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

Cheng Liu, Colin Yu Lin and Hayden Kwok-Hay So

***Abstract*—The QuickDough design framework is presented as a way to address productivity issues of developing high-performance FPGA accelerators. QuickDough utilizes a soft coarse-grained reconfigurable array (SCGRA) as an overlay on top of off-the-shelf FPGAs for rapid accelerator developments. Instead of compiling high-level applications directly to HDL circuits, the compilation step is reduced to a simpler operation scheduling task targeting the SCGRA overlay, significantly reducing compilation time and increasing possible numbers of debug-cycle-per-day as a result. The softness of the SCGRA allows highly customized domain-specific design while the regularity of the SCGRA makes the implementation scalable and reusable. When compared to a conventional design methodology using off-the-shelf high-level synthesis tools, QuickDough achieves competitive end-to-end application performance while reducing 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, FPGA accelerator design using the conventional HDL based design flow suffers extremely low design productivity due to the low-level design entry, lengthy compilation and poor design reuse, especially compared to the corresponding software solution.

Overlay which can be parametric HDL model, pre-synthesized or pre-implemented circuits promises to address the above challenges. It raises the abstraction level by using either high level language or data flow graph (DFG) as the design entry both of which are closer to software design. As the configuration granularity of an overlay is usually coarser, the compilation process is simplified and the compilation time can be reduced. Moreover, it can be resued easily across the FPGA devices and parts, which is essentially beneficial to FPGA accelerator design productivity.

In this work, an SCGRA is built as an FPGA overlay. Using this overlay, QuickDough a rapid FPGA accelerator design methodology is proposed. It translates compute kernel to DFG and then statically schedules the DFG to the target SCGRA. Finally, it integrates the scheduling result with a pre-implemented bitstream and accomplishes the accelerator design.

With the SCGRA overlay, QuickDough has the lengthy hardware compilation 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 implementation is required per application or application domain through the design iterations. In addition, the proposed SCGRA has quite regular hardware structure, and it scales well on both the implementation frequency and execution time in cycles, which are essential to the end-to-end run time. According to the experiments on Zedboard, FPGA accelerator using QuickDough achieves competitive performance compared to that using direct HLS, while the SCGRA does consume more hardware overhead especially the BRAM blocks.

In Section III, the proposed FPGA acceleration design methodology QuickDough will be elaborated. The SCGRA implementation and compilation will be presented in Section IV and Section V 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

FPGA design productivity remains a major obstacle that hinders the use of FPGA as computing devices from widespread adoption. Researchers have tried to approach the problem both by increasing the abstraction level of the design entry 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 designers to focus on high-level functionality instead of low-level implementation details. While the high-level abstraction helps to express the desired functionality, the low-level compilation time spent in synthesis, mapping, placing and routing is equally crucial to the design productivity. Researchers have approached the problem from many angles including the use of pre-compiled hard macros [21], partial reconfiguration and modular design flow [15].

On top of the above approaches, overlay, which can be parametric HDL Model, pre-synthesized or pre-implemented coarse-grained reconfigurable circuits over the fine-grained FPGA devices, promises both to raise the abstraction level and reduce the compilation time. Recent years have seen a number

of overlays [18, 19, 22, 24, 27, 29] with granularities ranging from multi-processors to highly configurable logic arrays.

Soft processors, which allow customization for target applications or application domains, have already been demonstrated to be efficient overlays on FPGA. The authors in [28] [2] and [3] mainly have micro-architecture parameters such as pipeline depth configurable, while the authors in [16] allow instruction set architecture (ISA) customizable. The authors in [20] developed a fine-grain virtual FPGA overlay specially for custom instruction extension, which makes the custom instruction implementation portable and fast. The authors in [22] proposed a many-core overlay with customizable data path. This overlay can be used to explore both the coarse-grain multi-thread parallelism and data-flow style fine-grain threading parallelism. The authors in [27] presented a multi-processor overlay with both micro-architecture and interconnection customizable. The authors in [24] and [29] took reconfigurable vector processors as the FPGA overlay to cover domain specific applications. The authors in [18] presented a GPU-Like overlay for portability. It can be used to explore both the data-level parallelism and thread level parallelism. These processor level overlays achieve the customization from diverse angles, but essentially they tend to expose a processor to the user. Thus the application development is quite close to a conventional software design, while the penalty is the hardware overhead and implementation frequency compared to a customized hardware design.

Another group of FPGA overlays are virtual FPGAs [8] [17] [13]. They are built on top of the off-the-shelf FPGA devices and typically have coarser granularity. 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 emphasize more on configuration capability to cover more applications, while FPGAs' inherent programmability greatly alleviate the concern. Accordingly, CGRAs on FPGA allow more intensive customization.

The authors in [19] developed WPPA a VLIW architecture based parameterizable CGRA overlay. There is an interconnection wrapper unit for each processing element (PE) and it can be used to dynamically configure the topology of the CGRA. The authors in [14] 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. In [25], QUKU a customized CGRA overlay was developed to improve reconfiguration speed of DSP algorithms. It can be used to bridge the gap between soft processor and customized IP core. The authors in [9] built a high speed mesh CGRA overlay using the elastic pipeline technique to achieve the maximum throughput. These CGRA overlays have demonstrated the promising performance acceleration capability for compute intensive applications. However, they mainly target the pure DFGs extracted from the compute kernels and it is insufficient for the FPGA accelerator design. In addition, the DFGs used are usually small and are limited to dozens of nodes to one or two hundred of nodes, while the benchmark in this work requires much larger DFGs.

