QuickDough: A Rapid FPGA Accelerator Design Method Using Soft Coarse-Grained Reconfigurable Array Overlay

Michael Shell, *Member, IEEE,* John Doe, *Fellow, OSA,* and Jane Doe, *Life Fellow, IEEE*

***Abstract*—FPGA accelerators used to offload compute intensive tasks of CPU offer both low power operation and great performance potential. However, they usually take much longer time to develop and are difficult to reuse, especially compared to the corresponding software design. Although the use of high-level synthesis (HLS) tools may partly alleviate this shortcoming, the lengthy low-level FPGA implementation process remains a major obstacle that limits the design productivity. To overcome this challenge, QuickDough, a rapid FPGA accelerator design method which utilizes soft coarse-grained reconfigurable arrays (SCGRAs) as an overlay on top of FPGA, is presented. Instead of compiling high-level applications directly as circuits implemented on the FPGA, the compilation process is reduced to an operation scheduling task targeting the SCGRA. Furthermore, the softness of the SCGRA allows domain-specific design of the processing elements, while allowing highly optimized SCGRA array be developed by a separate hardware design team. When compared to commercial high-level synthesis tools, QuickDough achieves competitive speedup in the end-to-end run time and reduces the compilation time by two orders of magnitude.**

***Index Terms*—Overlay, FPGA Accelerator, Soft Coarse Grain Reconfigurable Array, Design Productivity**

I. Introduction

The use of FPGAs as compute accelerators has been demonstrated by numerous researchers as an effective solution to meet the performance requirement across many application domains. However, despite years of research with numerous successful demonstrations, the use of FPGAs as computing devices remains a niche discipline that has yet to receive widespread adoption beyond highly skilled hardware engineers. Compared to a typical software development environment, developing an accelerator on FPGA is a lengthy and tedious process. While advancements in high level synthesis (HLS) tools have helped lower the barrier-to-entry for novel users, the irregularity of the synthesized circuit makes it difficult to compromise between execution time in cycles and implementation frequency. Furthermore, unless the HLS tool can directly synthesize the target FPGA configuration, vendor's back-end implementation tools must be employed for tasks such as floor planning and placing-and-routing. Compared to software compilation, the run-time of such back-end implementation tools is at least 2 to 3 orders of magnitude slower, hindering the productivity of the designers especially during early development phases.

M. Shell is with the Department of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta,GA, 30332 USA e-mail: (see http://www.michaelshell.org/contact.html).

J. Doe and J. Doe are with Anonymous University.

Manuscript received April 19, 2005; revised January 11, 2007.

To address the above challenges, QuickDough, a rapid FPGA accelerator design method that utilizes SCGRA as an overlay, is proposed to produce FPGA bitstream directly from applications written in high-level languages such as C/C++. SCGRA has quite regular hardware structure and scales well on both the implementation frequency and execution time in cycles for parallel compute intensive kernels. Meanwhile, instead of being synthesized to circuits on the FPGAs, application compute kernels are translated to data flow graphs (DFGs) and further be *statically* scheduled to operate on the SCGRA. The lengthy hardware implementation tool flow is thus reduced to an operation scheduling problem.

Although the design and implementation of the SCRGA based accelerator must rely on the conventional hardware design flow, only one instance of the SCGRA design is required per application or application domain. Subsequent application development may then be accomplished rapidly by executing a simple scheduling algorithm. In addition, the performance of the SCGRA can be carefully optimized by a separate experienced hardware engineering team. Since the physical implementation of the SCGRA remains unchanged across applications or design iterations, the physical performance of the design can be guaranteed. According to the experiments on Zedboard, SCGRA based FPGA accelerator achieves competitive performance compared to Vivado HLS based accelerator under limited on chip buffer. While the SCGRA does consume more hardware overhead especially the BRAM blocks.

In Section III, the proposed FPGA acceleration design method QuickDough will be elaborated. The SCGRA implementation and compilation will be presented in Section V and Section IV respectively. Experimental results are shown in Section VI. Finally, we will discuss the limitations of current implementation in Section VII and conclude in Section VIII.

II. Related Work

To improve the productivity of FPGA designers, researchers have approached the problem both by increasing the abstraction level and reducing the compilation time.

In the first case, decades of research in FPGA high-level synthesis have already demonstrated their indispensible role in promoting FPGA design productivity [8]. Numerous design languages and environments [7] have been developed to allow designers to focus on high-level functionality instead of low-level implementation details. While high-level abstraction may help a designers express the desired functionality, the low-level compilation time spent on synthesis, mapping, placing and routing for FPGAs remains a major hindrances to designs' productivity. Researchers have approached the problem from many angles, such as through the use of pre-compiled hard macros [17] in the tool flow, the use of a partial reconfiguration, and modular design flow [11].

On top of the above approaches, overlays, which can be parametric HDL Model, pre-synthesized or pre-implemented coarse-grained reconfigurable circuits over the fine-grained FPGA devices, promise both to raise the abstraction level and reduce the compilation time. Thus great research efforts have been attracted over the years and a number of overlays have been proposed [14, 15, 18, 19, 22, 24]. The granularity of these overlays ranges from multi-processors to highly configurable logic arrays.

Soft processors, which allow customization from various angles for target applications or application domains, have already been demonstrated to be efficient overlays for implementing an application on FPGA. [23],[2], and [3] employ general processors as the overlay and mainly have micro-architecture parameters such as pipeline depth configurable. [12] uses general processor with custom instruction extension as overlay, but hardware implementation is required whenever new custom instructions are added. [16] develops a fine-grain virtual FPGA overlay specially for custom instruction extension, which makes the custom instruction implementation portable and fast. [18] has customized data path on a many-core overlay, it could support both the coarse-grain multi-thread parallelism and data-flow style fine-grain threading parallelism. [22] adopts a multi-processor overlay with both micro-architecture and interconnection customizable. [19] and [24] develop reconfigurable vector processors as the FPGA overlay to cover larger domains of applications. [14] presented a GPU-Like overlay for portability and it could explore both the data-level parallelism and thread level parallelism. These processor level overlays typically approach the customization through instruction set extension or micro-architecture parameters tuning, and the application developers don't need much interaction with the low level hardware customization. Thus an application can be implemented rapidly, while the penalty is the hardware overhead and implementation frequency.

[5] and [13] build fine-grain components and mixed-grain components as a virtual FPGA overlay over the off-the-shelf FPGA devices. The virtual FPGAs allow the designers to reuse the virtual bitstream which is compatible across different FPGA vendors and parts. Particularly, the virtual FPGA with coarse granularity of components could also decrease the compilation time. [9] developed a family of intermediate fabrics which fits well for data parallel circuit implementation and the compilation time is almost comparable to software compilation. Apparently, the virtual FPGA overlays are beneficial to improving the design productivity and portability, though they do result in moderate hardware overhead and timing degradation.

