Received November 30, 2016, accepted January 23, 2017, date of publication February 20, 2017, date of current version March 15, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2672200

Dataflow in MATLAB: Algorithm Acceleration Through Concurrency

TRAVIS F. COLLINS AND ALEXANDER M. WYGLINSKI, (Senior Member, IEEE)
Wireless Innovation Laboratory, Department of Electrical and Computer Engineering, Worcester Polytechnic Institute, Worcester, MA 01609, USA

Corresponding author: T. F. Collins (traviscollins@wpi.edu)

This work was supported by MathWorks Inc.

ABSTRACT In this paper, we present a novel Data-Flow architecture for MATLAB. This architecture provides thread-level pipelining of MATLAB functions as well as general concurrency support. The proposed approach yields a significant speedup of current MATLAB implementations that rely on streaming data or employ data-dependent operations. Following the development of increased CPU core counts, this proposed framework will provide additional benefit only as this trend continues. A performance analysis of the proposed framework is performed, and we are able to demonstrate high-level throughput gains of specific applications. Discussions on implementation guidelines, as well as limitations of the framework, are proposed in this paper. Through the use of this tool, we have demonstrated a 802.11a receiver employing Software-Defined Radio hardware running in real time. From the user's perspective, this tool requires interaction only from the MATLAB language, handling all threading and data transfer without user intervention.

INDEX TERMS Software-defined radio, dataflow, HPC.

I. INTRODUCTION

With modern consumer CPU architectures growing in core count instead of higher core frequencies, system throughput can be efficiently maximized by expanding an algorithm over many cores. The applications of interest to the authors are with respect to signal processing, but such ideas can be expanded to other high performance compute applications, such as machine learning, data science, financial analysis, and other similar applications. Signal processing naturally fits into this coding architecture for applications such as signal recovery, tracking, and decoding. In these applications, the tasks can be pipelined, providing increased throughput or reduced output latency, assuming the algorithm is spanned over many cores instead of a single thread.

Parallelism can be performed in computation at many different levels: At the lowest level, modern single CPU cores can execute several basic operations simultaneously with single instruction, multiple data (SIMD) specialized commands [1], or more generically known as vector commands. Programming with a domain-specific language (DSL) provides fast development and natural problem formulation, but can be limiting on actual computational features. In this paper we focus primarily on MATLAB [2], but other DSL's such as *R* [3] and *Octave* [4] share in the fact that they provide limited multicore exploitation. This can be even difficult to realize in more general languages that maintain scientific libraries such as *Python* [5] with *SciPy* [6]. It is important to note that behind many functions in these languages/products are implementations of external BLAS (level 3) libraries that use multiple threads to do parts of basic vector and matrix operations in parallel [7], and even higher level functionality is available through *OpenMP* or *pthreads* [8] in these products.

With the growth of research areas such as Big Data and Machine Learning, it has become more important to utilize parallelism when possible. As a result, we have seen the introduction of Google's *TensorFlow* project [9], *Apache Beam* [10], and migrations to functional programming languages that naturally fit into flow based architectures [11]. Algorithmic acceleration has seen explosive growth in the hardware sectors with GPU and FPGA accelerators becoming more common, and their software architectures (CUDA, OpenCL, HDL/Verilog) becoming easier to use [12].

Focusing on MATLAB specifically, parallelism exploitation has been introduced in many forms but can be categories into two main areas. First are highly data independent implementations, where work can be spread over many cores and many machines. The two main projects in this area include pMATLAB [13] and MathWorks' own Parallel

Computing Toolbox [14]. Second are alternative MATLAB language compilers, which convert MATLAB code into C/C++, fortran, or specialized kernels which are parallelized. Such designs include *MATISSE* [15], *MEGHA* [16], and *StencilPaC* [17]. Additional extensions have been added to *MEGHA* introducing runtime thread management with Intel's Threading Building Blocks [18].

In this paper, we present an architecture to expand MATLAB in order to enable Data-Flow (DF) based parallelism and general concurrency for the purpose of stream processing acceleration and task concurrency. We provide a user configurable flowgraph, where functions produced from MATLAB are run concurrently and pass data between one another. The goal of this framework is to handle the threading and data passing between MATLAB functions, so that developers can focus on algorithmic aspects of their code rather than handling data flow and threading. MATLAB is a dominant tool in data sciences, engineering, and many other fields, reaching the top ten of IEEE's Spectrum programming languages [19]. Currently, MATLAB does not offer any DF or simpler cascaded tasked based parallelism that are available to the user. Expanding MATLAB to include this framework can provide significant acceleration of current scientific work, without difficulties of other languages/frameworks that do provide this capability.

A. CONTRIBUTIONS

This work provides the following contributions:

- Implementation of a novel DF framework via the MATLAB environment with significant language support.
- Performance analysis of the proposed framework itself and application level speedups possible over current serial code.
- A simplified workflow designed to fit algorithms and current MATLAB code into this framework, including load distribution over the cores of a given machine.
- Discussions on the limitations of the proposed framework and the drawbacks of utilizing some of the associated tooling.
- Presentation of a case study application of a 802.11a receiver implemented using this proposed framework.

II. PARALLEL ARCHITECTURE SYSTEMS

This work focuses on providing a DF parallelism framework to MATLAB, which we call MATLAB Data-Flow (MLDF). The goal of MLDF is to increase computational performance, but also limit latency through careful thread orchestration. DF, which can have several meanings in computing, is defined in this work as the division of computational stages into concurrent tasks that can have dependent or independent execution. Formally, a DF model consists of governing graph, where the nodes of the graph are computations, and connections between the nodes represent the data dependencies between the computations [20]. DF from a simplistic viewpoint is a combination of task and data based parallelism, utilizes both techniques in a cascaded arrangement to increase numerical computation through more general parallelism and concurrency. First, it is important to understand the differences between task parallelism (TP) and the more commonly implemented data parallelism (DP). In DP applications, we assume the same operations are performed on different sets of data in parallel across multiple cores. DP traditionally does not share state or handle data dependence across threads or processes, and such operations are considered very computationally expensive. In task based parallelism, each concurrently running operational unit can have a different function. Generally TP is the simultaneous execution on multiple cores of many different functions across the same or different datasets. Unlike data parallel functions, task parallel functions can be cascaded together in series, essentially pipelining operations, analogous to an assembly line. Existing frameworks such as OpenMP [8] provide this functionality through parallel for-loops in the case of DP, and as *jobs* or *spawns* in the case of TP. However, data handling can be limited or expensive in these libraries, with projects like *gnuradio* [21] or *Cilk* [22] providing more finely controlled data passing during inter-thread operations. Such problems have been researched extensively in the literature for more general languages (C/C++) with specialized implementations [23]–[25]. Now the goal in this work is to leverage this knowledge for an appropriate integration with MATLAB.

However, since this research is targeted at MATLAB users specifically, it is important to understand the performance benefits of DF. We approach this from the extreme perspectives of DP and TP. Each implementation has their advantages and drawbacks, but for fairness when comparing DP to TP we assume that the number of DP cores is equal to the number of TP functions, unless otherwise specified. When core communication is overlooked, DP and TP will always meet or outperform non-parallel implements, but this will be considered later in this work as well.

