Towards Transparent and Parsimonious Methods for Automatic Performance Tuning

Pedro Bruel

June 27, 2020

Contents

The Need for Autotuning

1.1 Introduction

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.

The contributions of this thesis are strategies to apply to program autotuning the statistical learning methods of *Design of Experiments* [1], or *Experimental Design*, and *Gaussian Process Regression* [2]. 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 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.

1.2 Historical Trends in Hardware Design

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 [3]. 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* [4], which spans an entire room.

Figure 1.1 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 [7], 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 [8] 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 1.1 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 [9].

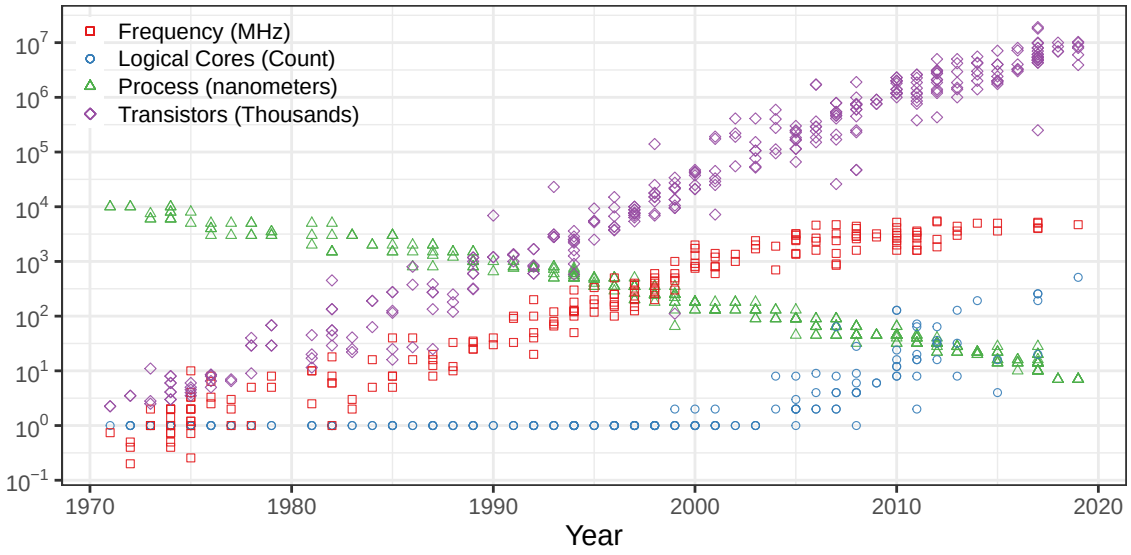

Figure 1.1: 49 years of microprocessor data, highlighting the sustained exponential increases and reductions on transistor counts and fabrication processes, the stagnation of frequency scaling around 2005, and one solution found for it, the simultaneous exponential increase on logical core count. Data from Wikipedia [5, 6]

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* [10], and presents significant challenges to current hardware designers and programmers [11, 12, 13].

The *Top500* [14] list gathers information about commercially available supercomputers, and ranks them by performance on the *LINPACK* benchmark [15]. Figure 1.2 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 1.3 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.

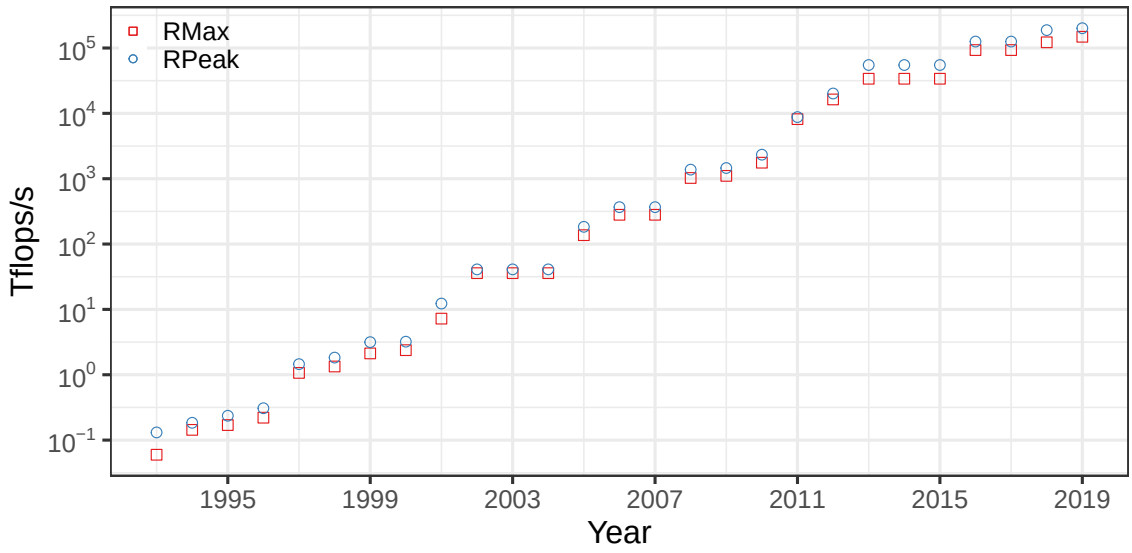

Figure 1.2: Sustained exponential increase of theoretical *RPeak* and achieved *RMax* performance for the supercomputer ranked 1st on TOP500 [14]

