Implementation and Design Space Exploration of a Turbo Decoder in High-Level Synthesis

Wesley Stirk
Electrical and Computer Engineering
Brigham Young University
Provo, UT, USA

Jeff Goeders
Electrical and Computer Engineering
Brigham Young University
Provo, UT, USA

***Abstract*—High-Level Synthesis (HLS) allows not only for quicker prototyping, but also faster and more widespread design space exploration. In this work we designed a turbo decoder using Vivado HLS, which has not previously been explored. Our turbo decoder was designed to allow for easy design space exploration, both of algorithmic turbo decoder parameters as well as HLS parameters. Data and analysis on the design space is presented for approximately 200,000 variations with an emphasis on the needed trade-offs when designing a turbo decoder.**

I. Introduction

High-Level Synthesis (HLS) has gained considerable traction in recent years as a design methodology that can provide significantly higher productivity than hand-crafted RTL, while still offering competitive circuit performance. HLS is now frequently the design methodology of choice for machine learning accelerators [1]–[3], video processing [4], [5], scientific computing [6], signal processing [7], communications [8], and more. In this paper, we explore an HLS implementation of a turbo decoder, which is the primary component in implementing turbo code based forward error correction (FEC) systems. Turbo codes are commonly used in mobile, deep space, and other noisy-channel communications system.

A turbo decoder does not guarantee zero error transmission, but rather is primarily evaluated by how low of an error rate is achieved in a given lossy transmission. As such, there are numerous trade-off opportunities. For example, a designer may consider reducing the fixed-point bit-width, while increasing the number of processing iterations, in hopes of trading an increase in latency for a reduction in area, while still maintaining the error-correcting performance. Such explorations are difficult when designing at the RTL abstraction level. In this work we show how HLS offers the ability to automatically explore the rich design space available in turbo decoder FPGA implementations.

The primary contributions of this paper are the following:

- A paramterized Turbo Decoder design, implemented in Vivado HLS.
- A demonstration of how HLS enables the automated design space exploration for this application.
- Experimental data of approximately 200,000 different design variants, evaluated against size, speed, error-correcting performance, and power metrics, with analysis of the major design trade-offs.

The implementation code, and all experimental data is made publicly available at *github.com/byuccl/hls_turbodecoder.*

II. Background

A. Turbo Decoder and Error Correcting Coding

Turbo Decoders are used to perform Error Correcting Coding (ECC), which is the process of correcting errors in digital data that have been added through transmission in a noisy communication channel. A turbo decoder relies on knowledge of the encoder and iterative feedback to interpret the noisy received message . The decoder is actually comprised of two separate decoding modules, each of which perform decoding based on the BCJR algorithm [9]. The BCJR algorithm relies on knowledge of the encoder, the received bits, and previous or prior information to produce additional or extrinsic information. On each iteration, the information is being passed around in a loop between the two decoders. As the information cycles through the decoders, the extrinsic information converges to the best solution or the most accurate message based on the available information.

Different variations of turbo decoders have various features of accuracy and complexity. While the structure and flow of information is the same, the implementations of the BCJR algorithm may be different. The "standard" *MAP* BCJR algorithm performs calculations on the probabilities of bits, while the *Max MAP* variant performs calculations on the logarithms of probabilities. The latter uses a computational approximation, allowing for significantly simpler calculations, but causes reduced accuracy.

Since turbo decoders can vary by MAP algorithm, number of iterations, message length, and data type, there is a rich design space to be explored, before even considering the optimization space offered by HLS. This is explored in Section IV-C.

We evaluate the quality of a turbo decoder implementation using the following metrics:

Bit Error Rate (BER) Number of bits in error divided by the total number of bits transmitted. This varies at different signal to noise ratios (SNR) and results in a curve as shown in Figures 1a, 1b and 1c.

Performance Baud rate of decoding (bits per second).

Implementation Cost Number of FPGA resources.

Power As estimated by the Xilinx Power Estimator (XPE).

