Enabling FPGA support in Matlab based Heterogeneous Systems

Sam Skalicky, Tyler Kwolek, Sonia Lopez, Marcin Lukowiak
Rochester Institute of Technology
Rochester, NY USA
{sxs5464,tjk9644,slaeec,mxleec}@rit.edu

Abstract—FPGAs have been shown to provide orders of magnitude improvement over CPUs and GPUs in terms of absolute performance and energy efficiency for various kernels such as Cholesky decomposition, matrix inversion, and FFT among others. Despite this, the overall performance of many applications suffer when implemented entirely in FPGAs. Combining FPGAs with CPUs and GPUs provides the range of capabilities needed to support diverse computational requirements of applications. Integrating FPGAs into these systems challenges application developers with constructing hardware kernel implementations and interfacing from the low level hardware logic in the FPGA to the high speed networks that connect processors in the system. In this work we extend the compute capabilities of Matlab by incorporating support for FPGAs and automating the parallel code generation. We characterize the system and evaluate the performance gains that can be achieved by adding the FPGA for two compute intensive applications. We present performance results for medical imaging and fluid dynamics applications implemented in a CPU+GPU+FPGA system and achieved over 40x improvement compared to the standard Matlab CPU+GPU environment.

I. INTRODUCTION

The systems used for solving complex engineering and scientific problems in today’s world are becoming increasingly diverse by including heterogeneous configurations of processors such as CPUs, GPUs, and FPGAs. Such configurations promise improved performance, lower power, or lower cost by taking advantage of different hardware platforms. However, developing solutions using such systems requires parallelization on two fronts: data parallelism and task/kernel parallelism. Applications must be decomposed into coarse-grain kernels that can be executed simultaneously by the processors in the system. Each kernel must also be implemented in a data parallel fashion to enable fine-grain parallelism at the individual processor level. To fully exploit data parallelism developers often need domain specific knowledge for each architecture and a firm understanding of the mathematical background to organize the work into a vectorized or matricized format. To fully exploit kernel parallelism, a background in computer architecture is needed in order to determine what kernels should be mapped to which processor.

The state-of-the-art solution is to take a two-pronged approach, separating the problem into: (1) designing and implementing kernels and (2) combining these kernels into an application. The latest versions of Matlab are a good example of this approach. The application developer in Matlab can currently make use of the provided CPU and GPU kernel implementations, as well as communicate between them using Matlab’s parallel environment. Currently, a library of kernel implementations is not provided for FPGAs.

Implementations for various types of computations are available for FPGAs either from: vendors using a core generator, 3rd party developers, or published academic work. But the lingering question is: how can we easily incorporate them into a functioning heterogeneous system? Support for PCI Express (PCIe) communication between CPUs, GPUs, and FPGAs has been a hot topic as of late, providing simple software APIs to communicate with hardware in the FPGA [1][2][3][4]. Moreover, with these recent developments, we can utilize FPGAs not just as secondary coprocessors but instead as first class citizens contributing processing power and improving performance to any part of an application. Thanks to these efforts, FPGAs can be integrated into heterogeneous systems and execute kernels with existing implementations leaving us now with the question: what must the developer provide in order to easily implement a heterogeneous execution of their application?

An application can be broken down into three main components: the types of kernels that will be executed, the data dependencies between these kernels, and the initial data the application begins operating on. In addition to this, the configuration of the system must to be specified, including the type of processors (CPU, GPU, or FPGA for example), quantity of each type, and data transfer rates between each processor. To connect the application to the system, the kernels will need to be scheduled onto the processors for execution. By providing these items separately as shown in Figure 1 we can generate the parallel Matlab code to execute the entire application. In this work we extend the compute capabilities of Matlab by incorporating support for FPGAs and automating the parallel code generation. This builds on the existing parallel Matlab support for CPU and GPU processors, while we provide an additional API to perform computations on the FPGA. We
demonstrate this approach using medical imaging and fluid dynamics applications.

The rest of this paper is organized as follows. Section II presents the related work. Section III provides an overview of the challenges involved in implementing an application in a heterogeneous system, followed by the parallel code generation approach in Section IV. Then we present how the FPGA interfaces with the rest of the system in Section V. Next, we characterize the performance of the data transfer in the system in Section VI. Benchmark applications are described in detail and the system performance results discussed in Section VII, and finally our conclusions in Section VIII.

