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.

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 [12]. Numerous design languages and environments [10] have been developed to allow

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.

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 [21] in the tool flow, the use of a partial reconfiguration, and modular design flow [15].

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 [18, 19, 22, 24, 27, 29]. 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. [28],[2], and [3] employ general processors as the overlay and mainly have micro-architecture parameters such as pipeline depth configurable. [16] uses general processor with custom instruction extension as overlay, but hardware implementation is required whenever new custom instructions are added. [20] develops a fine-grain virtual FPGA overlay specially for custom instruction extension, which makes the custom instruction implementation portable and fast. [22] 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. [27] adopts a multi-processor overlay with both micro-architecture and interconnection customizable. [24] and [29] develop reconfigurable vector processors as the FPGA overlay to cover larger domains of applications. [18] 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.

[8] and [17] 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. [13] 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 [26] [11]. 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.

[19] develops a VLIW architecture based parameterizable CGRA overlay called WPPA and provides an interconnection wrapper unit for each processing element to dynamically configure the topology of the CGRA. [14] proposes 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. [25] develops a CGRA overlay named QUKU to improve reconfiguration speed for DSP algorithms. The experiments show that the CGRA overlay bridges the gap between soft processor and customized IP core. [9] builds an high speed mesh CGRA overlay using the elastic pipeline technique and achieves the maximum throughput. These CGRA overlay work typically take DFG as input and it is also possible to extract DFG during the software compilation. Thus the CGRA overlay helps to raise the abstraction level compared to conventional hardware design. Given the CGRA overlay architecture, the compilation doesn't involve any circuit optimization such as timing and pipelining at all and it makes the design accessible to an application developer without much hardware design experience. Especially, unlike the soft processor overlay and virtual FPGA overlay, [9] shows that the CGRA overlay doesn't have to compromise between the implementation frequency and design productivity as well as design portability.

As demonstrated in previous CGRA work, one of the major motivations of using CGRA overlays is its promising performance acceleration capability for compute intensive applications. However, a complete accelerator design on a hybrid general purpose processor (GPP) + FPGA using CGRA overlay is still missing. In fact, communication between FPGA and GPP is expensive, it has significant impact on the overall performance acceleration and thus influence the design choices of the CGRA overlay as well.

In this work, we opt to utilize a fully pipelined synchronous SCGRA as the overlay. Then we further implement it as an accelerator on Zedboard [5] which is a hybrid ARM + FPGA system. The accelerator now is configurable and is capable to handle all of our four full compute intensive applications with diverse data sets. With this SCGRA overlay based accelerator design method, 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 an FPGA accelerator, where the processor handles tasks not-well suited to FPGA such as providing the OS environment and FPGA focuses on compute intensive kernels.

Figure 1 shows a typical FPGA acceleration architecture and all the experiments in this work stick 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 is a group of data buffers, an acceleration control block(Acc Ctrl), and a computation core. Data buffers are employed to store input data, output data and even temporary data of the computation. The Acc Ctrl block receives computation start signal probably from CPU via the bus when the input data is ready. Then it will trigger the computation core to start computation. When the computation is done, it will send the computation done signal back to interrupt CPU to collect data from the output buffer. The computation core is a customized circuit optimized for target compute kernel, which is supposed to be fast.

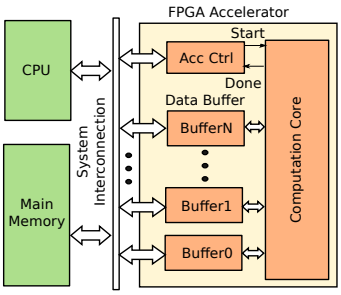

Fig. 1. A Typical FPGA Acceleration Architecture

B. QuickDough

In this section, QuickDough, which is developed to provide a customized acceleration solution to the target applications based on such a acceleration architecture, is presented. As shown in Figure 2, it starts from HW/SW partition where the compute intensive kernels of the applications are extracted. The extracted kernels are further transformed to DFGs, which are preferred in CGRA compilation. There are already work on both, which may help us to automate the processing [6] [4]. Currently, we just manually perform the HW/SW partition and DFG transformation in our preliminary research stage.

After the HW/SW partition, the design method can roughly be split into two parts. The part on top half is essentially the conventional software compilation except that we need to replace the compute kernel with accelerator drivers to mange the accelerator in the application, and the binary code is generated in the end. The part on the bottom half is basically the SCGRA compilation. Given the DFG of the compute kernel and accelerator configuration such as SCGRA size and on chip buffer capacity, the SCGRA compiler performs the operation scheduling, produces the configurations of the SCGRA overlay and generates the bitstream eventually based on the pre-implemented SCGRA overlay in the SCGRA library. Finally, the binary code is loaded to the host system, the bitstream is configured to FPGA and the application is implemented on the hybrid compute system.

