Autotuning Methods

Pedro Bruel

June 15, 2020

Contents

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 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 program optimization 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*, re-

lationships between parameters and performance. Choosing an autotuning method requires determining a balance between the exploration of a problem, when we would seek to discover and explain relationships between parameters and performance, and the exploitation of the best optimizations we can find, when we would seek only to minimize 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.

1.1 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 [5], who also postulated its continuity. The performance improvements produced by the design efforts to make Moore's forecast a self-fulfilling

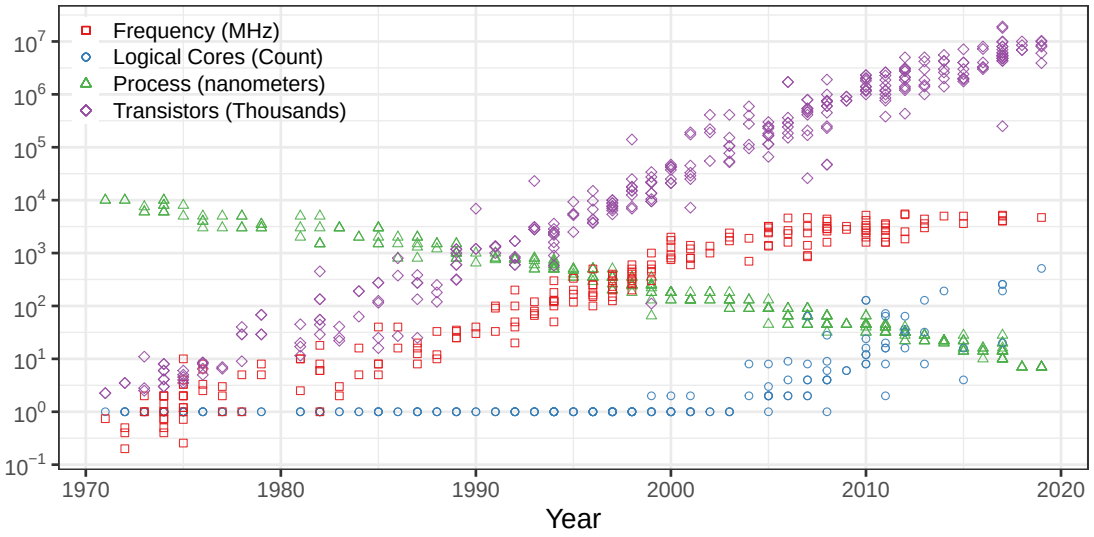

Figure 1.1: 49 years of microprocessor data, highlighting the sustained exponential increases and reductions on transistor counts and fabrication processes, the stagnation of frequency scaling around 2005, and one solution found for it, the simultaneous exponential increase on logical core count. Data from Wikipedia [6, 7]

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].

Current hardware is now past the effects of Dennard's scaling. The increase in logical cores, still 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 still presents significant challenges to hardware designers and programmers [11, 12, 13].

The *Top500* [14] list gathers information about commercially available supercom-

