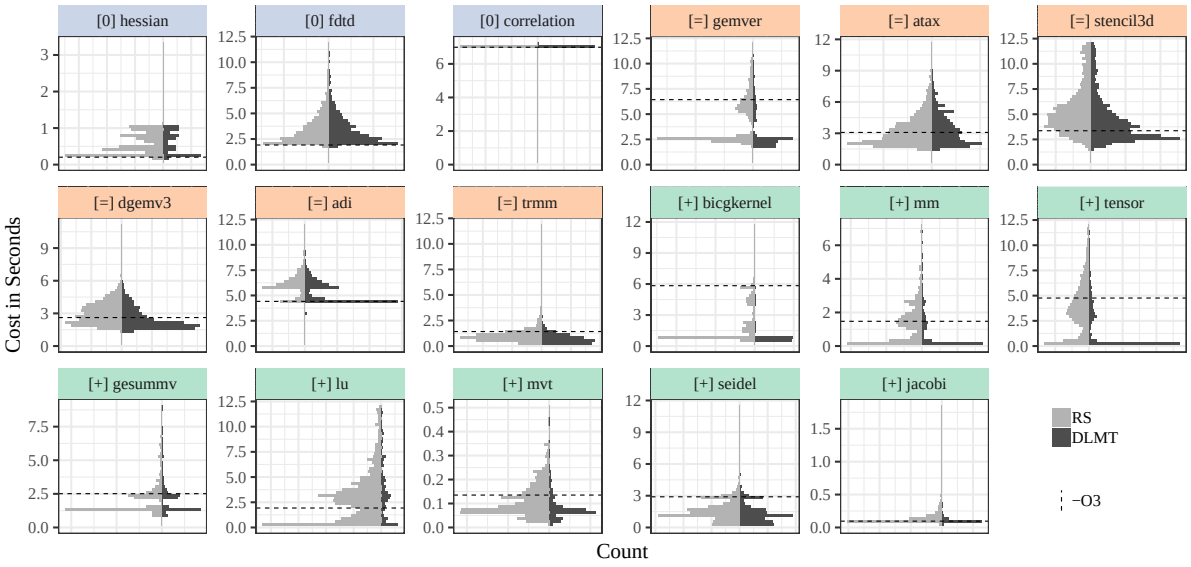

Figure 13.2: Histograms of explored search spaces, showing the real count of measured configurations. Kernels are grouped in the same way as in Figure 13.1. DLMT spent fewer measurements than RS in configurations with smaller speedups or with slowdowns, even for kernels in the orange group. DLMT also spent more time exploring configurations with larger speedups

Figure 13.2 shows the search space exploration performed by RS and DLMT. It uses the same color groups as Figure 13.1, and shows the distribution of the speedups that were found during all repetitions of the experiments. Histogram areas corresponding to DLMT are usually smaller because it always stopped at 4 iterations, while RS always performed 400 measurements. This is particularly visible in *lu*, *mvt*, and *jacobi*. We also observe that the quantity of configurations with high speedups found by DLMT is higher, even for kernels on the orange group. This is noticeable in *gemver*, *bicgkernel*, *mm* and *tensor*, and means that DLMT spent less of the budget exploring configurations with small speedups or slowdowns, in comparison with RS.

Analyzing the significant performance parameters identified by our automated approach for every kernel, we were able to identify interesting relationships between parameters and performance. In *bicgkernel*, for example, DLTM identified a linear relationship for OpenMP and scalar replacement optimizations, and quadratic relationships between register and cache tiling, and loop unrolling. This is an example of the transparency in the optimization process that can be achieved with a DoE approach.

Our approach used a generic initial performance model for all kernels, but since it iteratively eliminates factors and model terms based on ANOVA tests, it was still able to exploit global search space structures for kernels in the orange and green groups. Even in this automated setting, the results with SPAPT kernels illustrate the ability our approach has to reduce the budget needed to find good speedups by efficiently exploring search spaces.

13.4 Identifying Significant Factors for the *bicg* Kernel

The discussion on the following sections was not yet published at the time of writing of this thesis. We continued to study the application of Experimental Design methods to the optimization of SPAPT kernels, modifying the original DLMT algorithm by removing cubic terms to simplify the target performance model, fixing binary parameters to look for other significant effects, leveraging data collected in previous designs, running more optimization steps, and assessing the performance of quantile regression as an alternative to the linear model. In this study we sought to expose and analyze the inner workings of our approach.

In a non-automated setting, it becomes more clear how users could interfere and guide search space restriction and exploration, by intervening at each ANOVA step, for example. We started with the *bicg* SPAPT kernel because our approach achieved equivalent solutions using a smaller budget than a random sampling approach.

Figure 13.3 shows the execution times of the best kernel configurations, and the corresponding iterations where each of these points was found. Results are shown for Random Sampling, for our DLMT implementation from the CCGRID paper, and for four variations that we later explored, attempting to interpret and improve the optimization process. The y-axis on the top panel shows the execution time of the best points found in 10 distinct runs, and lighter colors mark points found with fewer iterations. Similarly, the y-axis on the bottom panel shows the iterations where points were found for the same data, with lighter colors marking points with lower execution times.

The next sections discuss incremental modifications to the algorithm published at CCGRID, which were performed to help identify significant factors and attempt to find better speedups. Although we did not significantly improve the speedups found by our method, we were able to present a deeper look into the inner workings of this white-box autotuning approach. The next section describe in detail the two incremental modifications below:

Removing Cubic Terms We initially used third-degree polynomials for the performance models of all kernels. Cubic terms were never eliminated when following significance levels reported by ANOVA, so we decided to drop cubic terms and fit a quadratic polynomial, which reduced the necessary design size and produced equivalent results.

Reusing Data In our initial approach, we performed ANOVA in each iteration using only the data from the subspaces to which we restricted exploration. This can increase flexibility because the models are free to vary outside the restricted regions, but this also wastes experimental data. In this modification we reused all measurements performed during optimization, leveraging all measurements regardless of their position in the search space, which increases the benefits of multiple runs.

The two other modifications shown in Figure 13.3 are:

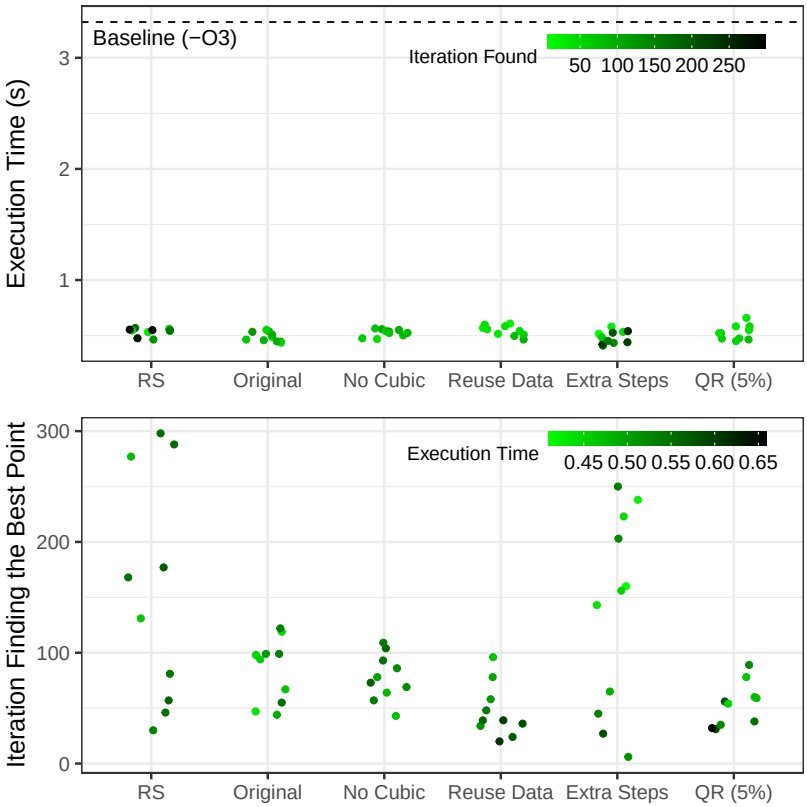

Figure 13.3: Summary of our DLMT results, compared to uniform Random Sampling (RS), showing the configurations with smallest speedups, found in 10 independent runs. The top panel compares execution time, with color-encoded iterations, and the bottom panel compares iterations where configurations were found, with color-encoded execution times.

Quantile Regression We identified a substantial amount of noise on the performance measurements of non-binary parameters. Models accounting for factor interactions could help, since there is a good chance that interactions are responsible for the noise. In this study, we decided to try Quantile Regression, which is a modification of Linear Regression where it is possible to weight points based on the quantiles they belong to, allowing the model to fit to a different quantile, opposition to fitting to the mean. We explored multiple quantiles, and Figure 13.3 shows results using the 5% quantile.

Running more Steps This is less of a modification, and involved simply running the original approach for twice the number of steps.

13.4.1 Removing Cubic Terms

The data in the figures shown next were obtained with a new set of 10 repetitions of the *bicg* kernel experiment from Figures 13.1 and 13.2, but using a performance model with only linear and quadratic terms. The results are similar to the ones from the CCGRID paper,

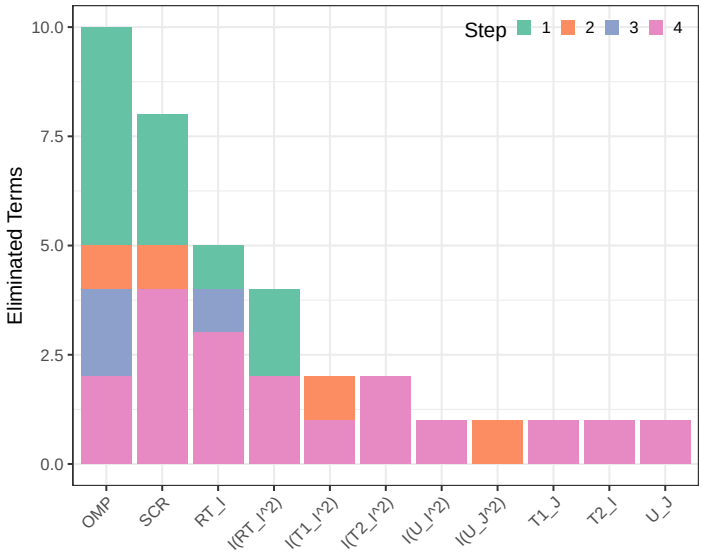

Figure 13.4: Count of the model terms eliminated in each of the 4 steps of 10 independent runs, without cubic terms

but our approach found slightly better configurations slightly faster. Random sampling also found a much better configuration much faster than before in one experiment. Figure 13.4 shows the *count of model terms* that were eliminated at each color-coded DLMT step. As before, we see that *OMP* and *SCR* were the most eliminated factors, especially on the first step.

Figure 13.5 shows the factors that were identified as significant by ANOVA, with significance threshold of 0.01, at each DLMT step. Identifying a factor removes the corresponding parameter from the model, regardless of the model term that was identified. Figures are grouped by each of the 10 experiments. Figure headers identify each run, and correspond to the names of the Grid5000 machines where experiments were run.

It is interesting that only *parasilo-19*, *parasilo-20*, and *parasilo-9* eliminated any factor other than *OMP* and *SCR* on the first step where any factor was eliminated, and also that those runs fixed the factor RT_I to the same value. We can also see that *OMP* seems to be the parameter behind the most extreme changes in the execution time of tested configurations. This is specially clear at *parasilo-13* and *parasilo-10*, where the explored configurations have relatively high execution time until after step 3, where *OMP* is fixed. A slight worsening of execution times, an increase, that is, can be seen at the fourth step at *parasilo-11*, after RT_I was fixed. The minimum execution time seems to be higher than in the third step.