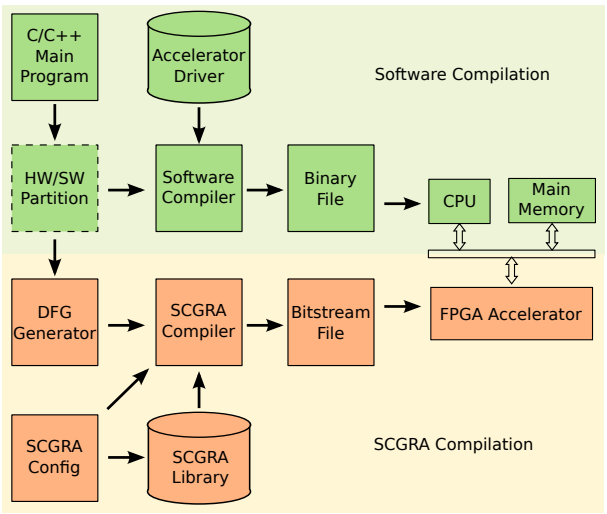

Fig. 2. QuickDough: An FPGA Accelerator Design Method Using SCGRA Overlay

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 the SCGRA overlay. The input/output data buffers and Acc Ctrl block are almost the same with those in the typical acceleration architecture 1, while the rest blocks are unique.

Particularly, the kernel part of the accelerator is a synchronous 2D torus SCGRA which consists of an array of PEs. Details of the PE will be illustrated in the next section. Another major difference is that two address buffers instead of the customized logic are used as the address generator to control the on-chip buffer accessing. With the address buffers, there is no need to develop any spcific address generator when the target application changes, as we can simply replace its content together with the SCGRA configuration memory according to the scheduling result. Therefore, it reduces the chance of FPGA implementation and is beneficial to improving the design productivity.

Finally, it is noted that input and output data buffers could accommodate more data than that used by a single SCGRA execution, so the SCGRA may iterate multiple times before it consumes all the data in input buffer or fills the output buffer. From the perspective of host processor, it is more efficient to control multiple SCGRA execution at a time instead of

each SCGRA execution independently. In this case, we have a control unit called SCGRA Ctrl to make the multiple SCGRA execution transparent to the Acc Ctrl. Also sharing the data sets among multiple SCGRA execution could make best use of the limited input/output buffer. In addition, the regular SCGRA overlay tends to run at higher frequency than the data buffer controller handling the system bus protocol, and two sets of synchronous registers are added to keep the computation core and the rest of the design to running at individual clock domains.

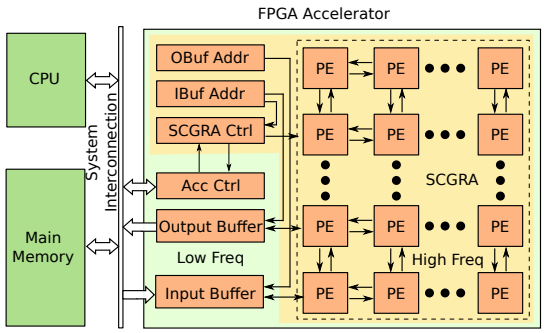

Fig. 3. SCGRA Accelerator

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 memory, is highly optimized for FPGA implementation. In addition, load/store paths are implemented on the PEs that are responsible for data I/O beyond the FPGA. Addr Ctrl is used to start and reset the SCGRA execution by changing the instruction memory read address sequentially, so it has a single bit global start input signal from the SCGRA Ctrl block.

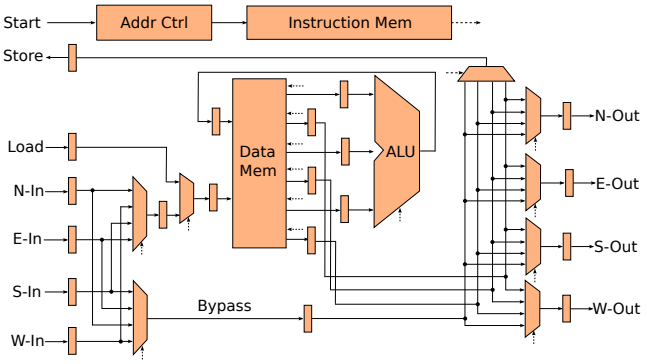

Fig. 4. PE structure

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 and the content of the ROM can be loaded directly into a pre-implemented bitstream. The address of the instruction ROM is determined by the Addr Ctrl. Basically, the SCGRA execution will proceed sequentially 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 seem to meet 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 using 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, eventually a multi-port data memory is used 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. Complex 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 create the ALU design, but it is possible to automate this step.

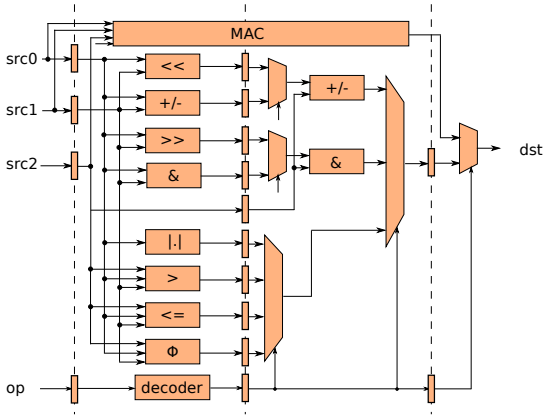

Fig. 5. ALU Example

C. Load/Store Interface

For the PEs that also serve as IO interface to the SCGRA, they have additional load path and store path as shown in 4. Load path and the SCGRA neighboring input share a single

TABLE I
OPERATION SET IMPLEMENTED IN ALU

Type	Opcode	Expression
MULADD	0001	dst = src0 × src1 + src2
MULSUB	0010	dst = src0 × 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

data memory write port, and an additional pipeline stage is added to keep the balance of the pipeline. Store path has an additional data MUX as well, but it doesn't have significant influence on the rest of the design.