In this work, we opted to utilize a fully pipelined synchronous SCGRA as the overlay and had it implemented as an accelerator on Zedboard [5] which is a hybrid ARM + FPGA system. The overlay can be customized either for each application of the benchmark or the whole benchmark. Moreover, we presented an end-to-end run time evaluation of the benchmark.

III. QuickDough Framework

A. System Context

This work assumes a hybrid computing architecture with both general purpose processor and FPGA, where the processor handles complex control intensive tasks such as providing the OS environment and FPGA focuses on compute intensive kernels. Figure 1 shows a typical FPGA acceleration system used in this paper. FPGA is attached to the system interconnection and it includes a group of independent accelerators customized for different compute kernels of the target application. In each accelerator, there are input/output data buffers used for buffering the communication data, a computation core customized for the compute kernels and an Acc-Ctrl block which can be used to start the computation core and acknowledge the computation core status as well.

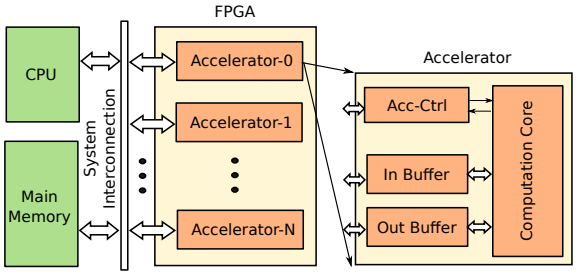


Fig. 1. Typical FPGA Acceleration System and Accelerator Architecture

B. QuickDough

On top of the system context, QuickDough, an SCGRA based FPGA accelerator design methodology, is presented in Figure 2. It starts from HW/SW partition where the compute intensive kernels of an application are identfied. Then it transforms the compute kernels in high level language program to DFGs for SCGRA compilation. There are already intensive work on both the HW/SW partition and DFG generation [6] [4]. We may automate the two steps in future, but currently we just manually handle the two steps in our preliminary research stage.

After the HW/SW partition and DFG generation, the design methodology in Figure 2 can roughly be split into

two parts. The part on the top half is mainly responsible for the conventional software compilation targeting general purpose processor of the hybrid compute system. In order to make use of the FPGA accelerators, we need to replace the original compute kernels with the accelerator drivers before the standard software compilation. When the software compilation is completed, the binary code targeting the processor will be generated.

The part on the bottom half mainly focuses on the SCGRA customization and compilation. Once the DFGs to be implemented on the SCGRA overlay are determined, SCGRA configuration such as operation type, PE pipeline, SCGRA size, on chip buffer capacity etc. should be decided accordingly. As the design space is large, delicate optimization algorithm is required to tackle the customization problem. At the moment, we just support the operation type customization and leave the rest design customization for future work. After the SCGRA customization, the SCGRA configuration can be decided and corresponding SCGRA compilation starts. If the SCGRA configuration is not available in the SCGRA library, hardware implementation is performed. After the implementation, the pre-implemented SCGRA will be put into the SCGRA library for reuse. Meanwhile, with the input DFG, SCGRA configuration and implementation, FPGA bitstream can be produced. When both the binary code and bitstream are ready, we complete the application specific FPGA acceleration system.

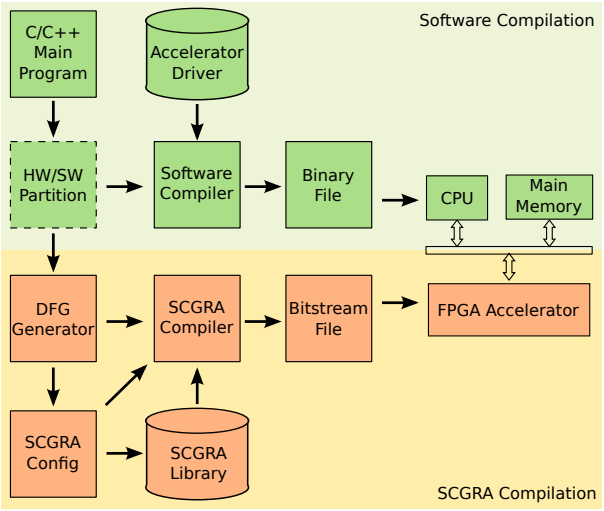


Fig. 2. QuickDough: FPGA Accelerator Design Methodology Using SCGRA Overlay

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 presents the proposed FPGA accelerator built on top of an SCGRA overlay. The input/output data buffers and Acc-Ctrl block are the same with those in the typical acceleration architecture in Figure 1, while the computation core is unique. It consists of an array of synchronous PEs which can be easily pipelined and run at higher frequency. Moreover, the regular array is able to maintain the high implementation frequency even when it scales up.

On top of the computation core, the accelerator has two address buffers (Addr IBuf/OBuf) instead of customized logic to control the on-chip data buffer accessing. When target application changes, the user can simply update the content of the address buffers to adapt to the change and thus can reuse the same hardware structure, which is beneficial to design reuse and improving design productivity.