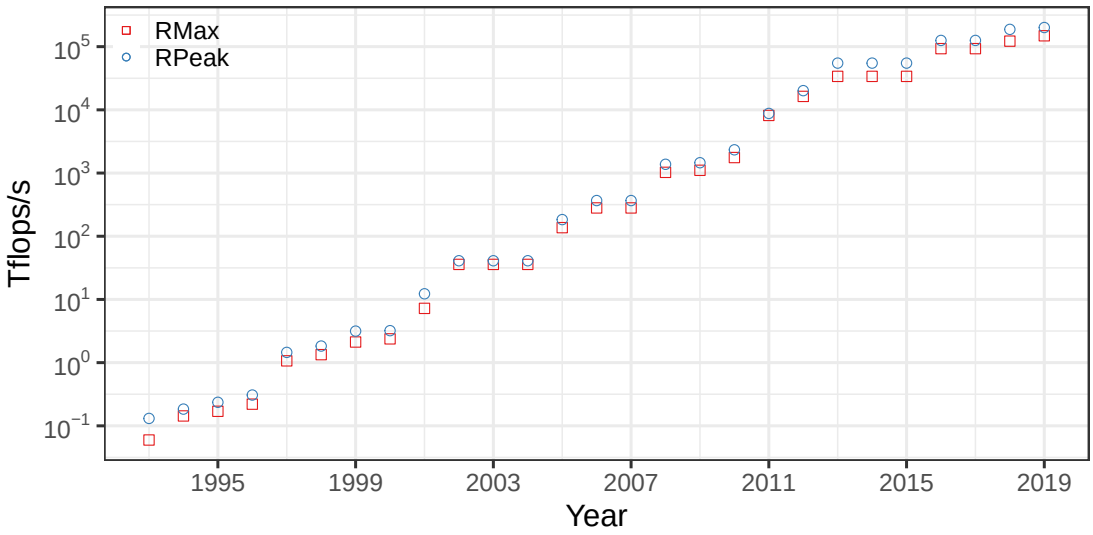

Figure 1.2: Sustained exponential increase of theoretical *RPeak* and achieved *RMax* performance for the supercomputer ranked 1st on TOP500 [14]

puters, and ranks them by their 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 continued its exponential climb, sustained mostly by software improvements.

Although hardware accelerators, such as GPUs and FPGAs, have also helped to sustain exponential performance increase, 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 of software 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

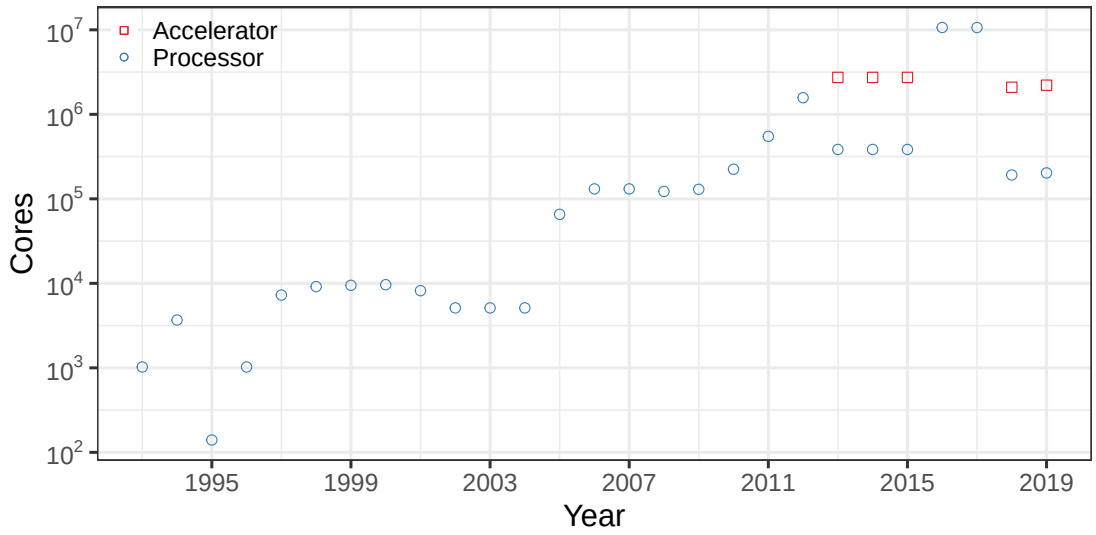

Figure 1.3: Processor and accelerator core count in supercomputers ranked 1st on TOP500 [14]. Core count trends for supercomputers are not necessarily bound to processor trends observed on Figure 1.1.

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 section introduces a simple autotuning problem, and presents an overview of autotuning search spaces from the size perspective. The section also briefly introduces methods used to explore autotuning search spaces, which are discussed in Chapter 3.