V. SCGRA COMPILATION

Figure 6 illustrates the detailed SCGRA compilation of QuickDough. As shown in the diagram, the compilation esentially is to compile the compute kernel of an application to a specifid SCGRA and generate the implementation bitstream.

The compilation starts from transforming the compute kernel probably written in high level language to a DFG as well as hyper block level IO mapping. When the DFG is determined, it can be scheduled to the specified SCGR using an operation scheduler. As the simulation performance of the DFG can be acquired from the scheduling, it is already enough to estimate the performance of the compute kernel. Moreover, taking the pre-built SCGRA implementation frequency and specified communication efficiency into consideration, it is possible to come up an even more accurate performance evaluation. Whether the performance design goal is met can be checked at this step. If no, we can go back to the DFG generation stage altering the DFG generation options such as loop unrolling factor. It may take multiple iterations to meet the design goal.

Once the design goal is achieved, the configuration words can be extracted from the scheduler and be further integrated into the pre-built SCGRA bitstream using the data2mem tool. Since the bitstream in the SCGRA library is bundled with specific FPGA device, HDL model will be used for porting to a new device and complete hardware implementation flow is required in that circumstance accordingly.

A. DFG Generator

When the data set is small, it is usually possible to fully unroll the loops in the compute kernel and transform the unrolled kernel to DFG directly. However, when the data set gets larger, delicate unrolling strategy is needed as it has critical impact on both the performance and hardware overhead such as on chip buffer size and instruction memory depth. While unrolling strategy is not the focus of this paper, we will mainly explain the major constrains of the loop unrolling and the DFG generator. Also we will clear the required dumping list for the following compilation step.

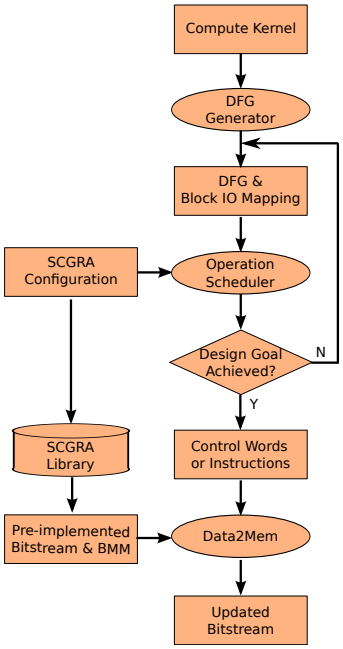

Fig. 6. SCGRA Compilation

Figure 7 shows how the DFG is extracted and executed on the SCGRA overlay based accelerator. The compute kernel can be performed by repeating the block execution, so the first step is to determine the block size that can be executed on the accelerator. Since the SCGRA overlay employs lock-step execution and the input/output data for each block execution must be fully buffered, the block size is mainly limited by the on chip buffer size. In this example, block size is set to be 10.

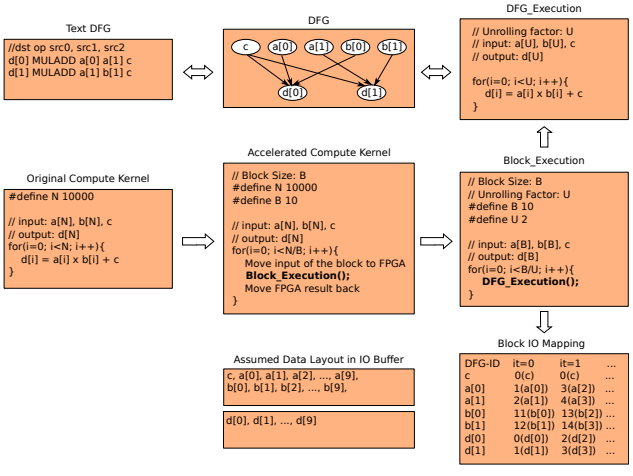

Fig. 7. DFG Generation

While the whole block can be completed by repeating the DFG execution, the next step is to decide the loop unrolling factor, such that the unrolled part can be transformed to DFG which can be executed on the SCGRA overlay at a time. In theory, the unrolling factor is probably limited by the instruction memory and data memory. Nevertheless, we have straightforward address buffers which store all the on chip buffer accessing addresses of the whole block execution.

Although it is already set to be twice larger than the data buffer, it still overflows easily and becomes another major limitation of both the block size and the unrolling factor. In addition, the compute kernel depends on the repeating of the block execution and the block execution depends on the repeating of the DFG execution. As a result, the total loop iteration number must be fully divided by the block size. Similarly, the block size must also be fully divided by the unrolling factor. This can be another unrolling and blocking limitation as well.

When the blocking and unrolling factor are determined, we can look at the dumping list of the DFG generator for the following compilation step. Apparently, DFG is required, as it is the input of the operation scheduler, And we store it in a text version as shown in Figure 7. The operation scheduler simply takes the DFG as input and it assumes the input/output of the DFG stored sequentially. However, it is more natural for the compute kernel to call the block computation directly. To bridge the gap, we have the block input/output stored sequentially, but have the address buffers to translate them back to the DFG preferred style. Therefore, the DFG generator must also provide the IO mapping information for each DFG execution as presented in Figure 7 as well.