Another important block of the accelerator is SCGRA-Ctrl. It can be used to iterate the SCGRA execution until the executions consume all the data in input data buffer or fill the output data buffer. As the executions can share the same input/output data buffer, it helps to increase the data reuse and reduce the communication cost.

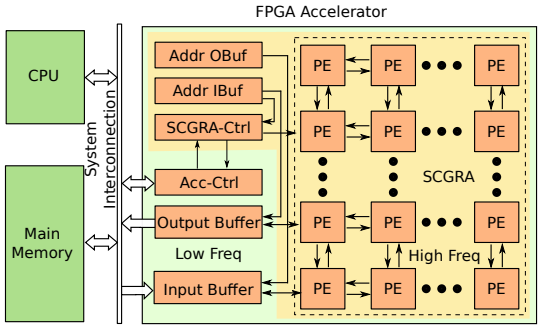


Fig. 3. SCGRA Overlay Based FPGA Accelerator

B. Processing Element (PE)

In this section, an instance of the pipelined 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. In order to synchronize the execution of the PE array, each PE has a single bit global start signal from the SCGRA-Ctrl block.

1) Instruction Memory and Data Memory: The instruction memory stores all the control words of the PE. Its content will not change at runtime, so ROM is used to implement this instruction memory. The address of the instruction ROM is determined by the Addr-Ctrl. Once the start signal is valid, the ROM address will increase by one every cycle and the SCGRA execution will proceed accrodingly. When the start signal is invalid, the ROM address will be reset to be 0 and the SCGRA execution will stop.

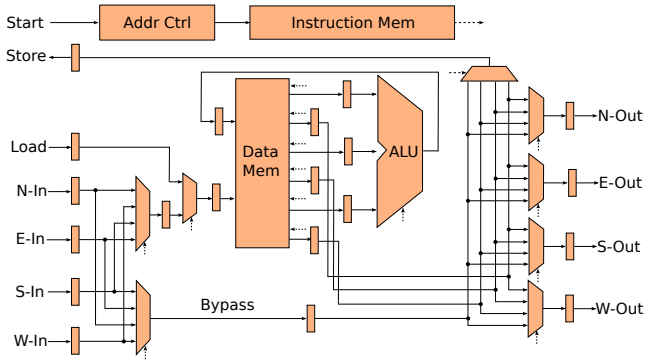


Fig. 4. Fully Pipelined PE structure

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 added to the data memory.

2) ALU: At the heart of the proposed PE is an ALU designed to cover the computations in the target applications. Figure 5 shows an example design that can support all the operations of our benchmark as listed in Table I. As the operations are usually different, each operation is provided with an independent data path. At the end of the data paths, a set of multiplexers are added to select the expected output. Currently, the maximum number of operations allowed in this template is 16 and an additional pipeline stage is added for the output selection. As the scheduler will make sure that the output port of the ALU is contention-free, the opcode in this ALU doesn't have to align with the input operations. Instead, it is merely used for output selection. Therefore, the proposed ALU can be easily customized and scaled.

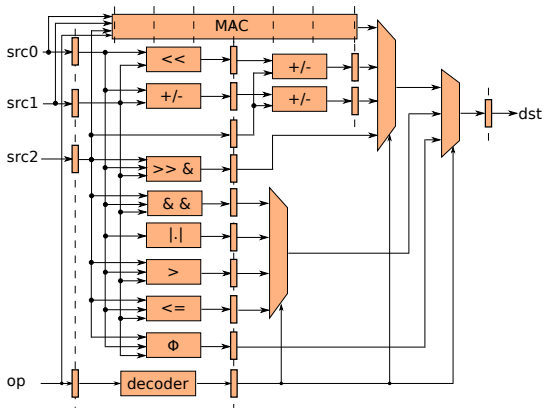


Fig. 5. ALU Example Supporting 12 Operations

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

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

V. SCGRA COMPILATION

A. Overview Of SCGRA Compilation

Figure 6 illustrates the detailed SCGRA compilation of QuickDough. As shown in the diagram, it aims to compile a compute kernel of an application to a customized SCGRA and generate the corresponding implementation bitstream.

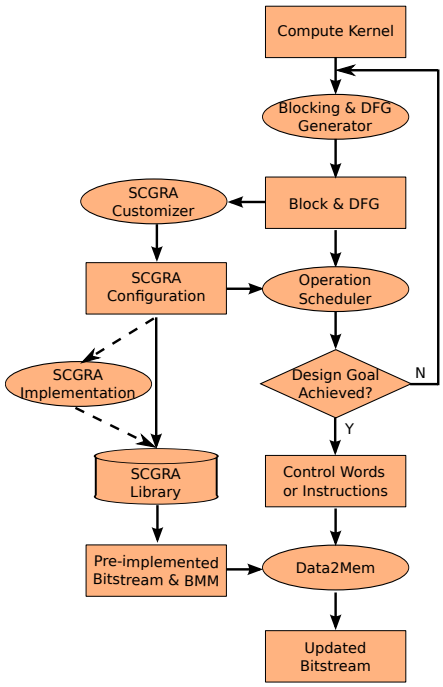


Fig. 6. SCGRA Compilation

The compilation starts from transforming the compute kernel probably written in high level language to a DFG as well as hyper block which is a number of consecutive DFG. Given the

DFG and block, a customized SCGRA configuration based on the template presented in previous section is determined. Then the DFG and block can be scheduled to the specified SCGRA using an operation scheduler. As the simulation performance of the DFG and block can be acquired from the scheduling, with the pre-built SCGRA implementation frequency and the communication efficiency of the compute system, we can obtain even accurate performance of the compute kernel and can further check whether the performance goal is met. If the design goal is not met, we can go back to the block and DFG generation stage altering the design options such as loop unrolling factor. Repeat these steps until the design goal is achieved.

