# QuickDough: A Rapid FPGA Accelerator Design Framework using Soft Coarse-Grained Reconfigurable Array Overlay

CHENG LIU, The University of Hong Kong
COLIN YU LIN, The University of Hong Kong
HAYDEN KWOK-HAY SO, The University of Hong Kong

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 regular structure 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.

General Terms: Design, FPGA, Overlay, CGRA

Additional Key Words and Phrases: Overlay, FPGA Accelerator, Soft Coarse Grain Reconfigurable Array, Design Productivity

**ACM Reference Format:**
Cheng Liu, Colin Yu Lin, and Hayden Kwok-Hay So, 2014. QuickDough: A Rapid FPGA Accelerator Design Methodology Using Soft Coarse-Grained Reconfigurable Array Overlay. *ACM Trans. Reconfig. Technol. Syst.* 0, 0, Article 1 ( 2014), 23 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

## 1. INTRODUCTION

By raising the abstraction level of the physical FPGA hardware, numerous researchers have demonstrated the potential of utilizing FPGA overlay architectures as a way to improve designers' productivity[Kingyens & Steffan 2011; Kissler et al. 2006; Lebedev et al. 2010; Severance & Lemieux 2012; Unnikrishnan et al. 2009; Yiannacouras et al. 2009]. Overlay architectures, similar to overlay networks in conventional network designs, are intermediate virtual architectures that are overlaid on top of the physical FPGA configurable fabric with purposes such as to improve design portability, security, and designer's productivity. Overlay architectures may be utilized at different stages of application implementation and have thus manifested in a number of different forms. Examples of overlays developed for such purposes range from parametric HDL models, pre-synthesized or pre-implemented circuits, to multicore processors or even arrays of coarse-grained processing units connected by an on-chip network. In practice, the ad-

This work is supported by xxx
Author's addresses: Cheng Liu, Colin Yu Lin, and Hayden Kwok-Hay So Department of Electrical and Electronic Engineering, The University of Hong Kong, Pokfulam Road, Hong Kong

DOI: http://dx.doi.org/10.1145/0000000.0000000

ditional overlay layer may in some cases result in implementations with less than optimal performance when compared to their hand-optimized alternatives. Nevertheless, the use of overlays remain highly anticipated as many early works have already demonstrated their potential as techniques to improve portability, security, and most importantly, productivity of application designers using FPGA-based reconfigurable computers.

In this work, the QuickDough rapid compilation and synthesis framework for applications targeting hybrid CPU-FPGA systems is presented. The goal of QuickDough is to provide a high-productivity compilation framework for applications that execute on both CPU and FPGA during run time. In particular, it automates the process of generating hardware accelerators and their associated CPU-FPGA communication infrastructure for loop kernels that are designated for FPGA acceleration by the user. By utilizing a soft coarse-grained reconfigurable array (SCGRA) as an overlay, QuickDough is able to generate the configuration bitstream for the targeted platform rapidly in the order of seconds.

QuickDough achieves this speed by first translating compute kernels into data flow graphs (DFGs), which are then statically scheduled onto the target overlay. It then integrates the scheduling result with a pre-implemented bitstream of the overlay to produce the final configuration bitstream with both the accelerator and its communication infrastructure.

When compared to a typical FPGA implementation flow that includes lengthy steps such as synthesis and place-and-route, the run time of QuickDough is almost 2 orders of magnitude shorter when compiling our benchmark programs. Although the implementation of the overlay must eventually rely on conventional hardware design tools, only one instance of the implementation is required per application or application domain throughout the design iterations. The lengthy hardware implementation process can therefore be amortized as the number of compile-debug-edit cycle increases.

Most importantly, despite the introduction of an overlay, the end-to-end run time of our benchmark in the generated system is comparable to the accelerators that are generated by a typical HLS flow. Furthermore, as the proposed overlay structure is very regular with short connections, the resulting implementation is highly scalable. In our experiments, the size of the proposed overlay can easily be scaled without major impact on implementation frequency. Our scheduler can also take advantage of the additional processing elements of larger overlays to improve execution time of most benchmark. Currently, the key limitation to the array size is hardware overhead due to the additional memory requirement of the proposed processing element array.

After exploring related work in next section, the QuickDough design methodology will be elaborated in Section 3. The design and implementation of the overlay will then be presented in Section 4 and the compilation framework will be illustrated in Section 5. In Section 6, experimental results will be shown. Finally, we will discuss the benefit and limitations of current implementation in Section 7 and conclude in Section 8.

## 2. RELATED WORK

Despite their promising performance advantage, the relatively low design productivity of developing FPGA applications remains a major obstacle that hinders widespread adoption of FPGAs as commodity computing devices. To address this problem, the design of QuickDough was inspired by the recent success in high-level synthesis tools. It also took advantage of modern FPGAs' capabilities to allow for an additional overlay architecture be employed for productivity sake.

### 2.1. High-Level Synthesis

To bridge the design productivity gap between software and hardware application development, many researchers have turned to the use of high-level synthesis (HLS) techniques [Cong et al. 2011]. By raising the abstraction level of the physical hardware, HLS allows designers to express hardware designs using familiar high-level, software-like description languages such as C, Java, or Python [Canis et al. 2011; Cardoso et al. 2010]. The low-level hardware implementations are then left to the tools to synthesize and optimize. Indeed, with decades of research, some early results in HLS have already found their ways into FPGA vendors' commercial tools in recent years [Chen et al. 2005; Xilinx 2014; Zhang et al. 2008].

Unfortunately, when considering the overall design productivity of developing hybrid software-gateware applications, the raised abstraction provided by HLS is only addressing part of the problem. While the high-level abstraction makes expressing complex functionalities as FPGA gateware easier, the lengthy low-level compilation time spent in synthesis, mapping, placing and routing remains a bottleneck to the overall design productivity for an application designer. Such long compilation time is particularly challenging for novice designers who are accustomed to the high speed of software compilation. Most importantly, it is significantly impacting the possible compile-debug-edit cycle achievable per day by a designer, negatively impacting the productivity of the designer.