B. Operation Scheduler

The operation scheduler adopts a classical list scheduling algorithm [23] to tackle the DFG scheduling. While scheduling operations to PEs closer with each other could reduce the communication cost but may lose the load balance, a scheduling metric compromising both the communication and load balance as presented in [30] is delicately adjusted to adapt to the proposed SCGRA overlay.

Algorithm 1 briefly illustrates the scheduling algorithm implemented in QuickDough. Initially, an operation ready list is created to store operations that can be scheduled. The next step is to select a PE from the SCGRA and an operation from the operation ready list using the compromised communication and load balance metric. When both the PE and the operation to be scheduled are determined, the OPScheduling procedure starts. It will figure out an optimized routing path, move the source operands to the selected PE along the path, and have the selected operation executed accordingly. After this step, the operation ready list is updated as the latest scheduling may produce more ready operations. Repeat the OPScheduling procedure as well as the operation ready list updating. The DFG scheduling will be completed when the operation ready list is empty.

Finally, the control words of each PE and the IO buffer accessing sequence will be dumped from the scheduler. They will be used for bitstream generation in the following compilation step.

Algorithm 1 The SCGRA scheduling algorithm.

```
procedure ListScheduling
  Initialize the operation ready list L
  while L is not empty do
    select a PE p
    select an operation l
    OPScheduling(p, l)
    Update L
  end while
end procedure

procedure OPScheduling(p,l)
  for all predecessor operations s of l do
    Find nearest PE q that has a copy of operation s
    Find shortest routing path from PE q to PE p
    Move operation s from PE q to PE p along the path
  end for
  Do operation l on PE p
end procedure
```

C. Bitstream Integration

The final step of the compilation is to incorporate the instruction for each PE as well as the IO buffer addresses obtained from the scheduling result 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 store the cycle-by-cycle configuration words using on-chip ROMs. The content of the ROMs are embedded in the bitstream and data2mem tool from Xilinx [1] can be used to update the ROM content of the pre-built bitstream directly. To complete the bitstream integration, BMM file that describes the organization and placed location of the ROMs in SCGRA overlay is also required and it can be extracted from the XDL file [7] of the pre-built SCGRA overlay automatically. While original SCGRA design needs around an hour to implement, the bitstream integration 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. To investigate the scalability of the accelerator design methods, 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 illustrated in the Table II.

The benchmark is implemented on Zedboard using both Vivado HLS based design method and QuickDough. Then the design productivity, implementation efficiency and performance of the two design methods are compared respectively.

A. Experiment Setup

All the runtimes were obtained from a laptop 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 i.e. IP core. Vivado 2013.3 is used to integrate the IP core and build the FPGA accelertor. The SCGRA is initially developed in ISE 14.7, and then the ISE project is imported as an IP core in planAhead 14.7. With the SCGRA IP core, the SCGRA overlay based FPGA accelerator is further built in

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

PlanAhead as well. Finally, the accelerators developed using both design methods target at Zedboard [5] and the system runs in the bare-metal mode.

Vivado HLS typically achieves the trade-off between hardware overhead and performance through altering the loop unrolling factors. Larger loop unrolling factors typically promise better performance, but this will not be guaranteed, because more hardware resources like DSP blocks will be required and the implementation frequency may degrade as well. In this work, we set the loop unrolling factor as large as possible and the detailed loop unrolling setup can be found in Table III.

As memory mapped IO buffers are used for the communication between FPGA and the accelerator, the compute kernel synthesized must have its input/output fully buffered. When the data set of the compute kernel is large, the on chip buffer will not be able to accommodate it. In this case, loop blocking is employed to divide the original compute kernel loop into loop blocks that fit the on-chip buffer. As presented in Table III, 2k-word buffer and 64k-word buffer will result in quite different blocking schemes for large data set. Since larger block size is beneficial to data reuse and amortizing the communication cost, the block size is set to be as large as possible within the on-chip buffer limit.

QuickDough has similar design choices to those of Vivado HLS based design method in terms of loop unrolling and blocking. However, the design constrains using the two design methods are different. The unrolled part in QuickDOugh is transformed to DFG which can further be scheduled to the SCGRA overlay, so the loop unrolling factor is mainly constrained by the resources of the SCGRA overlay such as SCGRA size, instruction memory size and data memory size. As for the loop blocking, the design constrain in QuickDough is relatively more complex. On the one hand, the block size is also limited by the size of input/output buffer, which is exactly the same with the constrain using Vivado HLS based design method. On the other hand, the address buffer size can be another major constrain, because we need to store all the IO buffer address sequence of the whole block instead of the DFG. Even though we set address buffer size twice larger than the data buffer size, it can still be a bottleneck in many occasions. The detailed SCGRA overlay configuration and corresponding loop unrolling as well as blocking setup using QuickDough are listed in Table IV and Table V respectively.

B. Experiment Results

In this section, design productivity, hardware implementation efficiency, performance and scalability of the accelerators using both design methodologies are presented and compared.

1) Design Productivity: Design productivity involves many different aspects such as the abstraction level of the design entry, compilation time, design reuse, and design portability, and it is difficult to compare all the aspects, especially some of the aspects could hardly be quantized. In this section, we mainly concentrate on the compilation time comparison and analyze the rest briefly.