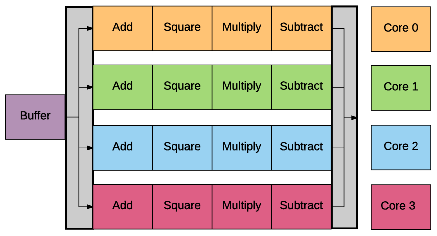


FIGURE 1. Data parallelism realization for equation (1) on a four core system. In the case of a streaming application the buffer block will be utilized to delay input data until four samples are available.

To provide a better understanding between these parallel architectures, let us consider a simple example. Note that in a real-world scenario the operations would be much more computationally intense. Fig. 1 and Fig. 2 contrast the different structures of parallelism for an evaluation of a simple

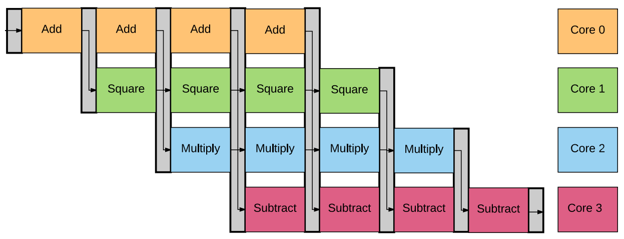


FIGURE 2. **Cascaded task parallelism realization for equation (1) on a four core system. If we condition on each core having a different function, this realization is the most efficient.**

example equation:

$$f(x) = 3(x+1)^2 - 4, \quad (1)$$

where we wish to compute four realizations given different input values (x). As you can see in Fig. 2, we split the work of each core to be a specific mathematical operation. Therefore, all data passes through each core. In Fig. 1, we instead supply different data to each core. The design or selection of parallel sections is not always simple to address or optimize due to mismatches in core count and available operations. Therefore, depending on the computational requirements of the algorithms, implementations of TP and DP can perform differently. From an implementers perspective, DP can be more easily applied to problems due to its lack of additional communication required between cores.

If all data to be processed is available at the start of the simulation, then DP has an obvious advantage, since there is no startup latency. This provides a direct speedup of $\sim N_c$ over the single core alternative where N_c is the number of available cores. However, this type of scenario is not always possible. For example, streaming applications generate data over time and tend to have latency or data dependency requirements. The amount of data to be processed in these types of applications can also be unknown. First, let us assume that the overall system is streaming based and is able to operate in real-time. We will also assume that DP must launch all processing threads at the same time, requiring the buffing of incoming data. Therefore, with TP the time between new data arrivals Δt_n and the largest processing time of any individual core/function for that much data $\Delta t_{TP,MAX}$ must have the following relation: $\Delta t_n > \Delta t_{TP,MAX} + \Delta t_{TP,OH}$. Where $\Delta t_{TP,OH}$ is the overhead for block data passing. For DP, the processing time of any individual core for that much data Δt_{DP} must have the following relation: $\Delta t_n > \Delta t_{DP}/N_c + \Delta t_{DP,OH}$. Where $\Delta t_{DP,OH}$ is the DP overhead. This is true since DP is able to buffer at most N_c data units. Now, the combined TP stages have equal latency as a single DP thread/process $\sum_{k=1}^{K} \Delta t_{TP,k} = \Delta t_{DP}$, where K is the number of TP stages, but the per data unit latency is different.

Furthermore, under these real-time requirements we can determine the average latency to process a unit of data. Note, for real-time applications the overall system throughput is bound by the data arrival rate. For the DP and TP cases where $K = N_c$, the mean real-time processing latencies $t_{LAT,RT}$ for data processed by the system are:

$$t_{LAT,RT} = \begin{cases} N_c \Delta t_{TP,OH} + \Delta t_{DP} & TP \\ \dfrac{1}{N_c} \sum_{n=0}^{N_c-1} n\Delta t_n + \Delta t_{DP} + \Delta t_{DP,OH} & DP. \end{cases} \quad (2)$$

In Equation (2) we observe the startup or buffering latency of DP as $(N_c - 1)\Delta t_n + \Delta t_{DP,OH}$.

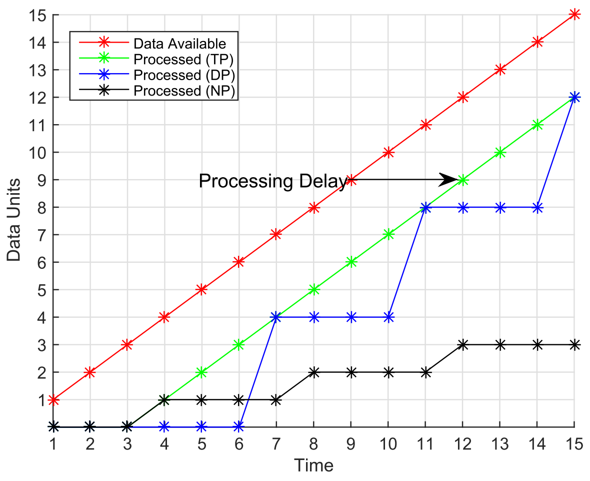


FIGURE 3. **Processing analysis for a streaming type application comparing the performance of three implementation: data parallel (blue), task parallel (green), and single threaded where no parallelism is exploited (black). Data arrives at the system every one time unit and takes two unit to be processed.**

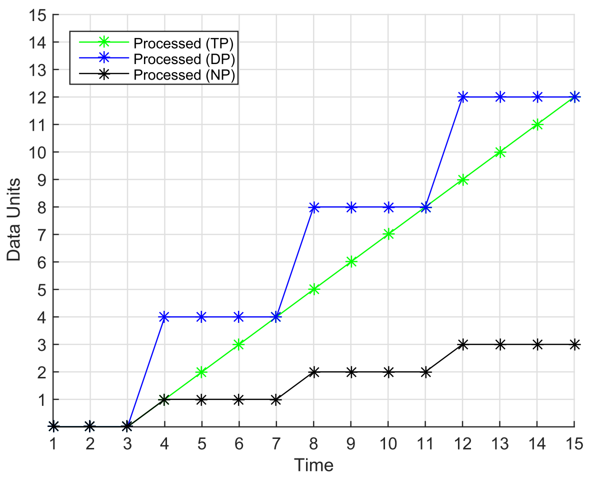


FIGURE 4. **Processing analysis for a off-line type application, where all data is available initially. Three implementations are compared: data parallel (blue), task parallel (green), and single threaded where no parallelism is exploited (black). Data parallelism can utilize all processing units (four) initially.**

Now, in the case of non-streaming applications with a fixed dataset that is entirely available at the start of a simulation, we provide a processing comparison to streaming applications in Fig. 3 and Fig. 4. In Fig. 3, DP needs to buffer inputs while TP can use them right away providing lower latency on average.

