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 application-specific design while the regular structure of the SCGRA makes the customization much easier. When compared to the execution on a general purposed processor, the accelerators generated using QuickDough achieves up to 9X performance speedup.

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), 26 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 additional overlay layer may in some cases result in implementations with less than optimal performance when compared to their hand-

This work is supported in part by the Research Grant Concil of Hong Kong, under the General Research Fund project 716510 and in part by the Croucher Innovation Award of the Croucher Foundation.
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

optimized alternatives. Nevertheless, the use of overlays remains 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.

Moreover, the softness of the overlay allows application-specific customization for better performance-overhead trade-off and the regularity of the overlay makes the customization much easier. According to the experiments, it takes around 10 minutes to 20 minutes to complete the customization and the customized accelerators achieve up to 9X speedup over the execution on a hard ARM processor. Although the overlay pre-implementation which must eventually rely on conventional hardware design tools and the customization over a large design space are relatively slow, they are required only once per application throughout the design iterations and can therefore be amortized as the number of compile-debug-edit cycle increases.

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, the compilation framework will be illustrated in Section 5 and the customization will be detailed in Section 6. In Section 7, experimental results will be shown. Finally, we will discuss the benefit and limitations of current implementation in Section 8 and conclude in Section 9.

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 HLS 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 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 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].

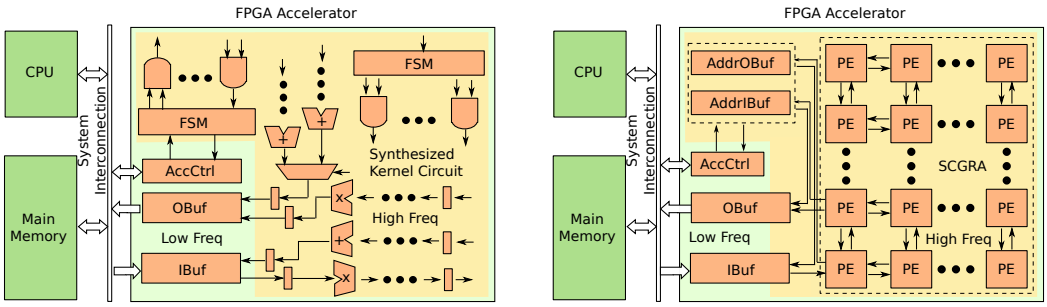
In this work, we opted to utilize a fully pipelined synchronous soft coarse-grained reconfigurable array (SCGRA) as an overlay to facilitate rapid FPGA accelerator generation in a hybrid CPU-FPGA system. Compared to previously proposed CGRAs, our overlay is designed to be *soft* as the size, processing element designs, as well as the interconnect topologies may all be customized as needed providing just enough resource for an application specifically. Moreover, the design of our overlay is regular and design parameters such as loop unrolling factor and overlay size have relatively predictable influence on the overlay performance and overhead, which makes the customization much easier and more efficient. Finally, 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.

3. QUICKDOUGH FRAMEWORK

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

Figure 1a shows the design of a typical accelerator system. In such system, on-chip memory is 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. 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 utilizes a regular SCGRA overlay instead of irregular random logic to implement the computation and the overlay can be reused during the design iterations.

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 a loop is identified, it is further compiled to an SCGRA overlay based FPGA accelerator



(a) Conventional FPGA Accelerator (b) SCGRA Based FPGA Accelerator

Fig. 1. Hybrid CPU-FPGA Computation System

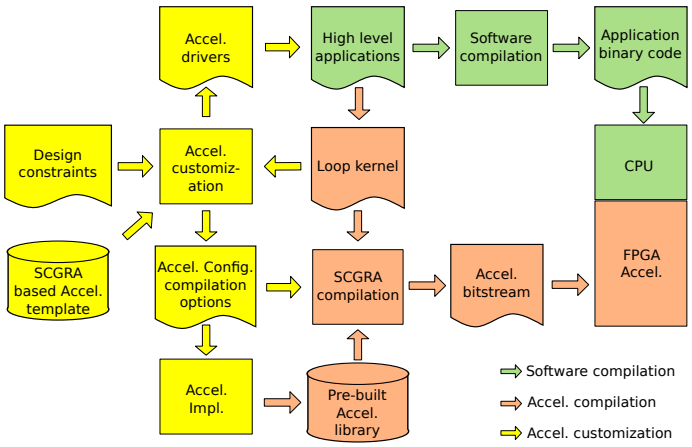


Fig. 2. QuickDough: FPGA Accelerator Design Methodology Using SCGRA Overlay. The compute intensive part of an application can be compiled to a customized SCGRA overlay based FPGA accelerator while the rest can be compiled to host CPU.

through SCGRA customization and SCGRA compilation while the rest part of the program is compiled to the processor through a conventional software compilation.

The SCGRA customization decides the optimized design parameters specifically to a loop kernel for better performance under given constraints such as resource budget. Because of the softness of the SCGRA 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. Also customized are routines that controls and transfers data to and from the accelerator. Most importantly, the customization makes all the hardware design details transparent to the users, which makes the whole system accessible to high-level designers. The user may choose to perform the customization only when the loop is changed dramatically and previous customization doesn't fit the updated loop.

Once the overlay design is determined, the SCGRA compilation starts to generates the final FPGA accelerator bitstream for the specified loop kernel. To begin, the compute kernel loops are statically analyzed to produce their corresponding data flow graphs (DFGs) with the customized loop unrolling factor. Then 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 binary code, 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 without significant performance speedup. 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 group 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.

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 OBuf external to the PE. At the same time, this data memory takes input from the ALU output, data arriving from

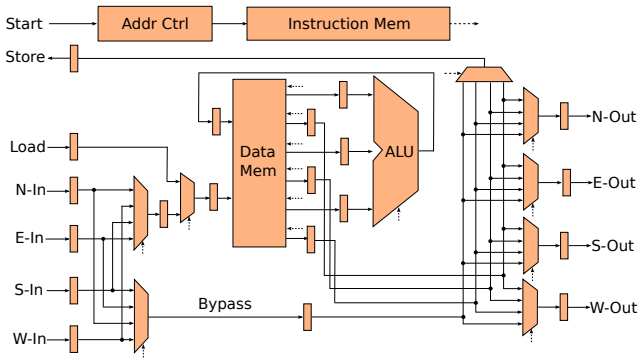


Fig. 3. Fully pipelined PE structure. Each PE can be connected to at most 4 neighbours.

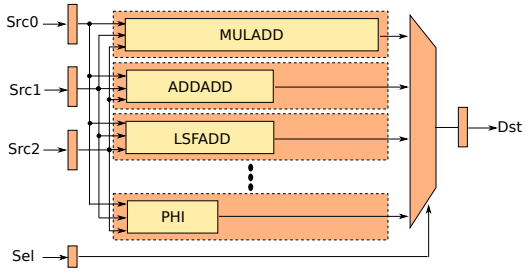


Fig. 4. The QuickDough ALU. It supports up to 16 fully pipelined 3-input operations.

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

Table I. Operation Set Implemented in ALU. It covers all the four applications used in the experiments.