In order to develop an FPGA accelerator for an application, Vivado HLS based design method roughly consists of the following 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 an HDL model according to the user's synthesis pragma. The next step is to further synthesize it and have it packed as an IP core. At the same time, the timing constrain should be met and the corresponding driver should be generated. Afterwards, the IP core can be integrated into the accelerator and the whole accelerator can be implemented on the FPGA accordingly. Finally, the application employing the FPGA accelertor should be compiled to binary code as conventional software.

The FPGA accelerator developed using QuickDough also needs four steps to complete the application acceleration. The first step is to translate the high level language program kernel to DFG. The second step is to schedule the DFG to the customized SCGRA overlay and dump the scheduling result for the following step. The third step is to integrate the scheduling result and the pre-built accelerator bitstream to produce the new bitstream for target application. Finally, the last step is also a conventional software compilation, which is similar to the fourth compilation step of the Vivado HLS based design method.

Figure 8 and Figure 9 present the compilation time of implementing the benchmark using both Vivado HLS based design method and QuickDough respectively. IP core generation and hardware implementation in Vivado HLS based design method is relatively slow. 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 pipelined large loop unrolling involved. The last step is essentially a software compilation, and the time consumed can be negligible. On the whole, the design method takes 20 minutes to an hour to implement an application. With pre-implemented SCGRA overlay, the processing steps except the DFG scheduling of QuickDough are 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 can still be completed in a few seconds.

TABLE III
LOOP UNROLLING & BLOCKING SETUP OF ACCELERATORS USING VIVADO HLS BASED DESIGN METHOD

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	$5 \times 4 \times 2$	$25000 \times 4 \times 2$	$5 \times 4 \times 2$	$1000 \times 4 \times 2$	$1000 \times 4 \times 2$

Basically, QuickDough could implement an application in 5 to 15 seconds and it is two magnitude faster than the SCGRA overlay based design method.

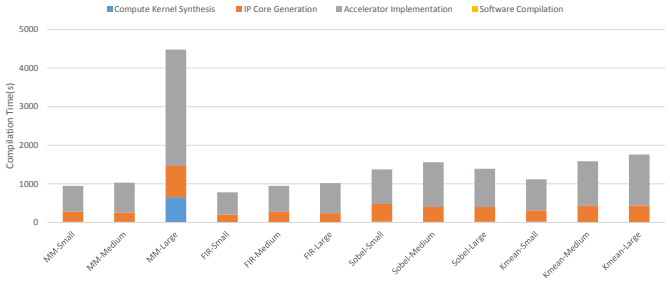

Fig. 8. Benchmark Compilation Time Using Vivado HLS Based Design Method

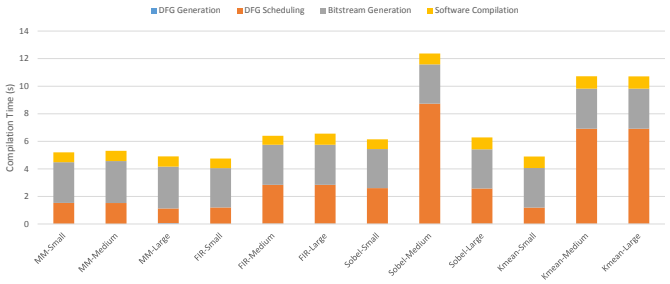

Fig. 9. Benchmark Compilation Time Using QuickDough

On top of the compilation time, the abstraction level of the design entry, design reuse and portability are also important aspects that influence the design productivity. Both Vivado HLS based design method and QuickDough adopt sequential high level language C/C++ as design input, but QuickDough still needs further efforts to have the DFG generation done automatically. Vivado HLS based design method needs the compute kernel be synthesized and implemented for each application instance. QuickDough requires compilation for each application instance as well, but could reuse the same hardware infrastructure across the applications in the same domain. It is possible for Vivado HLS based design method to port the synthesized HDL design among different devices and parts, but IP core generation and accelerator implementation depend on specific FPGA device. QuickDough's portability is also limited at HDL level, and complete hardware implementation is needed to port to a different FPGA device.

2) Hardware Implementation Efficiency: In this section, hardware implementation efficiency including the hardware resource overhead and implementation frequency of the accelerators using both Vivado HLS based design method and QuickDough are compared.

Table VI exhibits the hardware overhead using both accelerator design methods. It is clear that the accelerators using Vivado HLS based design method typically consume 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 and it limits the maximum loop unrolling factors for many applications. The accelerators using QuickDough usually cost comparable DSP48, more FF, LUT and particularly RAM36. Taking consideration of the total resource on FPGA, the overhead of DSP48, FF and LUT are still acceptable, while the RAM36 consumption is really large and it limits the maximum SCGRA that can be implemented on the target FPGA. Accordingly, it further constrains the maximum loop unrolling and blocking as well.