II. RELATED WORK

In this work, we present an approach for integrating support for FPGAs into a heterogeneous execution environment with CPUs and GPUs using Matlab for kernel implementations, communication and control. Below we present relevant works.

System Design. Adler et al. presented the Logic-based Environment for Application Programming (LEAP) to ease the development and execution of code on any reconfigurable logic that is connected to general purpose processors [5]. In their work, the application developer uses systems calls to manage the FPGA logic and memory, much like a regular embedded system. Similar to LEAP’s efforts to provide an abstracted development environment, Chung et al. presented Connected RAM (CoRAM)[6], an architecture to interface reconfigurable logic seamlessly with memory. CoRAM serves as an interface between the kernels responsible for the actual processing and the memory needed for the applications.

The Reconfigurable data-stream hardware software architecture (Redsharc) is a programming model, system architecture – including high performance on-chip networks, and build infrastructure to simplify development for multiprocessor systems-on-chips (MPSoCs) [7]. Recent efforts have also improved Redsharc to provide an API, build infrastructure, and runtime environment to design a system of simultaneously executing kernels in software or hardware that communicate across a seamless interface [8]. In their work on the Catapult project, Putnam et al. [9] describe an effort to provide a flexible acceleration of a varied problem set in a datacenter environment through the use of FPGAs. They presented a standardized support system for the compute logic, creating their Shell/Role scheme. Compared to CoRAM, they customized the Shell specifically for their problem domain rather than pure general applicability.

Programming. Two widely used APIs for parallel programming that allow one to specify the parallelism of parts of the application include OpenMP and OpenCL. OpenMP provides a set of compiler directives, library routines, and environment variables for the development and manipulation of high-level parallelism. However the role of application developer and kernel developer have not been segmented in this approach, requiring a wide range of knowledge from the application domain to computer architecture. OpenMP can utilize thread communication through the Message Passing Interface (MPI), though this would be a hybrid system requiring additional work by the developer. Further, the developer is required to focus additional time and effort on parallelizing different sections of their algorithm. This includes not only specifying code sections, but also ensuring data dependencies are met. Finally, though there has been additional support and focus on the addition of FPGAs accelerators [10] in OpenMP, the developer still needs to handle additional data and communication difficulties that may arise.

OpenCL, though similar to OpenMP, provides additional support for a wider range of devices, including FPGAs. It enables both kernel and data parallelism, and attempts to handle memory spaces between the various attached processors. Further, FPGA vendors must first provide the support in their tools for OpenCL, which will then be used to compile software down to FPGA implementations. One of the most widely used applications for engineering and scientific research, Matlab, is capable of providing a rich level of support to application developers including CPU and GPU compute libraries and multithreading support. Matlab provides an easy-to-use interface to operate on data in the GPU via Compute Unified Device Architecture (CUDA), taking advantage of the parallel computation elements that exist in the GPU. This allows for native processor support built into Matlab without the requirement of low level architectural complexities. Further, Matlab supports inter-task communication based on a subset of MPI. Through the combination of this support, Matlab enables application developers to achieve significant speedups without requiring them to be experts in the various processor architectures. However, Matlab does not currently provide any official FPGA implementation libraries, a target area that can significantly add to the speedups seen by CPU+GPU systems. An investigation into the potential benefits of incorporating FPGAs into existing systems was presented by Skalicky et al. [11]. They provided guidelines to help programmers select the best performing hardware platform based on various computational factors. Their results show orders of magnitude difference in performance between CPU, GPU, and FPGA architectures as a function of computation type and data size for linear algebra computations.

In many of the previous works, the goal was to enable developers to integrate FPGAs into their existing hardware systems or clusters. They focused on either embedded or high performance computing. This paper details an effort to not only simplify application development using FPGAs, but to improve upon an abstract programming model easing the design difficulties of integrating FPGAs in heterogeneous systems.

