Abstract—Compute-intensive applications incorporate ever increasing data processing requirements on hardware systems. Many of these applications have only recently become feasible thanks to the increasing computing power of modern processors. The Matlab language is uniquely situated to support the description of these compute-intensive scientific applications, and consequently has been continuously improved to provide increasing computational support in the form of multithreading for CPUs and utilizing accelerators such as GPUs and FPGAs. Moreover, to take advantage of the computational support in these heterogeneous systems from the problem domain to the computer architecture necessitates a wide breadth of knowledge and understanding. In this work, we present a framework for the development of compute-intensive scientific applications in Matlab using heterogeneous processor systems. We investigate systems containing CPUs, GPUs, and FPGAs. We leverage the capabilities of Matlab and supplement them by automating the mapping, scheduling, and parallel code generation. Our experimental results on a set of benchmarks achieved from 20x to 60x speedups compared to the standard Matlab CPU environment with minimal effort required on the part of the user.

I. INTRODUCTION

In the era of Dark Silicon where area is free and power is not, unconventional cores offer a new way forward [1]. Computing systems are becoming increasingly heterogeneous to optimize for performance, power, and cost [2]. Many systems now include GPUs, DSPs, FPGAs, or other custom ASICs in addition to general purpose processors. However, the heterogeneous hardware/software codesign for systems of such processors has been a long standing problem with increasing complexity and difficulty. Some non-trivial steps in the implementation flow include exploiting task parallelism, customizing the data parallelism within tasks, interfacing cores in a processor and processors in a system, and scheduling tasks effectively over the heterogeneous configuration [3]. Although these problems have been identified early on [3], many of them still persist and plague the development of current systems [4].

Of these non-trivial steps, some are no longer an issue in Matlab based implementations. Matlab already provides high performance data-parallel implementations for many computations [5][6] relieving the user of exploiting data parallelism. In addition, many types of processors including GPUs and FPGAs are already supported by either vendor drivers or academic efforts [7]. The biggest challenge for domain experts using heterogeneous systems stems from the need to find the computation-to-hardware assignments that maximize the overall application performance. This is not just a per-computation decision of which type of processor is better, but must include an overall analysis of the data dependencies between computations and incorporating overheads like data transfer and memory management in order to achieve high performance.

We present a compiler framework not to supplant existing tools or methods in Matlab, but to supplement them in order to create a more cohesive development and run time environment. We seek to ease the development complexities of using heterogeneous systems to enable higher performing implementations. The framework includes a front-end compiler that analyzes the application to identify its core computations. A simulator evaluates different schedules, provides early feedback on potential performance, and allows the developer to tune the configuration of processors in the system. Lastly, the framework produces the implementation of the application ready to be executed requiring no further effort on the part of the developer. The end result is a fully functioning, high performance, heterogeneous system implementation of the application. This paper makes the following contributions:

- A compiler framework specializing in computation identification and dynamic analyses,
- A simulator for analyzing scheduling policies, mapping and assigning computations to processors,
- A code generator that handles memory management, data transfer, and control for a system of heterogeneous processors.

We also present a generic run time environment and show how a developer uses the compiler framework to implement applications. As an example, a medical imaging application is compiled and tuned by manipulating the configurations of processors in the system. We achieve speedups of 20x to 60x compared to execution in a standard single CPU Matlab environment with minimal effort on the part of the user. The rest of this paper is organized as follows. Section II discusses the relevant related work. Section III presents our compiler framework flow. The run-time environment is discussed in Section IV. Next, we present some experiments and demonstrate the ease with which one can use the compiler framework in Section V. Section VI summarizes our contributions.

II. BACKGROUND AND RELATED WORK

Much previous work has investigated heterogeneous system development using lower level languages such as OpenCL or OpenMP that require architecture knowledge to be able to efficiently utilize the processor and all of its compute capabilities. At the other end of the spectrum, a high level domain-specific language like Matlab provides both abstrac-
This best of both worlds allows domain experts to accurately express their algorithm and architecture experts to provide high performance efficient implementations for different processors such as GPUs and FPGAs. In comparison to other works that generate implementations for each processor, we focus on a development environment with an extensible libraries of implementations for different processors. Other approaches such as cross-compiling or high-level synthesis could be used to facilitate extensions to these libraries by generating any missing implementations for other processors.