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

TABLE V
LOOP UNROLLING AND BLOCKING SETUP FOR ACCELERATORS USING QUICKDOUGH

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	$500 \times 4 \times 2$	$50000 \times 4 \times 2$

Figure 10 presents the implementation frequency of the benchmark using both accelerator design methods. Vivado HLS based design method takes timing constrain into consideration at the HLS step, and it could either synthesize the compute kernel to a lower frequency design with better simulation performance or a higher frequency design with worse simulation performance. Neither of them have a clear advantage over the other. While the AXI controller could work at around 100MHz by default and higher frequency design requires delicate placing and routing. Eventually, we set the HLS timing constrain at 100MHz and the whole accelerator could be synchronous. As there are so many different design options, the current design is not necessarily the optimal one, but it is representative. In addition, the synthesized IP core sometimes can be even slower though we set the timing at 100MHz during HLS. In this case, we don't spend time iterating the the HLS and implementing the whole design again. Instead, we just slow down the clock frequency until the timing constrain is met. As a result, sometimes the accelerator can only work at around 60MHz.

QuickDough utilizes the SCGRA overlay as the hardware infrastructure. Since the SCGRA overlay is regular and pipelined, the implementation frequency of the accelerator built on top of the SCGRA overlay is much higher than that of the accelerator produced using Vivado HLS. A 2x2 SCGRA based accelerator could run at 200MHz, and a 5x5 SCGRA based accelerator could work at 167MHz. The implementation frequency degrades slightly, because more than 95% of the BRAM blocks on target FPGA are used and the routing becomes extremely tight. However, the AXI controller block on Zedboard is slower and runs around 100MHz. To take advantage of the higher implementation frequency, simple synchronizers built with consecutive registers are inserted to divide the AXI controller and the SCGRA overlay into two

TABLE VI
HARDWARE OVERHEAD OF THE ACCELERATORS USING BOTH VIVADO HLS BSED DESIGN METHOD AND QUICKDOUGH

			FF	LUT	RAM36	DSP48
MM	2K Buffer	Small	4812	3390	4	84
		Medium	4804	4703	4	12
		Large	11107	11524	4	12
	Max Buffer	Small	4826	3390	128	84
		Medium	4251	4866	128	9
		Large	11024	24890	128	12
FIR	2K Buffer	Small	3736	3570	4	27
		Medium	3756	3872	4	27
		Large	3756	3872	4	27
	Max Buffer	Small	3742	3570	128	27
		Medium	3782	4246	128	27
		Large	3792	4426	128	27
Sobel	2K Buffer	Small	9556	6467	6	216
		Medium	7483	5520	6	144
		Large	7102	5501	6	144
	Max Buffer	Small	9564	6467	130	216
		Medium	7496	5711	130	144
		Large	7622	5904	130	144
Kmean	2K Buffer	Small	2826	3567	4	24
		Medium	6709	8088	4	120
		Large	6709	8088	4	120
	Max Buffer	Small	2852	3567	128	24
		Medium	6754	8122	128	120
		Large	6770	8205	128	120
SCGRA 2x2			9302	5745	32	24
SCGRA 5x5			34922	21436	137	150
FPGA Resource			106400	53200	140	220

separate clock domains. Also note that the implementation frequency of the SCGRA based accelerators in Figure 10 actually refers to the SCGRA overlay.

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 configuration is used instead. Figure 11 shows the performance comparison of four different sets of accelerator implementations including two accelerators built with QuickDough and two accelerators developed with Vivado HLS based design method. 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.

To gain insight into the performance of the two design methods, Figure 12 presents the execution time decomposition of the benchmark. As 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. The benchmark runs on an ARM + FPGA architecture 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 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.

According to this figure, accelerators using Vivado HLS based desgin method especially the one with max-buffer configuration perform better performance mainly due to the smaller overhead in communication and the others which are essentially the input/output data reorganization time. As presented in Table III, it is clear that accelerators using Vivado HLS based design method with max-buffer configuration could accommodate larger data sets and corresponding computation as well, which may improve the data reuse of the communication and deduce the amount of communication and data reorganization cost. This is one of the major reasons for lower communication cost while using Vivado HLS based design method. Moreover, part of the DMA transmission cost probably induced by DMA initialization routine is independent on the transmitted data size. Therefore, the average communication cost per data can be lower when the data size per DMA transmission is larger. Of course, the benefit will be negligible when the DMA transmission size is big enough to amortize the initialization cost. Because of the reasons mentioned above, for all the applications of the benchmark with medium and large data sets, the accelerators using Vivado HLS based design method especially with the Max-Buffer configuration have smaller communication and data reorganization cost.

Computation time is crucial to the overall execution time. It basically depends on both the simulation performance in cycles and implementation frequency of the 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. From this paper, it is clear that Vivado HLS based design method wins on MM-Large and Sobel while QuickDough outperforms on the rest of the benchmark. Referring to both the simulation performance in this figure and loop unrolling factors in Table III and Table V, it can be found that the simulation performance is quite relevant to the depth of the loop unrolling. More precisely, the simulation performance mostly depends on the depth of the loop unrolling instead of the specific hardware infrastructure. The only exception is the Sobel benchmark. QuickDough performs more intensive loop unrolling, but the simulation performance is not as good as Vivado HLS. The reason is probably due to the two Sobel operators used in the algorithm. Vivado HLS takes them as constant input and could have special optimization on it. While the DFG generator in QuickDough just takes them as normal variable input and more operations are required.

Loop unrolling factor is crucial to the simulation performance of the compute kernel, while the hardware infrastructure determines how much the loop can be unrolled, which eventually influences the simulation performance of the compute kernel. As presented in Table III and Table V, the loop unrolling factor with Vivado HLS is smaller than that in QuickDough in many cases. And this explains the comparison of the simulation performance using the two design methods.