III. APPLICATION SUPPORT FOR HETEROGENEOUS SYSTEMS

Matlab provides a high level abstraction that allows developers to describe kernels and data dependencies in their application. Behind the scenes, high performance parallelized implementations of kernels are called to perform the actual computation. In this paper we extend this abstraction for FPGAs, ease developer effort for system control and communication, and characterize the performance of a fully functioning CPU+GPU+FPGA system using Matlab.

Implementations for a wide range of kernels are already available in Matlab for the CPU and GPU, leaving the user the task of determining how to split the workload to achieve better performance. Given an application, implementing it sequentially in the CPU is very simple in Matlab. Extending this CPU version to also utilize the GPU only requires the use of two functions to interface with the GPU: gpuArray and gather that transfer data from the CPU to the GPU memory and vice-versa. Kernels are executed on the GPU
by first transferring a variable to the GPU, and later when this variable is operated on Matlab automatically performs the computation in the GPU, no other effort is required by the user. However, using this approach not only is the user responsible for application decomposition into kernels, they must also determine mapping and scheduling of kernels onto the various processors.

Parallelizing an application across a heterogeneous system is a complex and difficult task to perform manually. Figure 2 shows the standard two level compiler approach introduced with the Stream Virtual Machine (SVM) that was used in the DARPA Polymorphous Computing Architectures (PCA) program [12]. First, at a high level the kernels are identified from the application. Then the performance of each kernel is estimated, dictating which processor each should be assigned to. Then, the order in which each kernel should be executed is scheduled on the processors. The high level compiler produces an abstract representation of the application consisting of a dataflow graph (DFG) of the kernels and their data dependencies, and a schedule containing assignment and ordering information.

Although this abstract representation is human readable and is produced manually in this paper, the goal is for the high level compiler to produce it from sequential code automatically. The low level compiler/linker operates on the abstract representation, mapping kernels to implementations in existing libraries or provided by the user. From the schedule, the control threads are constructed to initiate computation and data transfers. Control threads are synchronized for correct operation by enforcing data dependencies using data transfers.

In this work, we focus on the last part (red items in Figure 2) of the development process: specifying the abstract representation, integrating the DFG, system configuration, and schedule, and generating the parallel implementation necessary to execute the application across multiple processors in a heterogeneous system. The strength of this approach is that the application developer is responsible only for the initial sequential application development. It is typically easier to develop an algorithm that does not require any extra communication or synchronization between operations, and then later determine which kernels should occur in parallel and on which processors for significant speedup. The goal of this approach is to allow the developer to focus on their application while gaining the performance improvement from utilizing different types of processors.

In addition to the regular application development issues, developing for parallel systems also increases the complexity by introducing problems such as deadlocks and memory management. By parsing the DFG, the high level compiler can also constrain the lifetime of temporary variables from initial write to the last read. Using this information we can automatically release the memory for these variables after the last read of the data. Given a cursory understanding of MPI communication, a developer can avoid deadlocks due to two threads trying to receive from each other before sending. But even experienced developers are still mired by deadlocks when non-blocking communication transitions to blocking.

The MPI_send function allows one processor to send a message to another. The data being transferred is held in an intermediary location, known as the “mailbox”, until the receiving processor calls the MPI_receive function to retrieve the data. As the size of the data being transmitted increases and no long fits in the amount of space allocated for the mailbox, this function becomes a blocking call, ultimately halting the progress of the sender until the receiver takes ownership of the data. This problem can be handled automatically for the developer by scheduling the send and receive function calls appropriately to not only prevent deadlocks but also minimize wait time for the other thread to begin communicating. This functionality is common in operating system schedulers using techniques such as gang scheduling or co-scheduling.

IV. Generating Parallel Implementations

The Generate stage from the implementation flow in Figure 1 performs the same duties as the low level compiler shown in Figure 2. This stage accepts the application representation (DFG and input data) in addition to the hardware specification (system configuration and schedule) producing the parallel Matlab code to execute the application. These inputs are collectively referred to as Abstract Representation. The DFG is stored using a representation such as adjacency matrix. In the schedule, the list of kernels appear in the order in which they should be executed for each processor in the system. The input data are the initial values that the first kernels will operate on in the application. Together, these are used to generate the parallel Matlab code. Next, we will describe the procedure used to generate this parallel code.