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 $\gg$ Src1) & Src2
LSFADD	1000	Dst = (Src0 $\ll$ Src1) + Src2
ABS	1001	Dst = abs(Src0)
GT	1010	Dst = (Src0 > Src1) ? 1 : 0
LET	1011	Dst = (Src0 $\leq$ Src1) ? 1 : 0
ANDAND	1100	Dst = (Src0 & Src1) & Src2

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 7. Figure 4 shows the current set of operation.

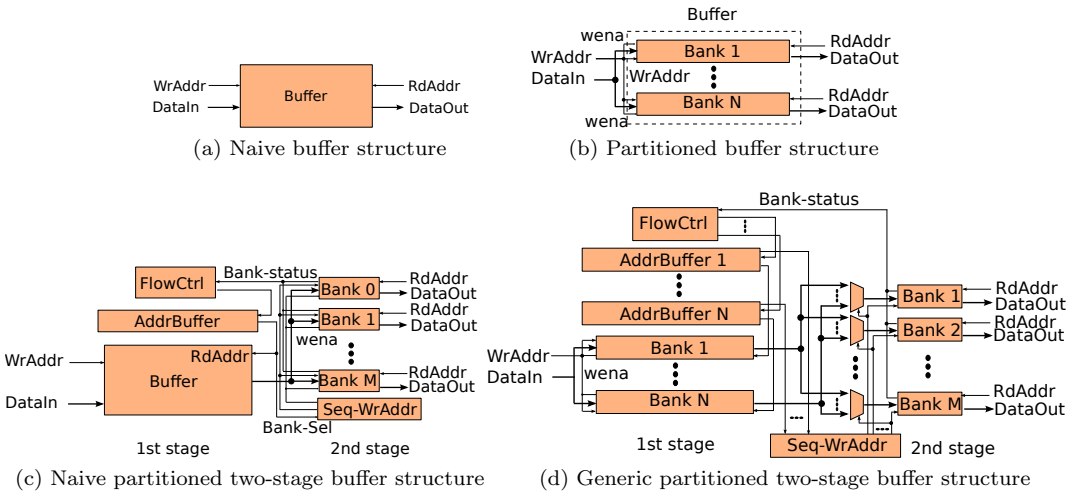
These operators in the ALU may execute concurrently in a pipelined fashion and must complete in a deterministic number of cycle. Given the deterministic nature of the operators, the QuickDough scheduler will ensure that there is never conflict at the output multiplexor.

4.2.3. 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 multiplier as well, but it doesn't influence the pipeline of the design.

4.3. Input/Output Buffer

Input/output buffer is used to store data transmitted between the FPGA accelerator and main memory. (The input and output buffers are quite similar, so we just present the input buffer structure here.) It is usually implemented with block RAM and a naive implementation as shown in Figure 5a provides limited bandwidth to the computing logic and may dramatically restrict the performance of the accelerator. A straightforward solution is to expose the primitive block RAM ports to the accelerator directly as shown in Figure 5b. However, the bandwidth is limited by the number of the primitive block RAM and it works only when the data is perfectly placed into different block RAM banks and each port of the accelerator accesses exactly the corresponding sub set of the data. However, we may have the input/output data of iterated DFGs extracted from the same loop stored in the input/output buffers when transmitted from/to main memory to amortize the initial communication cost, while different input/output of the DFGs may have diverse layout pattern. For instance, one DFG may load its first input data from partitioned bank 0 while the following DFG may have to load its first input data from bank 1. As a result, the two DFGs will not be able to reuse the same lock-step computing on the SCGRA overlay and providing each DFG with different scheduling will be extremely expensive.

To solve this problem, we have introduced an additional buffering stage and developed a scalable on-chip buffer structure as shown in Figure 5c and Figure 5d. The first stage is the basic on-chip buffer as mentioned in Figure 5a and Figure 5b. It stores data transmitted between the accelerator and main memory. The additional buffer stage stores the input/output of the DFG computation on the SCGRA overlay. It ensures the input/output data has exactly the same layout for all the iterated DFG computation and can be implemented with arbitrary number of partitions providing sufficient bandwidth to the computing logic.



(a) Naive buffer structure

(b) Partitioned buffer structure

(c) Naive partitioned two-stage buffer structure

(d) Generic partitioned two-stage buffer structure

Fig. 5. Input/output buffer structure

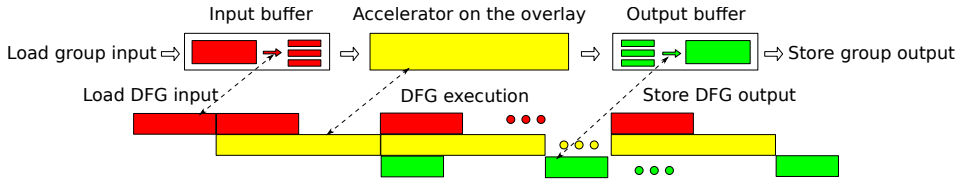


Fig. 6. Pipelined execution on the SCGRA overlay

Although the additional buffer stage makes the computation data path even longer, data movement between the first buffer stage and the second buffer stage and the iterated DFG computing can be pipelined as illustrated in Figure 6. As long as the second stage buffers are not full, FlowCtrl will keep the first stage buffer sending data to the second buffer stage with the pre-scheduled order which can be obtained from the SCGRA overlay scheduling and stored in the AddrBuffer. When the second buffer stage has all the input for a single DFG execution, FlowCtrl will start the accelerator. When the accelerator completes a DFG computing, it will stop until it is activated next time. On the output side, the buffer and the accelerator work similarly. When the number of the iterated DFGs is large enough and DFG computing time is larger than the data movement cost, the cost of the additional buffer stages can be ignored. In addition, to ensure the pipelining in Figure 6, the second buffer stage capacity is set to be twice the input/output of a single DFG.

5. APPLICATION COMPILATION

The QuickDough compilation process consists of four main inter-related steps as illustrated in Figure 7. In the first step, compute kernels of the application are extracted and transformed into their corresponding dataflow graphs (DFGs). Then, the corresponding overlay is created either by reusing an existing implementation or by synthesizing a new application-specific overlay based on an architectural template. In the third step, the DFGs and hyperblocks are scheduled to the generated overlay using an operation scheduler. Finally, the operation schedule is extracted and the corresponding configuration words are integrated into the configuration bitstream of the overlay. In our current implementation, the `data2mem` tool from Xilinx is used for this purpose.

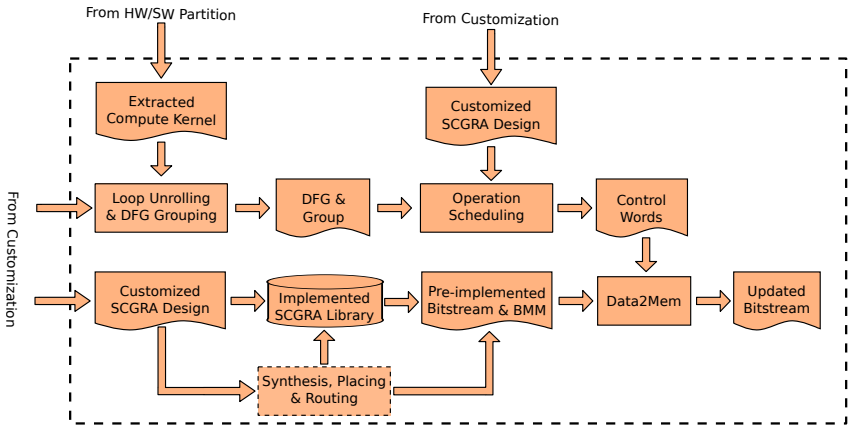


Fig. 7. SCGRA compilation with customized overlay and specified compute kernel.