Once the design goal is met, the configuration words can be extracted from the scheduler and be 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 accordingly.

B. Block and DFG Generation

The communication between the processor and the accelerator is costly. When the data size of a DFG is small, data transmission for each DFG execution may compromise all the benefits of the accelerator. To solve this problem, we have the accelerator to repeat the DFG execution multiple times and combine them as a block. Data transmission is performed with the granularity of a block instead of a DFG which helps to amortize the initial communication overhead especially for the DMA transmission. Figure 7 shows the relation among the compute kernel, block and DFG using a simple example.

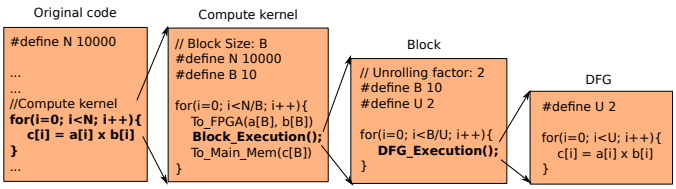


Fig. 7. Relation of Compute Kernel, Block and DFG

Since the SCGRA employs lock-step execution and the input/output data for each block execution must also be fully buffered, the block size is mainly limited by the data buffer size. Given the block size, we need further to decide the loop unrolling factor such that the unrolled part can be transformed to DFG which can be executed on the SCGRA. Usually, the unrolling factor is limited by the instruction memory and data memory.

In addition, 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. Finally, 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 loop count must be fully divided by the block size and the block size must also be fully divided by the unrolling factor. This can be another unrolling and blocking limitation as well.

C. SCGRA Customization

There are a lot of SCGRA design parameters such as operation type, SCGRA size, SCGRA topology and the number of data buffers to be customized. However, we are not going to solve all the customization problems in this work. Instead, we mainly provide implementations of a few different customized SCGRAs and investigate how the *softness* of SCGRA impacts on the overall performance and overhead. In this work, we implemented SCGRAs with different SCGRA size and operation type, while the rest design parameters are fixed.