Initialization. First, the Matlab workspace is cleared and configured for the number of threads needed for the application. Then the abstract representation is loaded from files. Using the system configuration, the spmd environment is configured and the initialization code for each processor written out into the parallel Matlab script. The single program multiple data (spmd) environment provided in Matlab supports multiple control/compute threads operating simultaneously. Since the

![Fig. 3: Number of PCIe lanes in AMD-based system with control and compute threads. The Compute Thread in the CPU is also its own control thread.](Image 50x627 to 300x738)
GPU and FPGA cannot operate directly, computation must be initiated by a Control Thread running on the CPU. In our implementation, each processor has a Control Thread and a Compute Thread. However, since the CPU can initiate its own computations, both control and compute functionality can be integrated into the same thread. Figure 3 shows this organization of threads.

**Control and Compute.** The next step is to generate each processor’s control and compute thread. For GPU or FPGA processors, initialization is performed using `reset(gpuDevice)` and `fpgaReset(fpgaId)`, respectively. The CPU control thread is integrated with its compute thread, whereas the GPU and FPGA are each controlled by threads that perform no computation themselves. Each control thread includes the associated device (GPU or FPGA) communications, as well as communication to other control threads and synchronization. When generating the code to execute a kernel, we check that the input data for that kernel exists on the processor and if not, we insert transfers from the processor that produced the data as needed. During this process, additional analyses are performed and transfers rearranged to prevent deadlocks. The amount of memory available on the particular device is also accounted for, and code to free memory allocated for temporary variables is also generated.

**Cleanup and Finalize.** The final step of the generation is to transfer any final results back to the control thread and perform cleanup of the processors. At the completion of this process, a single Matlab script is produced and can then be executed immediately by the user. The user can simply run this script, no other interaction is required. All of the control and compute threads are launched, communicate and compute, and transfer any results back to a single location upon completion.

### V. FPGA Interfacing and Hardware Kernel Support

The FPGA interfaces with other processors in the system using PCIe by leveraging existing frameworks that combine a hardware IP and a software driver. Many PCIe frameworks are now available including vendor reference designs, 3rd-party designs, or academic efforts [3][2] including the Reusable Integration Framework for FPGA Accelerators (RIFFA) [13]. We use RIFFA Version 2.02 in this work1 with a 64-bit interface. In addition to the HDL that interfaces the compute logic on the FPGA to PCIe interface, RIFFA also includes a kernel driver and a library for linking user software. We encapsulate the FPGA interface using the “Matlab executable” (MEX) API. This allows compiled C code to be called from Matlab scripts. These functions enable similar functionality to the built-in Matlab GPU functions, `gpuArray` and `gather` that transfer data to and from the GPU, respectively. The FPGA interface functions are described in Table I.

RIFFA provides the ability to direct data transfers to various channels on the FPGA, based on a user’s design. Each hardware core can interface with a separate channel to ease communication between multiple hardware cores and the CPU. This allows multiple simultaneous computations to execute on the FPGA, overlapping reconfiguration or data transfer with computation and providing a customized compute capability than the fixed multicore CPU architecture. The application developer can therefore focus on their algorithm while the domain specialist focuses on their implementation in the FPGA, without either needing to develop complex interfaces. The system’s PCIe interconnects are illustrated in Figure 3. The FPGA has been configured for a 4x lane PCIe 2.0 interface with a theoretical bandwidth of 2GBps. The GPU has a 16x lane PCIe 2.0 interface with a theoretical bandwidth of 8GBps. The Unified Media Interface (UMI) connects AMD’s CPU, now referred to as Accelerated Processing Units (APUs), and the chipset or Fusion Controller Hub (FCH). This link is based on PCIe 2.0 giving it a theoretical bandwidth of 2GBs, or enough to support the 2GBps bandwidth of the FPGA.