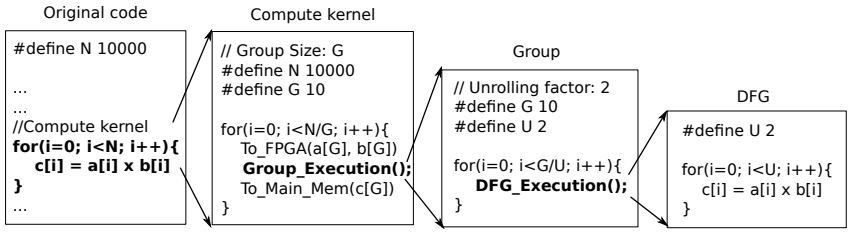


Fig. 8. Relation between compute kernel, group and DFG. Data transmission between FPGA and host CPU is performed for each group execution.

Among them, the physical implementation of the overlay is obviously the most time consuming. To ensure a rapid compilation experience, the user may want to generate a new overlay architecture only as needed, perhaps on a per-application domain basis. It is important to note, however, that the application remains functionally correct even when it is executed on a suboptimal overlay configuration. The user may therefore take advantage of the fast compilation to the overlay during initial development phases that demand rapid debug-edit-cycles. Once the function of the accelerator is confirmed, the user may proceed with customizing the overlay architecture for performance sake.

5.1. DFG Generation & Grouping

The main input to the QuickDough framework for acceleration are compute intensive kernels extracted from the user applications. The first step of the compilation process is therefore to extract dataflow graphs (DFGs) from these kernels that are often expressed as inner loop bodies. Depending on its structure, this loop may further be unrolled multiple times to increase the amount of operation parallelism in the generated DFG. During run time, the generated DFG will be executed repeatedly until the end of the original loop. Figure 8 illustrate this process.

In addition to unrolling the innermost loops, the QuickDough framework also allows users to balance computation and communication by batching data transfers for multiple executions of the same DFG into groups as shown in Figure 8. Specifically, after the loop is unrolled U times, G of them are grouped together for each data transfer. During each data transfer, input data for all G iterations will be sent to the accelerators. They are stored in the on-chip buffer so they can be accessed by the processing array within a single cycle.

While grouping data transfers helps amortize the communication cost between CPU and the accelerator, it also imposes additional requirement for on-chip memory to serve as buffer for the extra data transferred. As a result, the unrolling factor U and grouping factor G has to be co-optimized to balance performance and on-chip resource utilization. For instance, increasing U leads to a larger DFG to be executed in the QuickDough overlay, which may be benefited from a larger processing array. However, the increased processing array limits the amount of on-chip buffer available for data and address buffer. As a result, the amount of DFG grouping G is limited and may lead to higher performance penalty due to communication.

5.2. Overlay Generation

The QuickDough overlay is a highly regular processing array that can be generated easily according to a template. The overlay may be customized in many aspects, including the type of operation supported by each processing element (PE), the processing array size, its topology, as well as the number and capacity of the data buffers. For sake of rapid compilation, presynthesized overlay may be used. To improve performance, or to customize the type of operation for a new application, the user may opt to synthesize a new overlay design that may be reused subsequently. Similar to the case of DFG generation, the discussion of a fully automated optimization framework that optimizes the overlay design to the application is beyond the scope of this work. Instead, we examine multiple overlay configurations in the benchmark session to experiment tradeoffs that are involved.

5.3. Operation Scheduling

Once the overlay architecture is determined, the operations from the user DFG are scheduled to execute on the reconfigurable array. Since the processing elements in the QuickDough overlay execute in lock steps with deterministic latencies, a classical list scheduling algorithm [Schutten 1996] was adopted. The challenge in this scheduler is that data communication among the processing elements must be carried out via multi-hop routing in the array. As a result, while it is desirable to schedule data producers and consumers in nearby processing elements to minimize communication latencies, it is also necessary to utilize as much parallel processing power as possible for sake of load balancing. Building on top of our previous work presented in [Yu 2012], a scheduling metric considering both load balancing and communication cost was adopted in our current implementation.

Algorithm 1 briefly illustrates the scheduling algorithm implemented in QuickDough. Initially, an operation ready list is created to represent all operations that are ready to be scheduled. The next step is to select a PE from the SCGRA and an operation from the ready list using a combined communication and load balance metric. When both the PE and the operation to be scheduled are determined, the `OPScheduling` procedure starts. It determines an optimized routing path, moves the source operands to the selected PE along the path, and schedules the selected operation to execute accordingly. After this step, the ready list is updated as the latest scheduling may produce more ready operations. This `OPScheduling` procedure is repeated until the ready list is empty. Finally, given the operation schedule, the corresponding control words for each PE and the IO buffer accessing sequence will be produced. These control words will subsequently be used for bitstream generation in the following compilation step.

5.4. Bitstream Integration

The final step of the compilation is to generate the instructions for each PE and the address sequences for the I/O buffers according to the scheduler's result, which will subsequently be incorporated into the configuration bitstream of the overlay produced from previous steps. By design, our overlay does not have any mechanism to load instruction streams from external memory. Instead, we take advantage of the reconfigurability of SRAM based