Between the processor level overlays and virtual FPGA level overlays, CGRA overlays on FPGA have unique advantages of compromising hardware implementation and performance especially for compute intensive applications as demonstrated by numerous ASIC CGRAs **??** **??**. CGRAs on FPGA and ASIC have many similarities in terms of the scheduling algorithm and array structure, however, they have quite different trade-off on configuration flexibility, overhead and performance. Basically, CGRAs on ASIC need to emphasize more on configuration capability to cover more applications, while FPGAs' inherent programmability greatly alleviate this concern. Accordingly, CGRAs on FPGA could accept more intensive customization while design productivity comes up as a new challenge.

[10] proposed an heterogeneous CGRA overlay with multi-stage interconnection on FPGA, and the compilation can be done in milliseconds. While the CGRA size is quite limited and the implementation frequency is low due to the multi-stage interconnection. [20] [6] employ coarse-grained reconfigurable array (cgra) as overlays, these overlays typically take data flow graph as input and the compilation doesn't involve any circuit optimization such as timing and piplining at all. particularly, [6] shows that the cgra overlay could run at high frequency, which is different from the virtual fpga overlay. these overlays inherit the advantages of traditional asic cgras [21] in terms of exploring great parallelism as of the applications, preserve comparable implementation frequency and have more opportunities for cutomization thanks to the flexible fpga, therefore these cgra overlays present signigicant performance speedup, acceptable hardware overhead and extremely fast compilation. the benchmarks used are usually extracted from application kerenls and range from a few to dozens of nodes. and the evaluation is performed on a pure cgra overlay. nevertheless, using the cgra overlay as a fpga accelerator targeting a full application on a real system like a general processor (gpp) + fpga is still missing.

Building on top of many the above ideas, we have opted to utilize a fully pipelined synchronous SCGRA as the overlay. Then we further implement it as an accelerator on Zedboard which is a hybrid ARM + FPGA system. The acceleratoin system is configurable and is capable to handle all of our four full compute intensive applications with diverse data sets. With this SCGRA overlay based acceleration system, an application can be implemented in a short time and the performance is competitive.

III. QuickDough Framework

A. System Context

This work assumes a hybrid computing architecture with a host processor and a FPGA accelerator, where the processor handles tasks not-well suited to FPGAs such as providing the OS environment and the end user GUI and FPGA focuses on computation intensive kernels.

Figure 1 shows a typical FPGA acceleration architecture and all the experiments in this work sticked to it. In this system, FPGA accelerator is attached to the system bus and it could access main memory through the bus. Inner the FPGA accelerator, there are a group of data buffers, an acceleration control block(Acc Ctrl), and the computation core. Data buffers are employed to store input data, output data and even temporary data of the computation core. The Acc Ctrl block receives computation start signal from CPU and then triggers the computation core when the input data is ready. Also it sends computation done signal back to interrupt CPU for data collection when the computation core completes. The computation core is a hardware structure customized for the application compute kernel, which is supposed to be fast.

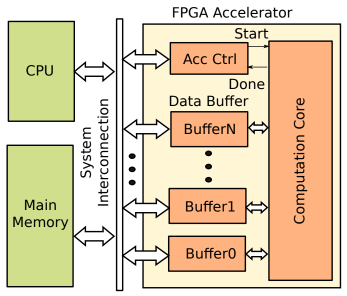

Fig. 1. A Typical FPGA Acceleration Architecture

B. QuickDough

Based on the typical FPGA acceleration architecture, a rapid FPGA acceleration framework is developed as presented in Figure 2. Basically, it starts from HW/SW partition so that the application compute kernel can be extracted. The extracted kernel can further be transformed to data flow graph (DFG), which is preferred to be used in CGRA compilation. There are already intensive research on both, which could help us to automate the processing. Currently, we just manually perform the HW/SW partition and DFG transformation in our preliminary research stage.

Afterwards, the design flow is split into two paths. The path on top half is basically a conventional software compilation flow except that we need to replace the compute kernel with CPU-FPGA accelerator communication drivers to manage the FPGA accelerator as an IO device.

The path on the bottom half is esentially the hardware design flow. Given the application compute kernel, the SCGRA optimizer is supposed to decide the optimal DFG size and corresponding SCGRA configuration. However, we haven't got a systematic solution for the optimizer yet and we will just choose two SCGRA configuration as an example to show the benefits of the SCGRA based acceleration framework. After this step, if the required SCGRA configuration happens to be in the SCGRA library, then the SCGRA compiler could generate the FPGA bitstream directly. If not, corresponding SCGRA HDL code will be generated automatically based on the pre-defined template, and the HDL model is further implemented on the target FPGA device. Then the implementation result is added to the SCGRA library. Finally, SCGRA compiler will handle the scheduling and bitstream generation. Through the design iterations of an application or even across a whole application domain, it is possible that a single hardware implementation meets the design goal. While SCGRA compilation can be done around a minute, the design productivity can be improved significantly.

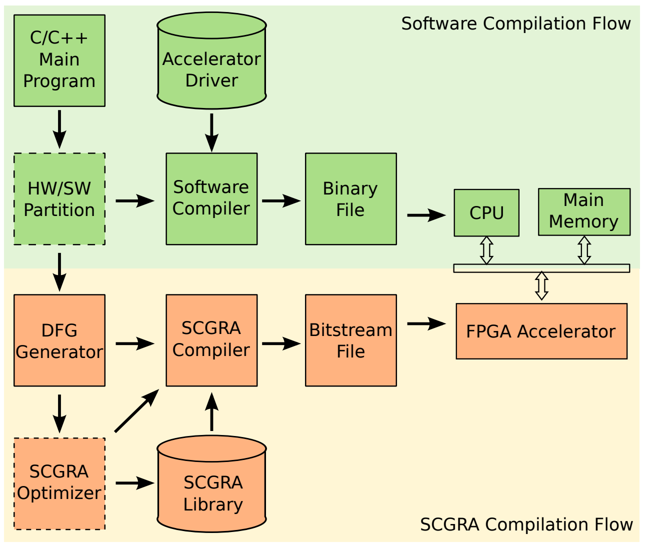

Fig. 2. QuickDough Framework

IV. SCGRA Overlay Infrastructure

One key idea of QuickDough is to rely on an intermediate SCGRA overlay to improve compilation time of the high-level user application. While the exact design of this SCGRA does not affect the compilation flow, its implementation does have a significant impact on the performance of the generated gateware.

A. SCGRA Based FPGA Accelerator

Figure 3 is the proposed FPGA accelerator built on top of SCGRA. Its communication interface with the host system follows exactly the typical FPGA acceleration system, while the computation core is a synchronous SCGRA which is esentially an array of homogeneous PEs. The topology of the SCGRA is one of the major design parameters. High dimension topology is preferred for kernels with heavy communication, while low dimension topolpgy is more scalable and can be implemented more efficiently. Currently, 2D Torus with both comparable scalability and communication bandwidth is used in this work.