978-1-7281-1957-1/19/$31.00 ©2019 IEEE

B. Previous Work

Since their invention in 1993 [10], turbo codes have been implemented in several different ways. While commonly used for mobile, defense, and satellite communications systems, these industrial implementations are not readily available. A few papers have discussed designing turbo decoders using RTL [11]–[14], either targeting an FPGA or VLSI implementation. Their work primarily focus on architectural changes to a turbo decoder design that can improve performance through hand optimization. In his Master's thesis [15], Conn designs three hand optimized turbo decoder designs using HLS, focusing on hardware acceleration of software defined radios (SDR). Rather than optimizing a few designs, we focus on a large turbo decoder design space and necessary tradeoffs in the design process.

Apart from error-correcting systems, many research projects have explored how one can intelligently explore a design space using HLS [16]–[18]. These techniques offer faster convergence to pareto-optimal designs. We have not used these "smart exploration" techniques in our work, as they are not yet available in commercial tools. However, when more easily accessible in the future, they would further strengthen the argument for using HLS for turbo code implementations.

III. Turbo Decoder Implementation

A. Initial Exploration

We began exploring using HLS to design a turbo decoder by using sample C++ code from the online resources from Todd K. Moon's popular textbook on error correcting codes [19], as this was one of the only open-source implementations of a turbo decoder. We modified this software to the point where it would be accepted by Xilinx's Vivado HLS tool.

This software was implemented as object-oriented C++ code, and relied heavily upon dynamic object allocation, thus requiring significant restructuring. While ultimately we were able to modify the original code in such a way that it would be accepted by the HLS tool, the resulting software still had a number of issues that made it less desirable including paramaterizing through class constructors and lack of readability due to our refactoring. The former was a problem because Vivado HLS had trouble optimizing the design for the parameters that were buried within the class constructors. Ultimately we decided to rewrite the entire code in a structure that would be easy for the compiler to analyze and optimize. This involved removing the class-based approach, and returning to simpler C-like software code.

While we are frequent advocates for HLS-based design flows (and indeed there are many benefits as demonstrated in subsequent sections), our experience in this application showed that it is not yet practical to expect a software designer to simply input their existing code into an HLS tool, and very significant code restructuring may be required.

B. Code Restructuring for HLS

Our code rewrite proved successful, and provided the groundwork needed to explore further HLS optimizations later. The rewritten code also required significantly fewer resources, and produced a much faster circuit (46% of the original latency). Design space exploration was facilitated by removing the parameterized software classes, and design parameters were instead implemented as pre-compiler (`#define`, `#ifdef`) statements. All of the standard parameters (message length, number of iterations, etc.) could be easily defined and modified prior to compilation. While modern software engineering often forgoes these precompiler directives in favor of parameterized or polymorphic classes and objects, we found the compiler could not optimize the latter as effectively.

Other computational changes were made, such as switching the computational order of some variables. While this does not change the algorithm, it does influence resource usage, speed, etc. Other works described in Section II-B demonstrate other computational changes that can influence the final result of the design.

C. HLS Directives

Once the code was restructured, we next focused on adding HLS optmization directives. All HLS directives were embedded inside of pre-compiler conditional statements to allow for design space exploration. Due to computational time and resources, not all possible or beneficial directives were included. Instead our emphasis was on systematically and automatically exploring a subset of the whole design space. Once this exploration is complete a designer can choose one of the designs to further optimize by hand.

The subset of the design space we explored focused on the many loops in the turbo decoder. After some initial exploration, it was determined that these optimizations had a larger influence on resource usage and speed than other HLS directives. Attempting to do a truly exhaustive search of all loops was determined to be unnecessary in order to save compute time. Some combinations were known to overrule each other, such as unrolling an outer loop automatically unrolling an inner loop, and therefore did not need to be simulated. In the end it was settled upon to do all combinations of unrolling and pipelining on the calculations on the probabilities in two different categories, normalizing those calculations, the normalization of the prior/extrinsic information that was passed between BCJR decoders (see Section II-A), and the BCJR algorithm as a whole.

IV. Experimental Results and Analysis

A. Design Parameters

In the design space exploration, we considered a number of parameters that affect the implementation of the turbo decoder:

Number of Iterations: The number of times that the turbo decoder repeats before deciding on an answer.

Message Length: Number of sent bits before processing begins.

Algorithm Variations: MAP versus Max MAP algorithms.