### 2.2. Overlay Architectures

To improve the speed of low-level implementation tools, researchers have explored various approaches over the past decades. Inspired by application specific integrated circuit (ASIC) design flows, researchers and vendors have developed modular design flow and explored the use of pre-compiled hard macros [Lavin et al. 2010, 2011] as implementation library. In addition, researchers have also exploited the use of dynamic partial reconfiguration capabilities in FPGAs [Frangieh et al. 2010] as a way to improve productivity. In recent years, there has been an increased interest in applying the concept of *overlay architectures* as a way to address this productivity challenge.

An overlay architecture is a virtual intermediate architecture that is overlaid on top of the physical configurable fabric of an FPGA. They are employed during the FPGA application implementation process for purposes such as to improve portability, security, and also productivity.

One of the most familiar categories of overlay consists of virtual FPGAs [Brant & Lemieux 2012; Coole & Stitt 2010; Grant et al. 2011; Koch et al. 2013]. They are built either virtually or physically on top of off-the-shelf FPGA devices and typically feature coarser configuration granularity than the physical device. Similar to virtual machines running on a typical computer, such virtual FPGA provides an additional layer that improves application portability and security. Furthermore, because of the coarser-grained configurable fabric, implementing designs on such overlay is relatively easier than on a fine-grained device. However, the additional layer imposes restrictions on the underlying fabrics' capability and usually results in moderate hardware overhead and timing degradation.

Another category of overlay architecture commonly employed is in the form of coarse-grained reconfigurable arrays (CGRAs). The use of CGRAs provides unique advantages of compromising hardware implementation and performance especially for compute intensive applications as demonstrated by numerous ASIC CGRAs [Tessier & Burleson 2001] [Compton & Hauck 2002]. Indeed, CGRAs on FPGA and ASIC have many similarities in terms of the scheduling algorithm and array structure. However, they have quite different trade-offs in terms of configuration flexibility, overhead and

performance. In a nutshell, CGRAs on ASIC emphasize more on configuration capability to cover more applications, while FPGAs' inherent programmability greatly alleviates the concern. Instead, CGRAs on FPGA may take advantage of the configurability of the underlying fabric to allow more intensive customization tailored to the target application.

The authors in [Kissler et al. 2006] developed WPPA (weakly programmable processor array), a VLIW architecture based parameterizable CGRA overlay. It featured an interconnection wrapper unit for each processing element (PE) that could be used for dynamic CGRAs topology customization. Unfortunately, programming and compilation on WPPA were not presented. The authors in [Ferreira et al. 2011] proposed a heterogeneous CGRA overlay with a global multi-stage interconnection on FPGA. Compiling applications onto the overlay took only milliseconds for smaller DFGs. However, the global multi-stage interconnection required multiple stages for communication between each pair of PEs and resulted in either low implementation frequency or large communication latency in terms of cycles. In addition, there was no intermediate storage except the pipeline registers in the CGRA and it limited the performance of the operation scheduling. In [Shukla et al. 2006], a customized CGRA overlay called QUKU was developed for DSP algorithms. It had two-level configuration capability, while the low-speed configuration was used for operator reuse within an application and high-speed reconfiguration was used for optimization between different applications. Nevertheless, the hardware infrastructure was consist of simple operation elements which can only be adapted to a few specified DSP algorithms. The authors in [Capalija & Abdelrahman 2013] built a more generic high speed mesh CGRA overlay using the elastic pipeline technique to achieve the maximum throughput. It adopted a data-driven execution flow and was suitable for smaller pipelined DFG execution, while it would be difficult to handle applications with random IO access.

In general, previous CGRA overlays have demonstrated the promising performance acceleration capability for compute intensive applications. They typically take DFG as design entry and focus on hardware infrastructure design as well as corresponding mapping and scheduling. However, they are still lack of consideration on proper loop unrolling for DFG generation, on-chip buffering, the communication with host and even end-to-end performance which are essential for FPGA accelerator design especially from a HW/SW co-design engineer's perspective.

Finally, a third category of overlay features soft-processor-like architectures with high degree of control and data parallelism suitable for FPGA accelerations. For example, in the work of MARC [Lebedev et al. 2010], a many-core overlay with customizable data path was proposed. Similarly, a GPU-like overlay was proposed in [Kingyens & Steffan 2011].

In this work, we opted to utilize a fully pipelined synchronous soft coarse-grained reconfigurable array (SCGRA) as an overlay to facilitate rapid FPGA accelerator generation in a hybrid CPU-FPGA system.

Compared to previously proposed CGRAs, our overlay is designed to be *soft* as the size, processing element designs, as well as the interconnect topologies may all be customized as needed. It also takes advantage of the large number of on-chip distributed memory on the FPGA for intermediate data storage and can handle large DFGs with thousands of nodes. Finally, the design of our overlay is highly scalable as it is regular with only short direct connections between PEs. It makes the proposed overlay easily portable among FPGAs with different sizes and scalable with the clock frequency improvements in future devices.

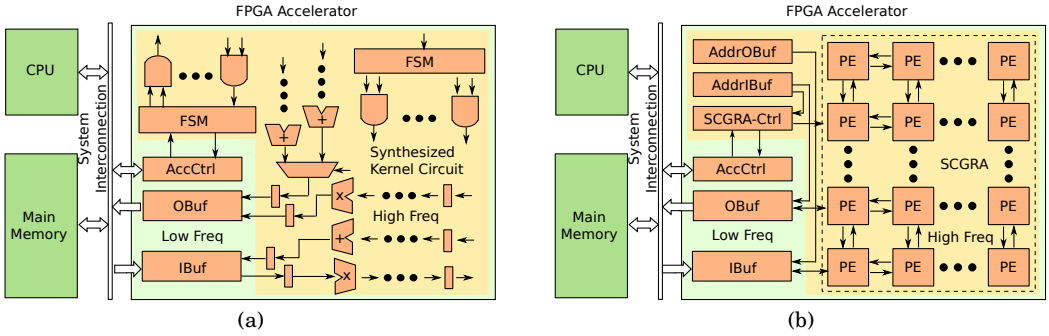


Fig. 1. SCGRA Based FPGA Accelerator and HLS Based FPGA Accelerator

## 3. QUICKDOUGH FRAMEWORK

QuickDough is a development framework for FPGA-accelerated applications. It generates FPGA accelerators for user-specific compute intensive kernels rapidly through the use of a soft coarse-grained reconfigurable array (SCGRA) overlay. QuickDough also generates the communication infrastructure between the CPU host and the accelerator automatically, integrating both software and hardware generation in a unified framework.