On top of the computation array, two address buffers instead of customized logic are provided as the address generator. When the high level application changes within the SCGRA's supporting set, we don't have to develop a new address generator for it and can simply replace its content together with the SCGRA configuration contexte. Therfore, it helps reduce the chance of FPGA implementation and eventually improves the design productivity of the FPGA acceleration framwork.

While usually the input and output data buffer could store more data than that used by a single SCGRA execution, thus the SCGRA may iterate multiple times before it consumes all the data in input buffer or fills the output buffer. From the perspective of software, it is more convenient to control all the computation involved in the data buffer instead of each SCGRA execution. In this case, we have a control unit called SCGRA Ctrl to make the multiple SCGRA execution transparent to the ACC Ctrl. Now, there are only two single bit signals between the ACC Ctrl and the rest of the accelerator.

Then two consecutive registers are applied to separate the clock domain of communication interface and the computation core. The optimized SCGRA could be easily pipelined and work at higher frequency, while the interface needs to handle complex protocol and works at lower frequency.

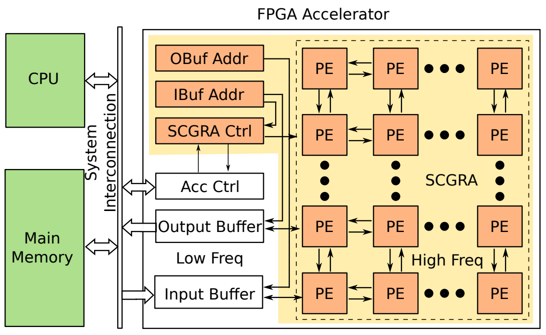

Fig. 3. SCGRA Accelerator

B. PE

In this section, an instance of PE is presented to demonstrate the feasibility of producing high performance gateware. As shown in Figure 4, the PE, centring an ALU block, a multiple-port data memory and an instruction ROM, is highly optimized for FPGA implementation. In addition, a load/store path is implemented on the PEs that are responsible for data I/O beyond the FPGA and an Addr Ctrl is employed to start as well reset the SCGRA execution.

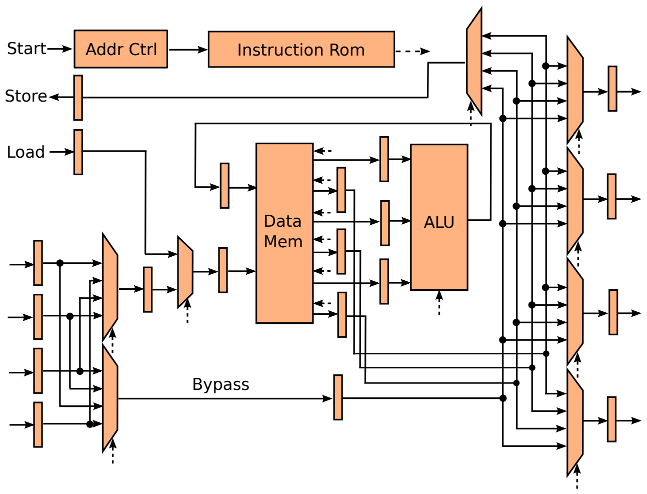

Fig. 4. PE structure

1) Instruction Memory and Data Memory: The instruction memory stores all the control words of the PE in each cycle. Since its content does not change during runtime, a ROM is used to implement this instruction memory. The content of the ROM is loaded together with the configuration bitstream. The address of the instruction ROM is determined by the Addr Ctrl. Basically, the SCGRA execution will proceed when the start signal is valid, and it will be reset when the start signal is invalid.

Data memory stores intermediate data that can either be forwarded to the PE downstream or be sent to the ALU for calculation. For fully parallelized operation, *at least* four read ports are needed – three for the ALU and one for data forwarding. Similarly, at least two write ports are needed to store input data from upstream memory and to store the result of the ALU in the same cycle. Although a pair of true dual port memories may seems to be able to satisfy this port requirement, conflicts may arise if the ALU needs to read the data while the data path needs to be written. As a result, a third dual port memory is replicated in the data memory.

Note that data memory here is usually implemented as a multiple port register file in many previous CGRA work. Although register file is even more flexible in terms of parallel reading and writing, the multi-port register file size is limited due to the inefficient hardware implementation. While we have a much larger DFG for scheduling and thus larger temporary storage is required, we eventually use a multi-port data memory instead.

2) ALU: At the heart of the proposed PE is an ALU designed to cover the computations in target application. Figure 5 shows an example design that could support all the operations of our benchmark which is listed in Table I. Complpex operations like MULADD/MULSUB are implemented with DSP core directly. Operations with moderate complexity like ADDADD, RSFAND etc. are divided into two stages naturally and hardware block reuse is also considered at the same time. Finally, simple operations that can be done in a single cycle can be put in either stage depending on the pipeline status. Note that MUX in data path has significant influence on timing as well, so small MUX should be inserted properly and large MUX should be avoided. Currently, we just manually optimize the ALU design, but it is possible to automate this step.

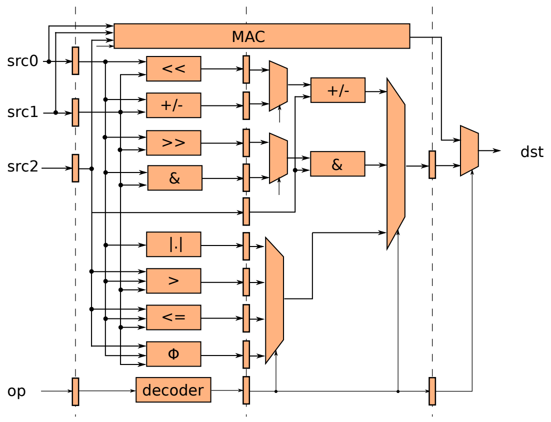

Fig. 5. ALU Example

C. Load/Store Interface

For the PEs that also serve as I/O interface to the SCGRA, there are also additional load path and store path as shown in 4. To keep the balance of the pipeline, an additional pipeline stage is added to chosse between neighboring input and SCGRA input interface. While store path has no influence on the rest of the design. In addition, as load path and store

TABLE I
OPERATIONS INVOLVED IN THE BENCHMARK

Type	Opcode	Expression
MULADD	0001	dst = src0 $times$ src1 + src2
MULSUB	0010	dst = src0 $times$ src1 - src2
ADDADD	0011	dst = src0 + src1 + src2
ADDSUB	0100	dst = src0 + src1 - src2
SUBSUB	0101	dst = src0 - src1 -src2
PHI	0110	dst = src0 ? src1 : src2
RSFAND	0111	dst = (src0 $>>$ src1) & src2
LSFADD	1000	dst = (src0 $<<$ src1) + src2
ABS	1001	dst = abs(src0)
GT	1010	dst = (src0 $>$ src1) ? 1 : 0
LET	1011	dst = (src0 $<=$ src1) ? 1 : 0
ANDAND	1100	dst = src0 & src1 & src2

path can be connected to the read port and write port of the I/O buffer at the same time, there is no further customization for input PE or output PE.