Data Type: Floating or fixed point. Fixed point types are labelled as fixed-x-y, where x is the total bit width, and y is the number of bits used for the integer.

HLS Optimization Directives: Which HLS optimizations (loop unrolling, pipelining) are chosen from Table II.

B. Methodology

In order to rapidly produce all of the different variations, a Vivado HLS Tcl script looped through all of the combinations and used Vivado HLS 2018.2 to run the synthesis. We used the synthesis report estimates to determine the resource usage and speed of the design. Power estimates were obtained by using Vivado 2018.2 to perform implementation and generate a power estimate. Some designs were too large to fit on the target chip. Vivado did not produce power estimates and power data is left blank for these designs. All synthesis was performed targeting the Xililnx Zynq-7000 FPGA (xc7z020clg400) with a target clock of 100 MHz. To reduce needed synthesis runs and computer time, approximately 100 of the variations with no HLS directives were chosen to synthesize using all combinations of the chosen HLS directives. These were chosen based on a cursory analysis for designs of interest - very small resource uses, extremely accurate, etc.

The exact same source code was also used to determine the error-correcting abilities of a particular implementation variant. This data is presented in BER plots, which demonstrate the error rate in the decoded data, given different signal-to-noise ratios (error rates in transmission). We ran the executable files generated for each turbo decoder variation (except for changes in HLS directives because there would have been no effect on the accuracy of the turbo decoder) through a simulated additive white Gaussian noise (AWGN) communication channel in python. The simulations were run until they had either decoded 250 bits incorrectly or they had transmitted 8 MB of data. The same messages and noise (at each SNR) were used for all variations of the turbo decoder to ensure consistent results among the different variations.

C. Results

Figures 1a and 1b show the effect that the number of iterations and message length can have on the accuracy of a turbo decoder. As the number of iterations increases, the turbo decoder is able to perform more calculations, resulting in more accuracy. As the message length increases, there is more information for the turbo decoder to operate on and so produces a more accurate message. The precision of the data type can also play a significant role in the success of the algorithm, as seen in Figure 1c. (The double, float, fixed-64-8, and fixed-128-16 all had the same accuracy in the simulation, and so cannot be differentiated in the graph). Floating point types allow greater precision, but this can be duplicated with extremely large floating point types.

These algorithmic considerations also affect the FPGA hardware required to synthesize it. Figure 2 show the consequences of the algorithm on the power, baud rate, FFs, and LUTs during the first HLS synthesis of the design (ie with no HLS directives included). Areas where power is not shown are areas that could not be synthesized in the target FPGA (without further HLS tuning) due to size constraints. In general, changes that tend to increase the accuracy of the received message make the design slower, and require more resources, although each parameter does so to varying degrees.

To evaluate the hardware costs of implementing a turbo decoder, the Accuracy-Resource Pareto frontier was found for each individual resource. The frontier at an SNR of 4 are shown in Figures 3b, 3c, 4 and 5. Zoomed in versions of the FF and LUT pareto frontiers are shown in Figure 3a. All of the figures show the difficulty in maintaining accuracy with limited resources. However, the rich variation of possibilities ensures that there will likely be a viable solution to any problem.

D. Notable Designs

Table I shows the best designs when optimizing for only a single objective Unsurprisingly, the smallest and fastest designs are very similar - low message length and using the Max MAP algorithm, although having the MAP algorithm as the minimum power breaks the pattern. We also defined a number of costs function in order to compare the various designs. The results can also be seen in Table I. Interestingly, all of the designs that had the best cost functions used a message length of 400 bits and used a float data type. A review of the table also shows which HLS directives are more important. 1b, 4, 5b, and 8 are the most common directive categories that are used (see Table II for what the various HLS directives refer to).