In Fig. 4, the throughputs are purely bound on Δt_{DP} and for equal comparison we define $\Delta t_{DP} = N_c \Delta t_{TP,MAX}$. When bounded by Δt_n the average throughput is better for TP, but when unbounded DP is better performing. This discrepancy in performance purely comes from the buffering that occurs in the system and processing lengths. However, if we remove this buffering constraint and allow DP to launch threads as data is made available, DP and TP will perform equally. Nonetheless, for many applications the segmentation of data with appropriate boundaries will not always be possible.

A. LIMITATIONS

Given these insight on the structures and relative performance of DP and TP to non-parallel operations, we can consider the disadvantages of these implementations. The first and most obvious is the overhead required, which is depicted as the gray sections between blocks in Figs. 1 and 2. Due to the asynchronous nature of threads, data must be passed between them in a safe method. This safety comes at the cost of speed. To reduce these overheads, we can either reduce the number of independent threads or increase the data passed at a given time. These have two downsides, first by reducing the number of threads we may not effectively utilize all processing cores. Second, by increasing the data passed between threads we increase the per-data unit latency. To provide gains over non-parallel implementations, for processing B data units we must maintain the following relation:

$$\begin{aligned} B\Delta t_{DP} > {} & \Delta t_{LAT,RT}(TP) \\ & + (B-1)(\Delta t_{TP,K} + \Delta t_{TP,OH}). \end{aligned} \tag{3}$$

For useful applications the system should handle a large amount of data units ($B \gg 1$).

When comparing DP and TP, DP implementations tend to be more load confining, but have a reduced communication overhead. From Figs. 1 and 2, we have 8 and 16 communications, respectively. In applications where we have an inherent data dependency, it may not be possible to have efficient data sizes evenly passed to DP threads. TP provides more flexibility and naturally handles data dependency. In particular, TP performs well when data passing between blocks is event driven and not deterministic, such as when streams of data have task boundaries that have time dependence. This is obvious for communication receiver algorithms which rely on all past data in order to perform actions. Such state sharing is difficult with DP, and comes with a significant performance penalties.

Since MLDF is a combination of TP and DP, we can utilizes the advantages of each. Primarily, we can cascade blocks where data dependency exists, and arrange other operation in parallel where there is no dependence. This independence did not exist in the first example we looked at in Equation (1), but instead becomes obvious in an alternative example:

$$f(x) = 2x^2 + x - 1. \tag{4}$$

Here in Equation (4), we can compute the squaring operation and multiplication concurrently with the subtraction, but the final addition can only be done once all others have completed. This concurrency flow is illustrated in Fig. 5. The considered Equations (1) and (4) are used here as simple examples to demonstrate concurrency of mathematical operations, and in our implementation we restrict these parallel components to the MATLAB functions only. In a realistic implementation, the operations outlined in Fig. 5 should be replaced by computationally intensive functions, since passing data between these concurrency/threaded domains is not without penalty. This measured penalty will be discussed later in Section IV-A of this paper. The defining aspect of this comparison comes down to data dependency. Whenever data dependence is required that portion of an algorithm should use TP, but without that requirement DP can be used.

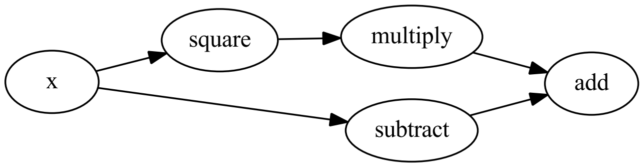


FIGURE 5. Mixed parallel flow for Equation (4), exploiting data parallel and task parallel advantages.

III. IMPLEMENTATION

Now that we have a general understanding of the certain parallel architectures, we will examine how DF was implemented through the MATLAB language. This will include the underlying threading model, cross-thread data sharing, and integration with MATLAB Coder. Once the implementation details have been provided we will move on to the performance analysis of applications and the underlying architecture.

A. MATLAB INTEGRATION AND RELATED WORK

In this work, we expand MATLAB through the code generation tool to provide DF. As stated before, since DF is a super set of TP and DP, we get those extreme cases for free. However, the general DF framework provided here can take advantage of both TP and DP within the same implementation using this framework. To provide concurrency with efficient data sharing, this DF framework must implement threading. Nonetheless, since MATLAB does not inherently have threading tools that are exposed to users in the MATLAB language,[1] and MATLAB's interpreter is not thread safe. Therefore, an alternative approach or language domain must be considered. Existing approaches in the literature have utilized external or custom compilers of the MATLAB language, where single vector operations are scaled across cores [18]. Alternatively, code replacement has been used to speedup individual functions or a language subset [15], [17], and even with targeting of hardware [16], [26]. These approaches do provide significant speedup, however they only provide an

[1]*par-for* relies on process forking for concurrency and cannot be used for general concurrency.

extremely limited subset of MATLAB's functionality. This is especially true when considering MATLAB's toolboxes, which have extensive system object use [27]. The usage of such toolboxes has become the main use cases of the product, beyond just matrix math.

All of the discussed approaches for providing DF-like parallelism to MATLAB involve C/C++ generation/translation, which we utilize here in this framework to provide access to threading libraries. However, instead of relying on a custom compiler we utilize MATLAB Coder, which has extensive (but not complete [28]) support for the MATLAB language. Inspirationally, this is how MATLAB provides data parallelism for one other built-in function *dspunfold*. Therefore, like *dspunfold* in this proposed framework, all functions must support Code Generation (CG) [29]. This framework only utilizes threading at the function level, since that is the minimum entry point for CG with MATLAB Coder. Building on top of MATLAB Coder has a clear advantage over custom compilers like *MEGHA* and *MATISSE*, since it is developed by MathWorks and will grow with the ever changing MATLAB language.

From the generated code, we provide C++ libraries, additional code, and tooling to automatically generate wrappers around these MATLAB generated functions (MGF). The header files produced by MATLAB Coder are parsed to extract the relevant function prototypes used in these wrapper functions. The wrapper functions create a common API for passing data between there eventual threads. The resulting code is tied together with a single source file describing the flow of data between functions, which we call the *flowgraph*. The C/C++ *flowgraph* is also automatically generated from a custom MATLAB class, which allows users to specify function level interactions, namely which functions pass to data to and receive data from which functions. In essence, this framework manages C/C++ code produced by MATLAB Coder and provides the threading infrastructure to parallelize the desired functions into a controlling graph. Structurally, each MGF is encapsulated with a wrapper functional which is a parameter of a generic C++ class. These classes, which we call *Workers*, launch a thread at runtime to process data in and out of these functionals independently. These *Workers* are combined into a secondary class called *Graph* who is responsible and manages all *Workers*.

This work utilizes the build systems already present within MATLAB Coder, providing even *Makefiles* when custom code needs to be edited or manipulated out of the scope of the auto generation process. From the perspective of a user, there is no necessary interaction required outside of MATLAB itself. Therefore, no knowledge of thread handling and efficient data transfer is required. However, function per thread separation is solely up to the user and will more than anything determine performance. Overall this framework provides function level parallelism, which is defined by the user. This function level point of parallelism is very different than the existing literature, which focuses on more instruction parallelism. We believe that this function level parallelism can be desirable since it limits latency and overhead compared with a more granular level of parallelism with a larger number of thread interactions. However, this can be application specific.