Banerjee et al. [8] presented the MATCH compiler for heterogeneous systems using Matlab as the targeted input language and generated C++ code in the back end for execution. Other works investigated different compiler designs [9], using a macro dataflow-style for identical CPUs [10]. Still other works have focused on empirically optimizing workload distribution for CPU+GPU systems [11]. Simulators have been developed for theoretical systems [12], abstract systems [13], heterogeneous CPU+GPU systems [14], and CPU+FPGA systems [15]. Hardware simulators are designed to provide accurate low level detailed operation of a particular hardware platform to aid in the design or tuning of architectural features. Yet this level of detail is more than is required to estimate the performance and efficiency of heterogeneous systems.

Lastly, scheduling algorithms for mapping computations from a directed acyclic graph (DAG) to heterogeneous processors have been studied and found to be NP-Hard for finding the optimal schedule [16]. Many works have presented heuristic scheduling approaches in lieu of an optimal algorithm. A few commonly used heuristic scheduling algorithms include HEFT [17], PEFT [18], serial scheduling [19], and adaptive greedy [20]. Braun et al. [21] presented eleven scheduling algorithms including opportunistic load balancing (OLB) and minimum execution time (MET) and evaluated them in a system with uniformly distributed random task compute times.

III. FRAMEWORK

We present a framework to aid the design, implementation, and performance estimation of an application in a heterogeneous system. Normally, a domain expert would develop their algorithm and begin numerical validation using some initial data. Once validated, development focus shifts to improving performance. But for many, the complexities of mapping and scheduling prevent them from achieving high performance. This is where our compiler framework can help. Our goal is to give the application developer, the domain expert, the same ability to tweak their implementation as if they were developing using a lower level programming model. We do not seek to enable domain experts to develop custom kernels, but to use any variety of built-in or external library supplied implementations for heterogeneous processor systems.

Figure 1 illustrates the overall structure of our framework. At a high level, the inputs to the parallelizing compiler are the sequential Matlab code and the configuration of processors in the system, and outputs are the performance report and the parallel Matlab code. Starting from the sequential Matlab implementation, kernels are identified by the front-end compiler, discussed in Section III-A. The dataflow graph of the kernels is passed to the simulator, described in Section III-B, which applies the scheduling policy and estimates overall application performance for the configuration of processors in the system. Finally, the implementation is produced by the code generator presented in Section III-C.

A. Front End Compiler

The front end of our framework includes a compiler that identifies kernels in the application and constructs a dataflow graph (DFG) representing the data dependencies between kernels. Although kernels can be identified statically we must perform a dynamic analysis to determine variable sizes (number of elements) and types such as: scalars, vectors or matrices. Our dynamic analysis is implemented by instrumenting the source code prior to sequential execution using overloaded operators that build the DFG in the background. Although this
Fig. 2: Progression of compilation process from initial source code (a) to the generated kernel DFG (d).

B. Simulator

The simulator is the core of our compiler framework. It analyzes different scheduling policies, provides feedback on potential performance, which allows developers to tune the choice of kernels in the application or processors in the system. Figure 3 shows the process of scheduling kernels from the DFG onto the processors in the system, resulting in a schedule that is later translated into the output parallel Matlab code. After constructing the DFG, the performance of kernels is estimated using processor models for CPU [22][23], GPU [24][25], and FPGA [26][27][28] processors. Compared to other approaches such as off-line training [29], profiling [30], or on-line work stealing [31] processor modeling offers the ability to estimate performance without any execution on the hardware. After performance is estimated, kernels are mapped to processors (represented by the coloring of the kernels in the DFG) using this performance information (represented by the table) as shown in Figure 3a. Then, the final assignments are made by applying the scheduling policy to include data transfer time and other system level overheads as shown in Figure 3b.

We focus on system configurations that fit the typical workstation form factor and include a CPU/Motherboard with PCI Express (PCIe) accelerator cards. Figure 3d shows the configuration of a modern CPU+GPU+FPGA system with PCIe connections both from the chipset and the CPU directly. Notice the GPU is connected directly to the CPU, while the FPGA is connected to the additional PCIe lanes available from the chipset. Given the mapped DFG, a scheduling policy, and the system configuration we simulate execution to determine the final schedule (Figure 3b) of the kernels in the application and when data transfers should occur. Performance statistics such as processor utilization, data transfer and scheduling delays among others are produced during simulation and constitute the Performance Report as shown in Figure 1. Using this information the application developer can make intelligent system configuration decisions (System Optimization Feedback Loop) and optimize the types of kernels in their application (Application Optimization Feedback Loop) by choosing different implementations (ie. regular matrix inversion or triangular matrix inversion).