V. SCGRA COMPILATION

Figure 6 depicts the idel SCGRA compilation flow in QuickDough. As shown in the diagram, the compilation esentially is supposed to compile an application compute kernel to an automaticaly customized SCGRA and generate the final bitstream.

The compilation starts from DFG generation transforming a specified compute kernel probably written in high level language to a DFG. When the DFG is determined, a proper SCGRA configuration is derived through a SCGRA optimizer compromising between the hardware constrain and performance. With both the DFG and SCGRA configuration specified, an operation scheduling algorithm is employed to map the DFG to this SCGRA. Meanwhile, the performance of the DFG running on the SCGRA can be acquired. With these information, it is easy to check whether the design goal is achieved. Usually, it may take multiple iterations to determine an optimized solution. Once the SCGRA configuration is determined, configuration context can also be extracted from the scheduling result. Also we will search the SCGRA library, where there are implementations of various SCGRA configurations. If the implementation of the specified SCGRA conbfiguration is found, then we can integrate the configuration context into the pre-implemented bitstream. If not, corresponding SCGRA HDL model should be generated and implemented before the final bitstream integration. Although it may take a long time to implementat the HDL model, the implementation process will be performed only once as long as the SCGRA configuration is decided.

As mentioned in previous sections, we have not got all the components in Figure 6 developed. Currently, we just manually generate the DFG of the compute kernel, and randomly specify two representative SCGRA configuration for all of our benchmark. Figure 7 shows the baseline compilation flow, which is the backbone of the ideal compilation flow. Details of the baseline compilation flow is further described in the following sections.

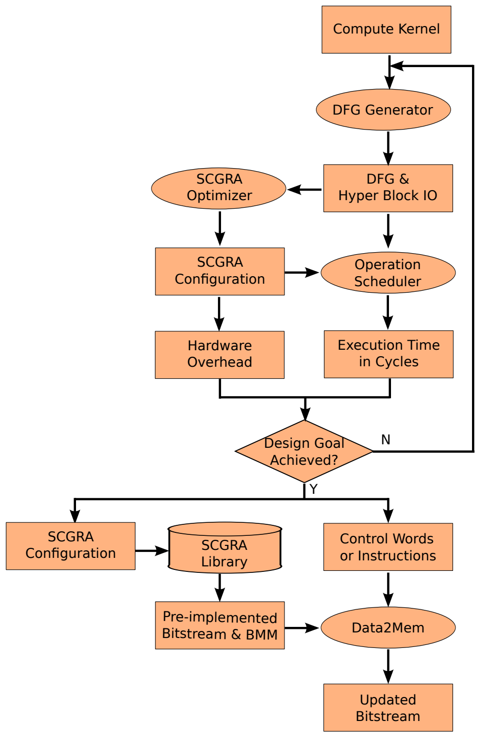

Fig. 6. Ideal SCGRA Compilation Flow

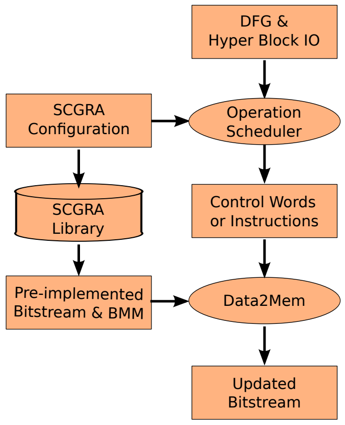

Fig. 7. Baseline SCGRA Compilation Flow

:

A. DFG Generation

To assist the DFG generation, we developed a group of classes to ease the DFG generation procedurs such as declaring an array of operands and operations, and automatically DFG verification. Eventually, the DFG is represented by three files including operand file, opcode file, and instruction file. The operand file stores details of each operand involved in the DFG such as logical ID of the input/output and data width. The opcode file just defines the meaning of each opcode. The instruction file stores all the operations of the DFG sequentially and each operation is represented as an instruction with destination operand ID, opcode, and source operand ID. Since the accelerator may have data set of multiple SCGRA execution stored in the input/output memory-mapped buffer and data set of differnt SCGRA execution may overlap, we have another io-mapper file to record the IO locations of each SCGRA execution.

B. Operation Scheduling

In the SCGRA scheduling step, a list scheduling algorithm similar to [25] is adopted to schedule the DFG to the SCGRA infrastructure. It statically schedules the entire DFG on the target SCGRA. Such statically scheduled architecture is crucial in keeping the target SCGRA simple and efficient. To adapt to the proposed SCGRA structure, the scheduling metric is delicately adjusted to compromise the communication cost and load balance. On top of the DFG scheduling, it should also be responsible for generating the addresses for each SCGRA execution using the IO-mapper file.

C. Bitstream Integration

The final step of the compilation flow is to incorporate the instruction for each PE obtained from the scheduling stage with the pre-compiled SCGRA bitstream. By design, our SCGRA does not have mechanism to load instruction streams from external memory. Instead, we take advantage of the reconfigurability of SRAM based FPGAs and stored the cycle-by-cycle configuration words using on-chip ROM. The content of these instruction ROMs are embedded in the configuration bitstream. In particular, the organization of the instruction ROM in the place-and-routed SCGRA design is obtained from its XDL file [4], which in turn allows us to create the corresponding BMM file. With this BMM file, the encoded instructions collected from the DFG scheduling may then be incorporated into the pre-implemented bitstream using the data2mem tool from Xilinx [1]. While original SCGRA design needs hours to implement, the bitstream updating scheme only costs a few seconds.

VI. Experiments

In this work, we take four applications including matrix multiplication (MM), FIR filter, Kmean and Sobel edge detector as our benchmark. In order to be more representative, each application is further provided with three different data sets ranging from small, medium to large ones. The basic parameters and configurations of the benchmark is explained in the Table II.

The benchmark is implemented on Zedboard using both Vivado HLS and QuickDough to build the FPGA acceleration system. Then design productivity, implementation efficiency and performance of the two design methods are compared respectively.

A. Experiment Setup

All the runtimes were obtained from a computer with Intel(R) Core(TM) i5-3230M CPU and 8GB RAM. Vivado HLS 2013.3 is used to transform the compute kernel to hardware IP Catalog. Vivado 2013.3 is then used to integrate the IP core and build the FPGA accelertor on Zedboard. The SCGRA is initially built using ISE 14.7 and then exported as an IP core. Afterwards, it is integrated into the SCGRA based FPGA accelerator using PlanAhead 14.7. Finally, the accelerators are implemented on Zedboard to run the benchmark on bare-metal system mode.