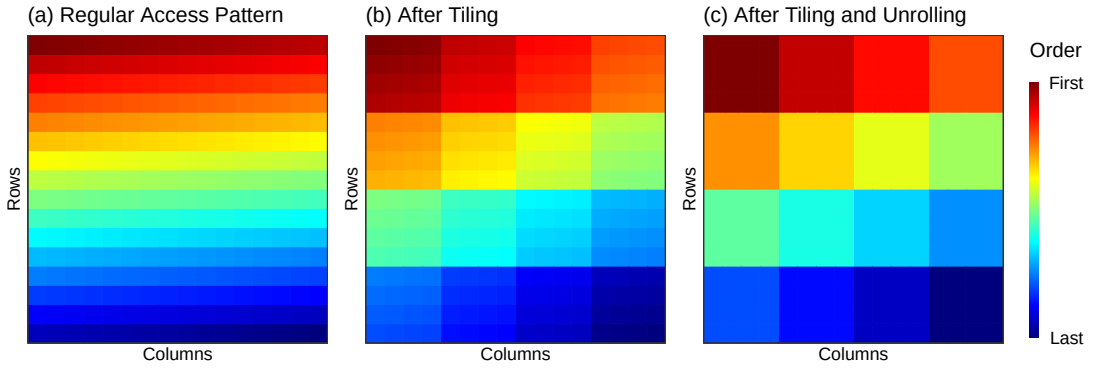

Figure 1.4: Access patterns on GEMM, for the *destination* matrix, using loop nest optimizations. Panel *(a)* shows the access order of a naive implementation, panel *(b)* shows the effect of loop *tiling*, or *blocking*, and panel *(c)* shows the compounded effect of loop *unrolling*

1.2 Characterizing Search Spaces

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], can be an interesting example of autotuning. One way to improve the performance of such 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 now briefly describe loop tiling and unrolling for GEMM, and how these methods generate a relatively large and complex search space, which we can explore with autotuning methods.

Figure 1.5 shows three GEMM implementations in C. Figure 1.5a shows a naive implementation, with three nested loops spanning the size of the square matrices, accessing elements of the *destination* matrix *C* in the order shown on the leftmost panel of Figure 1.4. If loop indices are split into blocks, as shown in Figure 1.5b, memory accesses will be performed in *tiles*, shown on the center panel of Figure 1.4, which can improve performance by using only elements loaded into the cache. Loop unrolling, shown conceptually in Figure 1.5c, is an additional source transformation that can be performed independently, or in addition to tiling, and that can improve performance by grouping iterations of a loop, as represented on the rightmost panel of Figure 1.4, ensuring that processor registers are used.

```
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++){
  for(j = 0; j < N; j++){
    for(k = 0; k < N; k++){
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

(a) Naive 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){
    for(k = 0; k < N; k++){
      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];
        }
      }
    }
  }
}
```