### VI. System Characterization

Data transfer to/from the FPGA is supported using our `fpgaSend` and `fpgaRecv` functions as presented in the previous section. These transfers move data between the processors control and compute threads. For example, to transfer data between the FPGA and the CPU compute thread, effectively data must be transferred twice: once from FPGA to the FPGA’s control thread, and again from the FPGA’s control thread to the CPU compute thread. This process is similar for communications between the CPU and GPU, while requiring a three transfer process for FPGA to GPU. Future efforts will work to integrate a direct GPU to FPGA communication path and also enable CPU threads to exchange data by passing pointers rather than copying data. This section shows the characterization of our system in terms of communication and data management. The specifications for each processor used in the system are shown in Table II.

<table>
<thead>
<tr>
<th>Name</th>
<th>Args</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fpgaDevice</code></td>
<td>ID</td>
<td>Enumerates the FPGA devices in the system providing detailed information about each. Equivalent to <code>gpuDevice</code></td>
</tr>
<tr>
<td><code>fpgaOpen</code></td>
<td>ptr</td>
<td>Initializes the device referenced by the given ID and returns a pointer to the device used to interface with that device</td>
</tr>
<tr>
<td><code>fpgaSend</code></td>
<td>channel A ptr</td>
<td>Sends the data at the specified channel on the FPGA referenced by the pointer <code>ptr</code>. Returns the number of bytes sent. Equivalent to <code>gpuArray</code> <code>numel</code></td>
</tr>
<tr>
<td><code>fpgaRecv</code></td>
<td>ptr channel A numeroel</td>
<td>Receives <code>numel</code> elements of data from the specified channel on the device referenced by the pointer <code>ptr</code> and returns an array of data. Equivalent to <code>gather</code></td>
</tr>
<tr>
<td><code>fpgaCompl</code></td>
<td>ptr CType</td>
<td>Initiates computation for the kernel specified by <code>CType</code> on the device referenced by the pointer <code>ptr</code></td>
</tr>
<tr>
<td><code>fpgaReset</code></td>
<td>ID</td>
<td>Resets the device with the given <code>ID</code>, including the state of the PCIe control logic and all transfers across all channels.</td>
</tr>
<tr>
<td><code>fpgaClose</code></td>
<td>ptr</td>
<td>Resets the device referenced by the pointer <code>ptr</code> and frees any allocated memory for data structures</td>
</tr>
</tbody>
</table>

### TABLE II: Processor Specifications

<table>
<thead>
<tr>
<th>Processor</th>
<th>Specifications</th>
<th>Implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td>AMD A10-5800K 3.8GHz 16GB DDR3 @ 1600MHz</td>
<td>MathWorks Matlab 2012a 32b</td>
</tr>
<tr>
<td><strong>GPU</strong></td>
<td>Nvidia GTX480 607MHz 1280MB GDDR5 @ 1.67GHz</td>
<td>MathWorks Matlab 2012a 32b</td>
</tr>
<tr>
<td><strong>FPGA</strong></td>
<td>Xilinx Kintex 325T, KC705 1GB DDR3 @ 1600MHz</td>
<td>[14][15][16][17][18][19]</td>
</tr>
</tbody>
</table>

---

1. While this paper was under review, a new version of RIFFA was released: v2.1 with Matlab support.
A. Inter-processor Communication

CPU to FPGA. After developing the MEX interface for the FPGA in Matlab, we analyzed the performance of data transfers and compared them to both the C++ implementation and the benchmarks provided in [13]. The difference in performance between the Matlab MEX and C++ implementations were negligible. Figure 4 shows the bandwidth achieved as the payload size was varied from 1024B to 1024MB. Notice the transfers to the FPGA were significantly faster than the receives. The previous work did not include the Kintex FPGA that we used, nor a 4x lane PCIe 2.0 configuration. Comparing to the 8x lane PCIe 1.0 which has an equivalent theoretical bandwidth, the performance we found was much lower.

![Fig. 4: Transfer bandwidths for CPU/GPU and CPU/FPGA as a function of payload size.](image)

CPU to GPU. Figure 4 shows the bandwidth achieved as the payload size was varied from 1024B to 256MB. We were only able to test up to 256MB transfers as Matlab produced an error when transferring 512MB payloads. Since our Nvidia GeForce GTX480 card has 1280MB of memory and can actually support both 512MB and 1024MB data sizes, we narrowed the source of this discrepancy to the infrastructure provided by Matlab and concluded that it is introducing these additional limitations. Even though the GPU has four times as many PCIe lanes, it only achieved 2x bandwidth over the FPGA.