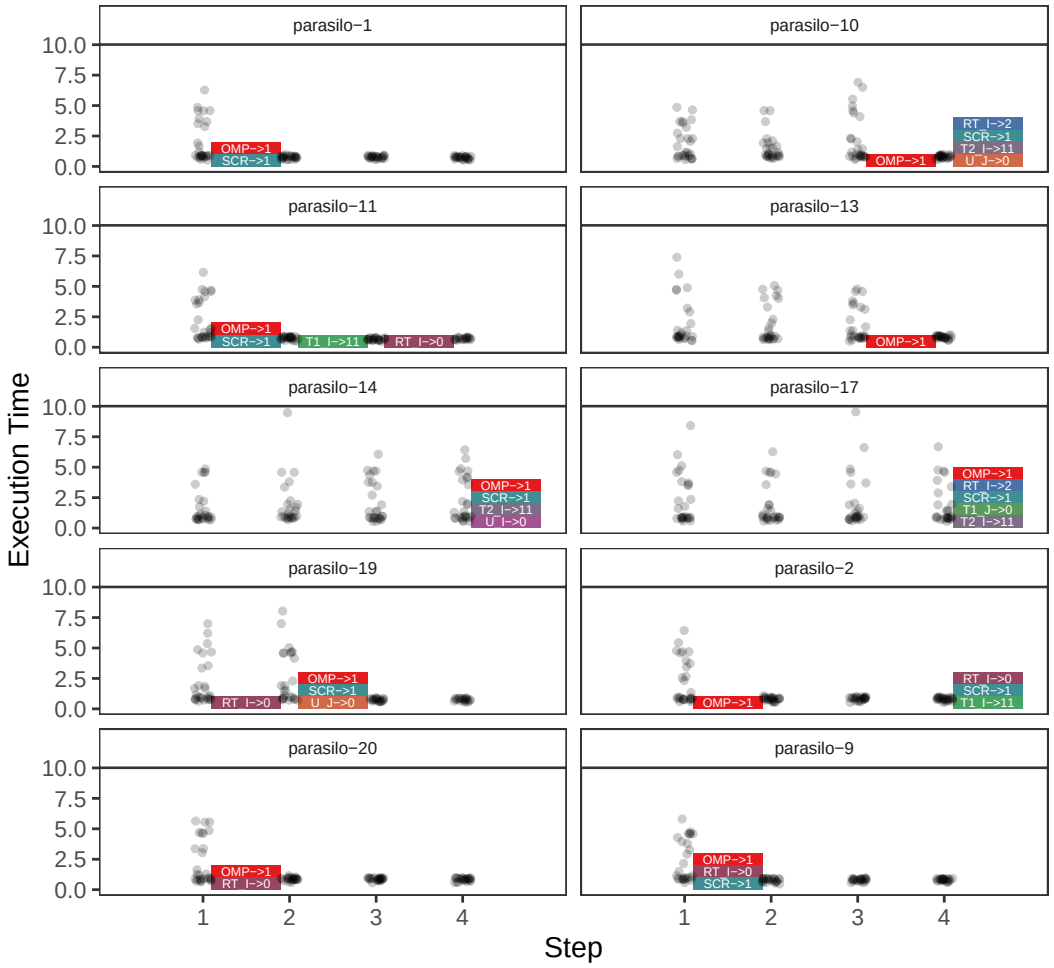

Figure 13.5: Measured execution time of all points in the designs constructed at each step, where panel headers mark the hostname of the machines used in each of the 10 separate experiments

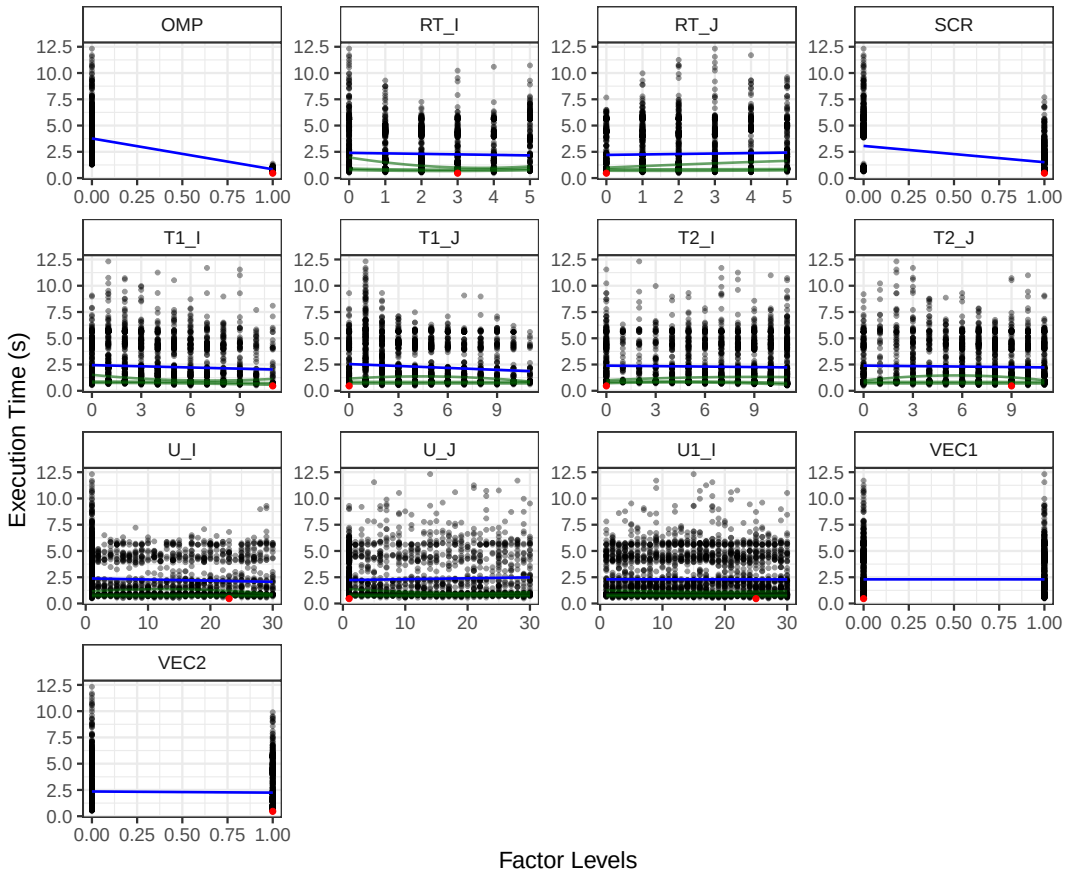

Figure 13.6: Fitting linear models and quadratic quantile regression, with $\tau = 0.05$, 0.25, and 0.5, to separate factors for 10 uniform random sampling runs

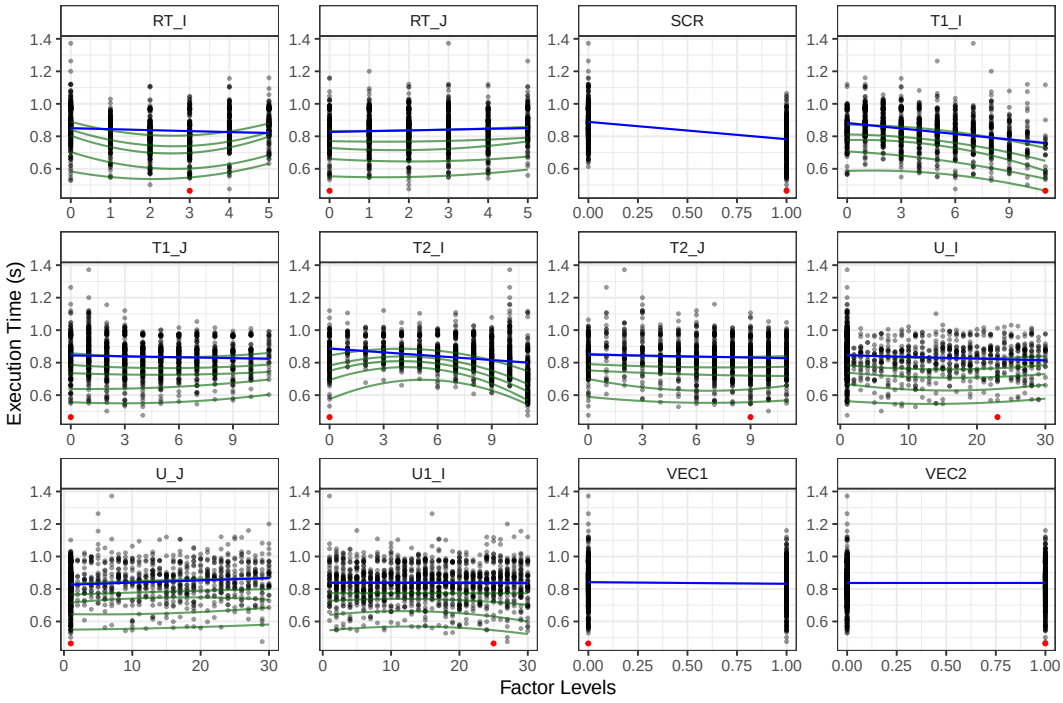

Figure 13.7: Same as above, for 10 uniform sampling runs with *OMP* fixed to *on*

Figures 13.6 and 13.7 contain all points measured in 10 uniform random sampling runs. Each panel the same execution time data in the y-axis, viewed from the perspective of the changes in a single factor, shown in the x-axis. The blue lines in both figures represents a linear model fit, using only linear terms for each factor separately. The green lines denote three quantile regression fits using quadratic model terms for each factor separately. We used the 5%, 25%, and 50% quantiles. Using the 50% quantile is equivalent to linear regression. Red dots mark the values of each factor in the point with the best performance across the 10 runs.

The *OMP* parameter controls whether OpenMP is used to run multithreaded execution, and has clearly the largest impact when its level is 1.0, which encodes the binary level corresponding to turning parallelization on. All measurements larger than 1.4 seconds happened when the parallelization flag was off, as can be seen on Figure 13.7, which shows all data collected in 10 uniform random sampling runs with the *OMP* factor fixed on the *on* level. The *SCR* parameter also has a strong effect. This parameter controls *scalar replacement*, which consists of moving array references to the stack, preferably into registers.

Fixing *OMP* to *on* makes the effects of other factors more pronounced and easier to detect using ANOVA. Most of the best factor levels in this example were near the limits of the search space. The RT_1 factor controls tiling of one of the loops in *bicg*, and its best level was exceptionally at the middle of the range.

We can also verify that the optimal design approach is restricting the exploration of the search space to regions with faster configurations. Figure 13.8 compares the performance predicted by the fitted model, represented by the green line, with the performance of the best point found at each design, represented by the blue line. The red line marks the best point found so far.

It is only at the fourth step of *parasilo-17* that the point with the best predicted performance was better than the best point on the design for that step, while also being better than the best point found so far. Although we seem to be effectively restricting the search space with our exploration, which is evidenced by the improvement that occurs as steps progress and by the best points being found inside designs, the models fit using experiment data are not able to improve the current best on the majority of cases. This is an indication that the model is not capable of describing the relationships between factors and response, perhaps because of the presence of interactions between parameters.