D. 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 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
```

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.

E. 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, Kmean and Sobel edge detector (Sobel) as our benchmark. To investigate the scalability of the accelerator design methodologies, 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 are illustrated in the Table II.

The benchmark is implemented on Zedboard using both direct HLS based design methodology and QuickDough. Then the design productivity, implementation efficiency, performance and scalability of the two design methodologies 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 was used to transform the compute kernel to hardware IP Catalog i.e. IP core. Vivado 2013.3 was used to integrate the IP core and build the FPGA accelerator. The SCGRA was initially developed in ISE 14.7, and then the ISE project was imported as an IP core in XPS 14.7. With the SCGRA IP core, the SCGRA overlay based FPGA accelerator was further integrated and implemented in PlanAhead 14.7. The accelerators developed using both design methodologies targets at Zedboard [5] and the system works at bare-metal mode.

Direct HLS typically achieves the trade-off between hardware overhead and performance through altering the loop unrolling factors. Larger loop unrolling factors typically promise better simulation performance while more hardware resources like DSP blocks will be required and the implementation frequency may also be affected. In this work, we set the loop unrolling factor large enough to reach the best simulation performance and the detailed loop unrolling setup can be found in Table III. Larger block size is beneficial to data reuse and helps to amortize the initial communication cost, so we set the block size as large as the data buffer size. Detailed block setup of the benchmark is presented in Table III as well.

QuickDough has similar design choices to those of direct HLS based design methodology in terms of loop unrolling and blocking. However, the design constrains using the two design methodologies 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 blocking, the design constrain in QuickDough is relatively more complex. First of all, the block size is also limited by the size of input/output buffer, which is exactly the same with the constrain using direct HLS based design methodology. In addition, the address buffer size can be another major constrain because we need to store all the IO buffer address sequence of the whole block execution instead of the DFG execution. Even though we set address buffer size twice larger than the data buffer size, it can still be a bottleneck in some 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. 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 them can hardly be quantified. In this section, we mainly concentrate on the compilation time while discussing the rest briefly.

In order to implement an application on a CPU + FPGA accelerator system, direct HLS based design methodology roughly consists of the following four steps including compute kernel synthesis, kernel IP generation, accelerator implementation and software compilation.

1) Compute kernel synthesis: High level language program kernel is translated to an HDL model according to the user's synthesis pragma.
2) Kernel IP generation: The compute kernel is synthesized and packed as an IP core. At the same time, the timing constrain is met and the corresponding driver is generated.
3) Accelerator implementation: The IP core is integrated into the accelerator and the whole accelerator is implemented on the FPGA.
4) Software compilation: The application employing the FPGA accelerator is compiled to binary code as conventional software.

Implementing an application using QuickDough also 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

TABLE III
LOOP UNROLLING & BLOCKING SETUP OF ACCELERATORS USING DIRECT HLS BASED DESIGN METHODOLOGY

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	SCGRA Size	Instruction Mem	Data Memory	I/O Data Buffer	Addr Buffer
Torus	2×2, 5×5	1024×72 bits	256×32 bits	2048×32 bits	4096×18 bits

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

volves four steps including DFG generation, DFG scheduling, bitstream generation and software compilation.

1) DFG generation: The high level language program kernel is translated to DFG.
2) DFG scheduling: The DFG is scheduled to the customized SCGRA overlay.