Vivado HLS typically achieves the trade-off between hardware overhead and performance through altering the loop unrolling factor and loop blocking size. Larger block size is beneficial to data reuse and amortizing the communication cost, but it is usually constrained by the size of on-chip buffer. Similarly, larger loop unrolling factor typically promises better performance, however, hardware resources like DSP blocks will soon be used up and implementation frequency will degrade as well. In this work, two different sets of input/output buffer configurations are applied for the Vivado HLS based acceleration design. One set has fixed 2k-word buffer for both input and output, and the other set has maximum available buffer size and it could use as much as 64k-word for both input and output. Table III shows the near optimal blocking and loop unrolling setup for the benchmark.

SCGRA based FPGA accelerator has similar design chocies to that of Vivado HLS in terms of loop unrolling and blocking. The difference is that the unrolled part of the loop is transformed to DFG which is further scheduled to the SCGRA infrastructure, and the loop unrolling factor is limited by the resources of the SCGRA overlay such as SCGRA size, instruction memory size and data memory size. While the block size is contrained by both the size of input/output buffer and the addr buffer. Currently, we set the data buffer to be 2k-word and addr buffer to be 4k-word. However, addr buffer size is the major constrain of the blocking size as useless addresses are also stored. In this work, both a 2x2 SCGRA and 5x5 SCGRA overlay are used for all these benchmark implementation. Table IV and Table V present the detailed SCGRA overlay configuration and loop unrolling as well as blocking of the benchmark respectively.

B. Experiment Results

In this section, design productivity, hardware implementation efficiency and performance using both design methodologies are presented.

1) Design Productivity: Since design productivity involves many different aspects such as the abstarction level of the design entry, compilation time, design reuse, and design portability, it is difficult to evaluate all aspects especially some of the aspects are not easy to be quantized. In this section, we mainly look at the compilation time and design reuse which are critical angles of design productivity.

In order to accelerate an application using FPGA, Vivado HLS based design flow mainly consists of four steps including compute kernel synthesis, kernel IP generation, overall system implementation and software compilation. At the compute kernel synthesis step, high level language program kernel is translated to a HDL model according to the user's synthesis pragma. Then the HDL model is further synthesized and packed to an IP core, which should meet the timing constrain and have the corresponding driver packed, at the kernel IP generation step. Afterwards, it is the system implementation

TABLE II
DETAILED CONFIGURATIONS OF THE BENCHMARK

Benchmark	MM	FIR	Sobel	Kmean
Parameters	Matrix Size	# of Input # of Taps+1	# of Vertical Pixels # of Horizontal Pixels	# of Nodes # of Centroids Dimension Size
Small	10	40/50	8/8	20/4/2
Medium	100	10000/50	128/128	5000/4/2
Large	1000	100000/50	1024/1024	50000/4/2

TABLE III
LOOP UNROLLING & BLOCKING SETUP FOR VIVADO HLS BASED ACCELERATOR DESIGN

Application		Max-Buffer		2k-Buffer		Complete Loop Structure
		Unrolling Factor	Block Structure	Unrolling Factor	Block Structure	
MM	Small	$2 \times 10 \times 10$	$10 \times 10 \times 10$	$2 \times 10 \times 10$	$10 \times 10 \times 10$	$10 \times 10 \times 10$
	Medium	$1 \times 1 \times 100$	$100 \times 100 \times 100$	1×100	10×100	$100 \times 100 \times 100$
	Large	1×500	50×1000	500	1000	$1000 \times 1000 \times 1000$
FIR	Small	2×50	40×50	2×50	40×50	40×50
	Medium	2×50	10000×50	2×50	1000×50	10000×50
	Large	2×50	50000×50	2×50	1000×50	100000×50
Sobel	Small	$1 \times 2 \times 3 \times 3$	$8 \times 8 \times 3 \times 3$	$1 \times 2 \times 3 \times 3$	$8 \times 8 \times 3 \times 3$	$8 \times 8 \times 3 \times 3$
	Medium	$1 \times 1 \times 3 \times 3$	$128 \times 128 \times 3 \times 3$	$1 \times 1 \times 3 \times 3$	$23 \times 128 \times 3 \times 3$	$128 \times 128 \times 3 \times 3$
	Large	$1 \times 1 \times 3 \times 3$	$75 \times 1024 \times 3 \times 3$	$1 \times 1 \times 3 \times 3$	$4 \times 1024 \times 3 \times 3$	$1024 \times 1024 \times 3 \times 3$
KMean	Small	$20 \times 4 \times 2$	$20 \times 4 \times 2$	$20 \times 4 \times 2$	$20 \times 4 \times 2$	$20 \times 4 \times 2$
	Medium	$5 \times 4 \times 2$	$5000 \times 4 \times 2$	$5 \times 4 \times 2$	$1000 \times 4 \times 2$	$1000 \times 4 \times 2$
	Large	$5 \times 4 \times 2$	$25000 \times 4 \times 2$	$5 \times 4 \times 2$	$1000 \times 4 \times 2$	$1000 \times 4 \times 2$

TABLE IV
SCGRA CONFIGURATION

SCGRA Topology	Torus
SCGRA Size	2×2, 5×5
Data Wdith	32 bits
Instruction Length	72 bits
Instruction Memory Depth	1024
Data Memory Depth	256
Input/Output Buffer Depth	2048
Addr Buffer Depth	4096

step, where the IP core is integrated into the target system and the accelerator is implemented on FPGA. Finally, the application using the hardware accelerator is compiled to binaray code. The SCGRA based FPGA accelerator design flow also needs four steps to complete the acceleration. The first step is translating the high level language program kernel to DFG, and the DFG is scheduled to a spcified SCGRA overlay at the second step. At the third step, the bitstream is generated and corresponding driver is prepared. The last softeware compilation step is similar to that of the Vivado HLS based design flow.

Figure 8 and Figure 9 shows the compilation time needed to implement the benchmark using both Vivado HLS based design flow and the SCGRA based design flow respectively. Vivado HLS based design method needs lenghty IP core generation and hardware implementation for each application instance, which are the major time consuming processes. Compute kernel synthesis is usually as fast as the software compilation and can be done in a few seconds, but it may take up to 10 minutes when there is a pieplined large loop unrolling. Eventually, it takes the whole design flow 20 minutes to an hour to complete an implementation. With pre-implemented SCGRA overlay, the processing steps except the DFG scheduling of the SCGRA based design flow are quite fast and don't change much across the different applications. DFG scheduling is relatively slower especially when the DFG size and CGRA size are large, but it still can be done in a few seconds. Finally, the SCGRA based design method could produce the bitstream in 5 to 15 seconds and it is typically two magnitude faster than the SCGRA overlay based design method.