It is important to discuss other existing tools within MATLAB in order to differentiate their functionality from this work. Those tools are primarily *par-for* and *dspunfold*. *par-for* is an extremely useful simulation tool for data parallel tasks, and will scale to clusters of computers. *par-for* relies on process forking and requires strict data independence between operations. Utilizing these tools also disables some of the built-in thread spawning functions such as *fft* and *eig* in favor of their single threaded options. Data communication is also rather expensive when working directly in MATLAB but is reduced when CG is applied over these functions. On the other hand, *dspunfold* is purely thread based but is only for data parallel tasks like *par-for*. *dspunfold* can automatically take advantage of data independence for the operation to be threaded, but only applies to a single function. That function must also support CG. *dspunfold* does not create free-running threads, instead they are explicitly launched for each call to a function utilizing *dspunfold*, thus relying on OpenMP under the hood. Both *dspunfold* and *par-for* are very useful tools that can provide significant speedup in DP applications. In the proposed implementation, threads only join the main thread at end of execution. Since this work desires to limit latency, and in essence thread overheads, this is an alternative to thread re-spawning. [22] can be used to implement threading in a similar way. However, to limit dependencies the threading model used was developed in house from *boost* [30] primitives, which are already a MATLAB dependence. This provided us with full knowledge of thread interaction without relying on external tools.

B. CODE ARCHITECTURE

Similar to other DF architectures, MLDF is completely characterized by a top-level flowgraph. This flowgraph completely describes function boundaries in the form of blocks, data passing between blocks in the form of connections, as well as sources and sinks inherently. Blocks without connections are called floating blocks. Application examples will focus on signal processing tasks, but as stated previously, can be extended to other applications. Fig. 7 provides a simple flowgraph that generates a signal, adds noise, filters the signal, and finally saves it to a file. The visual flowgraph was also automatically generated from line 16 of MATLAB code in Fig. 6, implemented by Graphviz [31].

Each of the blocks in the graph runs in their own thread, making them free-running. Blocks themselves can be considered identical from the perspective of the graph except for three properties: (1) the processing algorithm the block is assigned, (2) the number of inputs ports, and (3) the number of output ports. These three parameters, which are provided by the user, allow for automatic parsing of the MGF's function prototypes for template generation. Ports are simply protected FIFO queues of object pointers that

```matlab
%% Simple Flowgraph
g = GraphGenerator;

% Add Blocks To Graph
g.AddBlock('src',0,1)
g.AddBlock('awgn',1,2)
g.AddBlock('filter',1,1)
g.AddBlock('sink',2,0)

% Connect Blocks In Desired Order
g.Connect('src_0',0,'awgn_0',0)
g.Connect('awgn_0',0,'sink_0',0)
g.Connect('awgn_0',1,'filter_0',0)
g.Connect('filter_0',0,'sink_0',1)

g.DrawGraph(); % Visualize Graph
g.Build(); % Compile Graph
g.Run(); % Run Flowgraph
```

FIGURE 6. MATLAB flowgraph code for implementation of a simple noise filtering exampling. The MATLAB class *GraphGenerator* only contains five user visible methods, all of which are shown here.

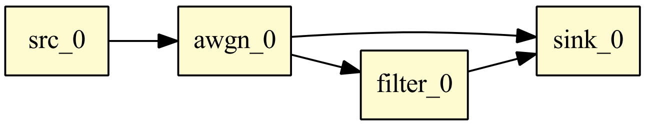


FIGURE 7. Flowgraph visualization shown when invoking the *DrawGraph* method as shown in Fig. 6. The end result is provided in a MATLAB figure.

are shared between blocks. The queues are type agnostic, which is generally a requirement when working with code produced by MATLAB Coder, due to its heavy use of custom type definitions. The queues only handle data pointers and never perform expensive data copies. Additionally, queue access was implemented in two flavors, a locking, and a non-locking style in order to reduce contention. Due to the fact that the queue model here is a single consumer and single producer between all blocks, we were able in include an efficient non-locking scheme based on [32]. In essence, we are taking advantage of memory barriers and ring buffers. Not all processors support such instructions, therefore the locking implementation is more portable. During operation when data is added to a queue by a block, that block is responsible for signaling downstream blocks. This prevents wasteful polling, as well as queue backlogging. Since the flowgraph generation process will enforce this threading model, queues will only be mapped to one consumer and one producer. Therefore, if the same data needs to be passed to different blocks downstream, an additional port should be added to the producing block to support this case. This is essentially performed by copying the necessary data in the source function in MATLAB. Under this enforcement side effects of data manipulation downstream can be ignored.

The graph itself manages block startup, shutdown, and any runtime management that is required, such as benchmarking. However, to limit overhead or performance penalties the graph does not interact with the blocks during runtime by default. Additional flags need to be set with the *GraphGenerator* class in order to enable these runtime options. Other projects, like *gnuradio*, utilize the graph to help manage flow between blocks during startup. The overall goal of this work is to minimize graph work, as well as latency between block information passing. Therefore, significant effort was put into optimizing inter-thread data passing. As a result, the blocks themselves can have only three states: waiting for new data, processing data, and passing data forward. State transitions only occur when all inputs or outputs are available, depending on the state. Again, when we utilize non-locking queues there should never be contention based waiting. A significant difference between this architecture and others, such as OpenMP *jobs*, is that this framework does not continually launch threads. This occurs once at the start of the flowgraph, and all threads are maintained for the life of the flowgraph.

In this framework, it is possible to implement scenarios where the producer block can run faster than the consumer block. If left unchecked, this would create an ever increasing input queue for the consumer block, consuming all system memory. To prevent this, back-pressure is provided at a configurable queue depth. Again, this is also provided through the *GraphGenerator* interface. When queues reach this level, the producer block will sleep periodically and poll the queue until space have freed up. This empirically has proven to be better performing in terms of latency than relying on conditional variable signaling between threads. During this period of time when the queue has reached the maximum queue depth additional computations are not performed on the producer and will remain in this state until data has been removed from the queue by the consuming thread.

C. MATLAB CODE GENERATION

Since 2011, MATLAB has been able to generate C/C++ code for a large portion of their functions [28], providing ISO compliant full C/C++ source code with no obfuscation through compiled libraries. However, CG is not simple to utilize, and has caveats of its own. This is especially true if the source MATLAB code is not strict about memory management. To help you handle that level of complexity, we have a recommended workflow that we have used to guide code to fit into the eventual MLDF architecture:

1) Confirm current MATLAB code has been tested and provides correct numerical results.
2) Split all code into separate functions that you wish to be individually threaded. Utilize assertions for all function inputs. Make sure again that this code provides correct numerical results, matching original implementation.
3) Utilizing *coder.screener*, evaluate each function to check for CG compatibility. Make necessary corrections where necessary. Again check numerical outputs of functions for correctness.
4) Now that functions are ready for CG, the flowgraph can be formed with the custom classes provided. These classes utilize the MATLAB build system and generate additional files used for wrapping MLDF's and connecting them together. This class also provides

a visual representation of the flowgraph by utilizing *Graphviz*.