The goal of QuickDough is to improve designers' productivity in developing mixed hardware-software applications by significantly reducing the time to generate FPGA accelerators. By using an SCGRA overlay, the QuickDough framework is able to produce a complete accelerated hardware-software system in the order of seconds. This software-like compilation speed allows user to iterate on design through rapid debug-edit-implement cycles.

Figure 1a shows the design of a typical accelerator system. In such system, on-chip memory are used to buffer data between the host CPU and the accelerator. A controller is also presented in hardware to control the operations of the accelerator as well as memory transfers. The entire design must be reimplemented every time a change is made to the accelerator design, going through the lengthy low-level hardware implementation tool flow. Also, users must manually manage the data transfer between CPU and the accelerator, implementing both the software and the communication hardware at the same time.

On the other hand, Figure 1b shows the system generated by QuickDough. While it features a similar overall design as a typical accelerator system, it differs in 2 important ways. First, instead of relying high-level synthesis tools to generate the acceleration circuits, QuickDough utilizes an SCGRA overlay to implement the computation. In addition, QuickDough systematically manages data transfer between the CPU and the accelerator. As a result, users only have to specify the region for acceleration and QuickDough will be able to generate the rest of the system automatically and rapidly.

Figure 2 summarizes the hardware-software compilation flow of QuickDough. Users begin by specifying the regions for accelerations in the form of compute intensive loops. Once these loops are identified, the QuickDough framework proceeds with two paths corresponding to software and hardware generations.

The first path of QuickDough compiles the overall software application. The compute kernels are replaced by calls to the accelerator drivers. Also generated in software are routines that controls and transfers data to and from the accelerator.

The second path of QuickDough is the focus of this work where the hardware accelerator is generated. To begin, the compute kernel loops are statically analyzed to

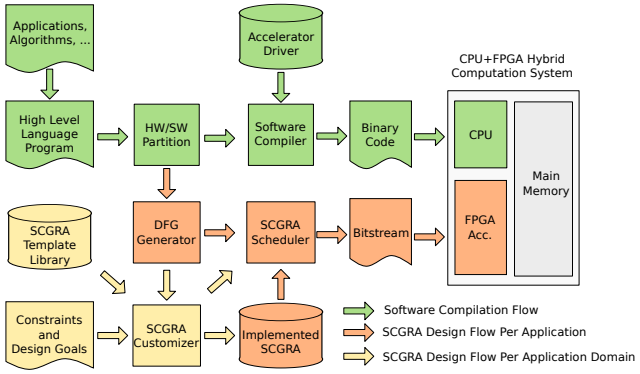


Fig. 2. QuickDough: FPGA Accelerator Design Methodology Using SCGRA Overlay

produce their corresponding data flow graphs (DFGs). These DFGs form the basis for accelerator generation, and are used to guide the operation of an SCGRA customizer that determines configuration of application-specific SCGRA overlays. As an overlay, most architectural parameters may optionally be customized, including the processing elements' operation, pipeline depth, size of array, as well as on-chip buffer size. While customization may lead to better power-performance for the final system, if the SCGRA configuration is not already available in the pre-implemented SCGRA library, lengthy hardware implementation must need to be performed. Therefore, the user may choose to perform customization only when needed. Once the overlay design is determined, the DFG is scheduled on the overlay with the scheduling result embedded directly into the SCGRA overlay to create the final FPGA configuration bitstream. This bitstream, in combination with the software created in the first path, forms the final application that will be executed during run time.

## 4. THE QUICKDOUGH OVERLAY

One key factor that determines the accelerator's performance rests on the design of the overlay. While the use of an overlay improves compilation speed and designers' productivity, the QuickDough framework will not be as useful if the performance of the generated system is not at least on par with similar systems created with conventional high-level synthesis tools. For that, QuickDough chose to utilize an overlay that is *simple*, *scalable* and *deterministic*.

The QuickDough overlay consists of an array of simple processing elements (PEs) connected by a direct network executing synchronously (Figure 1b). Each PE computes and forwards data in lock steps, allowing deterministic multi-hop data communication that overlaps with computations. The action of each PE in each cycle is controlled by an instruction ROM that is populated with instructions generated by the compiler. Finally, a data memory is featured on each PE to serve as a temporary storage for run-time data that may be reused in the same PE or be forwarded in subsequent steps.

Communication between the accelerator and the host processor is carried through a pair of input/output buffers. Accesses to these I/O buffers from the SCGRA array take place in lock step with the rest of the system. The exact buffer location to be accessed is control by the AddrIBuf and AddrOBuf blocks. Both of them are ROM populated with address information generated from the QuickDough compiler.