13.4.2 Reusing Data from All Steps

Significant changes were performed on the initial DLMT implementation. We decided that there was no good reason to not reuse the data obtained from evaluating designs at each step, and the various samples of the search space taken at different points. Now, all evaluated experiments compose a single growing design used by *aov* to identify the best

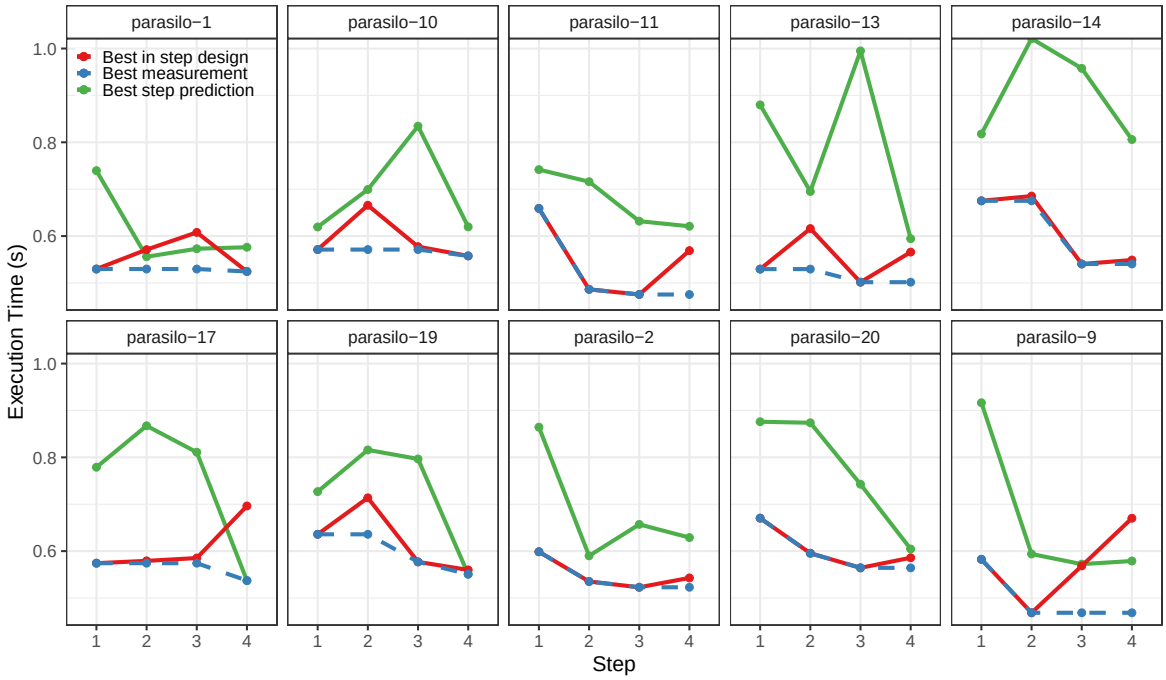

Figure 13.8: Performance of best points, step-by-step, by experiment

factors, and all samples from the search space compose a single data set used by *optFederov* to select experiments. The data set is pruned in both *aov* and *lm* analyses, to guarantee only experiments with the correct levels of fixed factors are used. This is crucial for both analyses, since having different fixed values of factors that are not in the model would imply that we would have inexplicable variance in the data set.

Using all experimental data on *aov* is interesting because it is always worth it to consider additional information on the factors that are currently being studied. On one hand, it might not allow for enough flexibility when we consider regression only on a small restricted subspace, because points outside the subspace would impact regression, and we would be interested in the significance of the factor inside the subspace at the moment. On the other hand, using all data available makes sense because we are also interested in exploring any global structures of the search space, and keeping points from other subspaces would increase the chance of "catching" the significance of a factor globally.

Using all sampled space, across all steps, as a candidate for *optFederov* has no downsides, provided we prune the search space to account for current constraints on factor levels fixed on previous steps. We increase the size of the available set of configurations that can compose a new design each time we sample a new subspace. This would hopefully improve the quality of designs produced as we progress.

13.4.3 Summary: Tuning SPAPT Kernels with Optimal Design

We could not significantly improve upon the speedups found by the uniform random sampling strategy, although we did find good speedups using a significantly smaller budget. This happens mainly because the DLMT approach quickly detects the large effect of the *OMP* binary parameter. Subsequent parameters that are detected as significant have much smaller effects, so consequently the performance does not improve by much.

We can conclude that the SPAPT search space structures defined by kernel parameters exposed and modified by Orio cannot be fully exploited by models based on second or third degree polynomials, even with well designed experiments. We could not determine whether kernel configurations with better performance than the ones we found exist in these search spaces. Additionally, the strongly biased models we used could be responsible for not finding better configurations. We attempted to find better configurations using a much more flexible approach, based on Gaussian Process Regression and Expected Improvement, which we discuss next. We also compared the performance of the *dgemv* SPAPT kernel with the peak theoretical performance for the *Xeon 2630v3* processor, indicating that better kernel configurations might exist.

13.5 Autotuning SPAPT kernels with Gaussian Process Regression and Expected Improvement

We used Gaussian Process Regression with an Expected Improvement acquisition function, denoted GPR, to try and find *bicg* kernel configurations improving upon those found by the uniform Random Sampling (RS) method. We found statistically but not practically significant improvements upon RS.

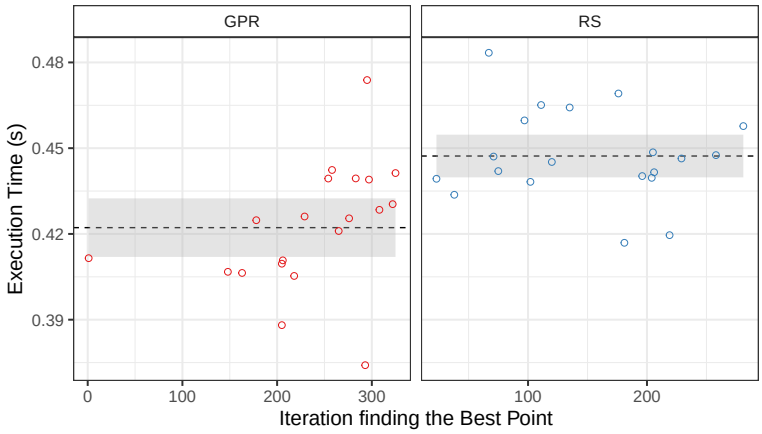

Figure 13.9: Best *bicg* configurations and iterations that found them, for Gaussian Process Regression with Expected Improvement acquisition function (GPR), and uniform Random Sampling (RS), with a budget of 400 measurements.

We used two *R* packages to construct our GPR algorithm [67]. We used the *DiceKriging* package to fit Gaussian Processes using *Matérn* covariance functions, and the *DiceOptim* package to compute the Expected Improvement metric at each step. Figure 13.9 compares the performance of the best points found by GPR and RS, across 20 independent tuning runs. The figure also shows the iteration finding the best point at each run. Dashed lines mark the estimate of the mean execution time of the best points across the 20 experiments, and the gray bands show the 95% confidence intervals for the mean estimate.

The GPR method found statistically significant better configurations than RS, but the practical difference is less than 0.1 seconds. Since Gaussian Process Regression is a much more flexible method than our previous Experimental Design approach, not finding expressively better configurations hints that it might not be possible to improve this kernel much more, in comparison to RS.

Figures 13.10 and 13.11 show the progression of all configurations evaluated during each of the 20 runs of GPR and RS, respectively. We can see that after an initial Sobol sample the GPR method quickly finds regions with better configurations. The RS method, as expected, continues to randomly explore the search space.

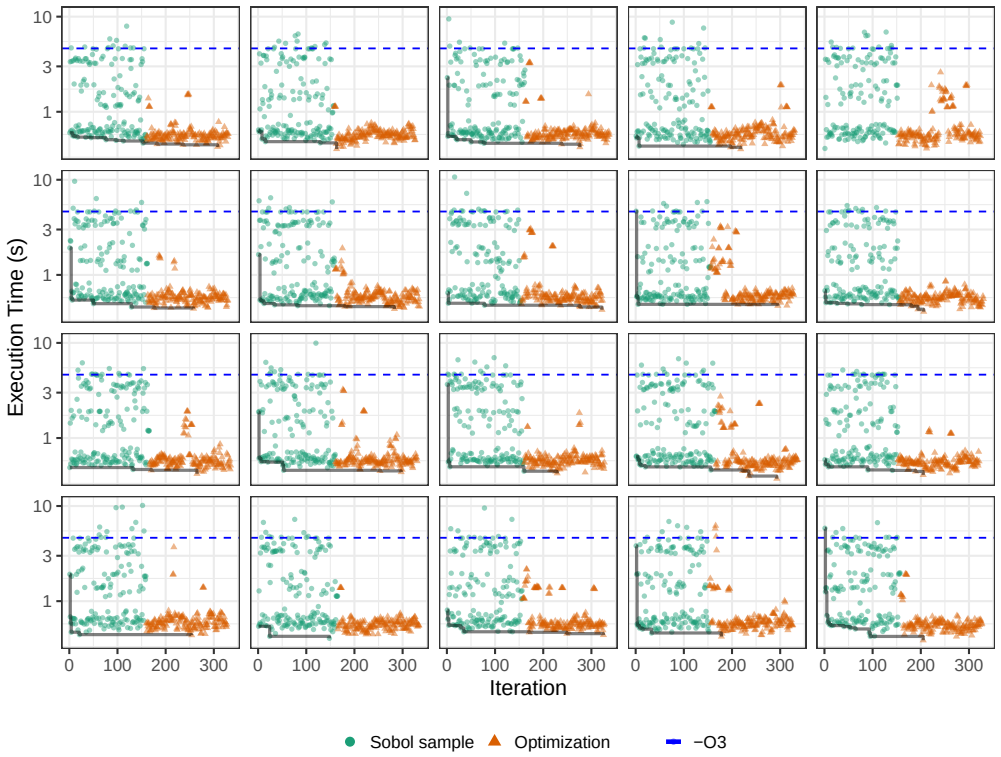

Figure 13.10: Execution time of points measured by GPR along iterations, with pareto border in red

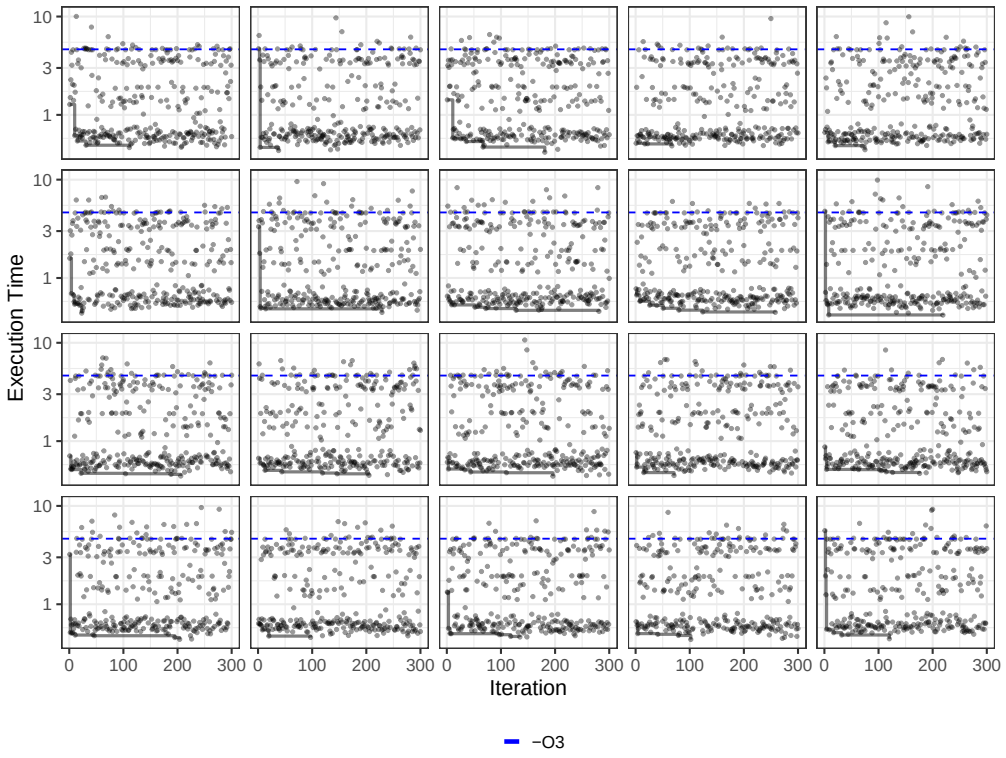

Figure 13.11: Execution time of points measured by RS along iterations

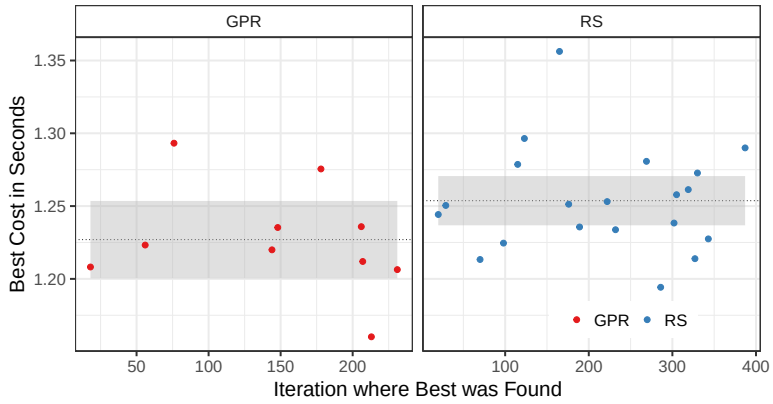

Figure 13.12: Best *dgemv3* configurations and iterations that found them, for Gaussian Process Regression with Expected Improvement acquisition function (GPR), and uniform Random Sampling (RS), with a budget of 400 measurements.