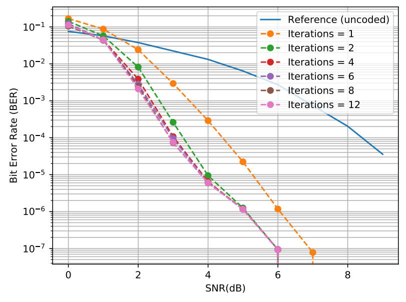
In addition to the single resource pareto frontier, we also performed a multi-dimensional pareto search to find all of the designs that were optimal in some way. From the approximately 200,000 designs that we started with, we found 2448 designs that were optimal in some combination of important parameters. The statistics for this optimal set of designs (of the design space that was explored) can be seen in Table III. It seems that designs that use 100-200 bits in the message and 3-4 iterations of the turbo decoder is a good spot to start many designs. While not shown in the table, HLS optimizations were found across the whole spectrum of attempted combinations.

E. Software Comparison

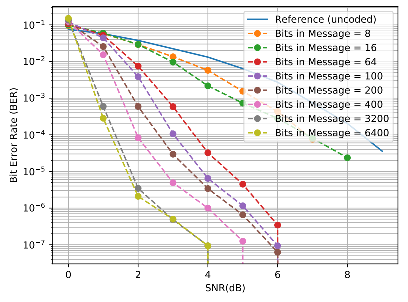
The HLS created hardware designs were also compared against software run times using the Zynq ARM processor.

Of the designs that were compared, 251 required no HLS directives in order to be faster than the software. However, all of them used fixed point data types, rather than a floating point type. Since the processor has no hardware support for fixed point, this makes sense. Of the 45 designs using the float data type that were chosen for further HLS refinement, 19 resulted in designs that were faster on the FPGA than on the processor.

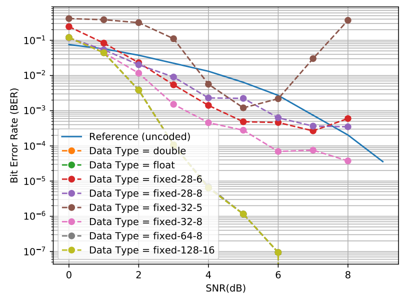
The designs that were chosen for further HLS directives had on average a 629% improvement in baud rate when compared to the software code. Of the 30 designs in this category that were still slower than the software, the average percent difference to the software was only 28%.



(a) Number of Iterations

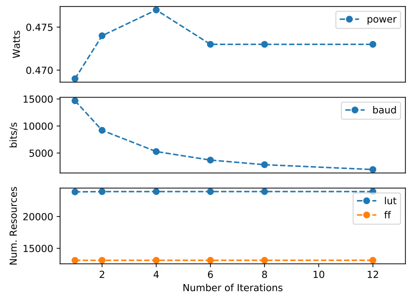


(b) Message Length

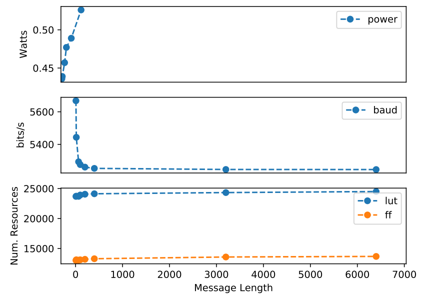


(c) Data Type

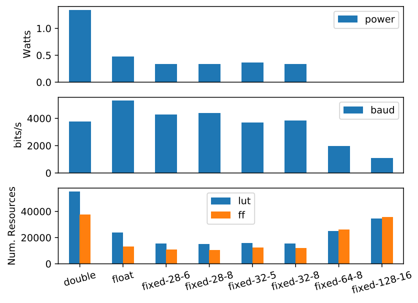
Fig. 1: BER Comparison for Different Parameters



