AutoFLEX: A Framework for Image Processing Applications on Multiple-FPGA Systems

Vijaykrishnan Narayanan, Ahmed Al Maashri, Kevin Irick, Michael DeBole and Sungho Park
Microsystems Design Lab, The Pennsylvania State University, University Park, PA, USA

Abstract - Field-Programmable Gate Arrays, FPGAs, are reconfigurable logic devices that afford rapid realization of hardware systems that are far less costly yet exhibit comparable performance to an ASIC. Consequently, FPGAs have become the preferred implementation platform for many complex and performance demanding application domains including Signal and Image Processing. The difficulties associated with employing FPGA technology has, however, created need for design tools that ease the process of building systems to a degree suitable for those working at the algorithm, application, and system levels. In this paper, we describe a hardware/software co-framework that aims to automate and accelerate the design of image processing applications on Multi-FPGA computing platforms. From a system composition perspective, the framework abstracts the details of RTL hardware design, communication infrastructure, and memory hierarchy, while providing useful forecasts of expected system performance and device utilization. Moreover, the framework’s scalable architecture allows for expansion of its base functionality including system specification input, device mapping methodology, and hardware exploration and optimization strategies.

Category: Panel Session

Keywords: FPGA Design, Image Processing

1 Introduction

Signal and image processing applications are based on algorithms that are usually computationally expensive. Running these algorithms on general purpose processing machines is impractical due to low performance. On the other hand, the fabrication complexity and time-to-market of ASIC chips discourages developers from choosing ASICs as an alternative for handling these algorithms. Therefore, the last decade has witnessed an increasing interest in reconfigurable computing machines such as FPGAs.

FPGAs are programmable integrated circuits that allow hardware designers to build customized architectures that target specific applications. FPGAs become the preferred choice for implementing signal and image processing application for a number of reasons including long-term availability, low power consumption and high configurability. Furthermore, multiple FPGA chips can be hosted on a single platform, in what is known as Multiple-FPGA System (MFS). MFS systems offer the hardware designers even higher computational power and larger fabric to map the design to. For example, the BEE3 platform [1] hosts 4 Virtex-5 chips [2] that are connected via high bandwidth channels allowing inter-module communication. However, designing a hardware architecture that targets an FPGA board requires certain skills and expertise that many designers may lack. Complexity gets even worse when mapping the design to a MFS, where more effort is required to furnish the design with the necessary communication infrastructure to ensure a successful operation. Hence emerges the need for design automation tools that assist the users in building their design easily and efficiently, saving them both time and effort.

In previous work, we introduced AlgoFLEX [3], a FPGA-based development framework intended for application developers who want to use FPGAs for their performance benefits, but do not possess the expertise of a seasoned hardware engineer. In this paper, we describe AutoFLEX, a software/hardware co-framework that utilizes the AlgoFLEX framework to automate and accelerate the process of building signal and image processing applications with a minimal effort at the user side. This framework abstracts the details of hardware RTL design, communication infrastructure and memory hierarchy as well as automating the synthesis of design and programming the FPGA chips. In addition, the framework provides the user with a friendly graphical user interface (GUI) that further simplifies the process of building an application.

The rest of the paper is organized as follows; the next section briefly describes the AlgoFLEX framework. Section 3 describes the automation process of the AutoFLEX framework, while section 4 discusses the software architecture of the AutoFLEX framework. Section 5 discusses results, followed by the Conclusion.
2 The AlgoFLEX Framework

In this section we describe a hardware infrastructure that we call AlgoFLEX. This hardware infrastructure provides the system designer with the capability to embed a collection of custom accelerator blocks without requiring intimate knowledge of how those blocks will be composed and interconnect. Figure 1 illustrates how the platform consists of a hybrid communication network comprised of a System Local Bus (SLB) and a packet-based on-chip router along with memory controllers, accelerators and other system components. While the SLB serves as the communication backbone within a single FPGA, the router enables both intra- and inter-FPGA modules communication. Each router has 8 ports; from which 2 are reserved for internal components. These two components are the mailbox and the DMAM unit described later. The router forwards packets based on a routing table. This routing table can be configured statically (i.e. pre-synthesis) or dynamically (i.e. at run time). This communication is primarily accomplished through the use of mailbox modules which provide messaging capabilities between various tasks executing on different FPGAs. Moreover, these mailboxes ensure proper dataflow between the tasks that are carried out on multiple FPGAs. In particular, messaging and synchronization is accomplished by uniquely binding a message to a task through the use of application-defined message identifiers. Upon receiving an incoming message the mailbox module sends an interrupt signal to a microcontroller which handles message interpretation for data synchronization. Once the appropriate handler has been initiated, the message is decoded and one or more operations are triggered in the hardware platform.