B. Deadlock

A common issue for programs that use MPI is communication deadlock. When the size of the data being transferred is larger than the size of the MPI “mailbox” the non-blocking communication functions transition to blocking implementations. This issue is present in Matlab and the equivalent function to MPI_send is labSend. We found that the labSend function does show this characteristic of transitioning to a blocking approach. We experimentally found that this limit to be 128KB, or a 128x128 double precision matrix. The subset of MPI that Matlab supports does not include the capability to configure the size of the mailboxes. Working around this limitation requires forward planning for sending and receiving data.

C. Memory Management

A secondary set of problems that is typically observed during parallel development is device memory management. In comparison to the CPU, both the GPU and FPGA have a limited amount of memory that can be utilized. Since Matlab’s abstraction presents limitations, we found that the GPU is typically more constrained than the FPGA, as the total memory available to the user may not be the actual total memory on board the chip. When operating on large data sizes, or after many kernels execute in the GPU this memory becomes fully utilized and errors arise when attempting to allocate more space. In comparison, typically the memory in the FPGA is partitioned as needed as inputs to each kernel implementation. For example, in the matrix addition kernel three matrices are stored: A, B, and C to compute $C = A + B$. The result of the addition is always stored in the same location in memory, overwriting any previous value. In addition to freeing unused temporary variables, it is also the responsibility of the user to ensure that their data sets can fit on the specified device.

VII. Application Performance Results

To analyze the overall performance benefits of including the FPGA in a Matlab based heterogeneous system, we implemented both medical imaging and fluid dynamics applications in various system configurations. In this section we introduce each application, describe its workload and DFG, and how it was implemented across various types of processors.

A. Noninvasive Transmural Electrophysiological Imaging

The NTEPI Algorithm employs a sequential maximum a posteriori (MAP) estimation of the transmural action potential (electrical propagation) distributions given the body-surface potential data (as measured from a standard ECG) [20]. At each time step when a new sample is available, a Cholesky decomposition is performed, then a set of sample vectors are generated. Each sample vector individually enters into simulation of the Alive-Panfilov models to predict a new set of sample vectors, to estimate future propagation of electrical potential. Since the sample ECG measurements contain electrical noise from sources other than the heart, such as the respiratory muscles that are located between the electrodes and the heart, a Kalman filter is used to reduce the impact of random noise from the data. The Kalman update process requires inverting an $M \times M$ matrix where $M$ is the dimension of the body surface data. This prediction and update processes is repeated iteratively. A typical patient analysis requires 2000-3000 iterations.

Each iteration requires twelve computations broken down into one Cholesky decomposition, one matrix inversion, and ten matrix-matrix multiplications. Between iterations there is no opportunity for overlap as every iteration is dependent on the previous update calculations. However, there is sufficient parallelism within each iteration to potentially keep all three CPU, GPU, and FPGA processors busy.

B. Shallow Water Simulation

Shallow water simulations utilize a set of hyperbolic partial differential equations to model the flow of liquid below a pressure surface, like the force of gravity on the surface of the ocean. These simulations typically cover both a large data set (large ocean-sized region) over a period of time, which leads to extensive processing requirements. These equations calculate the wave propagation using the total fluid column height, the water’s horizontal velocity as averaged across the vertical column, the acceleration due to gravity. Using an
approach similar to a Runge-Kutta method, at each time step the discrete values are used to calculate the propagation for a half-step and then used to compute new values for the next time step. Since these calculations operate on a grid of discretized points, all of the kernels in the application are element-wise computations on a matrix. Each iteration requires 15 matrix additions, 24 matrix subtractions, 4 element-wise multiplications, 29 matrix scalings, 15 matrix divisions, and 17 element-wise squaring kernels. Previous work on improving the performance of this application has mostly focused on multi-core CPUs and GPUs [21][22][23]. We investigate the performance benefit that adding the FPGA provides for these simulations.