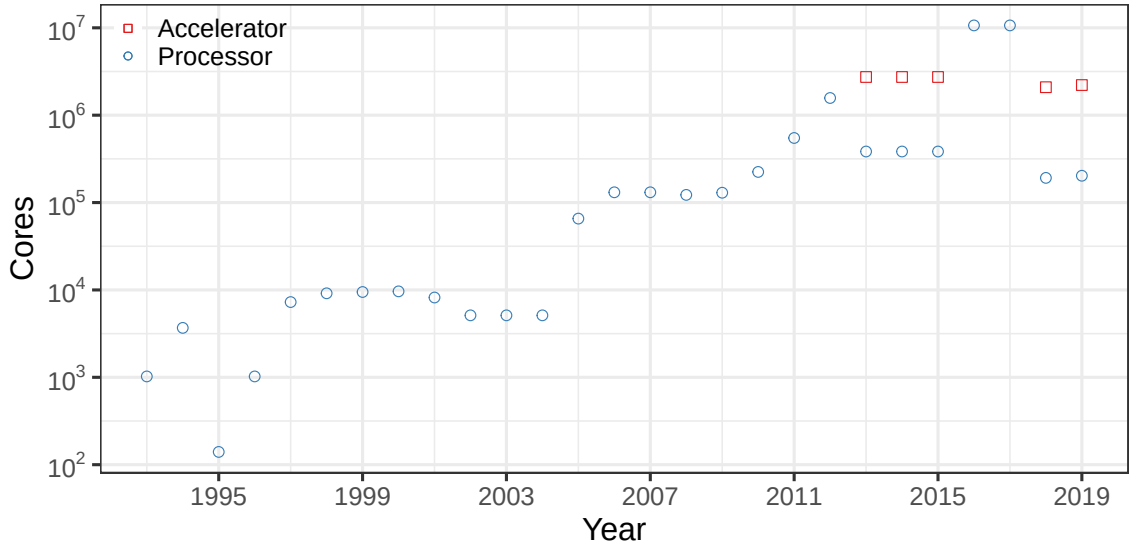

Figure 1.3: Processor and accelerator core count in supercomputers ranked 1st on TOP500 [14]. Core count trends for supercomputers are not necessarily bound to processor trends observed on Figure 1.1.

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 tuning 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 3.

1.3 Loop Nest Optimization as an Autotuning Problem

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) [16], and others from the original BLAS [17], 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 1.4 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 1.4a. The shaded elements on the top row of Figure 1.5 represent the elements that could be prefetched in iterations of Figure 1.4a.

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 1.4b.

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

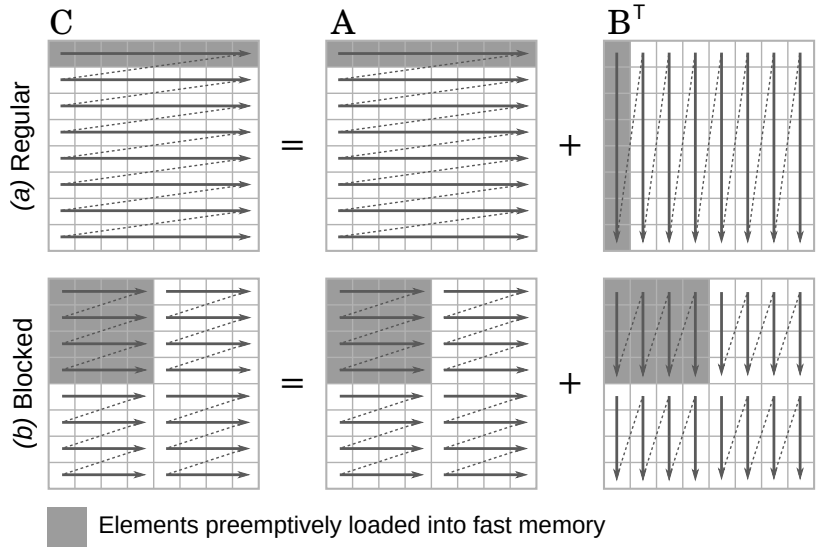

Figure 1.5: Access patterns for matrices in $C = A + B^\top$, with loop nest optimizations. Panel *(a)* shows the access order of a regular implementation, and panel *(b)* shows the effect of loop *tiling*, or *blocking*

Now, memory accesses are be performed in *tiles*, as shown on the bottom row of Figure 1.5. 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 1.4c.