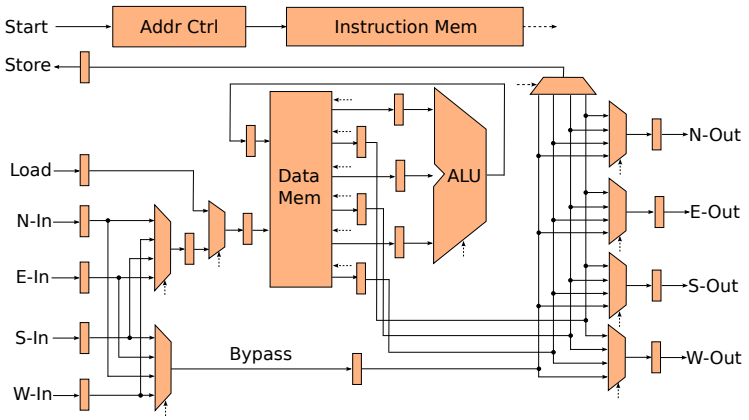


Fig. 3. Fully Pipelined PE structure

### 4.1. Reconfiguration

There are two levels of reconfiguration that may be applied to the overlay to address different application needs. The first and the quickest form of reconfiguration keeps the physical implementation of the overlay intact. To modify the function of the implemented hardware, the QuickDough overlay can be configured by changing (i) the content of the instruction ROM of each PE, (ii) the content of the input/output buffer, (iii) the content of the I/O buffer address ROM AddrIBuf and AddrOBuf, and (iv) the accelerator control AccCtrl. Among these 4 aspects, (i) and (iii) are modified by replacing the ROM content of the bitstream in-place using tools such as `data2mem`. On the other hand (ii) and (iv) are controlled by software during run-time. As a result, all 4 types of customization can be performed rapidly within seconds. They allow the same overlay implementation to be targeted to different applications as well as to different loop iterations of the compute kernel easily.

A second level of customization can be applied to the overlay implementation itself. As a *soft* overlay, many aspects of the array can be customized according to the input application to achieve different tradeoffs in area, power and performance. Customizations may involve, the size of the array, the type of supported operation in each PE, the size of data and instruction memory, and even the pipeline depth of the network and the PEs. When compared to the first level of customization, this level of customization involves reimplementation of the overlay and requires considerably longer run time. User may therefore opt for this level of customization only as needed. Note that in all cases, the overlay remains synchronous and deterministic to keep the overall flow of QuickDough intact.

### 4.2. Processing Element (PE)

The key design element of the QuickDough overlay is its processing element (PE). On one hand, the design of the PE must be simple with low overhead to reduce area and power consumption, and to improve performance. On the other hand, the design of PE must also be flexible enough that it can support all required operations in the target application.

Figure 3 shows the current implementation of a QuickDough PE that features an optional load/store path. At the heart of the PE is an ALU, which is supported by a multi-port data memory and an instruction memory. Three of the data memory's read ports are connected to the ALU as inputs, while the remaining ports are sent to the output multiplexors for connection to neighboring PEs and the optional store path to

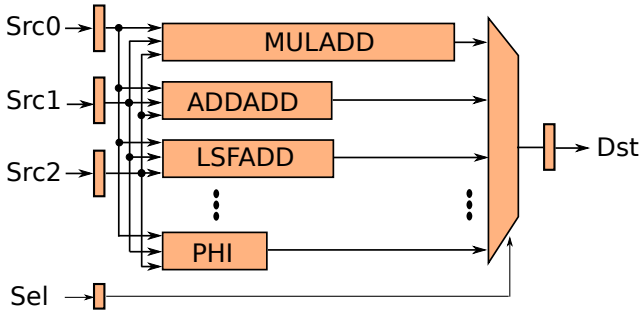


Fig. 4. The QuickDough ALU. It supports up to 16 fully pipelined 3-input operations.

OBuf external to the PE. At the same time, this data memory takes input from the ALU output, data arriving from neighboring PEs, as well as from the optional IBuf loading path. The action of the PE is controlled by the AddrCtrl unit that reads from the instruction memory. Finally, a global signal from the AccCtrl block controls the start/stop of all PEs in the array.

*4.2.1. Instruction Memory and Data Memory.* The instruction memory stores all the control words of the PE. As its content does not change at runtime, a ROM is used to implement this instruction memory. The address of the instruction ROM is determined by the AddrCtrl. Once the global `start` signal is valid, the ROM address will increase by one every cycle and the SCGRA execution will proceed accordingly. When the start signal is invalid, the ROM address will be reset to be 0 and the SCGRA execution will stop.

Data memory stores intermediate data that can either be forwarded to the neighboring PEs or be sent to the ALU for calculation. To support non-blocking operations in the PE, at least 4 read and 2 write ports are needed. In each cycle, 3 reads are needed for the ALU and 1 read is needed for data forwarding. At the same time, one write port is needed to store input data from neighboring PEs and another one is needed to store the computing result of the ALU within the same cycle. Currently, 3 true dual port memory blocks that contain replicated data are employed to implement this data memory.

*4.2.2. ALU.* At the heart of the proposed PE is the ALU that carries out the computations of the given application. As an overlay, the QuickDough ALU must be simple, regular, and flexible such that it may easily be customized with different operations specifically for any given user application. In addition, it must also be fully pipelined in order to achieve high clock frequency and thus higher overall performance. Figure 4 shows the current design of the ALU used in the QuickDough overlay.

The QuickDough ALU supports up to 16 fully pipelined 3-input operations. Depending on the area-performance requirements, the ALU may be customized with operations specifically designed for the application. It may also be customized to support the common set of operations for multiple compute kernels. For example, Table I shows the set of operations we have developed for all the benchmarks in Section 6. Figure 4 shows the current set of operation.

These operators in the ALU may execute concurrently in a pipelined fashion and must complete in a deterministic number of cycle. Given the deterministic nature of the operators, the QuickDough scheduler will ensure that there is never conflict at the output multiplexor.

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 |

### 4.3. Load/Store Interface

For the PEs that also serve as IO interface to the SCGRA, they have an additional load path and a store path as shown in 3. The data loading 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. Similarly, the data storing path has an additional data multiplexor as well, but it doesn't influence the pipeline of the design.

## 5. SCGRA COMPILATION

### 5.1. Overview Of SCGRA Compilation

Figure 5 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.