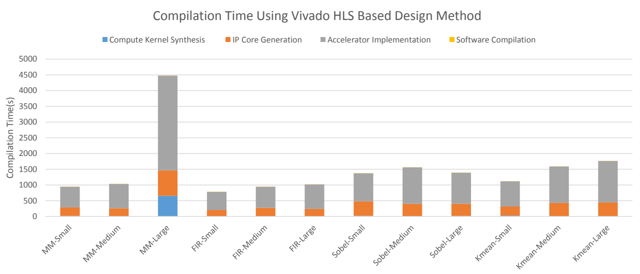

Fig. 8. Compilation Time Using Vivado HLS Based Design Method

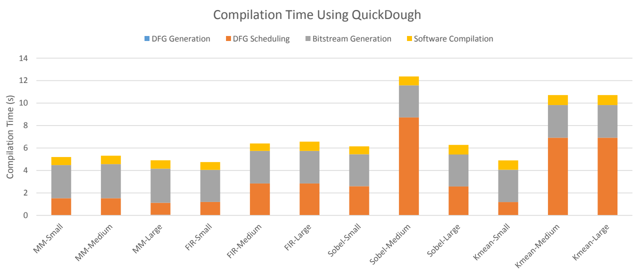

Fig. 9. Compilation Time Using SCGRA Overlay Based Design Method

2) *Hardware Implementation Efficiency:* In this section, hardware implementation efficiency including the hardware resource overhead, implementation frequency and implemen-

TABLE V
LOOP UNROLLING AND BLOCKING SETUP FOR SCGRA BASED FPGA ACCELERATOR DESIGN

Application		SCGRA 2x2			SCGRA 5x5			Complete Loop Structure
		Unrolling Factor	DFG(OP/IO)	Block Structure	Unrolling Factor	DFG(OP/IO)	Block Structure	
MM	Small	$10 \times 10 \times 10$	1000/301	$10 \times 10 \times 10$	$10 \times 10 \times 10$	1000/301	$10 \times 10 \times 10$	$10 \times 10 \times 10$
	Medium	5×100	750/606	10×100	5×100	750/606	10×100	$100 \times 100 \times 100$
	Large	200	301/402	1000	200	301/402	10×100	$1000 \times 1000 \times 1000$
FIR	Small	40×50	860/131	40×50	40×50	860/131	40×50	40×50
	Medium	20×50	1000/141	100×50	50×50	2500/201	250×50	$10^4 \times 50$
	Large	20×50	1000/141	100×50	50×50	2500/201	250×50	$10^5 \times 50$
Sobel	Small	$4 \times 8 \times 3 \times 3$	1080/39	$8 \times 8 \times 3 \times 3$	$8 \times 8 \times 3 \times 3$	2160/55	$8 \times 8 \times 3 \times 3$	$8 \times 8 \times 3 \times 3$
	Medium	$4 \times 8 \times 3 \times 3$	1080/39	$8 \times 8 \times 3 \times 3$	$23 \times 8 \times 3 \times 3$	6210/115	$65 \times 8 \times 3 \times 3$	$128 \times 128 \times 3 \times 3$
	Large	$4 \times 4 \times 3 \times 3$	540/31	$16 \times 4 \times 3 \times 3$	$16 \times 4 \times 3 \times 3$	2160/55	$16 \times 4 \times 3 \times 3$	$1024 \times 1024 \times 3 \times 3$
KMean	Small	$20 \times 4 \times 2$	920/62	$20 \times 4 \times 2$	$20 \times 4 \times 2$	920/62	$20 \times 4 \times 2$	$20 \times 4 \times 2$
	Medium	$25 \times 4 \times 2$	1144/72	$125 \times 4 \times 2$	$125 \times 4 \times 2$	5768/272	$500 \times 4 \times 2$	$5000 \times 4 \times 2$
	Large	$25 \times 4 \times 2$	1144/72	$125 \times 4 \times 2$	$125 \times 4 \times 2$	5768/272	$500 \times 4 \times 2$	$50000 \times 4 \times 2$

tation scalability using the two accelerator design methods are compared.

Table VI exhibits the hardware overhead using both accelerator design methods. It is clear that Vivado HLS based accelrator typically consumes much less FF, LUT and RAM36 due to the delicate customization for each application configuration. However, the number of DSP48 required increases dramatically with the expansion of the application kernel which is mostly induced by loop unrolling. As a result, DSP48 can be the bottleneck that constrains the performance of the accelerator through loop unrolling. SCGRA based accelerator usually costs more FF, LUT and particularly RAM36 as expected, while the DSP consumtion scales linearly with the SCGRA size. Note that the hardware overhead using Vivado HLS based design method in this table refers to the configuration with fixed 2k-word input/output buffer. The overhead with maximum input/output buffer configuration doesn't change the conclusion here and is skipped to save the space.

TABLE VI
HARDWARE OVERHEAD USING BOTH VIVADO HLS AND THE SCGRA BASED ACCELERATOR DESIGN METHODS

		FF	LUT	RAM36	DSP48
MM	Small	5099	3730	4	102
	Medium	2778	3217	4	9
	Large	8282	10228	4	9
FIR	Small	5287	4494	4	99
	Medium	5283	4577	4	99
	Large	5283	4577	4	99
Sobel	Small	8503	6436	4	216
	Medium	6737	5526	4	144
	Large	6755	5546	4	144
KMean	Small	2852	3762	4	24
	Medium	5530	5427	4	24
	Large	5530	5427	4	24
SCGRA 2x2		9302	5745	32	4
SCGRA 5x5		34922	21436	137	25
FPGA Resource		106400	53200	140	220

Figure 10 shows detailed the implementation frequency of the benchmark using both FPGA accelerator design methods. Vivado HLS based accelerator works at most 100MHz on the Zedboard system due to the slow AXI controller. Even though the accelerator may work at individual clock domain, the synthesized IP core can also be slow especially when relatively larger loop unrolling is employed at high level language program synthesis step. While the SCGRA based is regular and it could works at higher clock domain. To make use of this advantage, the slow AXI controller block and the SCGRA are divided into two separate clock domains. Currently, SCGRA 2x2 works at 200MHz while SCGRA 5x5 works at 167MHz due to the high utilization of BRAM blocks. The controlling logic is slow and works at 100MHz.