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, 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 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.* [18], and considers 128 blocking and unrolling values, in the interval $[0, 127]$, for the GEMM algorithm. The three panels of Figure 1.6 show conceptual implementations of loop blocking and unrolling for GEMM in *C*. A block size of zero results in the implementation from Figure 1.6a, 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 1.6, but the unrolling factor is not exposed as a parameter. To test different unrolling values we need 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 [19, 20, 21]. 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 1.7, was measured for a sequential GEMM implementation, using square matrices of size 400 [18]. 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 1.7, but local features jump to the eyes, such as the "valley" across

```
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

```
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

```
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 1.6: Loop nest optimizations for GEMM, in C

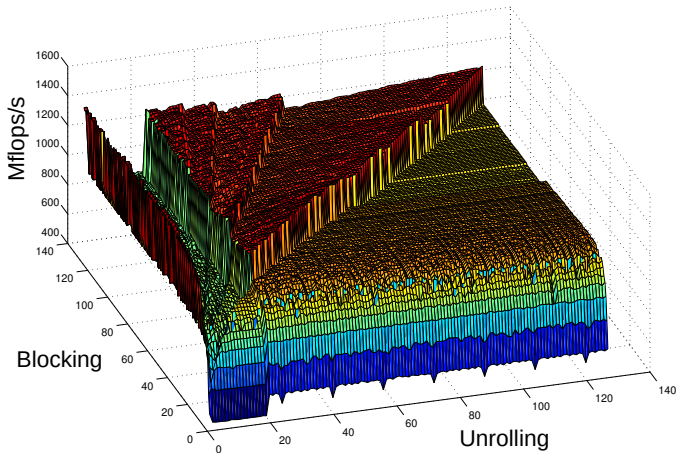

Figure 1.7: A search space defined for loop blocking and unrolling parameters, in a matrix multiplication kernel [18]

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

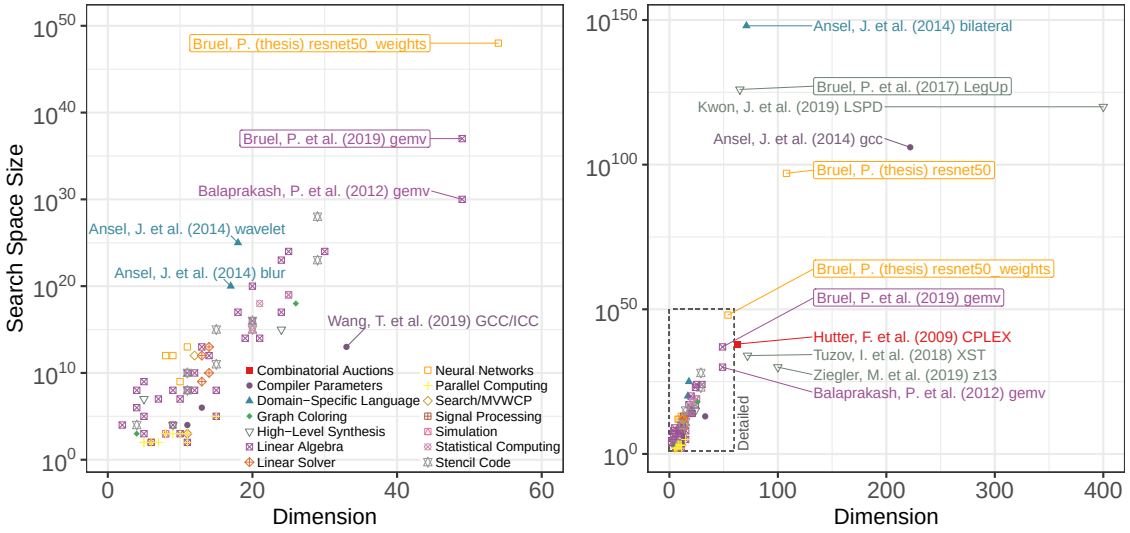

Figure 1.8: Dimension and search space size for autotuning problems from 14 domains [22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 18] The left panel shows a zoomed view of the right panel

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.

1.4 Characterizing Search Spaces

Autotuning methods are used to improve performance in an increasingly large variety of domains, from the initial 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. It is not far-fetched to link 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 1.8.

System	Domain	Method	Year
PhiPAC [42]	Linear Algebra	SH (Exhaustive)	1997
ATLAS [43]	Linear Algebra	SH (Exhaustive)	1998
FFTW [44]	Digital Signal Processing	SH (Exhaustive)	1998
Active Harmony [45]	Domain-Specific Language	SH	2002
OSKI [46]	Linear Algebra	SH	2005
Seymour, K. *et al.* [18]	Linear Algebra	SH	2008
PRO [33]	Linear Algebra	SH	2009
ParamILS [34]	Combinatorial Auctions	SH	2009
PetaBricks [21]	Domain-Specific Language	SH (Genetic Algorithm)	2009
MILEPOST GCC [47]	Compiler Parameters	ML	2011
Orio [22]	Linear Algebra	ML (Decision Trees)	2012
pOSKI [24]	Linear Algebra	SH	2012
INSIEME [48]	Compiler Parameters	SH (Genetic Algorithm)	2012
OpenTuner [23]	Compiler Parameters	SH	2014
Lgen [49]	Linear Algebra	SH	2014
OPAL [50]	Parallel Computing	SH	2014
Mametjanov, A. *et al.* [30]	High-Level Synthesis	ML (Decision Trees)	2015
CLTune [51]	Parallel Computing	SH	2015
Guerreirro, J. *et al.* [52]	Parallel Computing	SH	2015
Collective Mind [53]	Compiler Parameters	ML	2015
Abdelfattah, A. *et al.* [31]	Linear Algebra	SH (Exhaustive)	2016
TANGRAM [54]	Domain-Specific Language	SH	2016
MASE-BDI [55]	Environmental Land Change	SH	2016
Xu, C. *et al.* [32]	High-Level Synthesis	SH	2017
Apollo [56]	Parallel Computing	ML (Decision Trees)	2017
DeepHyper [26]	Neural Networks	ML (Decision Trees)	2018
Tuzov, I. *et al.* [36]	High-Level Synthesis	Design of Experiments	2018
Periscope [38]	Compiler Parameters	SH	2018