13.5.1 Peak Performance for the *DGEMV* kernel

We also performed GPR experiments with the *dgemv3* SPAPT kernel which, since we can compute its theoretical performance, allowed determining that the best kernel configuration we have found is still around 20 times slower than the theoretical peak, despite being around two times faster than the *-O3* flag. We did not perform a detailed analysis of the SPAPT kernel, which could find out whether more performance can be obtained from this kernel implementation.

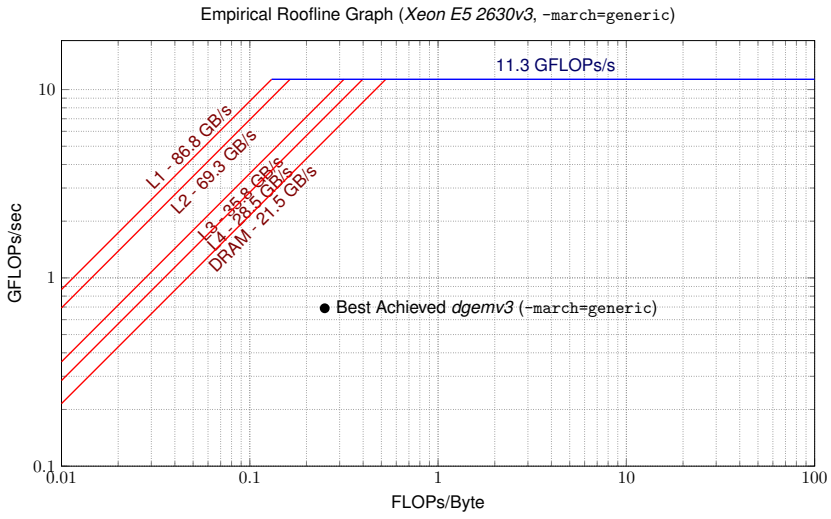

Figure 13.13: Empirical Roofline graph for the *Xeon E5-2630v3* processor [139, 140], showing the best point we found during all experiments with the memory-bound *dgemv3* SPAPT kernel

Figure 13.12 shows the best kernel configurations found across 20 repetitions of the GPR and RS methods. GPR could not find statistically or practically significant speedups in relation to RS for this kernel, although one specific GPR run found an outlier configuration. Figures 13.13 and 13.14 show the empirical Roofline [141] graphs for the processor targeted by our

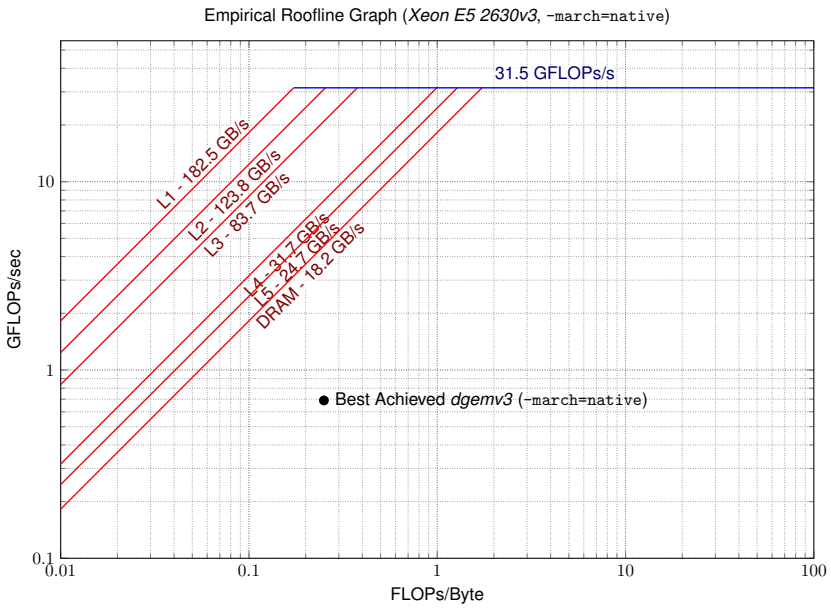

Figure 13.14: Empirical Roofline graph for the *Xeon E5-2630v3* processor [139, 140], showing the best point we found during all experiments with the memory-bound *dgemv3* SPAPT kernel

experiments, which was the same from our CCGRID paper, using *GCC -march=generic,native* options. The figure also shows the performance that was achieved by the best configuration we found for the *dgemv3* SPAPT kernel, the red outlier in Figure 13.12.

13.6 Summary

We have implemented an Experimental Design approach to autotuning based on ANOVA and D-Optimal Designs. We have applied this approach to a set of configurable kernels from the SPAPT benchmark suite and, although our approach found significant speedups in relation an *-O3* compiler baseline, we could not significantly improve upon the results found by an uniform Random Sampling algorithm using the same exploratory budget. Other experiments with the same kernels also showed than uniform random sampling [77, 78] performs well, especially when budgets are short.

We could identify and eliminate significant factors using our approach, but the largest effect was consistently that of the binary flags, especially *OMP*, which controlled parallelization. We could consistently detect significant albeit small effects for a few other numerical parameters.

The polynomial models coupled with optimal design construction have the potential for low prediction variance, but tend to have strong biases. We explored a Gaussian Process Regression approach which is much more complex, implying in high variance, but has small bias. We found small improvements upon Random Sampling with this new approach, but it is still unclear if Orio configurations are capable of further improving the kernel configurations

we already found. Analyzing the Roofline theoretical model for our target CPU hints that tweaks to the kernel implementation might be needed to improve the current performance.

A possible future direction for exploring the potential of this Experimental Design approach could involve developing a lower level set of configurable kernels, where performance measurements are always tied to a theoretical peak or to a known global optimum. As we discussed in Chapter 12, knowing the global optimum helps evaluating the performance of an autotuning approach.

Chapter 14

Gaussian Process Regression for Mixed-Precision Quantization in Convolutional Neural Networks

TODO: Fill in the sketched sections below

This chapter presents an application of Gaussian Process Regression with Expected Improvement acquisition function (GPR) to bit precision tuning for a Convolutional Neural Network (CNN). We compare GPR to a Reinforcement Learning approach which uses an actor-critic model (RL) [142], to a uniform random sampling algorithm, and to a low-discrepancy sampler using Sobol sequences. The objective function we attempted to optimize was the network accuracy over the ImageNet dataset [143, 144], subject to a limit imposed on network size. Our GPR approach achieved comparable results to the baseline RL approach, but the random samplers also performed well, especially the low-discrepancy Sobol sequence.

This work started during a two-month stay at *Hewlett Packard Enterprise* (HPE), former *HP Labs*, in Palo Alto, California. The problem of tuning mixed-precision quantization for CNNs is of interest to HPE researchers because it produces smaller and more energy-efficient networks, which are especially suited to embedded and custom-built hardware. We expanded the initial study in the last year of this thesis, still in collaboration with HPE researchers, but this work is still unpublished at the time of writing.

The remainder of this chapter is organized as follows. Section 14.1 describes the *HAQ* framework [142] for mixed-precision bit quantization, which we used to change the bitwidth of network layers, and which implements the baseline RL approach. Section 14.2 presents the *ResNet50* network used in our experiments and the ImageNet dataset. Section 14.4 discusses the implementation details of the methods we compared, and Section 14.5 evaluates their performance with respect to network accuracy. Section 4.3 summarizes the discussion and concludes the chapter.

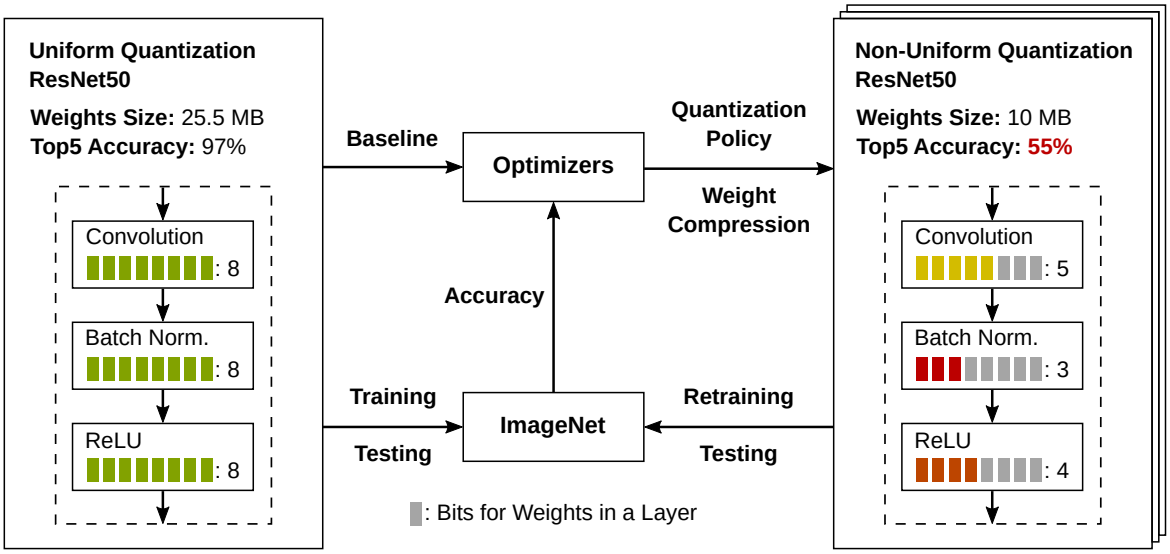

Figure 14.1: Optimizing nonuniform weight quantization policies for the *ResNet50* network, keeping total weight size below 10 *MB*, and attempting to keep accuracy. The optimization methods we compared were a Sobol Sampler, Reinforcement Learning [142], and Gaussian Process Regression with Expected Improvement

14.1 Autotuning Bit Precision for Convolutional Neural Networks

A Neural Network is composed of interconnected layers of "neurons". Each neuron can be interpreted as an agent that receives *weighted* inputs from connections with other neurons, combines them, and produces an output if the combined input crosses an *activation* threshold. The process of *quantization* of a network consists in choosing, according to a performance metric, the best bit precision for data representing weights and activation thresholds in each layer. Quantizing a network consequently determines the bit precision of the computations performed during training and inference. Quantization is increasingly relevant in Neural Network research [142, 145, 146, 147, 148, 149] because it reduces energy and space requirements, enabling deployment in embedded, mobile, or custom-engineered hardware.

Figure 14.2 shows the bit precision tuning experiments we present in this chapter. We used different methods to optimize quantization policies for the layer groups of the *PyTorch* [150] implementation of a *Residual Network* [151] with 50 residual blocks, called *ResNet50*. We used the *HAQ* framework [142] to quantize each layer with a bitwidth in the [1, 8] interval, producing a smaller network that was retrained on *NVIDIA* GPUs. We ensured that quantization policies respected size constraints, keeping total weight size under 10 *MB*, and allowing the comparison of our results to a reproduction of the approach of the original paper [142].

14.1.1 Further Applications of Autotuning to Neural Networks

Autotuning methods can be applied to optimize networks targeting different stages and structures. Online Gradient Descent coupled with backpropagation is a well established approach to network training [152, 153], but fine tuning of the network architecture and its parameters is also a promising target for the application of optimization methods.

A recent survey and taxonomy by Talbi [154] discusses work on *Neural Architecture Search* (NAS), and *Hyperparameter Tuning*. Neural Architecture Search consists of using optimization methods to find the modifications of the layers and connections of a network that best adapt to resource constraints or hardware architecture features. Hyperparameter Tuning refers to the optimization of any network parameters that have an impact on performance without necessarily requiring structure modification.

14.2 ResNet50 and ImageNet

14.2.1 ResNet50 Architecture

Residual Networks, or *ResNets*, were introduced by He *et al.* [151], and the intuition behind the idea is to add the original layer input, or *residual*, to the output of a stack of convolution layers. Keeping this residual component in a layer output enables better global optimization of the network. Figure 14.2 shows the *PyTorch* implementation of a 50-layer Residual Network, called *ResNet50*.