C. Code Generation

One of the benefits of Matlab, is the already available high performance kernel implementations. In many cases the kernel implementations for Matlab, whether provided by MathWorks or from third party libraries (ArrayFire, MAGMA, etc.), rival those of other high performance scientific libraries [5]. To use these implementations, a Matlab script is generated but the computations are replaced by data transfers to other processors for accelerated execution according to the schedule produced in simulation (Figure 3b). This leaves mainly the control flow and any kernels assigned to be executed by the CPU to be implemented in Matlab’s spmd environment (Figure 3c).

Another important process at this stage is to arrange message-passing sends and receives such that deadlocks will not occur. Matlab’s message-passing implementation suffers from an unseen problem where non-blocking transfers become blocking when the payload size is larger than some threshold [7]. However, we account for this possibility in our code generation procedure. We organize data transfers (each send with its accompanying receive) such that if the sender blocks on a transfer it will not deadlock the system. In addition, we also arrange the sends so that the processor will be idle as little as possible by scheduling the processor to perform some computation first and then the send operation later. Also during this stage, the code for our run time environment is integrated with the compute code to produce a fully functional
(a) Kernel/proc. mapping  (b) Applying scheduling policy  (c) Output parallel Matlab code  (d) Run time organization of threads.

Fig. 3: Parallelization process: mapping kernels (a), scheduling (b), code generation (c), and finally execution on the run time system (d).

implementation that does not require any further user input or manipulation. The final parallel script can be executed in a normal Matlab instance (using the regular run command) which sets up the environment, launches kernels, coordinates data transfers, and any other run time function to complete the execution of the application.

IV. Run Time Management

In heterogeneous systems, a comprehensive run time environment is needed to manage execution, data movement, and other system functions. Our run time environment is based on the message-passing-interface (MPI) capabilities present in Matlab. Each thread is given a static list of work to execute with a pre-determined order of computation and communication between other threads. We define two types of threads in our run time environment: control threads that manage which kernels to execute and when to communicate, and compute threads that receive commands from a control thread and actually execute kernels.

Figure 3d shows this two-thread control/compute organization for a three processor CPU+GPU+FPGA system. We integrate both control and compute functionality into a single thread for the CPU since both threads exist on the same processor. For the GPU and FPGA processors, their compute threads do not actually exist. Instead, the functionality present in the hardware to execute kernels is what we refer to as the compute thread. The control thread sends commands to the GPU or FPGA device to initiate computation, deliver or retrieve data. Since all control threads reside in the CPU, the data transfer between control threads is a zero-copy operation once the result of a kernel has been retrieved from the device.

This static run time environment has some advantages over other dynamic approaches. Since each processor is given the entire list of work at the beginning, there are no system level delays for a processor before being given its next task to execute by some central controller. Instead the only waiting that happens is for data transfers, either waiting to receive some data from another processor or when sending is blocked waiting for the receiver to begin accepting the data in transit. Thus, the run time overhead is very low since management functions to prevent deadlocks are not needed (recall that we handle MPI blocking/non-blocking deadlocks automatically in the code generation process).

V. Experiments and Results

In this section we present sample applications and demonstrate how our framework can parallelize the workload over a system of heterogeneous processors. First, we present three synthetic applications for initial validation, then we present a case study implementing a real-world medical imaging application and the performance improvements achieved.