In addition to custom accelerator modules, the AlgoFLEX infrastructure also allows for the extension of the underlying base platform with negligible impact on the data flow and control flow abstractions. For example, when migrating from the BEE2 to BEE3 platform, modifications to the base platform were limited to the memory controllers and the inter-FPGA physical interfaces, while the remaining accelerator and mailbox components went unchanged. The adaptability of the platform is due to the modular design of two key computational modules; the Direct Memory Access and Manipulate (DMAM) unit and the Universal Custom Accelerator Module (UCAM).

2.1 DMAM Unit

Figure 2 shows the Direct Memory Access and Manipulate unit. This unit supports burst memory transactions, DMA request initiation and packetizing/de-packetizing for on-chip and off-chip communications. In addition to being the primary facility for transporting data throughout the system, the DMAM can perform manipulations on data as it moves that data from source to destination. This is made possible through the inclusion of application specific Stream Operators. Stream Operators are the ideal implementation modality for common streaming operations including numerical conversion, color space conversion, coordinate system conversion, data scaling, computing histograms, computing vector differences, and data binning.
Stream Operator play an important role in a number of applications. For example, in an application where data should be geometrically binned by coordinates, a stream operator can evaluate the coordinates of streaming source data points and map each datum to its geometric neighborhood in the system memory by modifying the routing address of the containing packet.

In addition, DMAM supports data specific manipulation, where – in the packetization phase – it appends qualifiers to data packets to signal that the appropriate stream operator should operate on the incoming data. Moreover, the DMAM supports user defined coalescing and splitting of individual packets to minimize memory access overhead and increase network transaction efficiency.

2.2 UCAM

The Universal Custom Accelerator Module is a wrapper for integrating any accelerator into the AlgoFLEX framework. The UCAM defines a standard interface between the built-in facilities and an instance of a specific accelerator hardware module. The functional capabilities of the accelerator are exposed through the UCAM to any other UCAM compliant accelerator in the system. AlgoFLEX allows the system to be composed of a number of interconnected UCAM entities, with each UCAM offering custom acceleration to various parts of an algorithm. Each of these UCAM accelerators can then be composed at a system level to achieve the full functionality of the desired algorithm.

The UCAM design takes advantage of the fact that many signal processing applications can be characterized by the following: 1) at the application level a sequence of subtasks are repeatedly performed on incoming data and 2) the execution of subtasks consists of performing a static set of computationally intensive operations in a sequence that varies across any particular invocation of the subtask. For example, an optimized Fast Fourier Transform (FFT) implementation may selectively perform decomposition of data based on the dimensionality of the data where 1-Dimensional FFT operations will be performed in all cases, but the sequence of these operations can be radically different across different problem specifications. In the AlgoFLEX framework both the application and subtask control can be implemented in software on an embedded microprocessor, albeit with reduced system performance. To mitigate the overhead of software orchestration of the accelerator’s operations, the UCAM provides an alternative hardware Command Fetch Unit that facilitates fetching, decoding, and preprocessing of accelerator specific instruction sequences. These instruction sequences can be created and modified as necessary for a particular invocation. Consequently, applications benefit from the low overhead of direct hardware instruction fetching and decoding without the rigidity of a hardcoded state machine.

In order to integrate a usable core into the AlgoFLEX framework, the designer of each accelerator provides an HDL implementation that is compliant with the handshaking protocol and signaling needed by the UCAM interface. Furthermore, the designer of the module specifies the set of primitive commands supported by the accelerator module. At a more abstract level, the algorithm designer utilizes the module in different fashions by composing instruction sequences of primitive commands. Figure 3 illustrates the interface between the Application Specific Instruction Processor (ASIP) and an accelerator core.

The consumption of instructions can be throttled through the UCAMs handshaking protocol, allowing the accelerator to have fine grained control of the instructions presented to it by the Command Fetch Unit. In a typical scenario, the Command Fetch Unit fetches a command and dispatches it to an appropriate Command Handler module. The Command Handler performs any instruction specific preprocessing before presenting the command to the accelerator via the Processor Command bus along with the asserted Processor Command Valid signal. If the accelerator core is available to process the command it will assert the Processor Command Acknowledge signal along with any instruction specific response information via the Processor Command Response bus.

The UCAM provides a set of built-in Command Handlers that are useful for many algorithm accelerators. A window fetch handler supports optimized fetching of two-dimensional data with programmable column size, number of rows and columns, row and column offset, and inter-row and inter-column strides. An event signaling handler provides a mechanism for sending short messages to a list of UCAM modules. Moreover, the command handling capabilities of the UCAM can be extended by the inclusion of custom command handlers. For example, the framework contains a regridding core which contains a custom handler that loads an array of data into the accelerator by traversing a linked list of data elements from external memory.