We generated this visualization of ResNet50's structure with the *HiddenLayer* library [155], adding the underlying structure of *Bottleneck Residual Blocks* in panel *(a)*, and of *Residual Blocks* in panel *(b)*. Bottleneck blocks serve the same purpose as Residual blocks, adding an input component to layer output, but are favored in larger networks because they are computationally less expensive.

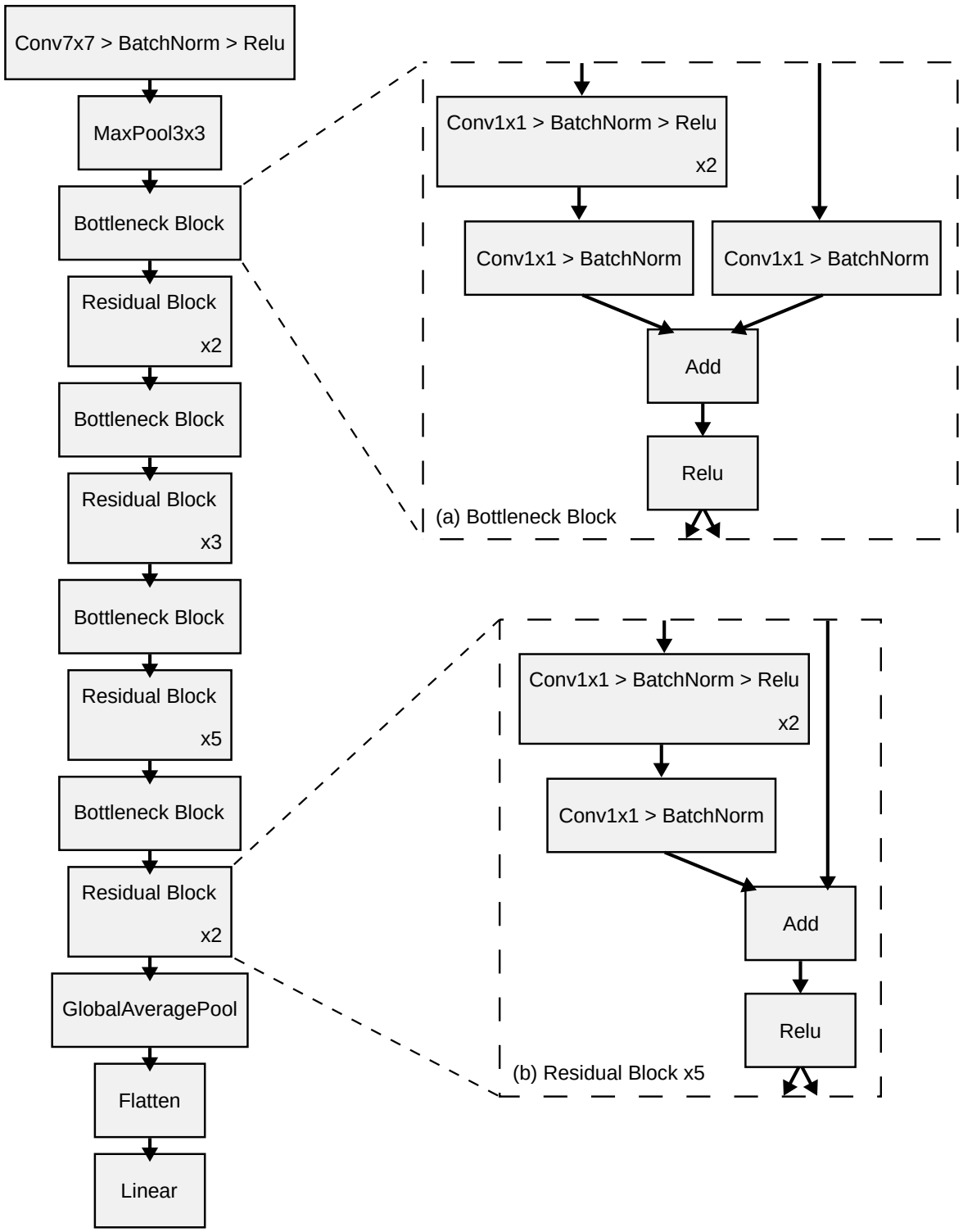

Figure 14.2: *ResNet50* architecture, implemented in *pytorch*

Figure 14.3: Images sampled from some of the categories in *ImageNet*, adapted from Deng *et al.* [143]

14.2.2 The ImageNet Dataset

- Add a figure and references
- Total size of dataset, categories
- Size of subset used, categories
- 2/3 training – 1/3 validation split

14.3 Search Space and Objective Function

- Targeted only weights, not activation
  - Activation code was released only after we did the study
- Size of the search space: $8^{54} \approx 10^{49}$ configurations
- Objective function: minimize accuracy loss with respect to 8-bit uniformly quantized ResNet50

14.4 Optimization Methods

14.4.1 Reinforcement Learning: The Baseline Method

- Describe on Online Learning Chapter, give small details here
- Adapt a figure from the HAQ paper?

Respecting Size Constraints

- Round-robin decrease of weights
- "Reverse-engineering": we found out that the first layers are less important, the authors might have encoded that knowledge in the round-robin procedure

14.4.2 Gaussian Process Regression with Expected Improvement

- Describe on Online Learning Chapter, give small details here
- Add a schematic figure or pseudocode of GPR algorithm

Using Space-Filling Designs

Respecting Size Constraints

14.4.3 Random and Quasi-Random Samplers

- Sobol Sequences

14.5 Performance Evaluation

14.5.1 Accuracy for Fixed Network Size

- Space-filling baseline
- Compare RL and GPR
- GPR achieves the same results in the same time
- GPR achieves more consistent quantization distributions across different optimizations

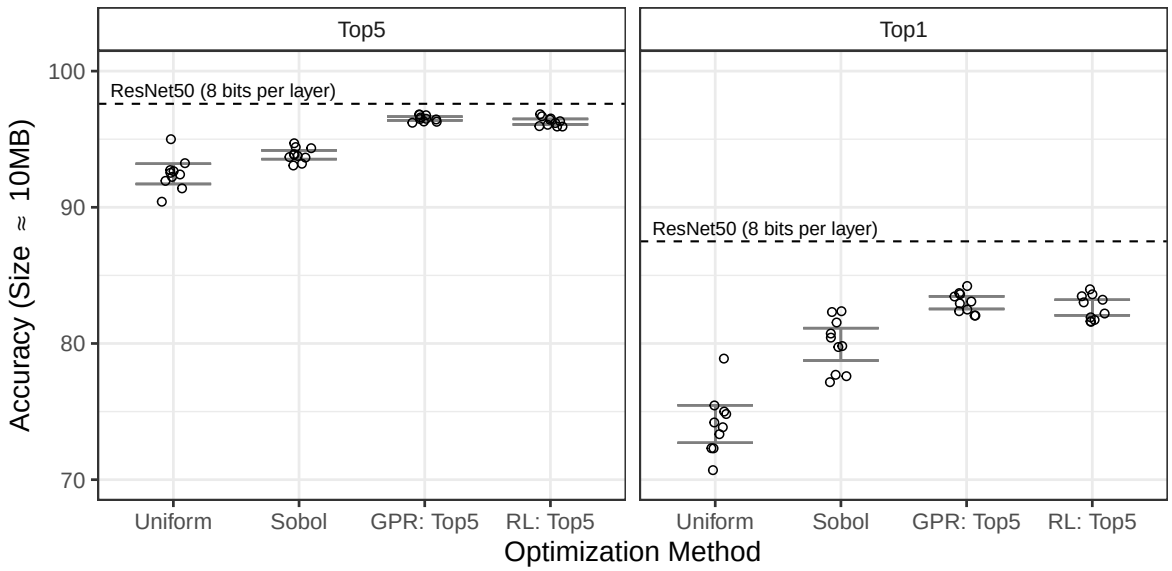

Figure 14.4: Mean estimates with 95% confidence intervals, for 10 repetitions of the 4 methods we compared.

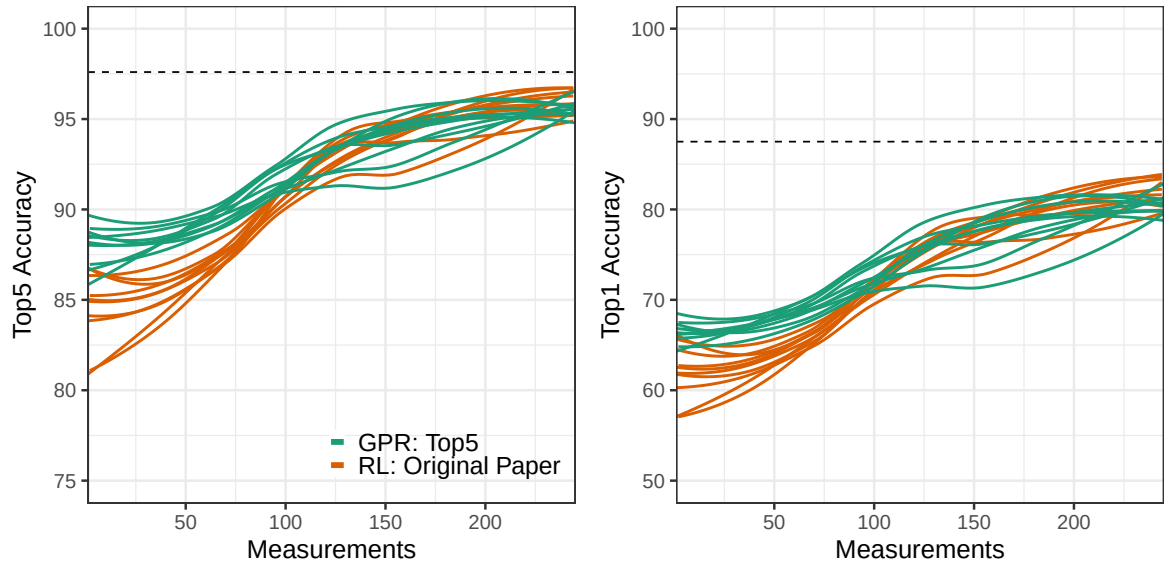

Figure 14.5: Best quantization policy found by each method during optimization, across 10 repetitions