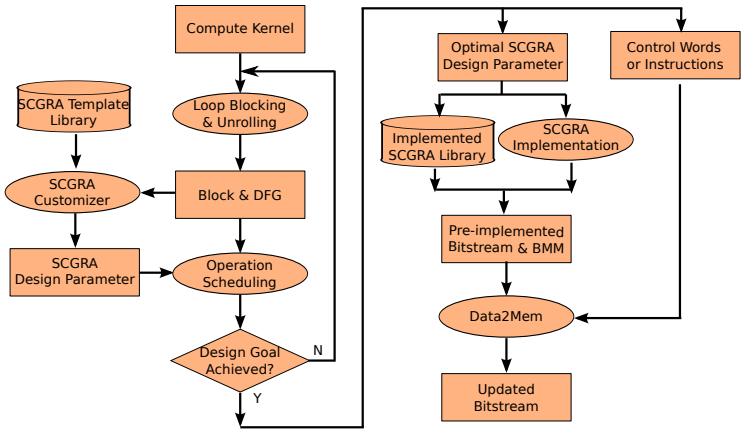


Fig. 5. SCGRA Compilation

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.

### 5.2. 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 6 shows the relation among the compute kernel, block and DFG using a simple example.

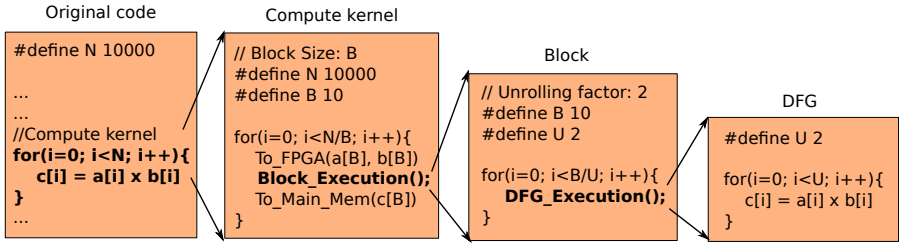


Fig. 6. Relation of Compute Kernel, Block and DFG

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.

### 5.3. 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.

### 5.4. Operation Scheduler

The operation scheduler adopts a classical list scheduling algorithm [Schutten 1996] 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 [Yu 2012] 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.

### 5.5. 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 [Xilinx 2012] 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 [Beckhoff et al. 2011] 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.

## 6. 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(S), Medium(M) to Large(L) ones. The basic parameters and configurations of the benchmark are illustrated in Table II and the complete loop structure of the benchmark is presented in Table III.

Table II. Detailed Configurations of the Benchmark

| Benchmark | MM | FIR | Sobel | Kmean |
|---|---|---|---|---|
| Parameters | Matrix Size | # of Input<br># of Taps+1 | # of Vertical Pixels<br># of Horizontal Pixels | # of Nodes<br># of Centroids<br>Dimension Size |
| S | 10 | 40/50 | 8/8 | 20/4/2 |
| M | 100 | 10000/50 | 128/128 | 5000/4/2 |
| L | 1000 | 100000/50 | 1024/1024 | 50000/4/2 |

Table III. Complete Loop Structure of the Benchmark

| Benchmark | MM | FIR | Sobel | Kmean |
|---|---|---|---|---|
| S | $10 \times 10 \times 10$ | $40 \times 50$ | $8 \times 8 \times 3 \times 3$ | $20 \times 4 \times 2$ |
| M | $100 \times 100 \times 100$ | $10000 \times 50$ | $128 \times 128 \times 3 \times 3$ | $5000 \times 4 \times 2$ |
| L | $1000 \times 1000 \times 1000$ | $100000 \times 50$ | $1024 \times 1024 \times 3 \times 3$ | $50000 \times 4 \times 2$ |

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.

### 6.1. 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 [Avnet 2014] 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 IV. 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 IV as well.

Table IV. Loop Unrolling & Blocking Setup Of Accelerators Using Direct HLS Based Design Methodology

| Application | | Max-Buffer | | 2k-Buffer | |
|---|---|---|---|---|---|
| | | Unrolling Factor | Block Structure | Unrolling Factor | Block Structure |
| MM | S | $2 \times 10 \times 10$ | $10 \times 10 \times 10$ | $2 \times 10 \times 10$ | $10 \times 10 \times 10$ |
| | M | $1 \times 1 \times 100$ | $100 \times 100 \times 100$ | $1 \times 100$ | $10 \times 100$ |
| | L | $1 \times 500$ | $50 \times 1000$ | 500 | 1000 |
| FIR | S | $2 \times 50$ | $40 \times 50$ | $2 \times 50$ | $40 \times 50$ |
| | M | $2 \times 50$ | $10000 \times 50$ | $2 \times 50$ | $1000 \times 50$ |
| | L | $2 \times 50$ | $50000 \times 50$ | $2 \times 50$ | $1000 \times 50$ |
| Sobel | S | $1 \times 2 \times 3 \times 3$ | $8 \times 8 \times 3 \times 3$ | $1 \times 2 \times 3 \times 3$ | $8 \times 8 \times 3 \times 3$ |
| | M | $1 \times 1 \times 3 \times 3$ | $128 \times 128 \times 3 \times 3$ | $1 \times 1 \times 3 \times 3$ | $23 \times 128 \times 3 \times 3$ |
| | L | $1 \times 1 \times 3 \times 3$ | $75 \times 1024 \times 3 \times 3$ | $1 \times 1 \times 3 \times 3$ | $4 \times 1024 \times 3 \times 3$ |
| Kmean | S | $20 \times 4 \times 2$ | $20 \times 4 \times 2$ | $20 \times 4 \times 2$ | $20 \times 4 \times 2$ |
| | M | $5 \times 4 \times 2$ | $5000 \times 4 \times 2$ | $5 \times 4 \times 2$ | $1000 \times 4 \times 2$ |
| | L | $5 \times 4 \times 2$ | $25000 \times 4 \times 2$ | $5 \times 4 \times 2$ | $1000 \times 4 \times 2$ |

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 V and Table VI respectively.