In the next section we describe the AutoFLEX framework that automates the process of composing a design based on the AlgoFLEX framework.

3 Phases of Automation Process

AutoFLEX is a software framework that is based on the AlgoFLEX framework. This tool automates the process of composing DSP applications, while hiding the complexity faced when mapping such applications to a MFS platforms. AutoFLEX implements a number of phases executed in sequence as illustrated in Figure 4. This section discusses these phases in details.
3.1 System Composition

The first phase is concerned with composing the system that represents the desired application. As illustrated in Figure 5, AutoFLEX offers a GUI interface that allows users to compose applications by simply drag-n-drop components from the components library into the design space. Each component can be configured by setting a number of parameters in the property grid. The components library is populated with the following built-in component categories:

- **Platforms**: This category includes the supported MFS platforms. For example, AutoFLEX comes with a built-in support for BEE3 platform.

- **Specifications**: Users can use components in this category to specify a number of parameters describing the application. For example, if the application is related to image processing, then the user can specify the number of dimensions and the size of the input images.

- **Configurations**: These components are not related to the application directly, but rather used to configure parameters related to logistics such as specifying the server at which the system is synthesized.

- **ASIPs**: Application Specific Instruction Processors are hardware accelerators performing particular operations (e.g. FFT) at a very high speed. The user adds these components to the design and connects them according to the desired application.

The next phase maps the ASIPs to the MFS platform.

3.2 Components Mapping

This phase performs two tasks: 1) checking the sanity of the design and 2) mapping ASIPs to the MFS platform. Sanity checking is done by traversing the flow from the input to the output to ensure that all components in the design are connected and that there are no discontinuities. In case a flaw was detected in the design, then the tool will abort operation and user will be notified by an informative message. On the other hand, if the composed design passes the sanity check, then the tool will assign each ASIP to an FPGA. The mapping strategy is based on First-Come-First-Served, in which the tool will traverse the design flow and whenever an ASIP is encountered, then that ASIP is allocated to the first available FPGA.

3.3 AlgoFLEX Infrastructure

The previous phase was responsible for mapping all ASIPs to the MFS platform. However, ASIPs cannot operate without an infrastructure facilitating inter-FPGA as well as inter-Process communication. Therefore, the tool uses the AlgoFLEX infrastructure framework to add the necessary hardware components to the FPGA chips hosted by the MFS platform.

3.4 Exploration & HDL Generation

AutoFLEX can provide the user with an estimate of the expected system performance and device utilization. These estimates are based on running algorithms that perform a trade-off between accuracy/performance vs. available fabric on FPGA. Then, the tool generates the HDL code based on the predictions made by the exploration phase.
Figure 4: The automation process phases. Note that the “Results Verification” is an optional phase that allows the user to check the correctness and accuracy of the results.

3.5 Design Decoration

This phase traces the flow of the design and complements the design by adding the necessary hardware components to ensure a successful operation. For instance, let us assume that the input data sent from the host machine to an FPGA is of single precision format. However, the user has added an ASIP that accepts data with fixed-point format only. In this case, the tool will add a Stream Operator as in interface between the FPGA input and the ASIP. This Stream Operator will convert the format of the input data into a suitable format to the ASIP.

3.6 Synthesis

At this stage, all the hardware components and infrastructure have been put in place and ready for synthesis. Since AutoFLEX depends on Xilinx ISE tools [4] to perform synthesis, hence, the tool needs to update some of the Xilinx-specific configuration files (e.g., .bbd, .pao and .mpd) so that the components introduced by AlgoFLEX are properly integrated to the framework. After that, the tool transfers the design project folders to the synthesis server that was identified earlier in the system composition phase. Next, the tool will connect to the synthesis server and
remotely invoke a set of instruction to launch the synthesis process. In addition, the software running on the processor in the FPGA will be compiled to generate the executables that will drive the operations on the FPGAs. Upon a successful synthesis, the generated bitstream used to program the FPGAs and the executables files that drive the design are copied from the synthesis server to the host machine to be used as discussed program the FPGAs as discussed in the following section. It is worth stating here is that all file transfers and remote method invocations are encrypted for protection against IP theft.

3.7 Programming

The framework uses the bitstreams and executables from the previous phase to program the FPGAs, which is done with the help of the Xilinx tools: IMPACT and xmd. The tool needs to copy the bitstreams and executables to the programming server, which is the machine to which the MFS platform is connected to through the JTAG cable. Then, the tool will generate and transfer scripts that contain instructions for the programming server. Next, the tool will remotely invoke these scripts to start the process of programming the FPGA boards. Once again, all data transfers and remote method invocations are encrypted.