5) Generate all code and build executable using the simple interface from the custom provided classes. The end product is a single executable, which is addressed through the *Run* method of the *GraphGenerator* class.[2]

Making code viable for CG primarily involves making memory allocation strict and removal or replacement of certain functions. It is also important to note that each function will be re-entered, therefore persistence should be utilized for additional speedup. This is true of code executed in MATLAB as well. In summary, this process involves only writing MATLAB code and enforces the CG of functions.

Once the *Run* method of the graph is called MATLAB's main thread will be consumed by this operation, and will remain blocked until the graph completes or is killed. The graph does not call back into MATLAB's interpretor because it is not thread safe and would inherently cause blocking if MATLAB's interaction was required. Therefore, all functions assigned to the graph are free running and require only data on their input buffers to trigger computations.

IV. TESTING AND EVALUATION

As mentioned in Section I, we primarily focus on signal processing and communication applications. Therefore, much of our testing focuses around these type of computations. This work originally provided a mechanism for providing concurrent operations for a communications system rather than a performance framework. However, when spreading work over threads we learned that significant performance enhancements could be gained, and attention was shifted to a more general framework. Therefore, to understand the speedup possible from this framework we provide testing from two perspectives. The first is from the top-level application of a flowgraph and the speedup provided over a non-concurrent implementation. The second is from under the hood of the framework itself, showing the consequences of managing concurrency. All tests outline in this sections are run on a Dual Xeon E5-2690v3, with 24 physical cores (48 Threads) at 2.60 GHz.

A. FRAMEWORK TESTING

We first examine the latency between blocks, specifically how long it takes for data to be added to the shared queue object between a pair of blocks and then popped off the same queue. In this scenario, we use a simple flowgraph with two blocks, a producer, and consumer sharing one connection. This is tested under two scenarios, the first where the source block produces data at a much slower rate than the downstream block can process it. Second, the source block's mean production interval is made roughly equivalent to the downstream block's mean consumption interval. Examining a case where the downstream block is much slower provides no perspectives on the framework, since the performance is solely gated on the processing time of the downstream block. Under the condition that the shared queue has already been filled. It also may be considered a poor design where downstream blocks are slower than upstream blocks in a constant streaming data scenario, since thread CPU utilization will be reduced. For an additional comparison, we provide results from our generic lock-based implementation to understand the benefits of the non-locking model. For all these tests we utilize a maximum queue depth of 300 pointers between blocks. However, this depth was never reached during testing. These tests were performed in series so there was no competition between threads in terms of resources, and for context only three threads were spawned at a given time including: the source block thread, the consuming block thread, and the monitoring main graph thread.

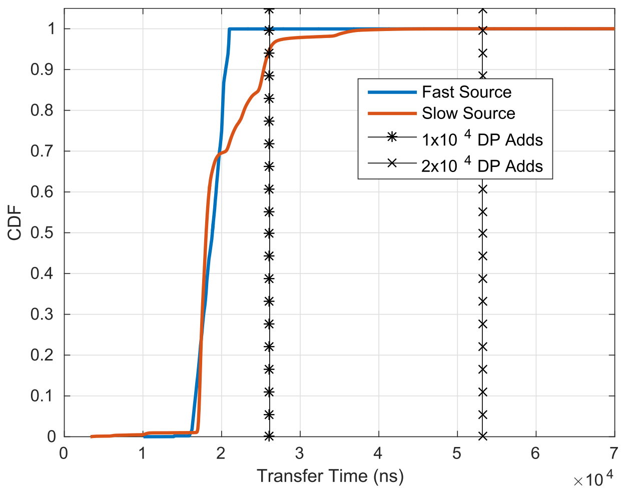


FIGURE 8. **Transfer latency between blocks for 10^6 iterations for locking queue design. Double precision additions are provided as references to the cost of an individual transfer on average. Their reference are provided as vertical lines since on a modern CPU they are roughly deterministic operations.**

Now given the implemented tests described above, these provide performance information relative to contention conditions between blocks. Since queues between blocks are shared resources, there is always a natural contention. Data manipulation between blocks only involves pointer movement, therefore no data copies occur. As a result, the transfer latency between blocks is not dependent on actual size or type of the data transfered. With this knowledge, Fig. 8 provides the results of 10^6 transfers for each of the test scenarios using the locking-queues. We utilize the *chrono* library within the C++ Standard Library to provide high resolution time information for these tests. For reference, we computed the average time to complete 100 , 10^4 and $2x10^4$ double precisions additions to the same memory location on the same machine averaged over 10^6 iterations. The transfer times for the Fast Source (FS) are almost deterministic and better performing than the Slow Source (SS). At first, this is counter intuitive since there is less contention between blocks in the

[2]The *Run* method simply uses the *system* command within MATLAB to call the associated compiled graph. Overall providing a simplified interface to the produced executable from MATLAB.

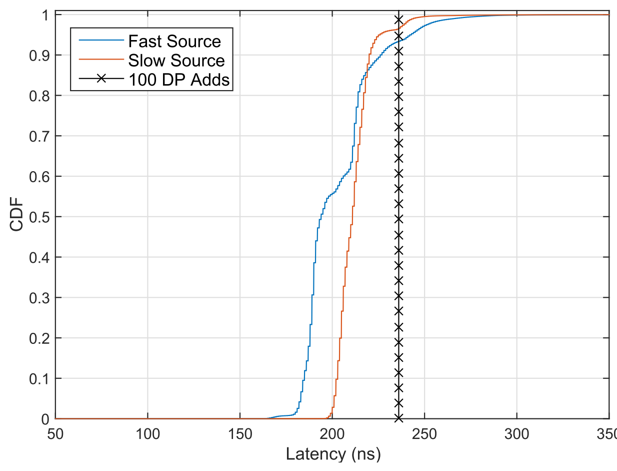


FIGURE 9. Transfer latency between blocks for 10^6 iterations for non-locking queue design. Double precision additions are provided as references to the cost of an individual transfer on average. Their references are provided as vertical lines since on a modern CPU they are roughly deterministic operations.

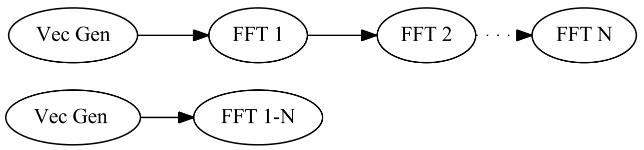


FIGURE 10. Throughput testing of cascaded FFT operations in the single threaded mode (bottom) and the MLDF implementation (top). These tests were run for two cascaded FFT's to twenty-two FFT's. The random vector generation (Vec Gen) was put into an independent thread to remove the additional computation required for those operations.