Table V. SCGRA Configuration

| SCGRA Topology | SCGRA Size | Instruction Mem | Data Memory | I/O Data Buffer | Addr Buffer |
|---|---|---|---|---|---|
| Torus | $2 \times 2$, $5 \times 5$ | $1024 \times 72$ bits | $256 \times 32$ bits | $2048 \times 32$ bits | $4096 \times 18$ bits |

### 6.2. 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.

- Compute kernel synthesis: High level language program kernel is translated to an HDL model according to the user's synthesis pragma.

Table VI. Loop Unrolling and Blocking Setup For Accelerators Using QuickDough

| Application | | SCGRA 2x2 | | | SCGRA 5x5 | | |
|---|---|---|---|---|---|---|---|
| | | Unrolling Factor | DFG (OP/IO) | Block Structure | Unrolling Factor | DFG (OP/IO) | Block Structure |
| MM | S | $10 \times 10 \times 10$ | 1000/301 | $10 \times 10 \times 10$ | $10 \times 10 \times 10$ | 1000/301 | $10 \times 10 \times 10$ |
| | M | $5 \times 100$ | 750/606 | $10 \times 100$ | $5 \times 100$ | 750/606 | $10 \times 100$ |
| | L | 200 | 301/402 | 1000 | 200 | 301/402 | 1000 |
| FIR | S | $40 \times 50$ | 860/131 | $40 \times 50$ | $40 \times 50$ | 860/131 | $40 \times 50$ |
| | M | $20 \times 50$ | 1000/141 | $100 \times 50$ | $50 \times 50$ | 2500/201 | $250 \times 50$ |
| | L | $20 \times 50$ | 1000/141 | $100 \times 50$ | $50 \times 50$ | 2500/201 | $250 \times 50$ |
| Sobel | S | $4 \times 8 \times 3 \times 3$ | 1080/39 | $8 \times 8 \times 3 \times 3$ | $8 \times 8 \times 3 \times 3$ | 2160/55 | $8 \times 8 \times 3 \times 3$ |
| | M | $4 \times 8 \times 3 \times 3$ | 1080/39 | $8 \times 8 \times 3 \times 3$ | $23 \times 8 \times 3 \times 3$ | 6210/115 | $65 \times 8 \times 3 \times 3$ |
| | L | $4 \times 4 \times 3 \times 3$ | 540/31 | $16 \times 4 \times 3 \times 3$ | $16 \times 4 \times 3 \times 3$ | 2160/55 | $16 \times 4 \times 3 \times 3$ |
| Kmean | S | $20 \times 4 \times 2$ | 920/62 | $20 \times 4 \times 2$ | $20 \times 4 \times 2$ | 920/62 | $20 \times 4 \times 2$ |
| | M | $25 \times 4 \times 2$ | 1144/72 | $125 \times 4 \times 2$ | $125 \times 4 \times 2$ | 5768/272 | $500 \times 4 \times 2$ |
| | L | $25 \times 4 \times 2$ | 1144/72 | $125 \times 4 \times 2$ | $125 \times 4 \times 2$ | 5768/272 | $500 \times 4 \times 2$ |

- 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.
- Accelerator implementation: The IP core is integrated into the accelerator and the whole accelerator is implemented on the FPGA.
- Software compilation: The application employing the FPGA accelerator is compiled to binary code as conventional software.

Implementing an application using QuickDough also involves four steps including DFG generation, DFG scheduling, bitstream generation and software compilation.

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

Figure 7 and Figure 8 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.

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

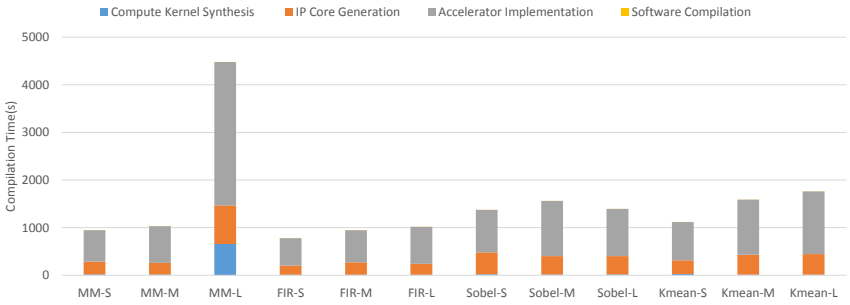


Fig. 7. Benchmark Compilation Time Using Direct HLS Based Design Methodology

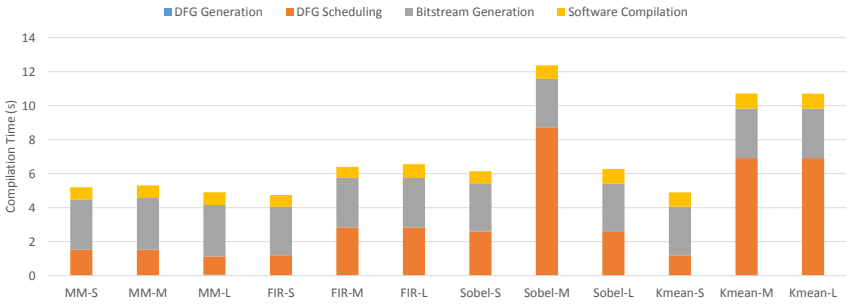


Fig. 8. Benchmark Compilation Time Using QuickDough

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.

### 6.3. 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 VII 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.

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

Table VII. 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 |

data width SCGRA overlay is customized for it. Kmean which covers almost all the operations adopts the original SCGRA overlay. Figure 9 shows the hardware saving using customized SCGRA overlay. The customized SCGRA overlay can save as much as 65% DSP48, 30% LUT and 15% FF.