SynTunSys [37]	High-Level Synthesis	SH	2019
Kwon, J. *et al.* [39]	High-Level Synthesis	ML	2019
FuncyTuner [40]	Compiler Parameters	SH	2019
Ol'ha, J. *et al.* [41]	Parallel Computing	Sensitivity Analysis	2019
Petrovic, F. *et al.* [25]	Linear Algebra	SH	2020
Chu, Y. *et al.* [35]	Search/MVWCP	SH	2020

1.5 Thesis Contributions

1.6 Text Structure

Chapter 3 presents background for the application of methods derived from search heuristics, mathematical optimization, and statistical learning to autotuning.

Efforts for Reproducible Science

2.1 Introduction

2.2 Computational Documents with Emacs

2.3 Versioning for Code, Text, and Data

Search Heuristics and Statistical Learning

3.1 Introduction

- Methods Tree

3.2 Search Heuristics

3.2.1 Underlying Implicit Hypotheses

3.2.2 Optimization without Surrogate Models

3.2.3 OpenTuner: Leveraging Ensembles of Heuristics

3.2.4 Software for Autotuning with Heuristics

3.2.5 Results with GPU Compiler Parameters

3.2.6 Results with High-Level Synthesis for FPGAs

3.3 Statistical Learning

3.3.1 Introduction

- Explicit surrogate models
- The Bias-Variance Trade-off
- Inference and prediction
- Supervised and unsupervised learning

- Parametric and Nonparametric Methods

3.3.2 Supervised Methods for Regression and Classification

- KNN, Linear Regression, Logistic Regression, Neural Networks
- Briefly mention others

3.3.3 Model Assessment

- Train and Test Sets
- Train and Test Mean Squared Error
- Cross Validation, Bootstrapping
- The question of choosing experiments X is never posed

3.3.4 Linear Regression

- Ordinary Least Squares: Gauss-Markov (BLUE)
- Ridge, Lasso
- Transformations of X

3.3.5 Software for Autotuning with Statistical Learning

3.3.6 Results

- Briefly mention results, link to following chapters

A Design of Experiments Methodology

4.1 Introduction

- Poses the question of choosing the experiments X
- Mixing categorical and numerical (continuous and discrete) factors

4.2 Inference with Analysis of Variance

4.3 Inference for Autotuning under a Tight Budget

4.3.1 Screening for Main Effects of GPU Compiler Parameters

4.3.2 Fractional Factorial Designs for Chip Design on FPGAs

4.3.3 Optimal Designs for a GPU Laplacian Kernel

4.4 Optimal Design

4.4.1 The KL Exchange Algorithm

4.4.2 Prediction and Inference for HPC Kernels

1. CCGRID Paper
2. Later developments

Gaussian Process Regression

5.1 Introduction

- Sampling functions
- Covariance Kernels: modeling uncertainty regions
- Expressing structure?

5.2 References

Bibliography

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

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

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

[4] 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].

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

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

[23] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014, pp. 303–316.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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*.

[29] ——, "Autotuning CUDA compiler parameters for heterogeneous applications using the OpenTuner framework," *Concurrency and Computation: Practice and Experience*, pp. e3973–n/a, 2017.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

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

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.