3) Bitstream generation: The scheduling result is further integrated with the pre-built accelerator bitstream to produce the new bitstream for the target application.
4) Software compilation: Application using the SCGRA accelerator is compiled to binary code.

Figure 8 and Figure 9 present the compilation time of implementing the benchmark using both direct HLS based design methodology and QuickDough respectively. IP core generation and hardware implementation in direct HLS based design methodology 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 is negligible. Basically, the direct HLS based design methodology 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 the time consumed doesn't change much across different applications. DFG scheduling is relatively slower especially when the DFG size and SCGRA size are large, but it can still be completed in a few seconds. Typically, QuickDough implements an application in 5 to 15 seconds and it is already two orders of magnitude faster than direct

HLS based design methodology.

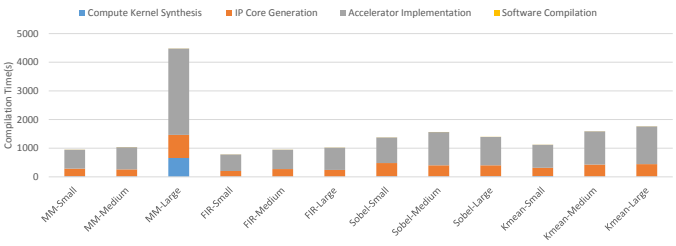


Fig. 8. Benchmark Compilation Time Using Direct HLS Based Design Methodology

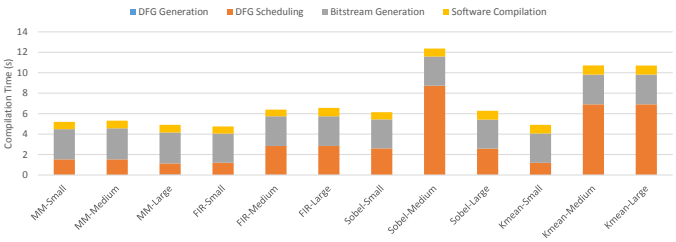


Fig. 9. Benchmark Compilation Time Using QuickDough

On top of the compilation time, the abstraction level of the design entry, design reuse and portability are also important aspects that affect the design productivity. Both direct HLS based design methodology 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. Direct HLS based design methodology needs the compute kernel be synthesized and implemented for each application instance. QuickDough requires compilation for each application instance as well, but it can reuse the same hardware infrastructure across the applications in the same domain. It is possible for direct HLS based design methodology to port the synthesized HDL design among different devices and parts, but IP core generation and accelerator implementation depend on specific FPGA device and they are needed for each application instance. QuickDough's portability is also limited at HDL level, and complete hardware implementation is needed to port to a different FPGA device.

C. Hardware Implementation Efficiency

In this section, hardware implementation efficiency including the hardware resource overhead and implementation frequency of the accelerators using both direct HLS based design methodology and QuickDough are compared. At the same time, SCGRAs with customized operations are implemented and the hardware resource saving is analyzed.

Table VI exhibits the hardware overhead using both accelerator design methodologies. It is clear that the accelerators using direct HLS based design methodology typically consume less FF, LUT and RAM36 due to the delicate customization for each application instance. However, the number of DSP48 required increases significantly 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 which limits the maximum SCGRA that can be implemented on the target FPGA and further constrains the maximum loop unrolling and blocking as well.

TABLE VI
HARDWARE OVERHEAD OF THE ACCELERATORS USING BOTH DIRECT HLS BSED DESIGN METHODOLOGY 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	12
SCGRA 5x5			34922	21436	137	75
FPGA Resource			106400	53200	140	110

To further investigate the hardware overhead, we divided the four benchmarks into three groups to implement customized SCGRAs for each of them. MM and FIR share the same operations, so they are implemented using the same SCGRA overlay. Sobel edge detector doesn't need complete 32-bit data with, and a mixed 16-bit and 32-bit data width SCGRA overlay is customized for it. Kmean which covers almost all the operations adopts the original SCGRA overlay. Figure 10 shows the hardware saving using customized SCGRA overlay. The customized SCGRA overlay can save as much as 65% DSP48, 30% LUT and 15% FF.

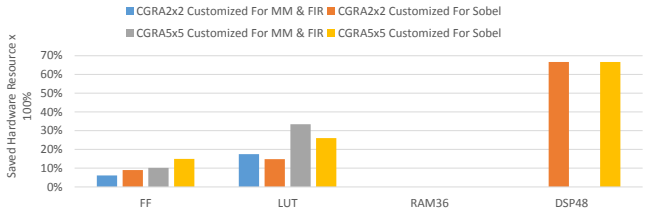


Fig. 10. Hardware Saving Of Customized SCGRA Overlay

Figure 11 presents the implementation frequency of the benchmark using both design methodologies. Direct HLS based design methodology takes timing constrain into consideration at the HLS step, and it can 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. The AXI controller on Zedboard typically works at 100MHz and higher frequency design requires delicate placing and routing. As a result, we set the HLS timing constrain at 100MHz and have the whole accelerator implemented synchronously. Although the current design option is not necessarily optimal, it is representative. In fact, the synthesized IP core sometimes can be even slower though we set the timing at 100MHz during HLS.