C. Performance Analysis

Each application was implemented for various system configurations. Single processor systems utilized only the CPU, GPU, or FPGA to execute computations. Processors were organized in the following way for two-processor system configurations: CPU+GPU, CPU+FPGA, CPU+CPU, and GPU+FPGA. We also investigated the benefit of a three-processor system containing CPU+GPU+FPGA. For each configuration, the same application DFG was used. Different system configurations and schedules were fed into the Generate flow to produce the appropriate parallel Matlab scripts.

Previous work has analyzed CPU+GPU+FPGA systems for medical imaging and DSP applications [11]. They found that the FPGA is best for computations on smaller data sizes and for kernels with complex control flow such as Cholesky decomposition.

The results for the Noninvasive Transmural Electrophysiological Imaging (NTEPI) application are shown in Figure 5. Contrary to the previous work [11], our initial results showed that the FPGA was not able to add any value to the system at any data size. In fact, the best performing system was the CPU+GPU since the CPU excelled at executing kernels quickly at smaller sizes and the GPU at larger sizes.

We investigated the source of the disappointing FPGA performance and found that the overhead required to initiate kernel computations made using the FPGA not beneficial. To address this issue, we implemented a simple controller in the FPGA to accept a list of kernels to be executed from the CPU control thread, and execute them in order without any more direction from the control thread. This same capability is not possible for the GPU since it would require merging the currently unavailable source code for kernel implementations to remove the inter-kernel control requirement. The results with reduced FPGA overhead are shown in Figure 6. Using this reduced overhead approach the single FPGA achieved speedups of 40x over CPU+GPU and 49x over GPU for smaller data sizes. This is due to the overhead of the GPU and extra transfers to/from the CPU that prevent the other two configurations from achieving the same performance as just the FPGA.

The results for the shallow water application are shown in Figure 7 with the additional overhead reducing the performance of the FPGA, and in Figure 8 with the improvements of adding an embedded controller to reduce control and communication overhead necessary to initiate computation. In this application the kernels were much simpler, containing almost no complex control flow, and resulted in very small execution times. Both the NTEPI and shallow water applications were evaluated with the same range of data sizes. Thus, the data transfers were much more of an impact on overall application performance of shallow water than NTEPI. This resulted...
in very few transfers, and in fact only in the CPU+CPU and GPU+FPGA configurations. The only configurations that utilized more than a single processor were the CPU+CPU and GPU+FPGA configurations. Compared to the single CPU system for the 2000 data size, the dual CPU+CPU achieved a 1.4x improvement. However for the approach with the overhead, the GPU+FPGA configuration performed worse than the single FPGA system due to the additional data transfers.

Overall, for both of these applications, the FPGA improved the performance of the system. We found that not necessarily all processors will be used simultaneously when trying to tailor the system for high performance. In fact, we also analyzed all three processors in a CPU+GPU+FPGA system and found that for small sizes in the NTEPI application, the best performance was achieved using only the FPGA, while only CPU and GPU were used for medium sizes, and only the GPU was used for larger sizes. Similarly for the shallow water application, only the GPU and FPGA were used for all sizes.

VIII. CONCLUSION

In this work we presented an effort to incorporate FPGAs into Matlab based heterogeneous systems and ease development by generating parallel Matlab code. Developers describe their algorithm in Matlab where simple sequential execution allows for focusing on the application and numerical debugging. The kernel abstraction that the Matlab environment provides, allows architecture experts to implement kernels for various processors including CPU, GPU, and FPGA. Our flow handles the implementation problems such as managing memory, communicating between processors, and generating parallel runnable code. We analyzed two compute intensive applications for various system configurations that showcased the performance gains that can be achieved using such heterogeneous systems. We presented performance results for medical imaging and fluid dynamics applications implemented in a CPU+GPU+FPGA system and achieved over 40x improvement compared to the standard Matlab CPU+GPU environment.

Moving forward, the development of applications in the Matlab environment has many benefits for both the high level developer and the kernel developer allowing them to more easily work together to achieve high performance in heterogeneous systems. Future work will focus on automating the entire design flow, from sequential Matlab code to a parallel implementation for a heterogeneous system. Additionally, we will add support for Partial Reconfiguration to the system and provide a standard interface to enable many hardware kernel implementations to be synthesized and utilized by application developers.

REFERENCES