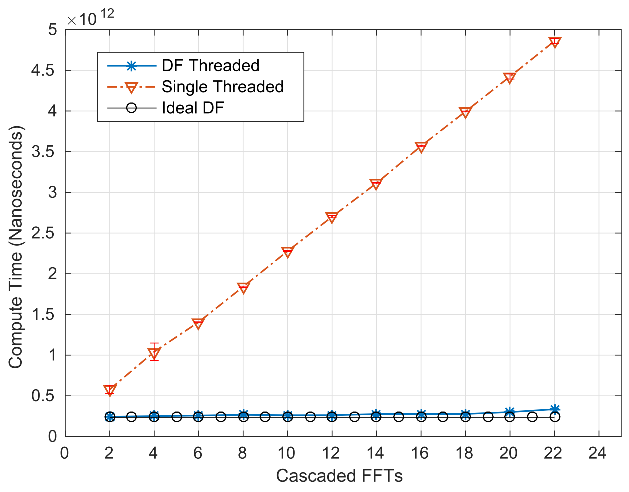


FIGURE 11. This test compares an increase in cascaded FFT operations in the MLDF case, the single threaded standard case, and an idealized case which has no communication overhead. At each number of cascaded FFT's the system processed 10^5 random complex double precision vectors of length 2^{15}. Each test was run four times, with little variance between them.

SS case. The reasoning behind this slowdown comes from the signaling delay that occurs when the downstream block sleeps if no data exists in the queue. Therefore, the sleeping block must wait for a signal from the conditional variables associated with the monitored mutexes. In the FS case, the block does not enter this sleep state as often, since the queue will have data ready to be process more often initially. Similarly, Fig. 9 provides the results for the non-locking case, and performs two orders of magnitude faster on average. In this case, the FS has more variation as expected due to the additional contention, but the SS is almost deterministic. The cases considered here are the most optimal since each block only has one port. The performance here will degrade since exiting the waiting for data state will require data from multiple threads.

B. FLOWGRAPH TESTING

Now that we have understanding of the low level thread interaction between concurrent functions, we can explore top-level speedup over single threaded non-concurrent versions. The goal is to understand system performance, in the form of data throughput, under significant computational tasks. To examine this we considered an FFT of length 2^{15} over complex double precision data as the basic unit of computation. This was chosen to represent an intensive workload over a large amount of data.

To evaluate the performance of this function, testing measured the processing time of 10^5 randomly generated complex vectors of length 2^{15} (matching the FFT size). Across the tests we cascaded additional copies of the FFT baseline function to increase complexity of the problem, ranging from 2 to 22. To remove the effects of the random vector generation, both the single threaded (ST) and multi-threaded (MT) versions were written in the MLDF framework. In the ST case, one block contained the random vector generation code, and a second contained a loop over multiple FFT's. In the MT case, the additional FFT's are applied in additional blocks. This comparison between the ST and MT implementation is shown in Fig. 10, with each block representing an individual thread. Fig. 11 provides the results of this test averaged over several iterations for each number of FFT's cascaded. Showing almost no variance in the results in both ST and MT cases. As can be seen, the MT scales significantly better than the ST version. As the number of FFT's does increase, but so does the processing time and individual overhead between blocks. Since the test machine does have 24 physical cores, no bottlenecks are observed. For reference an idealized result, which is simply an application of equation (2), is provided as well in Figure 11. This result would be attained if there was no communication overhead required between threads. However, as the flowgraphs become more complex and threads outnumber cores, performance evaluation will become difficult to predict. The ST case simply scales linearly with the increased workload, and will be independent of core count.

The speed advantage in Fig. 11 is substantial, but it is important to discuss side effects of the speedup. TP will have a constant latency through the system equal to the required N computations plus communication overhead between threads. On the other hand, if DP was applied to this problem, each input vector would have a varying latency. This was theoretically presented in Section II. The single thread case grows in latency since each subsequent vector must wait for each preceding vector to pass through all N FFT's before beginning computation.

V. DESIGNING FOR DATAFLOW

We examined the low-level consequences of using the MLDF architecture and the top-level performance gains it can provide. However, when implementing algorithms within this framework special consideration and thought needs to be provided to function placement and division. Multiple DP and TP sections of algorithms can exist but their implementation is not always necessary, since DF does have overhead. In Fig. 8 it was demonstrated under the best circumstances we can incur a penalty of $\sim 250ns$ per block transfer. Further synchronization of multiple port blocks can substantially increase this delay. Therefore, to minimize this effect computations of an individual block should take considerably longer than the data transfers. If the computation is not complex, in practice we will increase the amount of data passed per transfer to the block. Since this penalty is fixed, increasing the data to be processed will only increase the computation time of the block.

A second important aspect to consider is load distribution and gating that can occur. For cascade blocks specifically, when upstream blocks are significantly slower than downstream blocks this creates block starvation conditions. This will limit core utilization. A similar effect can happen in reverse when downstream blocks are much slower, causing upstream blocks to wait on full queues. Implementations should strive to balance work across blocks as much as possible, but at the same time without thread flooding and choking the processor. We typically approach this by first setting a target throughput rate, and then distributing the algorithm among more blocks until we reach our target.

A. LIMITATIONS

We have examined the possible speed up advantage of this MLDF framework, but there are consequences to using this framework specifically through the use of MATLAB Coder for code translation. First of all, in many circumstances CG does provide speed up over interpreted MATLAB code since it is compiled. However, for highly optimized built-in functions of the MATLAB language there can be a degradation. This comes from the fact that CG produces generic C/C++ code that does not heavily utilize vectorization or other SIMD type operations. Common advice in MATLAB is to vectorize code, which can provide increased speed up, since under the hood MATLAB will take advantage of processor specific extensions for vectorizations. However, this capability does not always translate into generated code with MATLAB Coder. Through the natural progression of this work we have examined a large amount of generated code, and have found that vectorization can be even tied to data size of each computation. For example, when low level mathematical operations are performed on vectors, there is a greater probability of insertion of specific BLAS calls to replace loops in the resulting generate code when the vectors contain large number of elements. This is also dependent on the operation itself and the profile selected by MATLAB Coder. Other projects can directly generate to SIMD and parallelized code from MATLAB code [15], [17], but as discussed previously, have very narrow language support. It may be possible to re-vectorize the produced code, but due to the level of obfuscation it will be very challenging. Nonetheless, MATLAB Coder is a proprietary tool and MathWorks provides no documentation on the internal CG process or access into the generation process.

Besides limited vectorization in generated code, plots and scopes are not available in any form of CG. Leaving only file outputs and textual feedback. It is important to note that plots and scopes in interpreted mode do cause additional latency in the system and hinder performance in the general sense. We are actively exploring other solutions to this problem, since scope can be invaluable for debugging. In this spirit, we are also considering hooks back into MATLAB at runtime through the MEX API [33].