(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){
    for(k = 0; k < N; k++){
      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];
    }
  }
}
```

(c) Tiled and unrolled

Figure 1.5: Loop nest optimizations for GEMM, in C

The two *parameters* involved in this example 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 available registers, but not so large that it forces memory to the stack.

The values of block size and unrolling factor that optimize performance depend on the cache hierarchy and register layout of the target processor, and on the memory access pattern of the target algorithm. An autotuner should also aim to account for the *interaction* between parameters, that is, for the fact that the best value for each parameter also depends on the value chosen for the other.

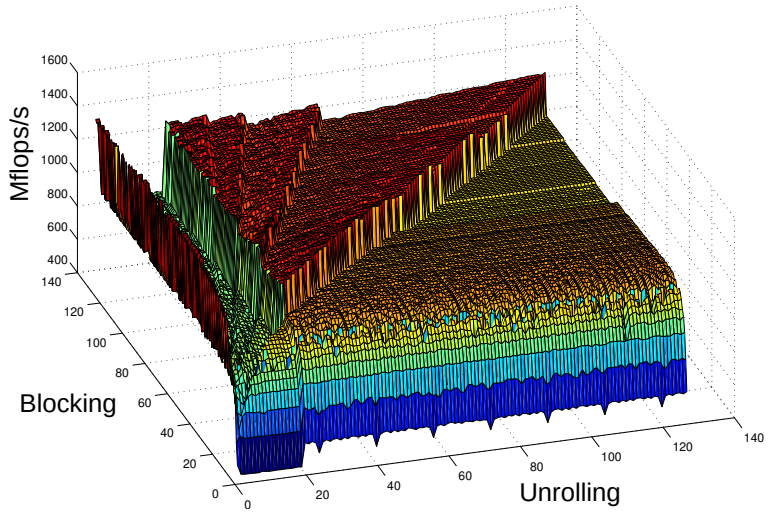

Figure 1.6: A search space defined for loop blocking and unrolling parameters, in a matrix multiplication kernel [18]

It is straightforward to change the block size of the implementations from Figure 1.5, 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. The loop nest optimization example in this section comes from Seymour *et al.* [18], and considers 128 integer values in the interval $[0, 127]$. A block size of zero results in the implementation from Figure 1.5a, and an unrolling factor of zero performs a single iteration per condition check.

The performance of each of the $128^2 = 16384$ combinations of blocking and unrolling factor, shown in *Mflops/s* in Figure 1.6, was measured for a sequential matrix multiplication algorithm, 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. An autotuner's objective is to find the *highest* point of this landscape, since the objective is to *maximize* Mflops/s, although it is more usual to transform the performance metric so that the objective is to *minimize* it.

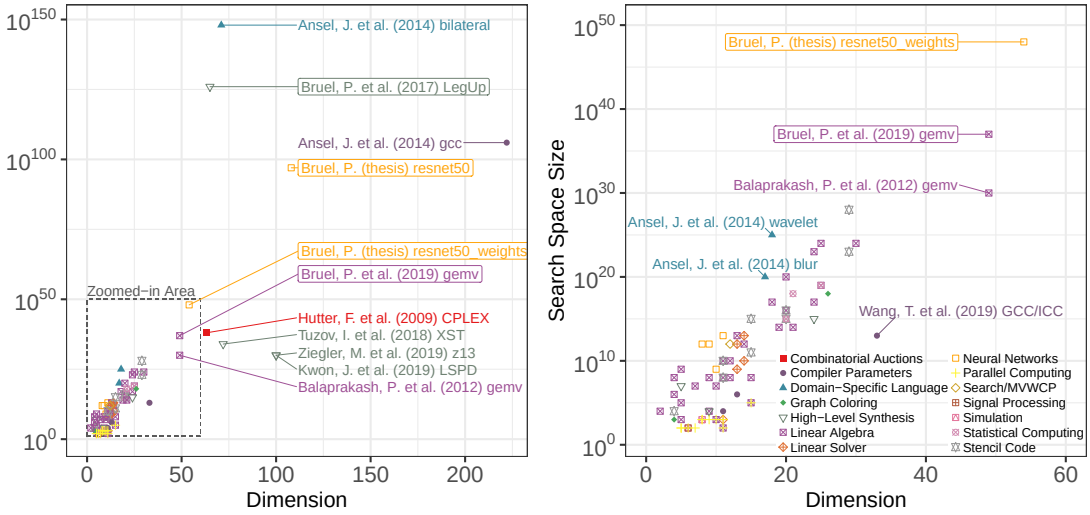

Figure 1.7: Dimension and search space size for autotuning problems from different domains [22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 18]. The left panel shows a zoomed view of the right panel

On a first look, there seems to be no apparent global search space structure in the landscape on Figure 1.6, 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 a curvature along the unrolling factor axis. 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 in most settings where autotuning methods are useful it is impossible to do so.

1.3 Thesis Contributions

1.4 Text Structure

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

Reproducible Science

Search and Statistical Learning for Autotuning

Design of Experiments

Gaussian Process Regression

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

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

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

[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] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.