Algorithm 1 The QuickDough 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
```

FPGAs and store the cycle-by-cycle configuration words using on-chip ROMs. The content of the ROMs are embedded in the bitstream and the `data2mem` tool from Xilinx [Xilinx 2012] is used to update the ROM content of the pre-built bitstream directly. To complete the bitstream integration, `BMM` file that describes the organization and placements of the ROMs in the overlay is extracted from `XDL` file corresponding to the overlay implementation [Beckhoff et al. 2011]. This bitstream integration process costs only a few seconds of the compilation time.

6. SCGRA OVERLAY BASED FPGA ACCELERATOR CUSTOMIZATION

Application-specific customization helps to improve the performance of the resulting accelerators. However, taking the system as a black box and exhaustively searching all the possible configurations can be inefficient and slow. In this work, by taking advantage of the regularity of the SCGRA overlay based FPGA accelerator, we can reduce the complex customization problem to a much simpler sub design space exploration (DSE) together with a simplified search problem and optimized application-specific nested loop accelerator can be produced efficiently.

6.1. Customization problem formulation

In this section, we will formalize the customization problem of the nested loop acceleration on an SCGRA overlay based FPGA accelerator within user specified hardware resource budgets.

Suppose $\boldsymbol{\Psi}$ represents the overall nested loop acceleration design space. $\boldsymbol{C} \in \boldsymbol{\Psi}$ represents a possible configuration in the design space and it includes a number of design parameters as listed in Table II. Assume that the loop to be accelerated has n nested levels and loop count can be denoted as $l = (l_1, l_2, ..., l_n)$. $R = (R_1, R_2, R_3, R_4)$ stands for the FPGA resource (i.e. BRAM, DSP, LUT and FF) that are available on a target FPGA and $Consumption(\boldsymbol{C}, i)$ denotes the four different types of FPGA resource consumption. $In(\boldsymbol{g})$ and $Out(\boldsymbol{g})$ stand for the amount of input and output of a group. Similarly, $In(\boldsymbol{u})$ and $Out(\boldsymbol{u})$ stand for the amount of input and output of a DFG. $DFGCompuTime(\boldsymbol{C})$ represents the number of cycles needed to complete the DFG computation. α_i and β_i are constant coefficients depending on target platform where $i = (1, 2, ...)$. $LoadDFG(\boldsymbol{C})$ and $StoreDFG(\boldsymbol{C})$ represent the time consumption of the initial loading and storing processes moving DFG input/output data between the first stage buffer and the second stage buffer.

Table II. Design Parameters of Nested Loop Acceleration

Design Parameters		Denotation / setting
Nested Loop Compilation	Loop Unrolling Factor	$\boldsymbol{u} = (u_0, u_1, ...)$
	Grouping Factor	$\boldsymbol{g} = (g_0, g_1, ...)$
Overlay Configuration	SCGRA Topology	2D Torus
	SCGRA Size	$r \times c$
	Instruction Mem	$imD \times imW$
	Data Mem	$dmD \times dmW$
	Input Buffer	$ibD1 \times ibW$ $ibD2 \times ibW$
	Input Buffer Partition	$ipNum1 \times ipNum2$
	Output Buffer	$obD1 \times obW$ $obD2 \times obW$
	Output Buffer Partition	$opNum1 \times opNum2$
	Input Address Buffer	$iabD \times iabW$
	Output Address Buffer	$oabD \times oabW$
	Operation Set	as needed
	Implementation Frequency	f, fixed
	Pipeline Depth	fixed

They can be obtained through the two-stage buffer scheduling. According to our experiments, $DFGCompuTime(\boldsymbol{C})$ is larger than $LoadDFG(\boldsymbol{C})$ and $StoreDFG(\boldsymbol{C})$ with buffer partition. In this case, the customization problem targeting minimum energy consumption can be formulated as follows:

Minimize

$$RunTime(\boldsymbol{C}) = CompuTime(\boldsymbol{C}) + CommuTime(\boldsymbol{C}) + LoadDFG(\boldsymbol{C}) + StoreDFG(\boldsymbol{C}) \tag{1}$$

subject to

$$\begin{aligned}
&Resource(\boldsymbol{C}, i) \leq R_i, i = 1, 2, 3, 4\\
&2 \times In(\boldsymbol{u}) = ipNum2 \times ibD2\\
&2 \times Out(\boldsymbol{u}) = opNum2 \times obD2\\
&In(g) \leq ipNum1 \times ibD1\\
&Out(g) \leq opNum1 \times obD1\\
&DFGCompuTime(\boldsymbol{C}) \leq imD\\
&\prod_{i=1}^{n} \frac{g_i}{u_i} \times In(u) \leq iabD\\
&\prod_{i=1}^{n} \frac{g_i}{u_i} \times Out(u) \leq oabD
\end{aligned} \tag{2}$$

$RunTime(\boldsymbol{C})$ represents the number of cycles needed to compute the loop on the CPU-FPGA system. It consists of both the time consumed for computing on FPGA and communication between FPGA and host CPU, and it can be calculated using (1).

Since the unrolled part of the loop will be translated to DFG and then scheduled to the SCGRA overlay. Thus the DFG computation time is essentially a function of $\mathbf{u}$, $ipNum2$, $opNum2$, r and c, and it can also be denoted by $DFGCompuTime(\mathbf{u}, ipNum2, opNum2, r, c)$. The nested loop is computed by repeating the same DFG execution, and the nested loop computation can be calculated using (3).

$$CompuTime(\boldsymbol{C}) = \prod_{i=1}^{n} \frac{l_i}{u_i} \times DFGCompuTime(\mathbf{u}, ipNum2, opNum2, r, c) \tag{3}$$

DMA is typically used for the bulk data transmission. Communication cost per data can be modeled with a piecewise linear function and thus DMA latency can be calculated using $DMA(x)$ where x represents the amount of DMA transmission. The communication time of the whole nested loop can be calculated by (4).

$$CommuTime(\boldsymbol{C}) = \prod_{i=1}^{n} \frac{l_i}{g_i} \times (DMA(In(\mathbf{g})) + DMA(Out(\mathbf{g}))) \tag{4}$$

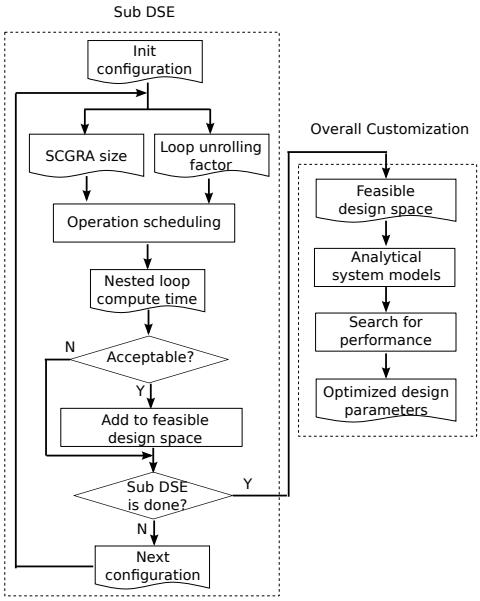


Fig. 9. System customization framework.

Hardware resource consumption on FPGA mainly includes DSP, LUT, FF and BRAM (block RAM). LUT, FF and DSP consumption can be roughly estimated with a linear function of SCGRA size and can be calculated using (5). BRAM consumption which is usually the resource bottleneck for SCGRA overlay based FPGA accelerator design can be calculated by (6). Note that the second stage buffers are implemented using distributed memory instead of block RAMs, so they are not included in the equation.

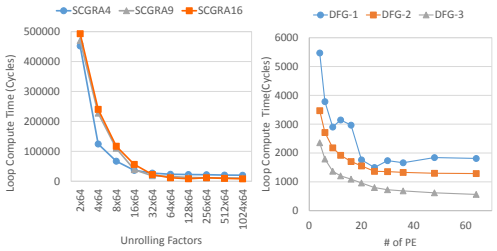
$$Resource(\boldsymbol{C}, i) = \alpha_i \times r \times c + \beta_i, (i = 2, 3, 4) \tag{5}$$

$$\begin{aligned} Resource(\boldsymbol{C}, 1) =& r \times c \times (imD \times imW + dmD \times dmW) + \\ & (ipNum1 \times ibD1 \times ibW + opNum1 \times obD1 \times obW) + \\ & (ipNum1 \times iabD1 \times iabW + opNum1 \times oabD1 \times oabW) \end{aligned} \tag{6}$$