B. REACHING REAL-TIME

With the limitations understood and known possible speed increases available, we implemented a real-time 802.11a receiver in MLDF framework. This work was based upon a serial implementation which is now available as part of the WLAN System Toolbox [34]. To reach the real-time objective, the flowgraph needed to handle a data throughput of 20×10^6 double precision floating point numbers per second at the most intensive sections. Data was provided into the flowgraph by a USRP-B210 [35], which displays overflow warnings when the system is unable to keep up with the data stream. To reach this performance goal, drastic modifications were made from the original serial code, primarily in the packet detection portions, which were the most computationally intensive. However, only by expanding into the MLDF framework did many of these bottlenecks become apparent. Many were rather hidden when looking at MATLAB profiles alone. Fig. 12 outlines the eventual flowgraph of the system. Both DP and TP type flows were applied as observed in Fig 12, and the overall structure was influenced by [36]. A pure DP implementation would not be efficient since frame boundaries are unknown and variable in the data streams. However, we leverage DP concepts by running the *Moving AverageSP*, *Moving AveragePP*, and *Delay* blocks in parallel on the same data. TP concepts are applied by pipelining the outputs of these parallel running blocks into other concurrently running blocks downstream, eventually funneling into *CFO*, *LLTFSearch*, and *DecodePacket*.

The workflow to reach real-time relied on the strategy discussed in Section V, where hotspots in the flowgraph were spread across more and more threads. An interesting aspect of this design is that the packet decode operations, which can be considered extremely computationally intensive due

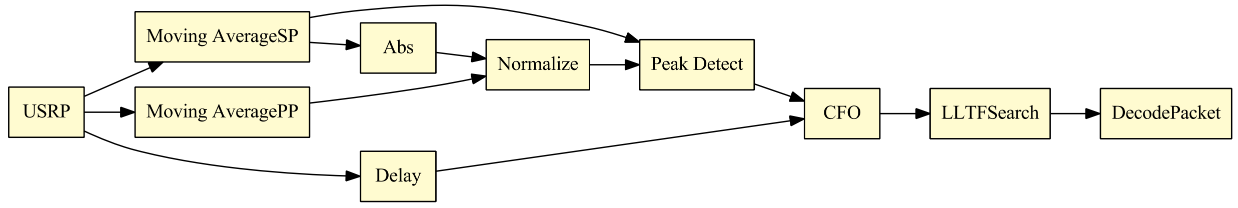


FIGURE 12. Complex flowgraph for WLAN 802.11a receiver, based upon the current design available in MATLAB. Significant resources where put into the packet detection phase of the flowgraph, which consists of six threads (Abs, Moving AverageSP, Moving AveragePP, Normalize, Peak Detect, Delay). This design was able to run in real-time and decode beacon frames from commercial routers. DecodePacket contains the majority of the code, but is called drastically less frequent than the upstream blocks.

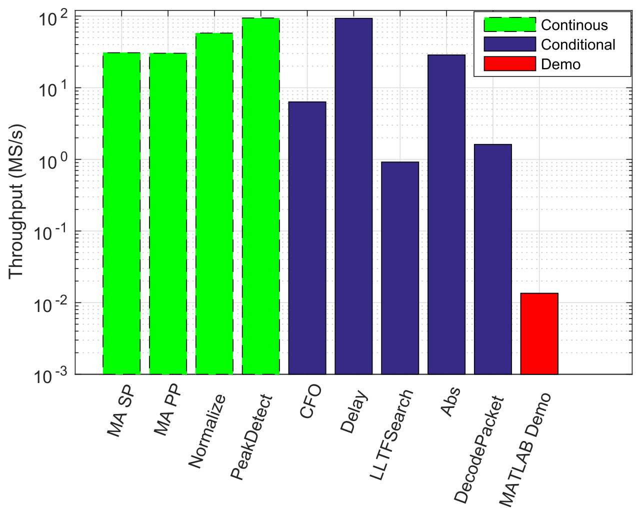


FIGURE 13. Relative throughput of computation blocks in WLAN receiver design. In green are blocks requiring continuous operations, in blue are blocks that have dependent operation, and in red is the original MATLAB demo implementation. The original MATLAB demo utilizes SIMD operations of certain mathematical functions, but does not support user controlled TP or DP operations.

to their reliance on a Turbo Decoder [37], are contained in a single thread. This was possible since the rate at which we observed packets was orders of magnitude below the rate at which the packet detection algorithms worked at. This is also a function of these more intensive blocks only conditionally running. Packets will not always exist in the received data, but searching for such packets is continuous. With this design, we were able to recover packets from commercial access points over the air without overflow for sustained periods of time. In Fig. 13 a breakdown of the relative blocks speeds are provided. The blocks with green bars are required to run continuously, however blocks with blue bars only conditionally run. The MATLAB only demo without the MLDF framework provided by MathWorks is shown in red. All continuously running blocks are well above the 20×10^6 samples per second throughput requirement.

VI. CONCLUSION

In summary, we have presented a framework to enable DataFlow parallelism from the MATLAB programming language. Several perspective of the framework's performance were evaluated to model possible algorithmic acceleration and penalties associated with the architecture. We have shown that it can provide significant speedup in certain applications, but is not without limitation. Included is a simple workflow to implement current code into this framework, which requires no knowledge of thread implementation or efficient data passing between those threads. We hope to open-source this tool, along with several example implementations using this framework, enabling others to experience such performance benefits in their own work.

ACKNOWLEDGMENT

The hardware was provided by the Wireless Innovation Laboratory at Worcester Polytechnic Institute, Worcester, MA, USA.

REFERENCES

[1] G. Conte, S. Tommesani, and F. Zanichelli, "The long and winding road to high-performance image processing with MMX/SSE," in *Proc. 5th IEEE Int. Workshop Comput. Archit. Mach. Perception*, Sep. 2000, pp. 302–310.

[2] S. J. Chapman, *MATLAB Programming for Engineers*. Boston, MA, USA: Cengage Learning, 2007. [Online]. Available: https://books.google.com/books?id=fhpotPvv7v8C

[3] R Development Core Team. (2008). *R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing, Vienna, Austria*. [Online]. Available: http://www.R-project.org

[4] J. W. Eaton, D. Bateman, S. Hauberg, and R. Wehbring. (2015). *GNU Octave Version 4.0.0 Manual: A High-Level Interactive Language for Numerical Computations*. [Online]. Available: http://www.gnu.org/software/octave/doc/interpreter

[5] G. Rossum, "Python reference manual," Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands, Tech. Rep. CS-R9526, 1995.

[6] E. Jones *et al.* (2001). *SciPy: Open Source Scientific Tools for Python*, accessed on Aug. 8, 2016. [Online]. Available: http://www.scipy.org/

[7] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, Mar. 1990. [Online]. Available: http://doi.acm.org/10.1145/77626.79170

[8] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Mar. 1998.

[9] M. Abadi *et al.* (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. [Online]. Available: http://tensorflow.org/

[10] A. Woodie. (Apr. 2016). *Apache Beam's Ambitious Goal: Unify Big Data Development*. [Online]. Available: https://www.datanami.com/2016/04/22/apache-beam-emerges-ambitious-goal-unify-big-data-development/

[11] T. Uustalu and V. Vene, "The essence of dataflow programming," in *Central European Functional Programming School*. Berlin, Germany: Springer, 2005, pp. 135–167.