3.8 Dynamic Configurations Generation

Some of the hardware components in the AlgoFLEX framework need to be dynamically configured at run time. For example, the command sequence – which is needed by ASIP in order to perform its designated task – is only generated at run time by the host machine. The generation of these configurations follows a standard format that was developed locally. The standard organizes these configurations hierarchically as shown in Figure 6. These configurations will be communicated to the FPGA chips in the next phase.

3.9 Application Running

This phase is concerned with running the application. Figure 7 illustrates the message exchange between the host machine and the target FPGA boards. First, the host machine will start by transferring the components’ configurations to the FPGAs boards. Once all the configurations are transferred, the host machine will wait until all FPGAs in the MFS platform are fully done with configuring all the components. After that, the MFS platform will request the host for an input data to be processed and the host machine will respond by sending the data along with a request for the output result. Once that input data is fully processed, the result will be sent back to the host machine. The MFS will keep requesting for input data and the host machine will respond to these requests until all input data are processed.

3.10 Results Verification

AutoFLEX offers an optional feature for verifying the results obtained from the MFS platform. The verification process will compare the output of an FPGA to a reference output (e.g. produced by MATLAB). The framework will display the results of the verification both textually (e.g. statistics) and visually (e.g. FPGA processed image).

![Message Types and Packet Lengths](image)

Figure 6: Any data exchanged between host machine and MFS follow a standard format. (a) Format of a packet holding component configurations. The “Msg Type” field is used to indicate that this data is for configuring a hardware component. The next header indicates which FPGA that these configurations are destined to. Then, the next set of headers indicates the component and/or subcomponent that needs to be configured. (b) Format of packet holding data. The “Msg Type” field indicates whether this packet is a request for data or a reply with some data. A set of routing parameters are needed to be used by intermediary components to forward the packet to the right destination.
4 AutoFLEX software Architecture

This section discusses the software architecture of the tool. We restrict the discussion to only two main design concepts that AutoFLEX was designed around.

4.1 Expandability

One of the main objectives of the automation tool is expandability. This means that the tool is able to host and operate with components developed by third parties. This allows users of this tool to expand its capability by simply adding more components to the framework.

AutoFLEX was designed in a way that it hosts a modular representation of the system. This representation includes a parent class that represents each FPGA in the system. Each FPGA object hosts a number of other objects that represents other components found inside the FPGA chip (e.g. Mailbox, Router, etc...). Figure 8 shows an illustration of the system modular representation as depicted by the tool.

To achieve expandability, software classes declare objects as interfaces. Interfaces are based on an Object-Oriented Programming concept that allows us to specify the set of properties and methods that we require a particular class/object to implement. Note that this means that we don’t care about the internal implementations of those components, as long as they are implemented by the developer. However, not every component hosted by the parent class is of type interface. Some of the standard components such as mailbox and router are provided by the infrastructure provider. On the other hand, ASIPs, DMAMs and Stream Operators are declared as interfaces without specifying the actual implementation. This makes the framework flexible allowing developers to come up with their own implementation to any custom component.

4.2 Graph Representation of design topology

The logical representation of the composed system is a set of nodes and edges that represent FPGAs that are connected to each other in the MFS platform. This logical representation allows us to employ some of the traditional algorithms for finding the path routes between the nodes. For example, AutoFLEX uses Dijkstra’s algorithm to generate a set of routing tables that can be used to configure the routers in the FPGAs at run time.

5 Results

AutoFLEX framework was used for accelerating applications [5, 6]. For instance, in [6], the framework was used for accelerating image regridding operations achieving up to 8X speedup when compared to Intel Xeon 2.33 GHz dual-core processor. Furthermore, in [6], the framework was utilized for running FFT accelerator, where it operated on a 2Kx2K source images with an execution time of 26.2 ms.

6 Conclusions

In this paper we introduced a software/hardware co-framework that automates the process of composing DSP applications in a multi-FPGA environment. The framework simplifies the design composition by offering a friendly GUI interface that is easy to use and can address the requirements of the user. As for the future work, this framework can be expanded further by supporting more MFS platforms and other types of high-speed interfaces (e.g. PCIe).

7 Acknowledgment

The authors would like to thank Chaitali Chakrabarti, Chi-Li Yu and Srinidhi Kestur for conducting experiments using the AutoFLEX framework. This work was supported in part by NSF Awards 07202617 and 0916887.

![Figure 8: Modular representation of the AutoFLEX infrastructure. Note that some of the components such as ASIPs and DMAMs are declared as interfaces for expandability purposes. Also, IDMAM are further composed of ISOP (i.e. Stream Operators declared as interfaces). ISOPs are not shown in figure.](image-url)
8 References