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 direct HLS. A 2x2 SCGRA based accelerator can run at 200MHz, and a 5x5 SCGRA based accelerator can work at 167MHz. The implementation frequency degrades slightly because more than 90% of the BRAM blocks on target FPGA are used and the routing becomes extremely tight. As mentioned before, the AXI controller block on Zedboard is slower and runs at 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 clock domains.

D. 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 direct HLS based implementation with 2k-Buffer configuration is used instead. Figure 12 shows the performance comparison of four different sets of accelerator implementations including two accelerators built with QuickDough and two accelerators developed with direct HLS based design methodology. According to this figure, direct HLS based design methodology presents better performance on 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 methodologies achieve similar performance on the rest of the benchmark.

To further investigate the performance of the benchmark, the distribution of the execution time including 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 is presented in Figure 13. Since the execution time of different applications with diverse data sets varies in a large range, the execution time used in this figure is actually normalized to that of a basic software implementation on ARM.

As shown in Figure 13, accelerators using direct HLS based design methodology especially the one with max-buffer configuration achieve better performance mainly through the smaller overhead in communication and 'the others' which are essentially the input/output data reorganization time. And the major reason for the lower communication and data reorganization cost is that it can accommodate larger data sets and corresponding computation for each acceleration execution, which essentially contributes to the large block size as shown in Table III because larger block size increases the data reuse between blocks and amortizes the initial DMA communication cost.

Computation time of QuickDough as illustrated in Figure 14 shows clear advantage over that of direct HLS based design methodology. As the computation time depends on both the simulation performance in cycles and implementation frequency of the hardware infrastructure, it is further analyzed from the two aspects in this section. Figure 15 shows the simulation performance which is the product of the block simulation performance and the number of blocks for each compute kernel. It can be found that direct HLS based design methodology performs better on MM-Large and Sobel while QuickDough outperforms on the rest of the benchmark. Comparing the simulation performance in this figure and loop unrolling factors in Table III and Table V, we can see 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 in the experiments is the Sobel benchmark. And the reason is that direct HLS takes Sobel operator matrices as constant input and has the implementation optimized while the DFG generator in QuickDough just takes them as normal variables and more computations are involved. As for the hardware infrastructure, QuickDough using regular SCGRA overlay typically can run at higher frequency than the circuit generated using direct HLS and the implementation frequency contributes a lot to the advantage of QuickDough computation time.

In summary, the accelerators using direct HLS based design methodology can 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 direct HLS based design methodology achieves better performance. QuickDough using SCGRA overlay can provide both higher simulation performance with larger loop unrolling capability in many cases and higher implementation frequency with its regular structure, so it outperforms when the target application has smaller data set or more intensive computation.

E. Scalability

On top of the design productivity, hardware implementation and performance, the scalability of the accelerator using both design methodologies is equally important. To further investigate the scalability of the two design methodologies, we use matrix multiplication with gradually increasing matrix size as a lightweight benchmark. Since FPGA resource on Zedboard is quite limited, Zc706 with abundant hardware resource is used as the target platform for scalability analysis.