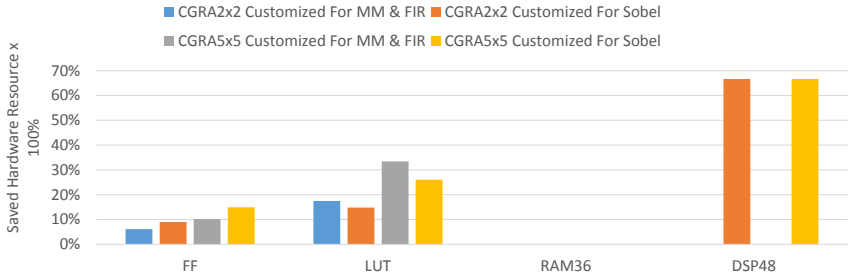


Fig. 9. Hardware Saving Of Customized SCGRA Overlay

Figure 10 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.

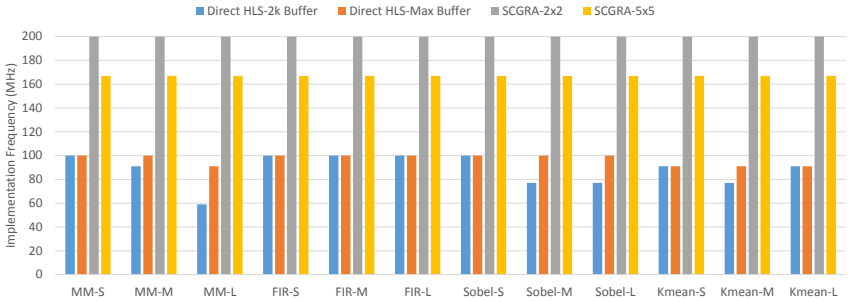


Fig. 10. Implementation Frequency of The Accelerators Using Both Direct HLS Based Design Methodology and QuickDough

### 6.4. 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 11 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.

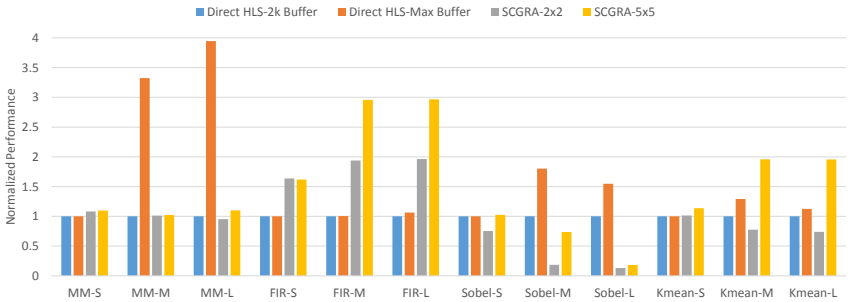


Fig. 11. Benchmark Performance Using Both Direct HLS Based Design Methodology and QuickDough

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 12. 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 12, 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 IV because larger block size increases the data reuse between blocks and amortizes the initial DMA communication cost.

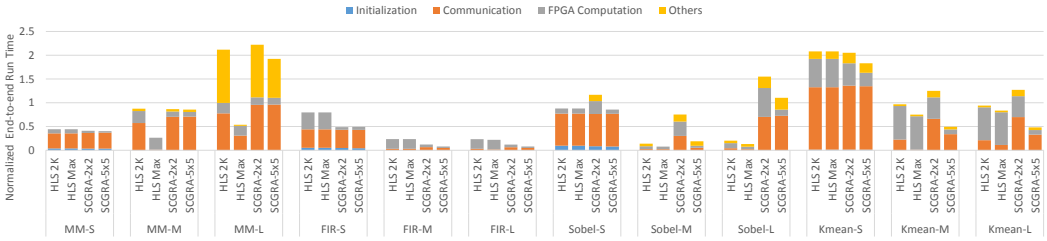


Fig. 12. Benchmark Execution Time Decomposition Of The Accelerators Using Both Direct HLS Based Design Methodology and QuickDough

Computation time of QuickDough as illustrated in Figure 13 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 14 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 IV and Table VI, 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

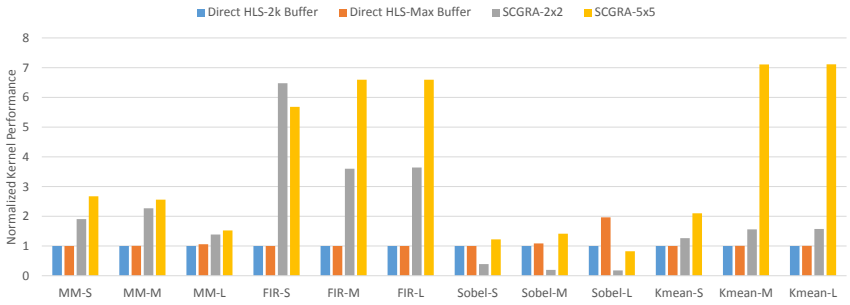


Fig. 13. Compute Kernel Performance Using Both Direct HLS Based Design Methodology and QuickDough

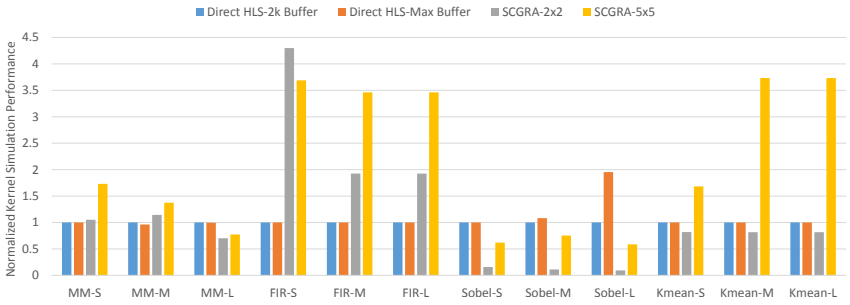


Fig. 14. Compute Kernel Simulation Performance Using Both Direct HLS Based Design Methodology and QuickDough

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.

### 6.5. 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 15 shows the simulation performance of matrix multiplication implemented using both design methodologies. Note that direct HLS with proper unrolling in this figure 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.