6.2. Customization framework

Figure 9 illustrates the overview of the customization framework. It can be roughly divided into two parts. In the first part, a sub DSE targeting loop execution time is performed and the feasible design space can be obtained. Since loop execution time is determined by the operation scheduling which simply depends on the loop unrolling factor, input/output buffer partition, SCGRA size, the sub DSE is much simpler compared to the overall system DSE which includes more than a dozen design parameters. In the second part, each configuration in the feasible design space will be evaluated. Instead of using simulation based methods, analytical models are employed to estimate the accelerator metrics such as performance and resource consumption. These analytical models are accurate because of the regularity of the SCGRA overlay. Even though the feasible design space is still large, it is fast to evaluate all the configurations in it. After the evaluation process, customization for best performance becomes trivial and the customized design parameters can be obtained immediately.

Suppose Φ denotes the feasible design space. ϵ indicates the percentage of the performance benefit obtained by the increase of loop unrolling or SCGRA size. It is a user defined threshold and must be small enough to prune the configurations that are inappropriate.



(a) (b)

Fig. 10. The design parameters typically have monotonic influence on the loop computation time and the computation time benefit degrades with the increase of the design parameter. (a) SCGRA Size, (b) Unrolling Factor

The configurations in Φ must satisfy (7) and (8).

$$\begin{aligned} &\forall \boldsymbol{C} = (..., \boldsymbol{u}, r, c, ...) \in \Phi, \boldsymbol{C'} = (..., \boldsymbol{u'}, r', c', ...) \in \Phi, \\ &(r + 1 == r' \text{ and } c == c') \text{ or } (r == r' \text{ and } c + 1 == c') : \\ &\frac{CompuTime(\boldsymbol{C}) - CompuTime(\boldsymbol{C'})}{CompuTime(\boldsymbol{C})} > \epsilon \end{aligned} \tag{7}$$

$$\begin{aligned} &\forall \boldsymbol{C} = (..., \boldsymbol{u}, r, c, ...) \in \Phi, \boldsymbol{C'} = (..., \boldsymbol{u'}, r, c, ...) \in \Phi, \\ &\boldsymbol{u} \text{ and } \boldsymbol{u'} \text{ are consecutive unrolling factors :} \\ &\frac{CompuTime(\boldsymbol{C}) - CompuTime(\boldsymbol{C'})}{CompuTime(\boldsymbol{C})} > \epsilon \end{aligned} \tag{8}$$

Each configuration $\boldsymbol{C} \in \Phi$ must have the corresponding scheduling result known, and thus the computation time of the loop kernel and minimum instruction memory depth are available as well. Then we can further evaluate the performance of each feasible configuration using the models built in previous section and obtain the optimized configuration through a simple search.

In addition, a series of experiments on Zedboard [Avnet 2014] as shown in Figure 10 demonstrate that SCGRA size and unrolling factor present a clear monotonic influence on the loop compute time. The performance benefit of loop unrolling and increase of SCGRA size drops gradually. This observation further helps to simplify the sub DSE with a simple branch and bound algorithm.

7. EXPERIMENTS

With an objective to improve designers' productivity in developing FPGA accelerators, the key goal of QuickDough is to reduce compilation and development time for such hardware-software system. However, to warrant the merit of such framework, the performance and implementation overhead of the generated acceleration system should remain competitive to ones that are developed with conventional manual design efforts. For that purpose, we have conducted a series of experiments on end-to-end application performance, compilation time, and implementation overhead of QuickDough. In each case, multiple configurations of the overlay were experimented in order to study their area-performance tradeoffs. The results are then compared against a typical manual accelerator design methodology that employs conventional vendor-provided high-level synthesis tools. Finally, we studied the scalability of the proposed overlay design and investigated how such regular design may benefit large accelerator designs in the future.

The experiment section is organized as follows. We will first briefly introduce the benchmark programs in the following subsection and explain the basic experiment setup in Sec-

Table III. Detailed Configurations of the Benchmark

Benchmark	MM	FIR	SE	KM
Parameters	Matrix Size	# of Input/ # of Taps+1	# of Vertical Pixels/ # of Horizontal Pixels	# of Nodes/Centroids/ Dimension
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 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. Complete Loop Structure of the Benchmark

Benchmark	MM	FIR	SE	KM
S	$10 \times 10 \times 10$	40×50	$8 \times 8 \times 3 \times 3$	$20 \times 4 \times 2$
M	$100 \times 100 \times 100$	10000×50	$128 \times 128 \times 3 \times 3$	$5000 \times 4 \times 2$
L	$1000 \times 1000 \times 1000$	100000×50	$1024 \times 1024 \times 3 \times 3$	$50000 \times 4 \times 2$

tion 7.2. Then we will elaborate the comparison of performance, compilation, implementation overhead and scalability in Section 7.3-Section 7.6 respectively.

7.1. Benchmark

Four applications were used as benchmark in this work, namely, matrix-matrix multiplication (MM), a finite impulse response (FIR) filter, a K-mean clustering algorithm (KM) and a Sobel edge detector (SE). For each application, three input with small (S), medium (M) and large (L) datasets were used, forming a total of 12 combinations. The basic parameters and configurations of the benchmark are illustrated in Table III and the loop structures of the programs are shown in Table V.

7.2. Experiment Setup

The Xilinx implementation tools were run on a computer with Intel Core i5-3230M CPU and 8 GB of RAM. The resulting hardware-software platform was targeted at the Zedboard with an XC7Z020 FPGA. Software components of the user applications ran on the embedded ARM processor, while the hardware accelerators were implemented in the programmable logic.

To develop accelerators with QuickDough, the SCGRA overlay was first developed in ISE 14.7 and was subsequently imported as an IP core in XPS 14.7. With the IP core, the SCGRA overlay based FPGA accelerator was further integrated and implemented in PlanAhead 14.7.