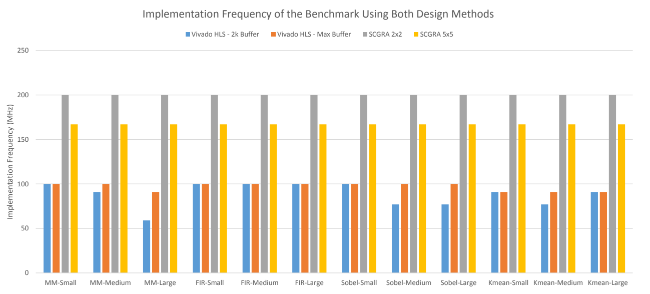

Fig. 10. Implementation Frequency of The Accelerators Using Vivado HLS and QuickDough

When the application data set gets larger, both the accelerator design mathods may further scale the hardware infrastructure through loop unrolling and extending the SCGRA size etc. to accommodate the increased parallelism. Therefore, the scalability of the hardware implementation is also essential for the FPGA acceleration system. Figure xxx shows the implementation frequency of the accelerators kernel on xxx which could accommodate larger design than Zedboard. Since the implementation frequency of the Vivado HLS based accelerator depends on the specific application, we just use the simple matrix multiplication as an example.

3) Performance: In this section, the execution time of the benchmark is taken as the performance metric. Since the execution time of different applications and data sets varies a lot, the performance speedup relative to Vivado HLS based implementation with 2k-Buffer is used instead. Figure 11 shows the performance comparison of four different sets of accelerator implementations including two SCGRA overlay based accelerators and two Vivado HLS based accelerators. According to this figure, Vivado HLS based design method wins the MM-Medium, MM-Large, Sobel-Medium and Sobel-Large, while QuickDough outperforms in FIR with all three data sets, Kmean-Medium and Kmean-Large. The two design methods achieve similar performance on the rest of the benchmark.

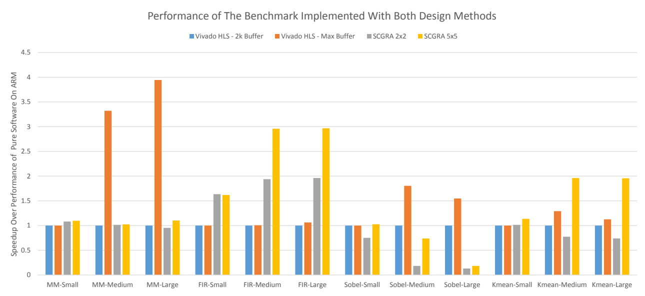

Fig. 11. Benchmark Performance Using Both Vivado HLS Based Design Method and The SCGRA Overlay Based Design Method

To insight in the comparison of the two design methods, Figure 12 presents the execution time decomposition of the benchmark. As the benchmark runs on an ARM + FPGA system where FPGA handles the compute kernel leaving the rest on ARM processor, the execution of the benchmark can be roughly divided into four parts: system initialization such as DMA initialization, communication time between FPGA and ARM processor moving input/output data to/from the FPGA on-chip buffers, FPGA computation and the others such as input/output data reorganization for DMA transmission or corner case processing. The system initialization time is relatively small and is about the same for both design methods. The rest three parts determine the performance of different implementations. In addition, note that the execution time of different applications with diverse data sets varies in a large range. To fit all the data in a single figure, the execution time used in this figure is actually normalized to that of a basic software implementation on ARM.

According to this figure, Vivado HLS based designs especially the designs with max-buffer configuration perform better performance mainly due to the smaller overhead in communication and others which are essentially the input/output data reorganization time. If we further look at the blocking details in Table III, it is clear that Vivado HLS based designs with max-buffer configuration could accommodate both larger data sets and corresponding computation, which could improve data reuse and deduce the amount of communication as well as data reorganization cost eventually. However, when there is little data reuse between the neighboring blocks, the design with larger data block couldn't reduce the amount of communication. Moreover, as the DMA is used to transmit the data between FPGA and Main Memory and it brings additional cost such as DMA setup, communication cost per data can be reduced when the data block per DMA transmission is larger. The benefit will be neglegible when the data set goes upto thousands and the additional DMA cost is amortized, while the exact numer depends on the platform. In all the applications of the benchmark with meidum and large data sets, Vivado HLS based designs especially the Max-Buffer configuration exihibit much smaller communication cost and data reorganization cost.

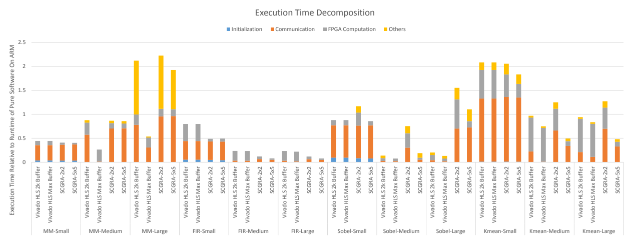

Fig. 12. Implementation Frequency of The Accelerators Using Vivado HLS and QuickDough

Computation time is one of the most significant part of the overal execution time. It basically depends on both the simulation performance in cycles and implementation frequency of the hardware infrastructure. As explained in previous sections, QuickDOugh using SCGRA overlay has regular structure and could provide scalable and higher frequency hardware infrastructure. Figure 13 shows the simulation performance which is the product of the block simulation performance and the number of blocks for each application. Apparently, QuickDough still fails to compete with the Vivado HLS based design in Sobel, but the performance gap of MM-Large is much smaller and particularly it shows better simulation performance on almost all the rest of the benchmark. The simulation performance comparison is quite relevant to the depth of the loop unrolling as shown in Table III and Table V. It indicates that the simulation performance depends heavily on the depth of the loop unrolling no matter HLS based customized circuit or SCGRA based overlay is used as the hardware infrastructure. While QuickDough using SCGRA overlay as the hardware infrastructure has advantages in terms of implementation freqeuncy, it enlarges its advatages in overall computation time as shown in Figure 14.

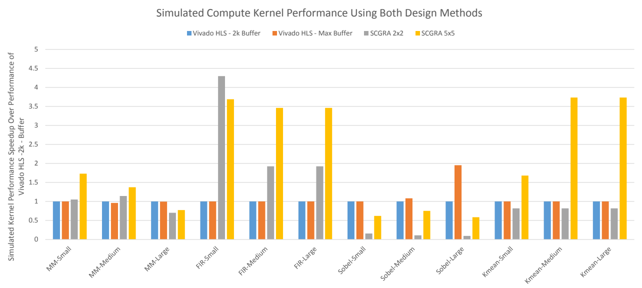

Fig. 13. Simulated Compute Kernel Performance Using Both Vivado HLS Based Design and QuickDough

In summary, Vivado HLS based design could afford larger buffer and accommodate larger block size, which further helps to reduce the communication time and the cost of input/output organization. Therefore, when there are more data reuse among neighboring blocks, the Vivado HLS based design is able to achieve better performance. QuickDough using SCGRA overlay could provide both higher simulation performance with larger loop unrolling in many cases and higher implementation frequency due to its regular structure, so it will win when the target application has smaller data set and more compute intensive kernels.