QuickDough using SCGRA overlay as the hardware infrastructure could run at higher frequency, and the advantage of computation time as shown in Figure 14 is enlarged on top of that in the simulation performance.

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

4) Scalability: On top of the design productivity, hardware implementation, and performance, the scalability of the accelerator using the design methods is also equally important. To investigate the scalability of the accelerators using both design methods, we adopt matrix multiplication with smoothly increasing matrix size as the benchmark.

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 method is designed to synthesize parallel compute kernels to execute on FPGAs

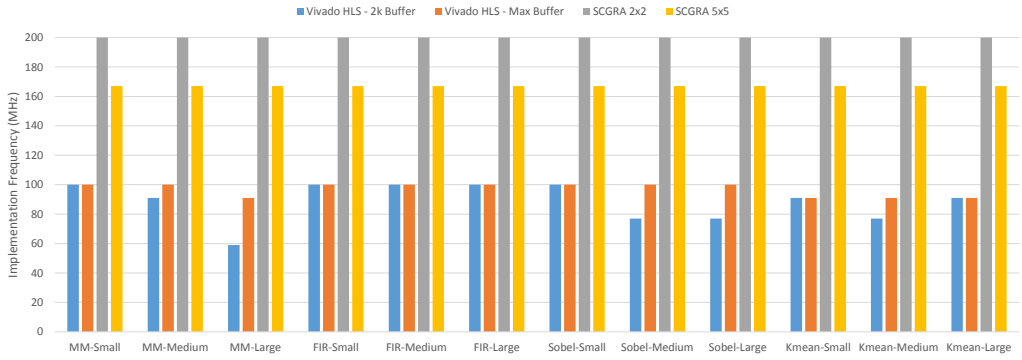

Fig. 10. Implementation Frequency of The Accelerators Using Both Vivado HLS Based Design Method and QuickDough

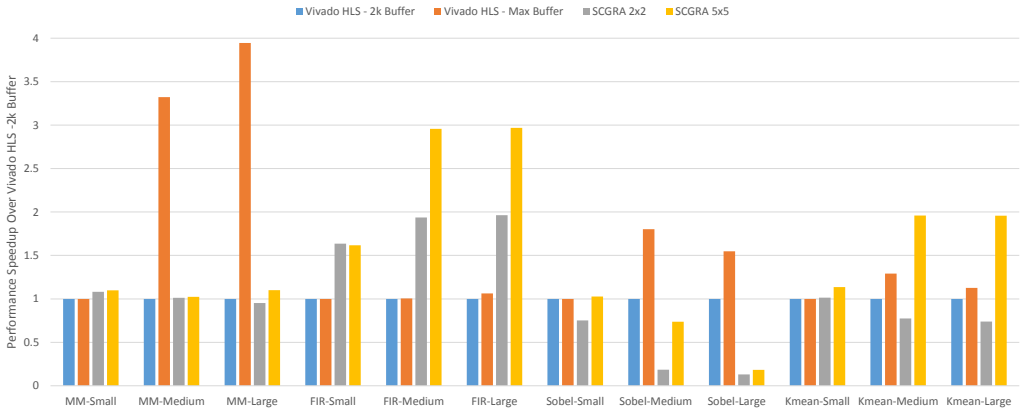

Fig. 11. Benchmark Performance Using Both Vivado HLS Based Design Method and QuickDough

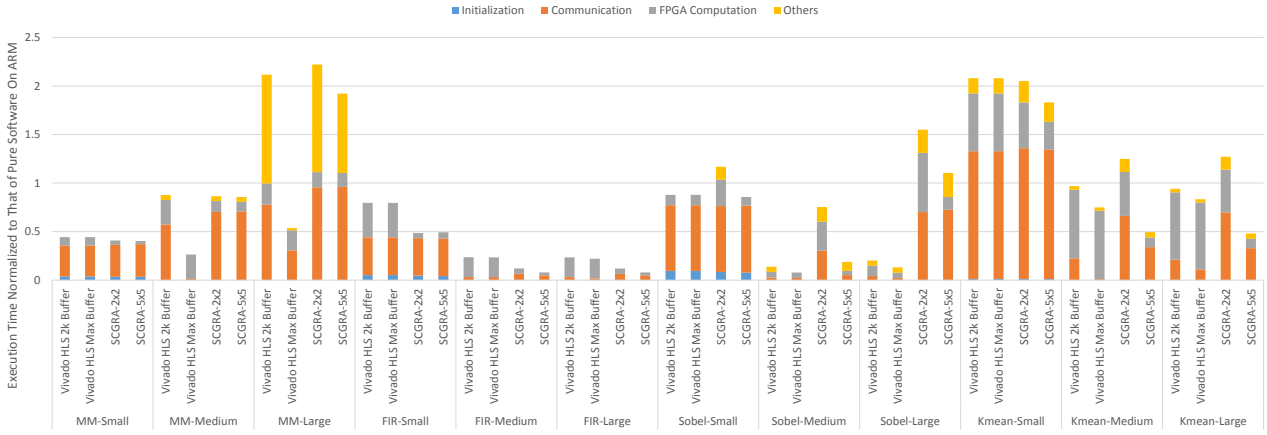

Fig. 12. Benchmark Execution Time Decomposition Of The Accelerators Using Both Vivado HLS Based Design Method and QuickDough

only. As such, it is not a generic method to perform HLS on random logic. Moreover, 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.