Figure 16 shows the simulation performance of matrix multiplication implemented using both design methodologies. Note that direct HLS with proper unrolling in this figure

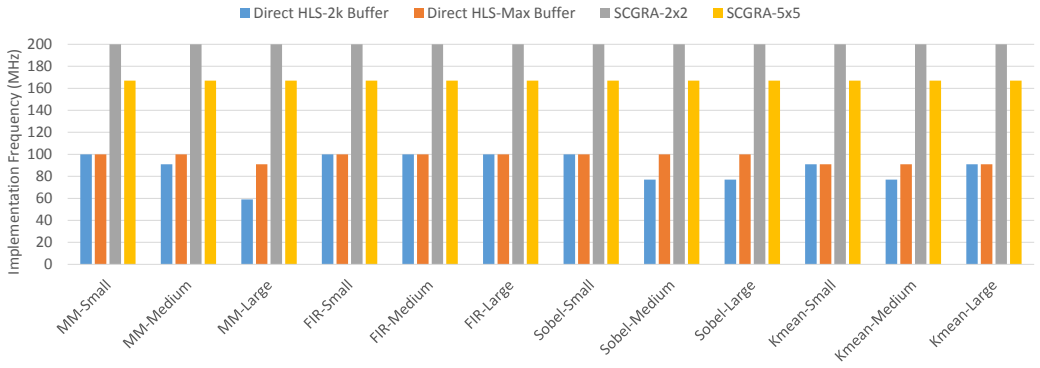


Fig. 11. Implementation Frequency of The Accelerators Using Both Direct HLS Based Design Methodology and QuickDough

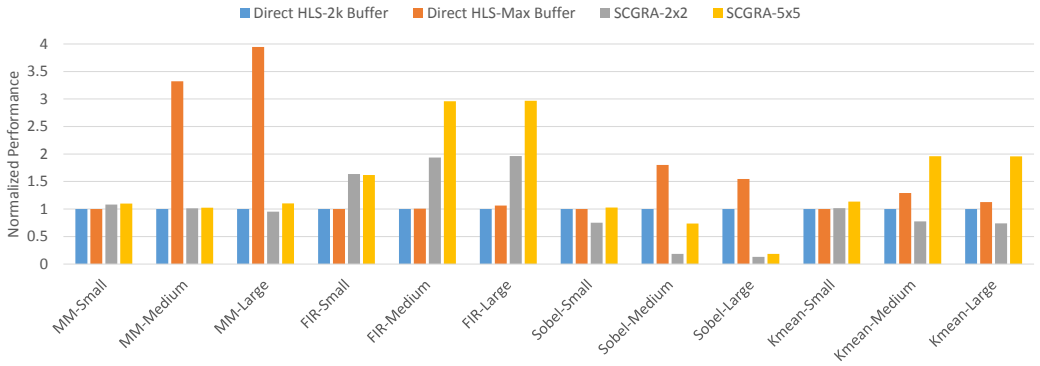


Fig. 12. Benchmark Performance Using Both Direct HLS Based Design Methodology and QuickDough

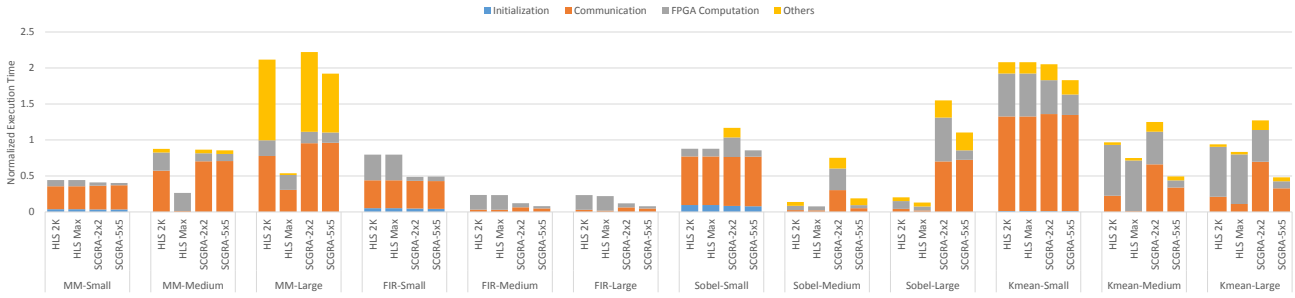


Fig. 13. Benchmark Execution Time Decomposition Of The Accelerators Using Both Direct HLS Based Design Methodology and QuickDough

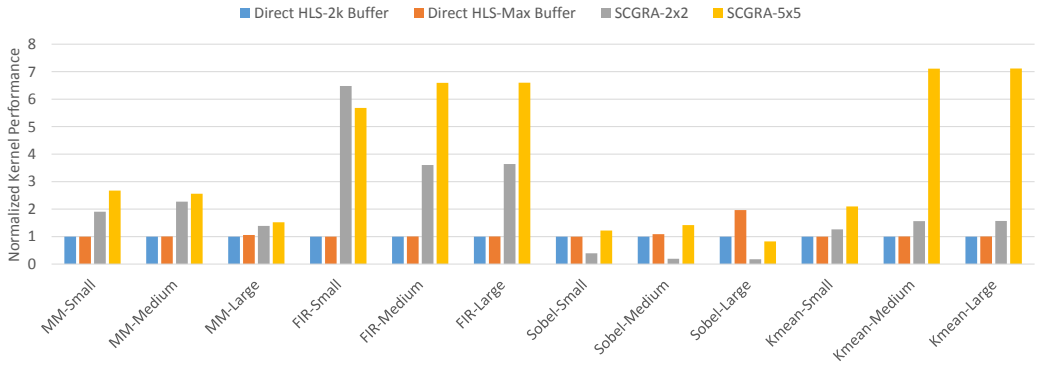


Fig. 14. Compute Kernel Performance Using Both Direct HLS Based Design Methodology and QuickDough

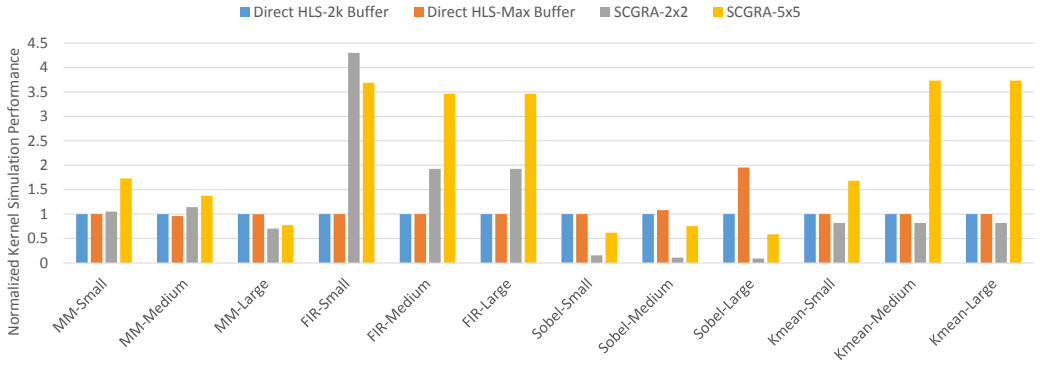
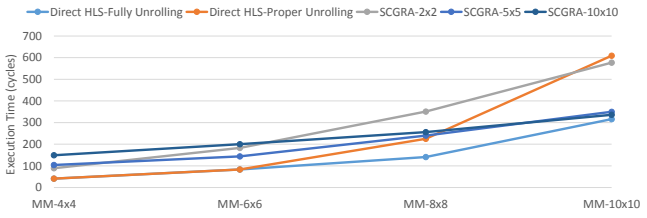
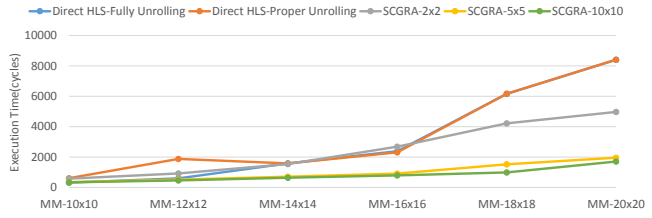


Fig. 15. Compute Kernel Simulation Performance Using Both Direct HLS Based Design Methodology and QuickDough

indicates the minimum loop unrolling that achieves the maximum performance under the hardware constrain. According to the figure, the performance of the accelerators using direct HLS based design methodology is much better than that using QuickDough when the matrix size is small enough for fully loop unrolling. When the matrix size gets larger, direct HLS can no longer afford the hardware overhead for intensive loop unrolling and the performance of the corresponding accelerator starts to degrade. In this experiment, MM-8x8 is the turning point that QuickDough begins to outperform, while the matrix size of the turning point on Zedboard is much smaller because of the limited hardware resource.



(a)



(b)

Fig. 16. Simulation Performance Of The Matrix Multiplication Using Both Direct HLS Based Design Methodology And QuickDough

When the matrix size gets to 18 as shown in Figure 16b, the simulation performance using proper loop unrolling and that using fully loop unrolling starts to overlap. The major reason is that direct HLS develops both loop level parallelism through pipeline and data level parallelism through loop unrolling. When the matrix is small, direct HLS depends more on loop unrolling for performance enhancement. When the matrix is larger, loop level parallelism becomes significant to the performance and may even reach the IO bound. To further prove this statement, we have a 20x20 matrix multiplication synthesized using direct HLS with pipelining and gradually increasing loop unrolling. Figure 17 shows the influence of loop unrolling and pipelining on both performance and hardware overhead. It can be found that loop unrolling typically can improve performance and requires more hardware overhead. When the loop level parallelism is big enough to consume the IO bandwidth, loop unrolling is not necessary for improving the performance. IO bandwidth instead of hardware resource turns to be the performance bottleneck.

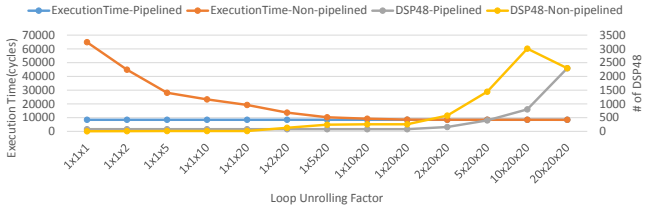


Fig. 17. MM 20x20 Implemented Using Direct HLS Based Design Methodology With Diverse Loop Unrolling And Pipelining

However, as shown in Figure 16b, when the matrix size is larger than 8x8, the accelerator using QuickDough achieves better performance than that using direct HLS with loop unrolling and pipelining. The advantages mainly lie on the following two aspects. On the one hand, SCGRA overlay can accommodate larger loop unrolling and are less prone to reach the hardware resource bottleneck, though it usually consumes larger amount of hardware resource in general. Figure 18 shows the hardware overhead with increasing SCGRA size and it proves its scalability on hardware overhead. On the other hand, the SCGRA overlay has distributed data memory to store temporary data and allows larger loop unrolling. Thus it requires less IO bandwidth compared to the direct HLS based design and is less sensitive to the IO bound.

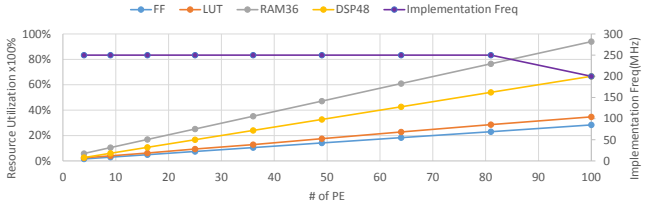


Fig. 18. Hardware Overhead With Increasing SCGRA Size

Finally, we also compare the implementation frequency using both design methodologies. Since the maximum clock

available is 250MHz and the speed level of FPGA on Zc706 is -2, the implementation using both design methodologies present similar implementation frequency. QuickDough allows SCGRA ranging from 2x2 to 9x9 running at 250MHz on Zc706. SCGRA 10x10 degrades slightly and can still work at 200MHz. Direct HLS has all the implementation running at 200MHz.

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 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.

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.