[12] Z. Or-Bach. (Apr. 2014). *FPGAs as ASIC Alternatives Past and Future*. [Online]. Available: http://www.eetimes.com/author.asp?doc_id=1322021

[13] N. T. Bliss and J. Kepner, "pMatlab parallel Matlab library," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 336–359, Aug. 2007.

[14] W. Gao, Q. Kemao, H. Wang, F. Lin, and H. S. Seah, "Parallel computing for fringe pattern processing: A multicore CPU approach in MATLAB environment," *Opt. Lasers Eng.*, vol. 47, no. 11, pp. 1286–1292, Nov. 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0143816609001079

[15] J. A. Bispo, L. Reis, and J. M. P. Cardoso, "Multi-target C code generation from MATLAB," in *Proc. ACM SIGPLAN Int. Workshop Libraries, Lang., Compil. Array Program. (ARRAY)*, New York, NY, USA, 2014, pp. 95:95–95:100. [Online]. Available: http://doi.acm.org/10.1145/2627373.2627389

[16] A. Prasad, J. Anantpur, and R. Govindarajan, "Automatic compilation of MATLAB programs for synergistic execution on heterogeneous processors," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 152–163, Jun. 2011. [Online]. Available: http://doi.acm.org/10.1145/1993316.1993517

[17] J. Spazier, S. Christgau, and B. Schnor, "Automatic generation of parallel C code for stencil applications written in MATLAB," in *Proc. 3rd ACM SIGPLAN Int. Workshop Libraries, Lang., Compil. Array Program. (ARRAY)*, New York, NY, USA, 2016, pp. 47–54. [Online]. Available: http://doi.acm.org/10.1145/2935323.2935329

[18] P. Ratnalikar and A. Chauhan, "Automatic parallelism through macro dataflow in high-level array languages," in *Proc. 23rd Int. Conf. Parallel Archit. Compil. (PACT)*, New York, NY, USA, 2014, pp. 489–490. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628131

[19] S. Cass. (Jul. 2015). *The 2015 Top Ten Programming Languages*. [Online]. Available: http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages

[20] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Comput. Surv.*, vol. 36, no. 1, pp. 1–34, Mar. 2004. [Online]. Available: http://doi.acm.org/10.1145/1013208.1013209

[21] *GNU Radio Website*, accessed on Apr. 2014. [Online]. Available: http://www.gnuradio.org

[22] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *J. Parallel Distrib. Comput.*, vol. 37, no. 1, pp. 55–69, Aug. 1996.

[23] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proc. 5th ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPOPP)*, New York, NY, USA, 1995, pp. 207–216. [Online]. Available: http://doi.acm.org/10.1145/209936.209958

[24] M. Frigo, "Multithreaded programming in Cilk," in *Proc. Int. Workshop Parallel Symbolic Comput. (PASCO)*, New York, NY, USA, 2007, pp. 13–14. [Online]. Available: http://doi.acm.org/10.1145/1278177.1278181

[25] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "SLAW: A scalable locality-aware adaptive work-stealing scheduler for multi-core systems," in *Proc. 15th ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPoPP)*, New York, NY, USA, 2010, pp. 341–342. [Online]. Available: http://doi.acm.org/10.1145/1693453.1693504

[26] P. Yalamanchili *et al.* (2015). *ArrayFire—A High Performance Software Library for Parallel Computing With an Easy-to-Use API, Atlanta*. [Online]. Available: https://github.com/arrayfire/arrayfire

[27] The MathWorks Inc. (Feb. 2016). *What are System Objects?* [Online]. Available: https://www.mathworks.com/help/dsp/gs/what-are-system-objects.html

[28] The MathWorks, Inc. (Feb. 2016). *Functions and Objects Supported for C and C++ Code Generation*. [Online]. Available: http://www.mathworks.com/help/simulink/ug/functions-supported-for-code-generation–categorical-list.html

[29] The MathWorks, Inc. (Feb. 2016). *MATLAB Coder*. [Online]. Available: http://www.mathworks.com/help/coder/index.html

[30] B. Schäling, *The Boost C++ Libraries*. Cambridge, MA, USA: XML Press, 2011.

[31] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, *Graphviz—Open Source Graph Drawing Tools Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 2001, pp. 483–484.

[32] D. Vyukov. (Jan. 2009). *Single-Producer/Single-Consumer Queue*. [Online]. Available: https://software.intel.com/en-us/articles/single-producer-single-consumer-queue

[33] The MathWorks, Inc. (Feb. 2016). *MEX File Creation API*. [Online]. Available: http://www.mathworks.com/help/matlab/call-mex-files-1.html

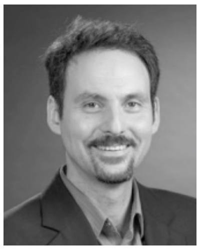
[34] The MathWorks, Inc. (Feb. 2016). *IEEE 802.11 WLAN—OFDM Beacon Receiver With USRP Hardware*. [Online]. Available: http://www.mathworks.com/help/supportpkg/usrpradio/examples/ieee-802-11-tm-wlan-ofdm-beacon-receiver-with-usrp-r-hardware.html

[35] Ettus Research, a National Instruments Company. (Jun. 2016). "USRP B200/B210 Bus Series." [Online]. Available: https://www.ettus.com/content/files/b200-b210_spec_sheet.pdf

[36] B. Bloessl, M. Segata, C. Sommer, and F. Dressler, "An IEEE 802.11a/g/p OFDM receiver for GNU radio," in *Proc. 2nd ACM SIGCOMM Workshop Softw. Radio Implement. Forum (SRIF)*, Hong Kong, Aug. 2013, pp. 9–16.

[37] P. H.-Y. Wu, "On the complexity of turbo decoding algorithms," in *Proc. IEEE VTS 53rd Veh. Technol. Conf. (VTC Spring)*, vol. 2. May 2001, pp. 1439–1443.

TRAVIS F. COLLINS received the B.S., M.S., and Ph.D. degrees from Worcester Polytechnic Institute, Worcester, MA, USA, all in electrical and computer engineering. For the past four years, he has been collaborating with software-defined radio industry leaders, including analog devices, ettus research, and MathWorks, Inc. in commercial and academic domains. His research interests include small-cell interference modeling, phased-array direction finding, and high-performance compute for SDR applications. He has been a member of the Wireless Innovation Laboratory since 2011.



ALEXANDER M. WYGLINSKI received the B.Eng. and Ph.D. degrees from McGill University, Montreal, QC, Canada, in 1999 and 2005, respectively, and the M.Sc. (Eng.) degree from Queen's University, Kingston, ON, Canada, in 2000, all in electrical engineering. He is an Associate Professor of Electrical and Computer Engineering and an Associate Professor of Robotics Engineering with the Worcester Polytechnic Institute, Worcester, MA, USA, where he is the Director of the Wireless Innovation Laboratory. Throughout his academic career, he has published more than 35 journal papers, more than 80 conference papers, nine book chapters, and two textbooks. His current research interests include wireless communications, cognitive radio, software-defined radio, dynamic spectrum access, spectrum measurement and characterization, electromagnetic security, wireless system optimization and adaptation, and cyber physical systems.