Secondly, the DFG is still manually generated, and a general front-end compilation that could transform high level language program kernel to DFG is still missing. Thridly, 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 SCGRA configuration. An SCGRA optimizer will be developed to perform the SCGRA customization automatically in future. Finally, the capacity of the address buffers used in the accelerator limits the block size that can be adopted to the FPGA in a few cases. However, there are a large number of invalid address entries in it and this will be fixed in future.

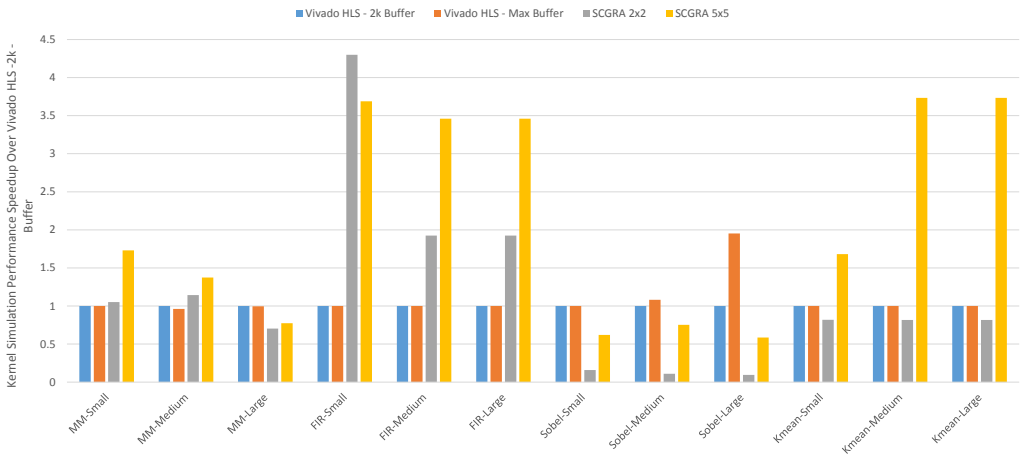

Fig. 13. Simulated Compute Kernel Performance Using Both Vivado HLS Based Design and QuickDough

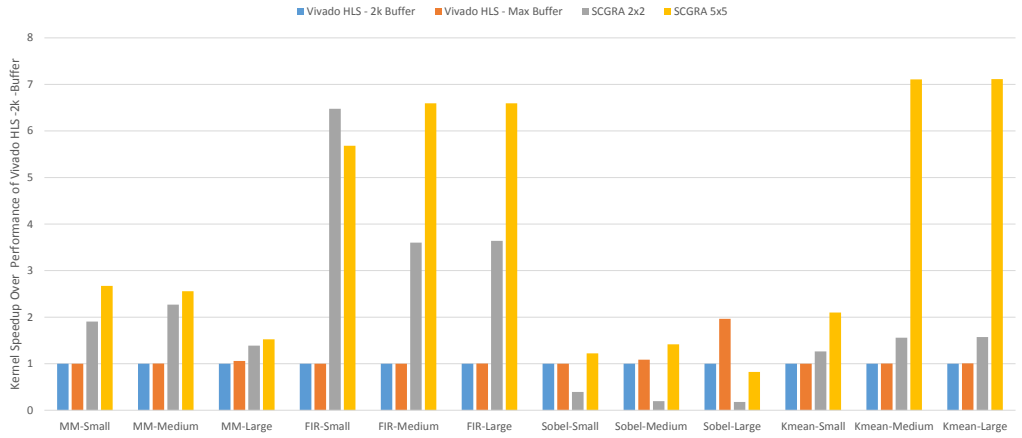

Fig. 14. Compute Kernel Performance Using Both Vivado HLS Based Design Method and QuickDough

VIII. Conclusions

In this paper, we have proposed QuickDough, an SCGRA overlay based FPGA accelerator design method, to compile compute intensive applications to a CPU+FPGA system. With the SCGRA overlay, the lengthy low-level implementation tool flow is reduced to a rapid operation scheduling problem. The compilation time from high level language application to the CPU+FPGA system is reduced by around 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 operations to overlap with pipeline latencies provides competitive performance compared to that using a commerical 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/nio-index.html. [Online; accessed 25-June-2014].

[4] Roccc2.0. http://www.jacquardcomputing.com/roccc/. [Online; accessed 19-January-2014].

[5] Zedboard. http://www.zedboard.org/. [Online; accessed 25-June-2014].

[6] Massimo Baleani, Frank Gennari, Yunjian Jiang, Yatish Patel, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Hw/sw partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, pages 151–156, New York, NY, USA, 2002. ACM.

[7] 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.

[8] 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.

[9] 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.

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

[11] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (csuR)*, 34(2):171–210, 2002.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] Dmitrij Kissler, Frank Hannig, Alexey Kupriyanov, and Jürgen Teich. A dynamically reconfigurable weakly programmable processor array architecture template. In *ReCoSoC*, pages 31–37, 2006.

[20] 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.

[21] 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.

[22] 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.

[23] JMJ Schutten. List scheduling revisited. *Operations Research Letters*, 18(4):167–170, 1996.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.