As a comparison, we also developed accelerators following a typical implementation flow. In particular, Vivado HLS 2013.3 was used to transform the compute kernels into hardware IP Catalogs. They were then integrated into the rest of the FPGA acceleration system using Vivado 2013.3.

Furthermore, 2 accelerator configurations were employed in each design methodology. In the case of building custom accelerators with Vivado HLS, 2 systems, called HS and HL in the results, which included a 2 kB and 64 kB I/O buffer respectively were used. In the case of QuickDough, the 2 systems, labeled as QS and QL were developed, which included a 2×2 and 5×5 processing arrays respectively. The basic configurations of the QS and QL are listed in Table IV.

7.2.1. Loop unrolling and grouping. In our benchmark, multi-level loops form the compute intensive sections that are accelerated by FPGAs. The structure for these loops are shown in Table V.

To increase the amount of concurrent compute operations available, these loops were first unrolled by a factor of U. Input and output data for G invocations of the unrolled loop body

Table VI. Loop Unrolling & Grouping Setup Of Accelerators Using Direct HLS Based Design Framework

Application		HL		HS	
		Unrolling Factor	Group Structure	Unrolling Factor	Group Structure
MM	S	$2 \times 10 \times 10$	$10 \times 10 \times 10$	$2 \times 10 \times 10$	$10 \times 10 \times 10$
	M	$1 \times 1 \times 100$	$100 \times 100 \times 100$	1×100	10×100
	L	1×500	50×1000	500	1000
FIR	S	2×50	40×50	2×50	40×50
	M	2×50	10000×50	2×50	1000×50
	L	2×50	50000×50	2×50	1000×50
SE	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	$1 \times 1 \times 3 \times 3$	$75 \times 1024 \times 3 \times 3$	$1 \times 1 \times 3 \times 3$	$4 \times 1024 \times 3 \times 3$
KM	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	$5 \times 4 \times 2$	$25000 \times 4 \times 2$	$5 \times 4 \times 2$	$1000 \times 4 \times 2$

Table VII. Loop Unrolling and Grouping Setup For Accelerators Using QuickDough

Application		QS		QL	
		Unrolling Factor	Group Structure	Unrolling Factor	Group Structure
MM	S	$10 \times 10 \times 10$	$10 \times 10 \times 10$	$10 \times 10 \times 10$	$10 \times 10 \times 10$
	M	5×100	10×100	5×100	10×100
	L	200	1000	200	1000
FIR	S	40×50	40×50	40×50	40×50
	M	20×50	100×50	50×50	250×50
	L	20×50	100×50	50×50	250×50
SE	S	$4 \times 8 \times 3 \times 3$	$8 \times 8 \times 3 \times 3$	$8 \times 8 \times 3 \times 3$	$8 \times 8 \times 3 \times 3$
	M	$4 \times 8 \times 3 \times 3$	$8 \times 8 \times 3 \times 3$	$23 \times 8 \times 3 \times 3$	$65 \times 8 \times 3 \times 3$
	L	$4 \times 4 \times 3 \times 3$	$16 \times 4 \times 3 \times 3$	$16 \times 4 \times 3 \times 3$	$16 \times 4 \times 3 \times 3$
KM	S	$20 \times 4 \times 2$	$20 \times 4 \times 2$	$20 \times 4 \times 2$	$20 \times 4 \times 2$
	M	$25 \times 4 \times 2$	$125 \times 4 \times 2$	$125 \times 4 \times 2$	$500 \times 4 \times 2$
	L	$25 \times 4 \times 2$	$125 \times 4 \times 2$	$125 \times 4 \times 2$	$500 \times 4 \times 2$

are subsequently grouped into individual transfers to amortize the cost for communication between hardware and software. As explained, the unrolling factor U and grouping factor G have significant impact on the overall application performance for both QuickDough and for any typical HLS design methodology.

Table VI shows the loop unrolling and grouping arrangement for generating accelerators using typical high-level synthesis tools in this work. In these cases, while larger loop unrolling factors typically result in better simulated performance, additional hardware resource must usually be consumed. The increased resource usage may lead to a lowered implementation frequency on the FPGA and thus lower overall performance. At the same time, larger group size is beneficial to data reuse and helps to amortize the initial communication cost. However, additional on-chip memory resources must be utilized as I/O buffer instead of being spent on computation, potentially limiting the performance of the hardware accelerator. In this work, we set the loop unrolling factor just large enough to reach the best simulated performance. Furthermore, we also set grouping factor to large enough to fully utilize the on-chip buffer in the 2 accelerator configurations.

Similar loop unrolling and grouping sizes were chosen carefully for the experiments targeting the QuickDough overlay as shown in Table VII. In the case of the QuickDough overlay, a large unrolling factor leads to a larger input DFG, which affects the SCGRA array size, instruction ROM size, as well as data memory size. The group size has direct impact on I/O buffer size, address buffer size, and communication latency. In this work, we manually adopted the unrolling factor and group size to the two SCGRA configurations to balance computation and communication.

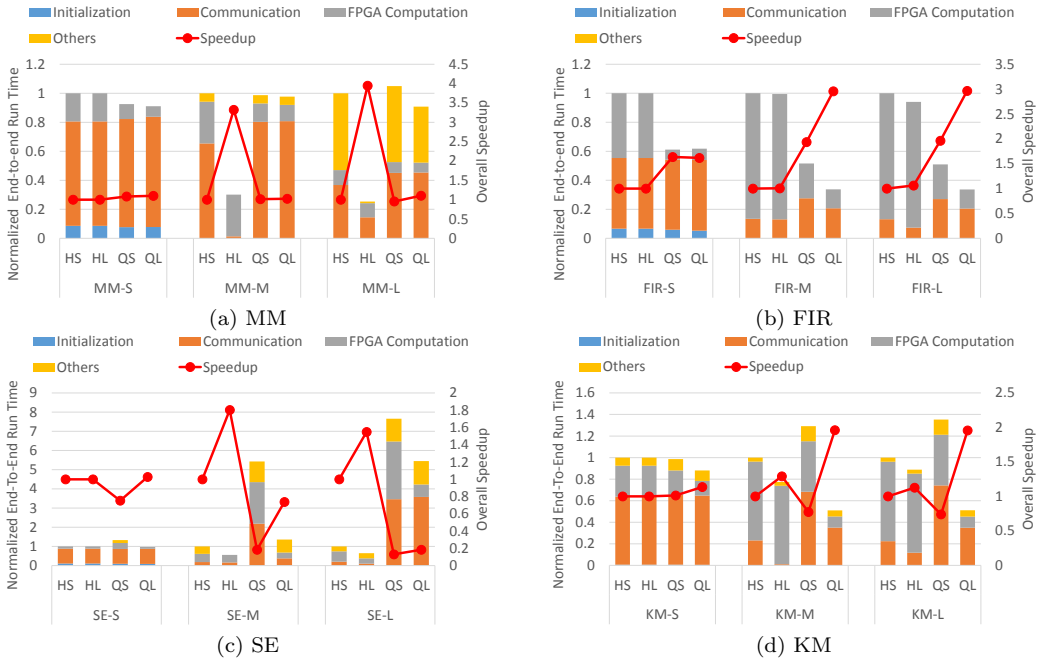


Fig. 11. Benchmark Speedup and Execution Time Decomposition Using Both Direct HLS Based Design Framework and QuickDough. HS is used as the baseline design to calculate the speedup as well as the normalized execution time.

7.3. Performance

While improving designers' productivity is the primary goal of QuickDough, the FPGA accelerators it generates must remain competitive in performance to those generated using manual efforts through typical HLS tools. In this section, the end-to-end performance of the target benchmark applications when executed on the targeted Zedboard is compared.

Figure 11 shows the performance speedup and execution time decomposition of the 4 benchmark programs, normalized to the performance of HS. The reported end-to-end execution time includes time spent on system initialization such as DMA initialization, I/O data communication between FPGA and the ARM processor, FPGA computation, as well as other tasks such as to marshal input/output data for DMA transmission and to process corner cases.

The results in the figure illustrate the complex relationship between accelerator configurations and the structure of the application on performance. Typically, performance of the accelerators generated using QuickDough is in the same order of those created using a direct HLS-based methodology. In the extreme cases, such as FIR-M and FIR-L, the QuickDough accelerator's performance is 3× better, while in other cases, such as SE-M and SE-L, they can be up to 7.3× slower. The variation in performance is due to the intricate interaction between the accelerator architecture, the computation and data reuse pattern of the user application.