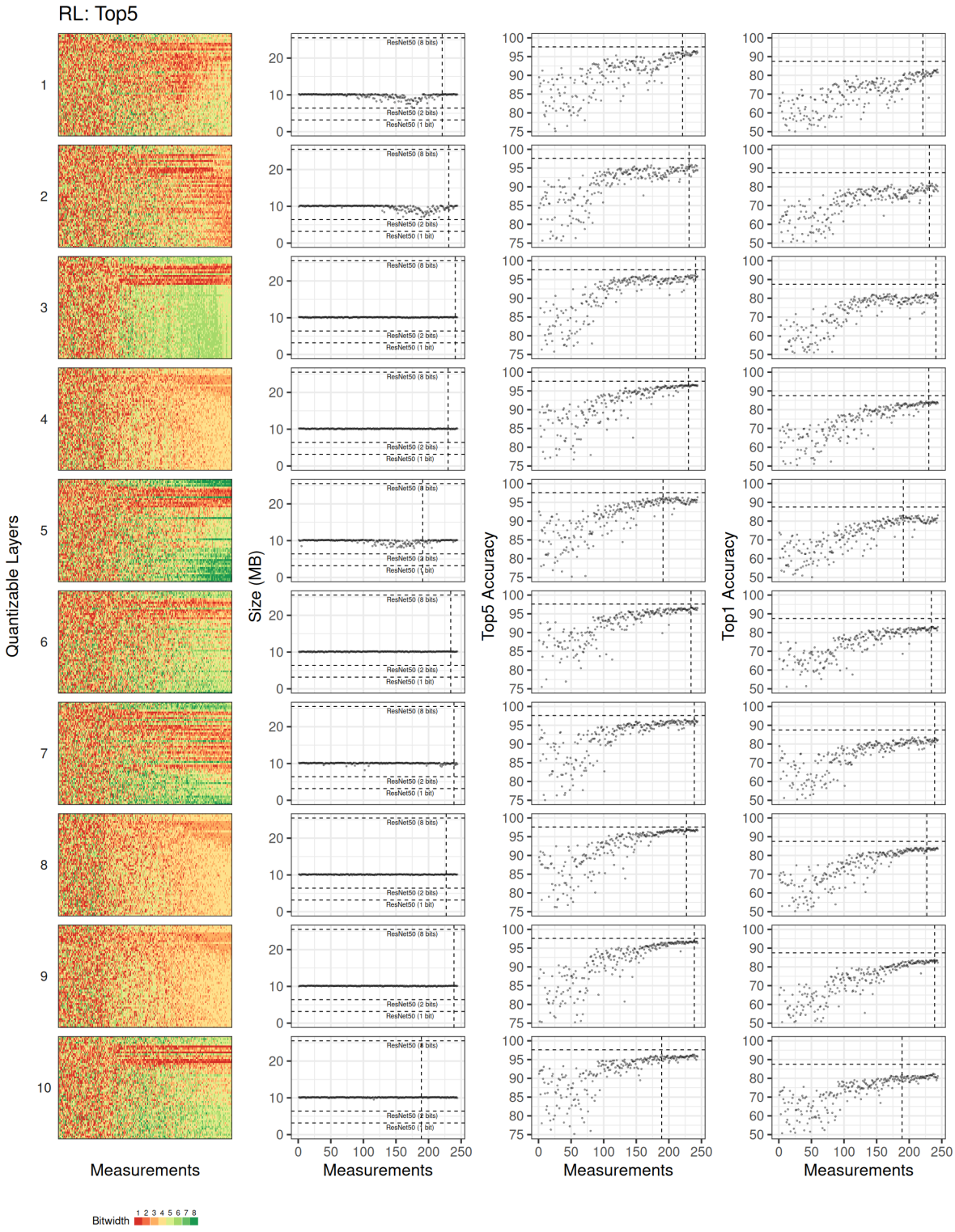

Figure 14.6: Results with the baseline Reinforcement Learning method

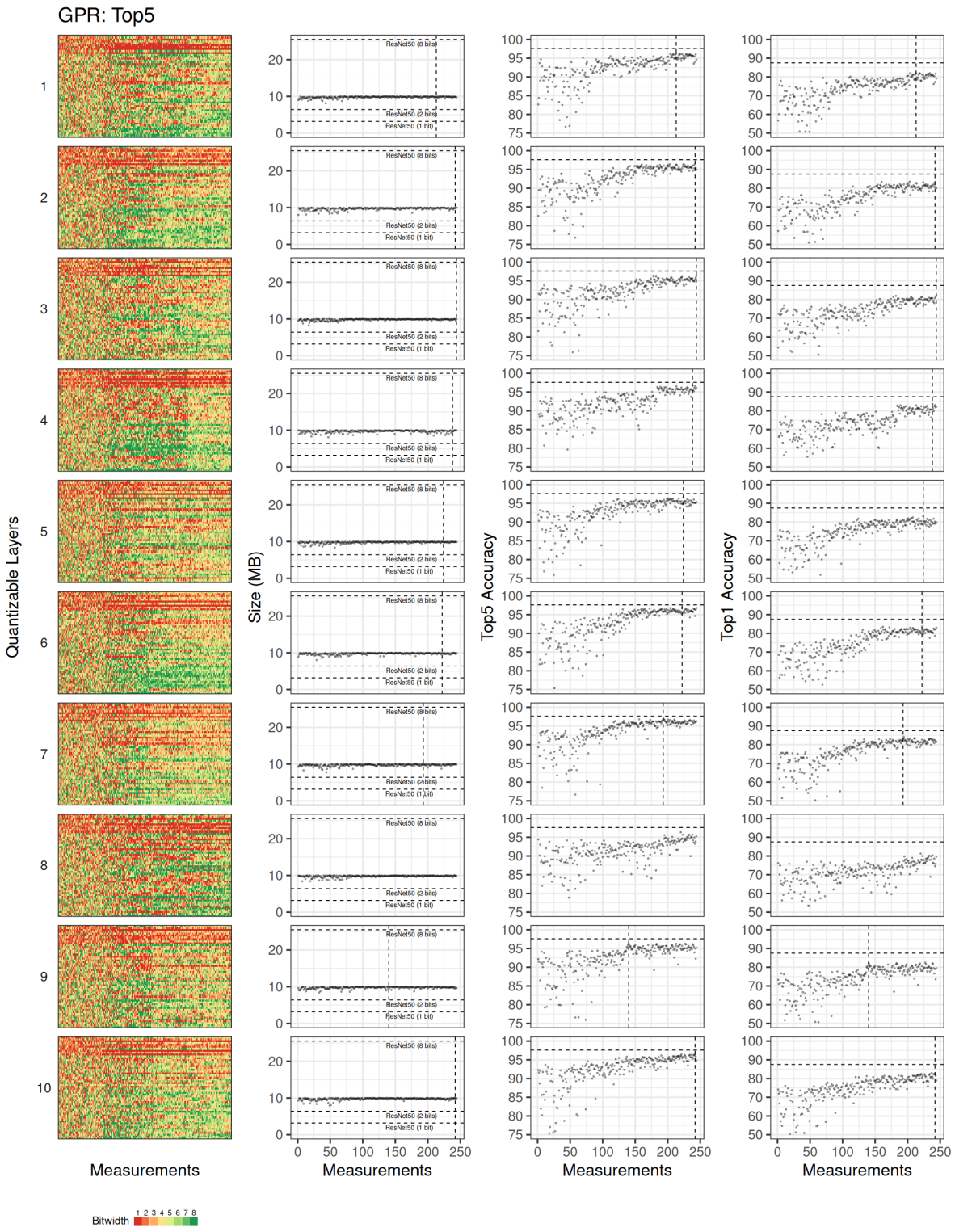

Figure 14.7: Results with Gaussian Process Regression

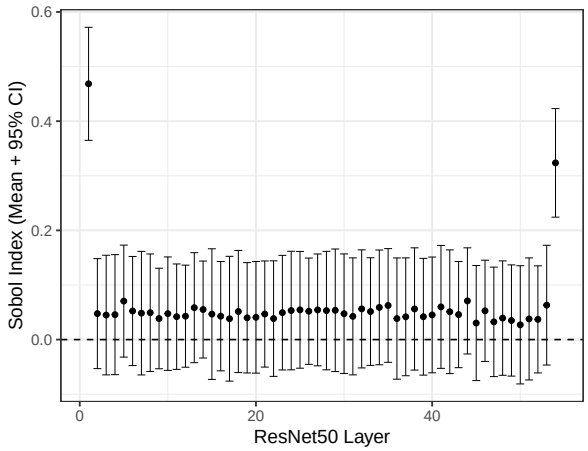

Figure 14.8: Sobol indices computed for the impact on *Top5* accuracy of each *ResNet50* layer, using Sobol samples

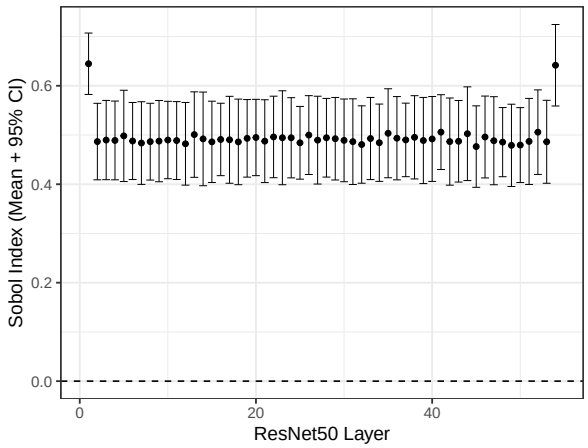

Figure 14.9: Sobol indices computed for the impact on *Top5* accuracy of each *ResNet50* layer, using Sobol samples and search paths performed by *GPR*

14.5.2 Running GPR with All Collected Data

- GPR can be restarted at any point and easily leverage past executions

14.5.3 Measuring the Performance of the GPR Method

- Training and testing the Net is the bottleneck
- Model fit times becomes more important with more data
  - Due to the R implementation? (it uses RCpp)

Detecting Significance with Sobol Indices

14.5.4 Perspective: Optimizing Accuracy and Weight

14.6 Summary

- Overfitting for both methods:
  - Variability of optimizations, Robustness
- Extensibility:
  - We can apply the same method with a different objective function and still get sensible results
- Restarting GPR with past experiments

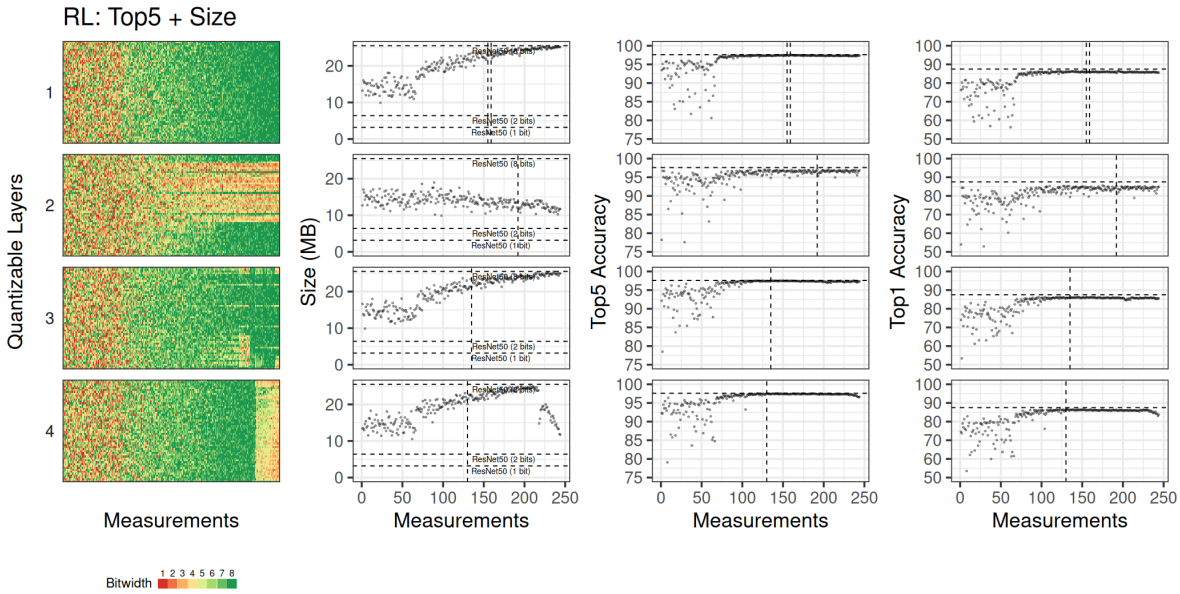

Figure 14.10: Optimizing accuracy and weight with the modified baseline Reinforcement Learning method

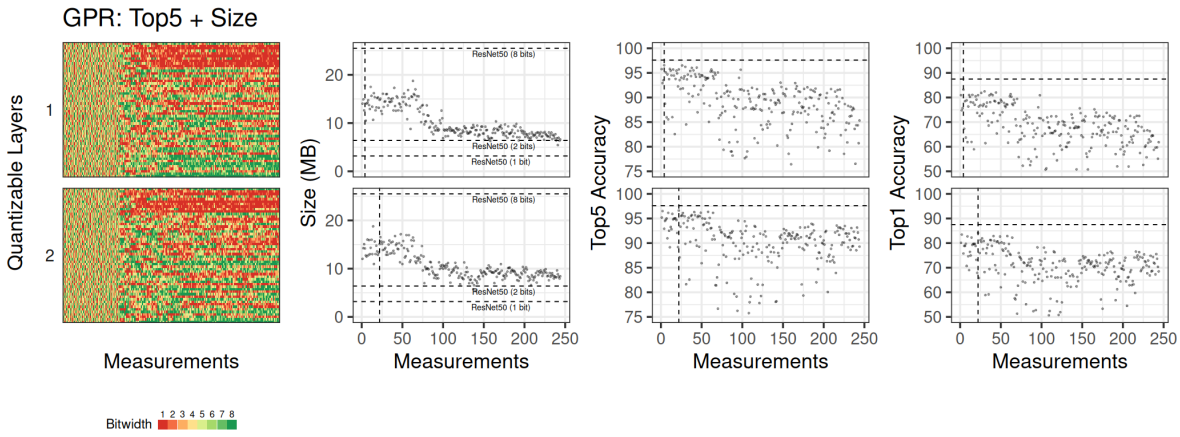

Figure 14.11: Optimizing accuracy and weight with Gaussian Process Regression with EI.