(a) Number of Iterations

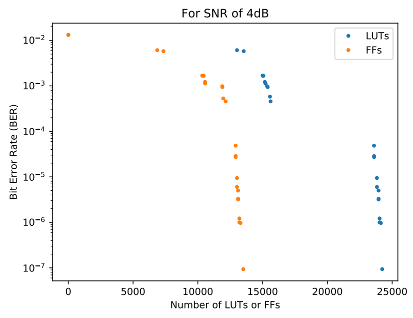


(b) Message Length

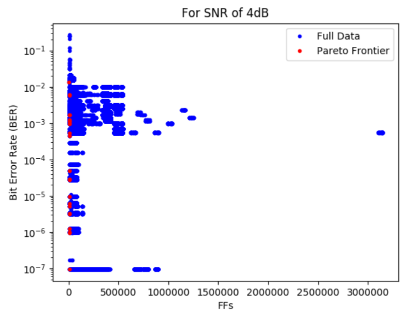


(c) Data Type

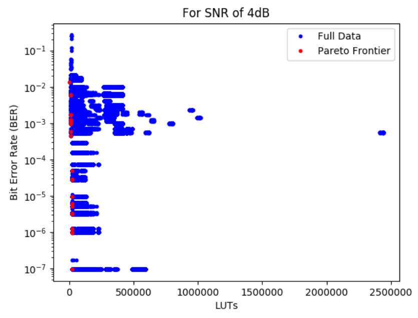
Fig. 2: Resource Comparisons for Different Parameters



(a) FF and LUT



(b) FF



(c) LUT

Fig. 3: Accuracy-Resource Pareto Comparisons

V. Conclusion

In this paper we demonstrated how algorithmic exploration of turbo decoders could be combined with HLS optimization exploration in order to generate a rich set of design variants. The result was a series of best designs in 9 different categories, and 2448 designs that were optimal in some combination of parameters.

References

[1] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An OpenCL deep learning accelerator on Arria 10," in *International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 55–64.

[2] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," in *International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 65–74.

[3] H. Nakahara, A. Jinguji, T. Fujii, and S. Sato, "An acceleration of a random forest classification using Altera SDK for OpenCL," in *International Conference on Field-Programmable Technology*, Dec 2016, pp. 289–292.

[4] S. Neuendorffer, T. Li, and D. Wang, "Accelerating OpenCV applications with Zynq-7000 all programmable SoC using Vivado HLS video libraries," *Xilinx Inc., August*, 2013.

[5] K. Denolf, S. Neuendorffer, and K. Vissers, "Using c-to-gates to program streaming image processing kernels efficiently on FPGAs," in *2009 International Conference on Field Programmable Logic and Applications*, Aug 2009, pp. 626–630.

TABLE I: Best Design Variants

Objective	# LUTs	# FFs	Power	Baud	BER @ SNR=4	Algorithm	Msg. Len.	Iter.	Data Type	# HLS Directives
Min. LUTs	**12996**	6872	0.28	22650.1	0.02	Max MAP	8	1	float	5b
Min. FFs	13025	**6866**	0.30	21974.4	0.006	Max MAP	64	1	float	5b
Min. Power	16509	11602	**0.19**	2074.3	0.006	MAP	8	12	fixed-28-8	1b,3a,8a
Max Baud	20896	12881	0.36	**218459.9**	0.02	Max MAP	8	1	float	1b,4a,7b,8a
Min. BER	55252	37663	1.45	2630.9	**9.69E-07**	MAP	400	6	double	
Max Baud/BER	41817	25804	0.54	28799.2	1.22E-06	MAP	400	2	float	1b,4a,5b,7b,8a
Min Power*BER	33968	20662	0.42	7165.7	0.000001	MAP	400	4	float	1b,4a,8b
Min. LUT*BER	24112	13468	0.51	5066.7	0.000001	MAP	400	4	float	1b,4b
Max Baud/(BER*LUT*power)	24983	14858	0.49	21890.2	1.22E-06	MAP	400	2	float	1b,4b,5b,6b,8b