To further investigate the computing performance of the accelerators, Figure 12 further breaks down the results, showing only the simulated computing time spent in the accelerators. The performance impacts due to communication, data organization, as well as implementation frequency are not shown.

In the case of MM, the QuickDough generated accelerators performed similar to HS with all matrix sizes. In these 3 cases, the QuickDough accelerators exhibit slightly higher com-

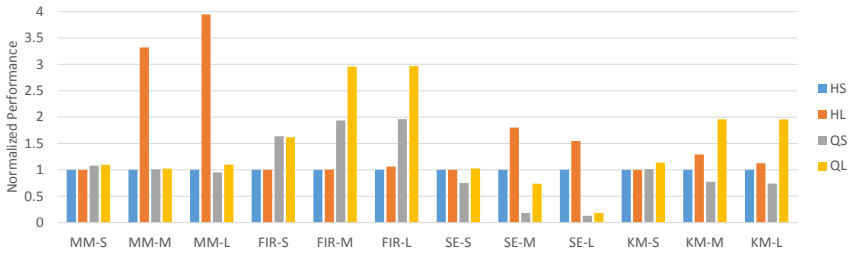


Fig. 12. Compute Kernel Simulated Performance Using Both Direct HLS Based Design Framework and QuickDough. The performance is normalized to that of HS.

munication cost, which is compensated by their slightly shorter time spent in computation. In fact, the QuickDough accelerators perform slightly better in both MM-S and MM-M as can be seen in Figure 12. However, HL outperform all cases by up to a factor of 4. It is because its large input buffer is able to buffer all input data on-chip, essentially eliminating the communication cost between the CPU and FPGA. The QuickDough overlay cannot afford buffer of this size as on-chip memories are devoted to the overlay execution.

In the case of FIR, we see that the QuickDough accelerators outperform the rest regardless of the input data size. In these cases, the communication cost is comparable while the QuickDough accelerators have a clear advantage in computing time as shown in Figure 12. Part of the reason is that the FIR application can be unrolled at least 20 times more in the QuickDough accelerators than the HLS-based accelerators, shown in Table VI and Table VII. The heavily unrolled loops provide far more operations to be carried out in parallel that lead to lower computing time. On the other hand, unrolling is limited in HLS tools due to the limited available resources. The commercial HLS tools were not very effective in time-sharing the generated blocks.

In the case of KM, similar observation can be made in terms of loop unrolling. As shown in Figure 12, the larger processing array in QL is able to take advantage of the heavily unrolled loops, while the smaller array size in QS limits the achievable speedup. However, because of the higher communication cost, the overall speedup is limited.

Finally, in the case of SE, the QuickDough generated accelerators perform poorly compared to those generated using HLS tools. Comparing Figure 11 and Figure 12, we can see that the QuickDough accelerators not only spend more time in communication, but they are also slower in computing. In terms of computation, we see that the high-level synthesis tools are capable of simplify the user-provided computation effectively through constant propagation. On the other hand, the constant multiplications must be carried out regardless in the QuickDough overlay, generating a lot more operations comparatively. At the same time, the large number of additional compute operations in the QuickDough accelerators significantly increases the compute-to-I/O ratio of the accelerators. As a result, only small portion of input data can be transferred in groups before a new batch of data must be transferred. This increased number of transfer between FPGA and CPU significantly limits the overall system performance.

7.3.1. Implementation Frequency. One advantage of employing a simple and regular overlay architecture is that it allows for highly optimized physical implementations. In particular, when compared to the random hardware generated by HLS tools, the QuickDough overlay is regular and highly pipelined, allowing implementations to run at much higher frequencies. The increased running frequency in turns results in higher overall performance of the system.

Figure 13 shows the implementation frequency of the benchmark using both design methodologies. Direct HLS based design framework takes timing constrain into consideration at the HLS step, and it can either synthesize the compute kernel to a lower frequency

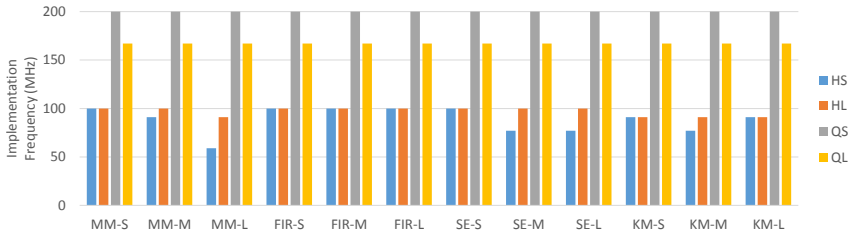


Fig. 13. Implementation Frequency of The Accelerators Using Both Direct HLS Based Design Framework and QuickDough

design with better simulated performance or a higher frequency design with worse simulation performance. Neither of them have a clear advantage over the other. In these experiments, the implementation tools were targeting a 100 MHz clock that the AXI controller on the Zedboard typically works at. Even at this rate, some of the synthesized IP cores could barely meet timing.

On the other hand, 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. QS with a 2×2 SCGRA overlay can run at 200 MHz while QL with a 5×5 SCGRA overlay can run at 167 MHz. The implementation frequency degrades slightly with a large design because more than 90 % of the BRAM blocks on target FPGA are used and the routing becomes extremely tight.

7.4. Compilation Time

In this section, the compilation time of the QuickDough framework is compared against that of using a direct HLS synthesis methodology. It is used as an indicator on the designer's productivity as it greatly limits the number of debug-edit-implementation cycles achievable per day.

7.4.1. Direct HLS-based design. The baseline direct HLS-based design framework consists of 4 main steps:

- Compute kernel synthesis: The compute kernel is translated to the corresponding HDL model.
- Kernel IP generation: The compute kernel is synthesized and packed as an IP core.
- Accelerator implementation: The FPGA accelerator built on top of the IP core is implemented.
- Software compilation: The application employing the FPGA accelerator is compiled to binary code.

The time to generate our benchmark accelerators through these steps is shown in Figure 14. From the figure, it is clear that IP core generation and hardware implementation dominate the compilation time, taking upward of almost an hour to implement some benchmark applications. Comparatively, the time for software compilation and kernel synthesis are negligible in most cases. Unfortunately, under this design methodology, every implementation iteration must go through all 4 steps, even when only minor modifications were needed. This long debug-edit-implement cycle greatly limits the designer's productivity.