Part IV

Conclusion

Bibliography

[1] 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*. ACM, 2014, pp. 303–316.

[2] P. Bruel, M. Amarís, and A. Goldman, "Autotuning CUDA compiler parameters for heterogeneous applications using the OpenTuner framework," *Concurrency and Computation: Practice and Experience*, pp. e3973–n/a, 2017.

[3] P. Bruel, A. Goldman, S. R. Chalamalasetti, and D. Milojicic, "Autotuning High-Level Synthesis for FPGAs Using OpenTuner and LegUp," in *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2017.

[4] P. Bruel, S. Quinito Masnada, B. Videau, A. Legrand, J.-M. Vincent, and A. Goldman, "Autotuning under Tight Budget Constraints: A Transparent Design of Experiments Approach," in *The 19th Annual IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing (CCGrid 2019)*. IEEE/ACM, 2019.

[5] K. Seymour, H. You, and J. Dongarra, "A comparison of search heuristics for empirical code optimization." in *CLUSTER*, 2008, pp. 421–429.

[6] R. A. Fisher, *The design of experiments*. Oliver And Boyd; Edinburgh; London, 1937.

[7] P. N. D. Bukh, *The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling*. JSTOR, 1992.

[8] D. C. Montgomery, *Design and analysis of experiments*. John wiley & sons, 2017.

[9] G. E. Box, J. S. Hunter, and W. G. Hunter, *Statistics for experimenters: design, innovation, and discovery*. Wiley-Interscience New York, 2005, vol. 2.

[10] C. K. Williams and C. E. Rasmussen, *Gaussian processes for machine learning*. MIT press Cambridge, MA, 2006, vol. 2, no. 3.

[11] P. E. Ceruzzi, E. Paul *et al.*, *A history of modern computing*. MIT press, 2003.

[12] O. R. N. Laboratory, "Summit – Oak Ridge Leardership Computing Facility," https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit, 2020, [Online; accessed 08-Jun-2020].

[13] Wikipedia, "Transistor count," https://en.wikipedia.org/wiki/Transistor_count, 2020, [Online; accessed 08-Jun-2020].

[14] ——, "Microprocessor Chronology," https://en.wikipedia.org/wiki/Microprocessor_chronology, 2020, [Online; accessed 08-Jun-2020].

[15] G. E. Moore *et al.*, "Cramming more components onto integrated circuits," 1965.

[16] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of solid-state circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[17] D. J. Frank, R. H. Dennard, E. Nowak, P. M. Solomon, Y. Taur, and H.-S. P. Wong, "Device scaling limits of si mosfets and their application dependencies," *Proceedings of the IEEE*, vol. 89, no. 3, pp. 259–288, 2001.

[18] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual international symposium on computer architecture (ISCA)*. IEEE, 2011, pp. 365–376.

[19] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing and the quest for computing efficiency," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2015, pp. 1–6.

[20] H.-Y. Cheng, J. Zhan, J. Zhao, Y. Xie, J. Sampson, and M. J. Irwin, "Core vs. uncore: The heart of darkness," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2015, pp. 1–6.

[21] J. Henkel, H. Khdr, S. Pagani, and M. Shafique, "New trends in dark silicon," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (Dac)*. IEEE, 2015, pp. 1–6.

[22] Top500, "Top500 List," https://www.top500.org/, 2020, [Online; accessed 08-Jun-2020].

[23] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

[24] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 16, no. 1, pp. 1–17, 1990.

[25] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[26] B. Videau, K. Pouget, L. Genovese, T. Deutsch, D. Komatitsch, F. Desprez, and J.-F. Méhaut, "BOAST: A metaprogramming framework to produce portable and efficient computing kernels for hpc applications," *The International Journal of High Performance Computing Applications*, p. 1094342017718068, 2017.

[27] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-based empirical performance tuning using Orio," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–11.

[28] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: a language and compiler for algorithmic choice," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 38–49, 2009.

[29] P. Balaprakash, S. M. Wild, and B. Norris, "SPAPT: Search problems in automatic performance tuning," *Procedia Computer Science*, vol. 9, pp. 1959–1968, 2012.

[30] J.-H. Byun, R. Lin, K. A. Yelick, and J. Demmel, "Autotuning sparse matrix-vector multiplication for multicore," *EECS, UC Berkeley, Tech. Rep*, 2012.

[31] F. Petrovič, D. Střelák, J. Hozzová, J. Ol'ha, R. Trembeckỳ, S. Benkner, and J. Filipovič, "A benchmark set of highly-efficient cuda and opencl kernels and its dynamic autotuning with kernel tuning toolkit," *Future Generation Computer Systems*, 2020.

[32] P. Balaprakash, M. Salim, T. Uram, V. Vishwanath, and S. Wild, "Deephyper: Asynchronous hyperparameter search for deep neural networks," in *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE, 2018, pp. 42–51.

[33] P. Bruel, M. Amarís, and A. Goldman, "Autotuning gpu compiler parameters using opentuner," in *Proceedings of the WSCAD (Simpósio em Sistemas Computacionais de Alto Desempenho), 2015*.

[34] A. Mametjanov, P. Balaprakash, C. Choudary, P. D. Hovland, S. M. Wild, and G. Sabin, "Autotuning fpga design parameters for performance and power," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. IEEE, 2015, pp. 84–91.

[35] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Performance, design, and autotuning of batched gemm for gpus," in *International Conference on High Performance Computing*. Springer, 2016, pp. 21–38.

[36] C. Xu, G. Liu, R. Zhao, S. Yang, G. Luo, and Z. Zhang, "A parallel bandit-based approach for autotuning fpga compilation," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 157–166.

[37] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth, "A scalable auto-tuning framework for compiler optimization," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[38] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "ParamILS: an automatic algorithm configuration framework," *Journal of Artificial Intelligence Research*, vol. 36, no. 1, pp. 267–306, 2009.

[39] Y. Chu, C. Luo, H. H. Hoos, Q. Lin, and H. You, "Improving the performance of stochastic local search for maximum vertex weight clique problem using programming by optimization," *arXiv*, pp. arXiv–2002, 2020.

[40] I. Tuzov, D. de Andrés, and J.-C. Ruiz, "Tuning synthesis flags to optimize implementation goals: Performance and robustness of the leon3 processor as a case study," *Journal of Parallel and Distributed Computing*, vol. 112, pp. 84–96, 2018.

[41] M. M. Ziegler, H.-Y. Liu, G. Gristede, B. Owens, R. Nigaglioni, J. Kwon, and L. P. Carloni, "Syntunsys: A synthesis parameter autotuning system for optimizing high-performance processors," in *Machine Learning in VLSI Computer-Aided Design*. Springer, 2019, pp. 539–570.

[42] M. Gerndt, S. Benkner, E. César, C. Navarrete, E. Bajrovic, J. Dokulil, C. Guillén, R. Mijakovic, and A. Sikora, "A multi-aspect online tuning framework for hpc applications," *Software Quality Journal*, vol. 26, no. 3, pp. 1063–1096, 2018.

[43] J. Kwon, M. M. Ziegler, and L. P. Carloni, "A learning-based recommender system for autotuning design fiows of industrial high-performance processors," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[44] T. Wang, N. Jain, D. Beckingsale, D. Boehme, F. Mueller, and T. Gamblin, "Funcytuner: Auto-tuning scientific applications with per-loop compilation," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[45] J. Ol'ha, J. Hozzová, J. Fousek, and J. Filipovič, "Exploiting historical data: pruning autotuning spaces and estimating the number of tuning steps," in *European Conference on Parallel Processing*. Springer, 2019, pp. 295–307.

[46] 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 International Conference on Supercomputing, Vienna, Austria*, 1997.

[47] J. J. Dongarra and C. R. Whaley, "Automatically tuned linear algebra software (ATLAS)," *Proceedings of SC*, vol. 98, 1998.

[48] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the fft," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3. IEEE, 1998, pp. 1381–1384.

[49] C. Ţăpuş, I.-H. Chung, J. K. Hollingsworth *et al.*, "Active harmony: Towards automated performance tuning," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 2002, pp. 1–11.

[50] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*, vol. 16, no. 1. IOP Publishing, 2005, p. 521.

[51] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois *et al.*, "Milepost gcc: Machine learning enabled self-tuning compiler," *International journal of parallel programming*, vol. 39, no. 3, pp. 296–327, 2011.

[52] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch, "A multi-objective auto-tuning framework for parallel codes," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 2012, pp. 1–12.

[53] D. G. Spampinato and M. Püschel, "A basic linear algebra compiler," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 2014, p. 23.

[54] C. Audet, K.-C. Dang, and D. Orban, "Optimization of algorithms with opal," *Mathematical Programming Computation*, vol. 6, no. 3, pp. 233–254, 2014.

[55] C. Nugteren and V. Codreanu, "Cltune: A generic auto-tuner for opencl kernels," in *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2015 IEEE 9th International Symposium on*. IEEE, 2015, pp. 195–202.

[56] J. Guerreiro, A. Ilic, N. Roma, and P. Tomás, "Multi-kernel auto-tuning on gpus: Performance and energy-aware optimization," in *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*. IEEE, 2015, pp. 438–445.

[57] G. Fursin, A. Memon, C. Guillon, and A. Lokhmotov, "Collective mind, part ii: Towards performance-and cost-aware software engineering as a natural science," *arXiv preprint arXiv:1506.06256*, 2015.

[58] L.-W. Chang, I. El Hajj, C. Rodrigues, J. Gómez-Luna, and W.-m. Hwu, "Efficient kernel synthesis for performance portable programming," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016.

[59] C. G. Coelho, C. G. Abreu, R. M. Ramos, A. H. Mendes, G. Teodoro, and C. G. Ralha, "Mase-bdi: agent-based simulator for environmental land change with efficient and parallel auto-tuning," *Applied Intelligence*, vol. 45, no. 3, pp. 904–922, 2016.

[60] D. Beckingsale, O. Pearce, I. Laguna, and T. Gamblin, "Apollo: Reusable models for fast, dynamic tuning of input-dependent code," in *The 31th IEEE International Parallel and Distributed Processing Symposium*, 2017.

[61] M. J. Kochenderfer and T. A. Wheeler, *Algorithms for optimization*. Mit Press, 2019.

[62] A. J. Dobson and A. G. Barnett, *An introduction to generalized linear models*. CRC press, 2018.

[63] A. Agresti, *Foundations of linear and generalized linear models*. John Wiley & Sons, 2015.

[64] G. Seber, *The Linear Model and Hypothesis: A General Unifying Theory*. Springer, 2015.

[65] C. M. Bishop, *Pattern recognition and machine learning*. springer, 2006.

[66] Roustant, Olivier and Ginsbourger, David and Deville, Yves and Chevalier, Clement and Richet, Yann, "DiceKriging." [Online]. Available: https://cran.r-project.org/web/packages/DiceKriging/index.html

[67] O. Roustant, D. Ginsbourger, and Y. Deville, "DiceKriging, DiceOptim: Two R Packages for the Analysis of Computer Experiments by Kriging-Based Metamodeling and Optimization," vol. 51, no. 1, 2012, pp. 1–55.

[68] I. M. Sobol', "Sensitivity estimates for nonlinear mathematical models," *Math. Model. Comput. Exp*, vol. 1, no. 4, pp. 407–414, 1993.

[69] A. Saltelli, M. Ratto, T. Andres, F. Campolongo, J. Cariboni, D. Gatelli, M. Saisana, and S. Tarantola, *Global sensitivity analysis: the primer*. John Wiley & Sons, 2008.

[70] A. Saltelli, P. Annoni, I. Azzini, F. Campolongo, M. Ratto, and S. Tarantola, "Variance based sensitivity analysis of model output. design and estimator for the total sensitivity index," *Computer physics communications*, vol. 181, no. 2, pp. 259–270, 2010.

[71] R. L. Plackett and J. P. Burman, "The design of optimum multifactorial experiments," *Biometrika*, vol. 33, no. 4, pp. 305–325, 1946.