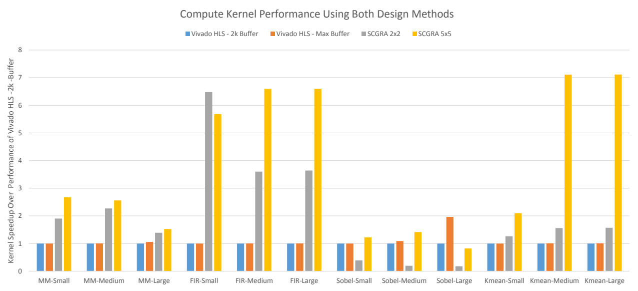

Fig. 14. Compute Kernel Performance Using Both Vivado HLS Based Design and QuickDough

VII. Limitations and Future Work

While the current implementation of QuickDough has demonstrated promising initial results, there are a number of limitations that must be acknowledged and possibly addressed in future work.

First and foremost, the proposed methodology is designed to synthesize parallel computing kernels to execute on FPGAs only. As such, it is not a generic methodology to perform HLS on random logic. Furthermore, the proposed method is intended to serve as part of a larger HW/SW synthesis framework that targets hybrid CPU-FPGA systems. Therefore, many high-level design decisions such as the identification of compute kernel to offload to FPGAs are not handled in this work. Also to guarantee the design productivity, a general front-end compilation that transforms high level language program kernel to DFG is still missing.

Currently, we just specify two SGCRA configurations for all the benchmark, while it is difficult for a high-level software designer to figure out an appropriate hardware configuration. An SCGRA optimizer will be developed to perform the SCGRA customization automatically in future.

Finally, the capacity of the address buffer used in QuickDough limits the block size that can be adapted to the FPGA in many cases. However, there are a large number of invalid address entries in it and this will be fixed in future.

VIII. Conclusions

In this paper, we have proposed QuickDough using a SCGRA overlay for compiling compute intensive applications to Zedboard. With the SCGRA overlay, the lengthy low-level implementation tool flow is reduced to a relatively rapid operation scheduling problem. The compilation time from an high level language application to the hybrid GPP+FPGA system is reduced by two magnitudes, which contributes directly into higher application designers' productivity.

Despite the use of an additional layer of SCGRA on the target FPGA, the overall application performance is not necessarily compromised. Implementation with higher clock frequency resulting from the highly regular structure of the SCGRA, in combination with an in-house scheduler that can effectively schedule operation to overlap with pipeline latencies provides competitive performance compared to a conventional HLS based design method.

References

[1] data2mem. http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/data2mem.pdf. [Online; accessed 19-September-2012].

[2] Microblaze soft processor core. http://www.xilinx.com/tools/microblaze.htm. [Online; accessed 25-June-2014].

[3] Nios embedded processor. http://www.altera.com/products/ip/processors/nios/nios-index.html. [Online; accessed 25-June-2014].

[4] C. Beckhoff, D. Koch, and J. Torresen. The Xilinx design language (XDL): Tutorial and use cases. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, pages 1–8. IEEE, 2011.

[5] A. Brant and G.G.F. Lemieux. Zuma: An open fpga overlay architecture. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 93–96, 2012.

[6] D. Capalija and T.S. Abdelrahman. A high-performance overlay architecture for pipelined execution of data flow graphs. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8, Sept 2013.

[7] J.M.P. Cardoso, P.C. Diniz, and M. Weinhardt. Compiling for reconfigurable computing: A survey. *ACM Computing Surveys (CSUR)*, 42(4):13, 2010.

[8] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, 2011.

[9] J. Coole and G. Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 13–22, Oct 2010.

[10] R. Ferreira, J.G. Vendramini, L. Mucida, M.M. Pereira, and L. Carro. An fpga-based heterogeneous coarse-grained dynamically reconfigurable architecture. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, pages 195–204. ACM, 2011.

[11] T. Frangieh, A. Chandrasekharan, S. Rajagopalan, Y. Iskander, S. Craven, and C. Patterson. PATIS: Using partial configuration to improve static FPGA design productivity. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –8, april 2010.

[12] Mariusz Grad and Christian Plessl. Woolcano: An architecture and tool flow for dynamic instruction set extension on xilinx virtex-4 fx. In *ERSA*, pages 319–322, 2009.

[13] David Grant, Chris Wang, and Guy G.F. Lemieux. A cad framework for malibu: An fpga with time-multiplexed coarse-grained elements. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 123–132,

New York, NY, USA, 2011. ACM.

[14] Jeffrey Kingyens and J. Gregory Steffan. The potential for a gpu-like overlay architecture for fpgas. *Int. J. Reconfig. Comp.*, 2011, 2011.

[15] D. KISSLER, F. HANNIG, A. KUPRIYANOV, and J. TEICH. A dynamically reconfigurable weakly programmable processor array architecture template. In *PROCEEDINGS OF THE 2ND INTERNATIONAL WORKSHOP ON RECONFIGURABLE COMMUNICATION CENTRIC SYSTEM-ON-CHIPS (RECOSOC), MONTPELLIER, FRANCE*, pages 31–37, 2006.

[16] D. Koch, C. Beckhoff, and G.G.F. Lemieux. An efficient fpga overlay for portable custom instruction set extensions. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8, Sept 2013.

[17] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. HMFlow: Accelerating FPGA compilation with hard macros for rapid prototyping. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 117 –124, may 2011.

[18] I. Lebedev, Shaoyi Cheng, A. Doupnik, J. Martin, C. Fletcher, D. Burke, Mingjie Lin, and J. Wawrzynek. MARC: A many-core approach to reconfigurable computing. In *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, pages 7 –12, dec. 2010.

[19] A. Severance and G. Lemieux. Venice: A compact vector processor for fpga applications. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 261–268, Dec 2012.

[20] S. Shukla, N.W. Bergmann, and J. Becker. Quku: a two-level reconfigurable architecture. In *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*, pages 6 pp.–, March 2006.

[21] R. Tessier and W. Burleson. Reconfigurable computing for digital signal processing: A survey. *The Journal of VLSI Signal Processing*, 28(1):7–27, 2001.

[22] D. Unnikrishnan, Jia Zhao, and R. Tessier. Application specific customization and scalability of soft multiprocessors. In *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, pages 123–130, April 2009.

[23] P. Yiannacouras, J.G. Steffan, and J. Rose. Exploration and customization of fpga-based soft processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):266–277, Feb 2007.

[24] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. Fine-grain performance scaling of soft vector processors. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, pages 97–106, New York, NY, USA, 2009. ACM.

[25] Hayden Kwok-Hay. Yu, Colin Lin. So. Energy-efficient dataflow computations on FPGAs using application-specific coarse-grain architecture synthesis. In *Highly Efficient Accelerators and Reconfigurable Technologies, The 4th International Workshop on*. IEEE, 2012.