When the matrix size gets to 14 as shown in Figure 15b, the simulation performance using proper loop unrolling and that using fully loop unrolling starts to overlap. The

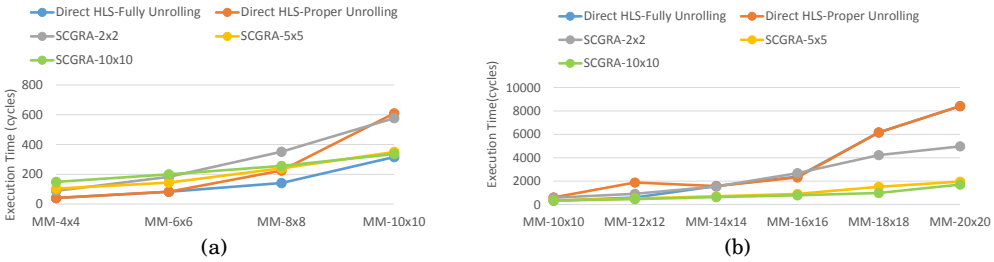


Fig. 15. Simulation Performance Of The Matrix Multiplication Using Both Direct HLS Based Design Methodology And QuickDough

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 16 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.

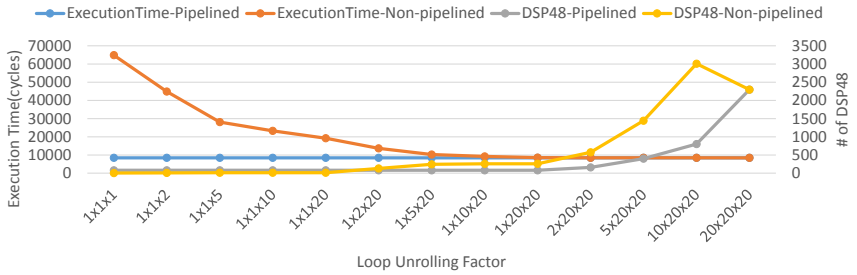


Fig. 16. MM 20x20 Implemented Using Direct HLS Based Design Methodology With Diverse Loop Unrolling And Pipelining

However, as shown in Figure 15b, 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 17 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.

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

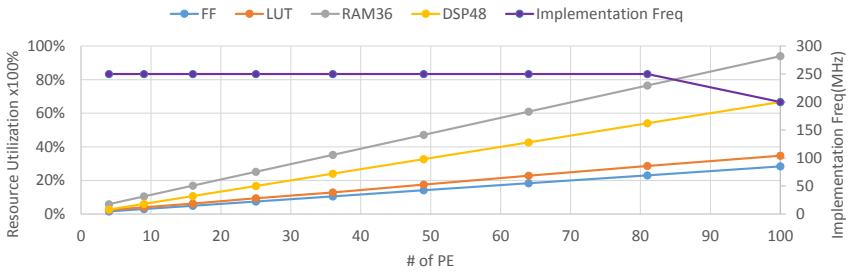


Fig. 17. Hardware Overhead With Increasing SCGRA Size

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.

## 7. 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.

## 8. 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

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

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

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

Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Brown, S., & Czajkowski, T. (2011). LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, (pp. 33–36)., New York, NY, USA. ACM.

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

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

Chen, D., Cong, J., Fan, Y., Han, G., Jiang, W., & Zhang, Z. (2005). xpilot: A platform-based behavioral synthesis system. *SRC TechCon*, *5*.

Compton, K. & Hauck, S. (2002). Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (csuR)*, *34*(2), 171–210.

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

Coole, J. & Stitt, G. (2010). 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*, (pp. 13–22).

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

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

Grant, D., Wang, C., & Lemieux, G. G. (2011). 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, (pp. 123–132)., New York, NY, USA. ACM.

Kingyens, J. & Steffan, J. G. (2011). The potential for a gpu-like overlay architecture for fpgas. *Int. J. Reconfig. Comp.*, *2011*.

Kissler, D., Hannig, F., Kupriyanov, A., & Teich, J. (2006). A dynamically reconfigurable weakly programmable processor array architecture template. In *ReCoSoC*, (pp. 31–37).

Koch, D., Beckhoff, C., & Lemieux, G. (2013). An efficient FPGA overlay for portable custom instruction set extensions. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, (pp. 1–8).

Lavin, C., Padilla, M., Ghosh, S., Nelson, B., Hutchings, B., & Wirthlin, M. (2010). Using hard macros to reduce FPGA compilation time. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, (pp. 438–441). IEEE.

Lavin, C., Padilla, M., Lamprecht, J., Lundrigan, P., Nelson, B., & Hutchings, B. (2011). HMFlow: Accelerating FPGA compilation with hard macros for rapid prototyping.

In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, (pp. 117 –124).

Lebedev, I., Cheng, S., Doupnik, A., Martin, J., Fletcher, C., Burke, D., Lin, M., & Wawrzynek, J. (2010). MARC: A many-core approach to reconfigurable computing. In *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, (pp. 7 –12).

Schutten, J. (1996). List scheduling revisited. *Operations Research Letters*, *18*(4), 167–170.

Severance, A. & Lemieux, G. (2012). Venice: A compact vector processor for fpga applications. In *Field-Programmable Technology (FPT), 2012 International Conference on*, (pp. 261–268).

Shukla, S., Bergmann, N., & Becker, J. (2006). Quku: a two-level reconfigurable architecture. In *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*.

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

Unnikrishnan, D., Zhao, J., & Tessier, R. (2009). Application specific customization and scalability of soft multiprocessors. In *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, (pp. 123–130).

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

Xilinx (2014). Vivado hls. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/. [Online; accessed 18-October-2014].

Yiannacouras, P., Steffan, J. G., & Rose, J. (2009). 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, (pp. 97–106)., New York, NY, USA. ACM.

Yu, Colin Lin. So, H. K.-H. (2012). 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.

Zhang, Z., Fan, Y., Jiang, W., Han, G., Yang, C., & Cong, J. (2008). Autopilot: A platform-based esl synthesis system. In *High-Level Synthesis* (pp. 99–112). Springer.