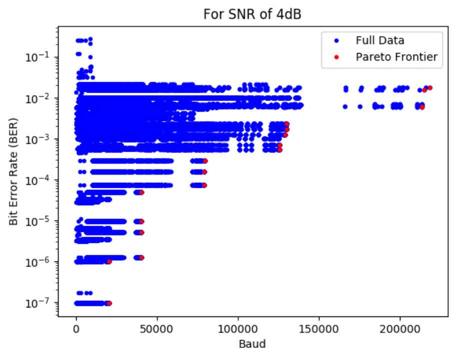


Fig. 4: Accuracy-Baud Rate Pareto Frontier

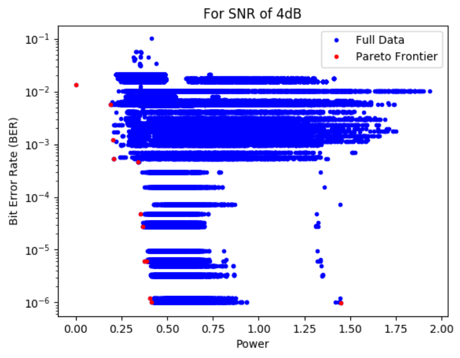


Fig. 5: Accuracy-Power Pareto Frontier

TABLE II: HLS Directive Reference

1	Inner BCJR Loop
2	Outer BCJR Loop
3	Outer Probability Calculation Loop
4	Inner Probability Calculation Loop
5	Probability Normalization
6	Inner PreProbability Calculation Loop
7	Outer PreProbability Calculation Loop
8	Information Normalization Loop

A - Loop was Unrolled, B - Loop was Pipelined

TABLE III: Pareto Optimal Statistics

	Average	Maximum	Median	Minimum	Std. Deviation
Msg. Len.	115.5	400	64	8	119.6
Iter.	3.2	12	2	1	3.2
Power	0.4	1.655	0.427	0.19	0.2
Clock (ns)	10.7	17.57	9.723	8.239	2.6
Baud	24073.6	218459.9	14086.0	1033.7	30559.9
BRAM	35.7	212	23	7	30.4
DSP	81.0	461	88	20	33.2
FF	19310.9	73754	14730	6866	11479.3
LUT	29686.9	103332	26680	12996	12603.8

[6] D. Weller, F. Oboril, D. Lukarski, J. Becker, and M. Tahoori, "Energy efficient scientific computing on FPGAs using OpenCL," in *International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 247–256.

[7] H. Wang, Ming Zhang, P. Thiagaraj, and O. Sinnen, "FPGA-based acceleration of FDAS module using OpenCL," in *2016 International Conference on Field-Programmable Technology (FPT)*, Dec 2016, pp. 53–60.

[8] J. Andrade, N. George, K. Karras, D. Novo, F. Pratas, L. Sousa, P. Ienne, G. Falcao, and V. Silva, "Design space exploration of ldpc decoders using high-level synthesis," *IEEE Access*, vol. 5, pp. 14 600–14 615, 2017.

[9] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate (corresp.)," *IEEE Transactions on information theory*, vol. 20, no. 2, pp. 284–287, 1974.

[10] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near shannon limit error-correcting coding and decoding: Turbo-codes. 1," in *IEEE International Conference on Communications*, vol. 2, May 1993, pp. 1064–1070 vol.2.

[11] K. Manjunatha and V. A. Meshram, "Design and FPGA implementation of power efficient turbo decoder for 4g lte standards," *International Journal of Applied Engineering Research*, vol. 12, no. 21, pp. 10 921–10 925, 2017.

[12] T. V. Vardhan, B. Neeraja, B. P. Kumar, and C. S. Paidimarry, "Implementation of turbo codes using verilog-hdl and estimation of its error correction capability," in *IEEE Asia Pacific Conference on Postgraduate Research in Microelectronics and Electronics (PrimeAsia)*. IEEE, 2015, pp. 75–79.

[13] V. Belov and S. Mosin, "Fpga implementation of lte turbo decoder using max-log map algorithm," in *2017 6th Mediterranean Conference on Embedded Computing (MECO)*. IEEE, 2017, pp. 1–4.

[14] Y. Tong, T.-H. Yeap, and J.-Y. Chouinard, "VHDL implementation of a turbo decoder with log-map-based iterative decoding," *IEEE Transactions on Instrumentation and measurement*, vol. 53, no. 4, pp. 1268–1278, 2004.

[15] B. E. Conn, "Master's thesis: Exploring high level synthesis to improve the design of turbo code error correction in a software defined radio context," 2018.

[16] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *design automation conference*, 2013, p. 50.

[17] B. C. Schafer and K. Wakabayashi, "Design space exploration acceleration through operation clustering," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 29, no. 1, pp. 153–157, 2009.

[18] ——, "Machine learning predictive modelling high-level synthesis design space exploration," *IET computers & digital techniques*, vol. 6, no. 3, pp. 153–159, 2012.

[19] T. K. Moon, *Error correction coding: mathematical methods and algorithms*. Wiley, 2016.