A. Synthetic Applications

Figures 4b-d show the workloads of three synthetic applications highlighting each kernel’s best processor. Figure 4a shows the ideal mapping of the kernels used in these experiments on a kernel-to-processor basis without data transfer or other system level delays. The goal of this experiment is to demonstrate that our compiler properly accounts not only for the best kernel-to-processor mapping (the naive approach) but also for the data transfer cost. Both the kernel performance and data transfer cost are dependent on the data size (Matlab uses 64bit double precision by default). When manually parallelizing the application, it is difficult to keep all of these factors in mind. Consider the example workload shown in Figure 4b for data sizes of 50 (meaning 50 element vectors or 50x50 sized matrices). Upon initial inspection, it can be easily determined that it is best to keep the matrix-vector multiply kernel in the CPU since data transfer would eliminate the performance benefit of execution in the FPGA. Then, the process continues for the rest of the kernels in the DFG. Our compiler carefully incorporates the cost of data transfer looking not only one kernel ahead, but two or more (as defined by the scheduling policies). This is especially beneficial in the case of the workload shown in Figure 4d in which the performance of one kernel is best in the GPU, but the feeding and consuming kernels are best executed in the FPGA. Conventional wisdom says that matrix-matrix multiplication is the bread and butter of GPU processors. However, at this data size (100) the difference in performance is small and not enough to warrant the additional data transfer cost round-trip between processors. If the situation were different and kernel 4 (inversion) was not present in the application, then in fact
executing that matrix-matrix multiplication in the GPU would provide the best performance. On the contrary, given that the matrix-matrix multiply kernel will execute in the GPU for the workload in Figure 4c, data transfer to the FPGA for the matrix inversion is not beneficial thus it is best to keep the execution of that kernel in the GPU as well. In order to achieve high performance developers not only have to keep in mind the data sizes that their kernels operate on, but also the resting location of data, where it would need to be transferred, and the associated data transfer rate between those two processors.

These three synthetic applications (Figures 4b-d) represent cases in which the best processor for a kernel is not always the best in terms of overall application performance. For these applications, our compiler generates single processor implementations for best performance. Moreover, our compiler generated code outperforms the sequential CPU implementation (except for the workload in Figure 4b which is best executed entirely in the CPU) and other possible naïve implementations.

B. Use Case: Medical Imaging Application

We analyze a medical imaging application that simulates the propagation of the electrical signals from the heart to the body surface, called Noninvasive Transmural Electrophysiological Imaging (NTEPI) [32]. The workload of the application has already been analyzed for implementation in a CPU+GPU system [33], and thus is a good candidate for parallelization among CPU, GPU and FPGA processors together. We vary the configuration of the system by adding or removing processors to validate the mappings chosen. Figure 4a shows the ideal mapping of the kernels used in these experiments on a kernel-to-processor basis ignoring data transfer and other system level delays. The workload for the medical imaging application is shown in Figure 5a. The data size for this application is proportional to the size of the 3D mesh that represents the heart tissue structure, so we evaluate a range of data sizes from 5 to 2000 (64bit double precision, 5 to 2000 element vectors and 5x5 to 2000x2000 element matrices). Notice that for this range, the best processor for matrix inversion (Inv) and matrix-matrix multiply (M-M) varies based on the data size as shown in the kernel design space chart in Figure 4a. Yet, when we implement the application in systems without one of the best processors, the 2nd or 3rd best may have to be used further complicating the implementation for the developer.

We compiled the NTEPI application for three configurations of systems with accelerators: CPU+FPGA, CPU+GPU, and CPU+GPU+FPGA. We modified the source code for each data size and the choice of processors in a configuration file. Both the source code and configuration files are input to the compiler to produce the output parallel code. After executing the parallel code in hardware, we collected the overall application performance as shown in Figure 5b. This chart shows the speedup of each of these systems compared to sequential execution of a single CPU system for the range of data sizes. Notice that for smaller sizes the FPGA provides speedups of up to 60x, and the GPU provides speedups of up to 20x for larger data sizes. For the medium data sizes the capability of the FPGA is stretched beyond its limit but there is not yet enough data to fully utilize the GPU, thus the potential speedup over the single CPU diminishes to 1.6x. When combined together the CPU+GPU+FPGA system is best, achieving performance similar to the higher performing system at any data size. Overall, these speedups are achieved with no effort on the part of the user. Previously, the user would need to manually guess and check trying various different configurations in order to discover the best implementation. Our compiler analyzes the application and parallelizes the execution of kernels to achieve the best performance possible for any given system configuration.
VI. CONCLUSION

We presented a framework to parallelize sequential Matlab programs over a heterogeneous system of processors. Our framework takes as input sequential Matlab scripts, identifies computations, maps and schedules them, estimates application performance prior to execution, and generates fully functional parallel implementations. Our run time environment handles data transfer and communication between processors, memory management, and other system level organizational details. We analyzed three synthetic applications and verified that for the worst case, our compiler does not impair the performance for programs that cannot be parallelized. We also showed speedups from 20x to 60x for a CPU+GPU+FPGA system, that were achieved with our compiler framework for a standard sequential Matlab implementation of a medical imaging application. Our compiler enables users to produce high performance parallel implementations easily and with very little effort.

REFERENCES