[72] U. Grömping, "R package FrF2 for creating and analyzing fractional factorial 2-level designs," *Journal of Statistical Software*, vol. 56, no. 1, pp. 1–56, 2014. [Online]. Available: http://www.jstatsoft.org/v56/i01/

[73] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2018. [Online]. Available: https://www.R-project.org/

[74] V. V. Fedorov, *Theory of optimal experiments*. Elsevier, 1972.

[75] B. Wheeler, *AlgDesign: Algorithmic Experimental Design*, 2014, R package version 1.1-7.3. [Online]. Available: https://CRAN.R-project.org/package=AlgDesign

[76] R. Carnell, *lhs: Latin Hypercube Samples*, 2018, R package version 0.16. [Online]. Available: https://CRAN.R-project.org/package=lhs

[77] P. Balaprakash, S. M. Wild, and P. D. Hovland, "Can search algorithms save large-scale automatic performance tuning?" in *ICCS*, 2011, pp. 2136–2145.

[78] ——, "An experimental study of global and local search algorithms in empirical performance tuning," in *International Conference on High Performance Computing for Computational Science*. Springer, 2012, pp. 261–269.

[79] P. Balaprakash, A. Tiwari, S. M. Wild, and P. D. Hovland, "AutoMOMML: Automatic Multi-objective Modeling with Machine Learning," in *High Performance Computing: 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, M. J. Kunkel, P. Balaji, and J. Dongarra, Eds. Springer International Publishing, 2016, pp. 219–239.

[80] M. Parsa, A. Ankit, A. Ziabari, and K. Roy, "PABO: Pseudo Agent-Based Multi-Objective Bayesian Hyperparameter Optimization for Efficient Neural Accelerator Design," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.

[81] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 44–54.

[82] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for gpu architectures," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011, pp. 382–393.

[83] M. Amaris, D. Cordeiro, A. Goldman, and R. Y. de Camargo, "A simple bsp-based model to predict execution time in gpu applications," in *High Performance Computing (HiPC), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 285–294.

[84] T. T. Dao, J. Kim, S. Seo, B. Egger, and J. Lee, "A performance model for gpus with caches," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 7, pp. 1800–1813, 2015.

[85] J. Picchi and W. Zhang, "Impact of l2 cache locking on gpu performance," in *SoutheastCon 2015*. IEEE, 2015, pp. 1–4.

[86] D. Sampaio, R. M. d. Souza, S. Collange, and F. M. Q. Pereira, "Divergence analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 35, no. 4, p. 13, 2013.

[87] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for gpu architectures," in *ACM Sigplan Notices*, vol. 45, no. 5. ACM, 2010, pp. 105–114.

[88] P. Guo and L. Wang, "Auto-tuning cuda parameters for sparse matrix-vector multiplication on gpus," in *Computational and Information Sciences (ICCIS), 2010 International Conference on*. IEEE, 2010, pp. 1154–1157.

[89] Y. Li, J. Dongarra, and S. Tomov, "A note on auto-tuning gemm for gpus," in *Computational Science–ICCS 2009*. Springer, 2009, pp. 884–892.

[90] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–10.

[91] A. Chaparala, C. Novoa, and A. Qasem, "Autotuning gpu-accelerated qap solvers for power and performance," in *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*. IEEE, 2015, pp. 78–83.

[92] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on gpus," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 99–108.

[93] "GitHub Repository for Autotuning CUDA Compiler Parameters for Heterogeneous Applications using the OpenTuner Framework," https://github.com/phrb/gpu-autotuning, accessed: 18 January 2021.

[94] "CUDA Compiler Driver," https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/, accessed: 18 January 2021.

[95] C. S. Ferreira, R. Y. Camargo, and S. W. Song, "A parallel maximum subarray algorithm on gpus," in *Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2014 International Symposium on*. IEEE, 2014, pp. 12–17.

[96] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek *et al.*, "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, no. 10, pp. 56–67, 2009.

[97] C. E. Alves, E. N. Cáceres, and S. W. Song, "Bsp/cgm algorithms for maximum subsequence and maximum subarray," *PVM/MPI*, vol. 3241, pp. 139–146, 2004.

[98] G. Holmes, A. Donkin, and I. H. Witten, "Weka: A machine learning workbench," in *Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*. IEEE, 1994, pp. 357–361.

[99] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim *et al.*, "A cloud-scale acceleration architecture," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–13.

[100] X. Yu, Y. Wang, J. Miao, E. Wu, H. Zhang, Y. Meng, B. Zhang, B. Min, D. Chen, and J. Gao, "A data-center fpga acceleration platform for convolutional neural networks," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 151–158.

[101] N. Tarafdar, T. Lin, E. Fukuda, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "Enabling flexible network fpga clusters in a heterogeneous cloud data center," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 237–246.

[102] R. Shu, P. Cheng, G. Chen, Z. Guo, L. Qu, Y. Xiong, D. Chiou, and T. Moscibroda, "Direct universal access: Making data center resources available to {FPGA}," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 127–140.

[103] J. Zhang, Y. Xiong, N. Xu, R. Shu, B. Li, P. Cheng, G. Chen, and T. Moscibroda, "The feniks fpga operating system for cloud computing," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, 2017, pp. 1–7.

[104] N. Tarafdar, N. Eskandari, V. Sharma, C. Lo, and P. Chow, "Galapagos: A full stack approach to fpga integration in the cloud," *IEEE Micro*, vol. 38, no. 6, pp. 18–24, 2018.

[105] "Unleash Your Data Center for Intel Xeon CPU with FPGAs, howpublished = https://www.intel.com/content/www/us/en/data-center/products/programmable/overview.html, note = Accessed: 25 January 2020."

[106] "Adaptable Acceleration for the Modern Data Center, howpublished = https://www.xilinx.com/applications/data-center.html, note = Accessed: 25 January 2020."

[107] D. Singh, "Implementing fpga design with the opencl standard," *Altera whitepaper*, 2011.

[108] T. Feist, "Vivado design suite," *White Paper*, vol. 5, 2012.

[109] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 8–17, 2009.

[110] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, p. 24, 2013.

[111] "LegUp Computing, now closed-source, paid software," https://www.legupcomputing.com/, accessed: 26 January 2021.

[112] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[113] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "Chstone: A benchmark program suite for practical c-based high-level synthesis," in *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1192–1195.

[114] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in c with roccc 2.0," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*. IEEE, 2010, pp. 127–134.

[115] P. Coussy, G. Lhairech-Lebreton, D. Heller, and E. Martin, "Gaut a free and open source high-level synthesis tool," in *IEEE DATE*, 2010.

[116] V. V. Kindratenko, R. J. Brunner, and A. D. Myers, "Mitrion-c application development on sgi altix 350/rc100," in *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*. IEEE, 2007, pp. 239–250.

[117] A. Antola, M. D. Santambrogio, M. Fracassi, P. Gotti, and C. Sandionigi, "A novel hardware/software codesign methodology based on dynamic reconfiguration with impulse c and codeveloper," in *Programmable Logic, 2007. SPL'07. 2007 3rd Southern Conference on*. IEEE, 2007, pp. 221–224.

[118] S. Loo, B. E. Wells, N. Freije, and J. Kulick, "Handel-c for rapid prototyping of vlsi coprocessors for real time systems," in *System Theory, 2002. Proceedings of the Thirty-Fourth Southeastern Symposium on*. IEEE, 2002, pp. 6–10.

[119] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. PP, no. 99, pp. 1–1, 2016.

[120] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed *et al.*, "Vtr 7.0: Next generation architecture and cad system for fpgas," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 2, p. 6, 2014.

[121] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, N. Calagar, S. Brown, and J. Anderson, "The effect of compiler optimizations on high-level synthesis-generated hardware," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 3, p. 14, 2015.

[122] S. W. Nabi and W. Vanderbauwhede, "A fast and accurate cost model for fpga design space exploration in hpc applications," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 114–123.

[123] A. C. Canis, "Legup: Open-source high-level synthesis research framework," Ph.D. dissertation, University of Toronto, 2015.

[124] S. Hadjis, A. Canis, J. H. Anderson, J. Choi, K. Nam, S. Brown, and T. Czajkowski, "Impact of fpga architecture on resource sharing in high-level synthesis," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, 2012, pp. 111–114.

[125] "LegUp Documentation, now closed-source, paid software," https://www.legupcomputing.com/main/resource, accessed: 18 January 2021.

[126] "GitHub Repository for Autotuning High-Level Synthesis for FPGAs using OpenTuner and LegUp," https://github.com/phrb/legup-tuner, accessed: 18 January 2021.

[127] "GitHub Repository for the dockerfile used in Autotuning High-Level Synthesis for FPGAs using OpenTuner and LegUp," https://github.com/phrb/legup-dockerfile, accessed: 18 January 2021.

[128] S. Q. Masnada, "Semi-Automatic Performance Optimization of HPC Kernels," Master's thesis, Université Grenoble Alpes, Jun. 2016. [Online]. Available: https://hal.inria.fr/hal-01579422

[129] "BOAST Source Code," https://github.com/Nanosim-LIG/boast, accessed: 15 March 2021.

[130] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan, "POET: Parameterized optimizations for empirical tuning," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, 2007, pp. 1–8.

[131] "BOAST Laplacian Sample GPU Kernel," https://github.com/Nanosim-LIG/boast/blob/master/sample/Laplacian.rb, accessed: 15 March 2021.

[132] J. Fox and S. Weisberg, *An R Companion to Applied Regression*, 2nd ed. Thousand Oaks CA: Sage, 2011. [Online]. Available: http://socserv.socsci.mcmaster.ca/jfox/Books/Companion

[133] S. Addelman and O. Kempthorne, "Some main-effect plans and orthogonal arrays of strength two," *The Annals of Mathematical Statistics*, pp. 1167–1176, 1961.

[134] U. Grömping and R. Fontana, "An algorithm for generating good mixed level factorial designs," Beuth University of Applied Sciences, Berlin, Tech. Rep., 2018.

[135] L.-Y. Deng and B. Tang, "Generalized resolution and minimum aberration criteria for plackett-burman and other nonregular factorial designs," *Statistica Sinica*, pp. 1071–1082, 1999.

[136] A. C. Atkinson and A. N. Donev, "The construction of exact d-optimum experimental designs with application to blocking response surface designs," *Biometrika*, vol. 76, no. 3, pp. 515–526, 1989.

[137] J. Kiefer, "General equivalence theory for optimum designs (approximate theory)," *The annals of Statistics*, pp. 849–879, 1974.

[138] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science, I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan, Eds. Springer International Publishing, 2013, vol. 367, pp. 3–20.

[139] "APP Metrics for Intel Xeon Processors," https://www.intel.com/content/dam/support/us/en/documents/processors/APP-for-Intel-Xeon-Processors.pdf, accessed: 04 March 2021.

[140] "Intel Xeon Processor E5-2630 v3," https://ark.intel.com/content/www/us/en/ark/products/83356/intel-xeon-processor-e5-2630-v3-20m-cache-2-40-ghz.html, accessed: 04 March 2021.

[141] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[142] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: Hardware-Aware Automated Quantization With Mixed Precision," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 8612–8620.

[143] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[144] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[145] A. Elthakeb, P. Pilligundla, F. Mireshghallah, A. Yazdanbakhsh, S. Gao, and H. Esmaeilzadeh, "ReLeQ: An automatic reinforcement learning approach for deep quantization of neural networks," in *NeurIPS ML for Systems workshop, 2018*, 2019.

[146] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *arXiv preprint arXiv:1612.01064*, 2016.

[147] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[148] A. Mishra, E. Nurvitadhi, J. J. Cook, and D. Marr, "Wrpn: Wide reduced-precision networks," *arXiv preprint arXiv:1709.01134*, 2017.

[149] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[150] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[151] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[152] D. R. Wilson and T. R. Martinez, "The general inefficiency of batch training for gradient descent learning," *Neural networks*, vol. 16, no. 10, pp. 1429–1451, 2003.

[153] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.

[154] E.-G. Talbi, "Automated design of deep neural networks: A survey and unified taxonomy," *ACM Computing Surveys (CSUR)*, vol. 54, no. 2, pp. 1–37, 2021.

[155] "HiddenLayer: A lightweight library for neural network graphs and training metrics for PyTorch, Tensorflow, and Keras," https://github.com/waleedka/hiddenlayer, accessed: 24 March 2021.