7.4.2. QuickDough Compilation Time. Every implementation iterations in QuickDough also involves 4 steps:

- DFG generation: The compute kernel is translated to corresponding DFG.
- DFG scheduling: The DFG is scheduled to the customized SCGRA overlay.

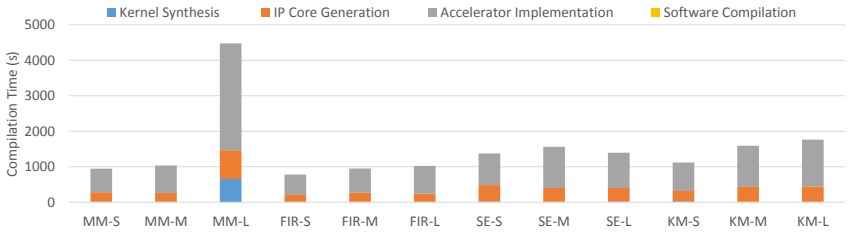


Fig. 14. Benchmark Compilation Time Using Direct HLS Based Design Framework

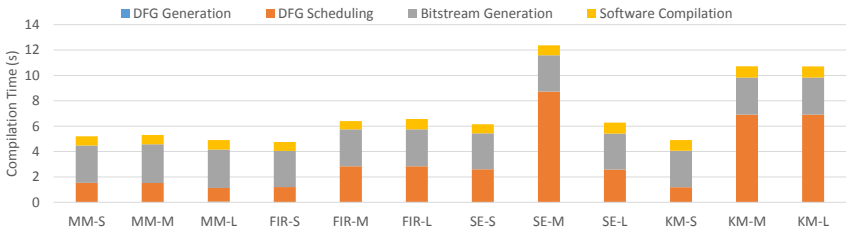


Fig. 15. Benchmark Compilation Time Using QuickDough

- Bitstream generation: The scheduling result is embedded into a pre-built accelerator bitstream to produce the final FPGA bitstream of the compute kernel.
- Software compilation: Application using the SCGRA accelerator is compiled to binary code.

Figure 15 shows the compilation time of our benchmark using QuickDough. Except for DFG scheduling, all remaining steps are fast and consume about constant time across different applications. DFG scheduling is relatively slower and its run time depends on the DFG and SCGRA overlay size. Yet, this longest step does not last longer than a few seconds for all our benchmarks. Overall, QuickDough is able to reimplement an application in 5 to 15 seconds, a two orders of magnitude improvement over our baseline direct HLS-based design framework.

Clearly, the designer must spend the time to physically implement the overlay architecture on the target FPGA, spending considerable time on the implementation tools. However, this can be done only as needed, for example, on a per-application-domain basis. The designer may iterate via the above rapid steps during design and debugging phases using a initial overlay implementation. Once the functionality is frozen, the designer may then opt to further optimize performance through overlay customization and reimplement a different overlay architecture. We argue that the ability to separate functionality and optimization concern, and the possibility of performing rapid debug-edit-implement iterations in QuickDough are crucial factors that contribute to a high-productivity design experience.

7.5. Implementation Overhead

In this section, hardware implementation overhead in both design methodologies are compared. Furthermore, we examined the impact on overlay customization on implementation overhead.

7.5.1. General Overlay Architecture. Table VIII shows the hardware overhead of both accelerator design methodologies across the benchmark applications. In these cases, an identical and general overlay architecture was employed across all benchmark applications.

From the table, it is clear that the accelerators using direct HLS based design framework generally consume fewer flip-flops, LUTs and RAM36s due to the delicate customization for

Table VIII. Hardware Overhead of The Accelerators Using Both Direct HLS Bsed Design Framework and QuickDough

			FF	LUT	RAM36	DSP48
MM	HS	S	4812	3390	4	84
		M	4804	4703	4	12
		L	11107	11524	4	12
	HL	S	4826	3390	128	84
		M	4251	4866	128	9
		L	11024	24890	128	12
FIR	HS	S	3736	3570	4	27
		M	3756	3872	4	27
		L	3756	3872	4	27
	HL	S	3742	3570	128	27
		M	3782	4246	128	27
		L	3792	4426	128	27
SE	HS	S	9556	6467	6	216
		M	7483	5520	6	144
		L	7102	5501	6	144
	HL	S	9564	6467	130	216
		M	7496	5711	130	144
		L	7622	5904	130	144
KM	HS	S	2826	3567	4	24
		M	6709	8088	4	120
		L	6709	8088	4	120
	HL	S	2852	3567	128	24
		M	6754	8122	128	120
		L	6770	8205	128	120
QS			9302	5745	32	12
QL			34922	21436	137	75
Total FPGA Resource			106400	53200	140	110

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.

On the other hand, the accelerators using QuickDough usually require comparable number of DSP48s, more FFs, LUTs and RAM36s. In particular, the large consumption of RAM36s often becomes the bottleneck limiting the maximum SCGRA size that can be implemented on the target FPGA. This limited size in turn constrains the maximum loop unrolling and blocking of the application and thus the overall performance.

7.5.2. Overlay Customization. To investigate the effect of overlay customization on hardware overhead, we further divided the four benchmarks into three groups. In the first group, since MM and FIR share the same set of compute operations, they were implemented using the same customized overlay. In the second case, since SE does not require a complete 32-bit data path, a customized overlay with a mixed of 16-bit and 32-bit data path was used. Finally, since KM requires almost all the operations presented in the original overlay, no change was made.

The experiments were rerun using the respective customized overlay and the resulting hardware savings are shown in Figure 16. Overall, the customized overlays provide as much as 65 % of DSP48, 30 % of LUT, as well as 15 % of FFs saving over the original generic design.

7.6. Scalability

Finally, we studied the scalability of the proposed overlay architecture in terms of problem size as well as processing array size in anticipation for future FPGA devices.

7.6.1. Problem Size. To study the effect of problem size on the overlay performance, MM with matrix sizes ranging from 4×4 to 20×20 were used. In the case of QuickDough overlay, 3 different SCGRA sizes were studied: 2×2, 5×5, 10×10. They were targeted at the larger `zc706` FPGA with abundant hardware resource. In the case of direct HLS synthesis, 2 scenarios were considered. In the first scenario, the matrix multiplication was

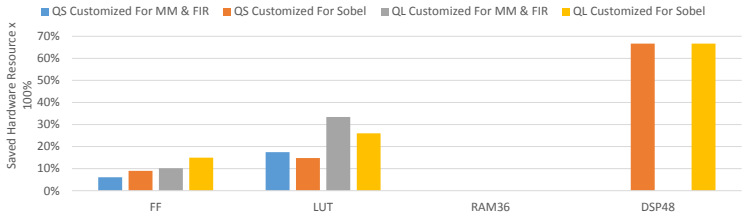


Fig. 16. Hardware Saving Of Customized SCGRA Overlay

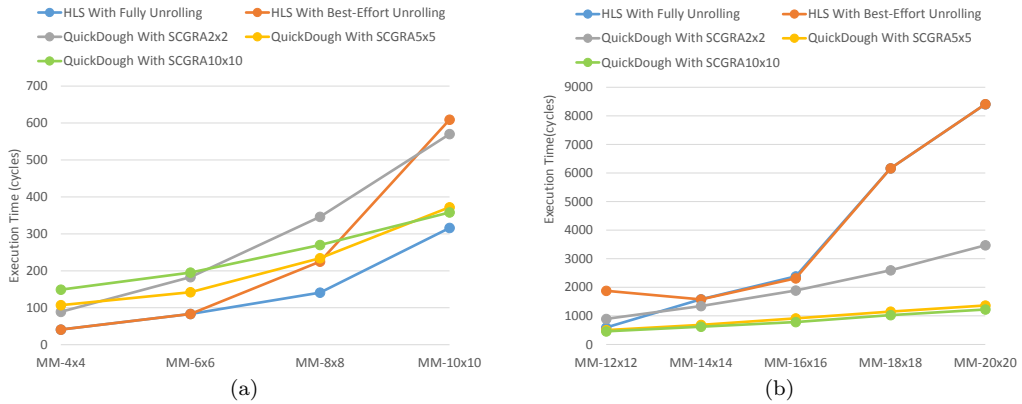


Fig. 17. Simulated Performance Of MM Implemented Using Both Direct HLS Based Design Framework And QuickDough

fully unrolled. In the second scenario, a best-effort loop unrolling that results in maximum performance under hardware constraints was used. In the case of the SCGRA overlay, the MM operations were fully unrolled. Figure 17 shows the simulated performance of the resulting accelerators generated by both design methodologies. Communication cost was not taken into account as they remain comparable in both cases.

Results from the figure show that accelerators using direct HLS based design framework perform much better than the QuickDough accelerators when the matrix size is small enough for the loops to be fully unrolled. However, as the matrix size grows, direct HLS can no longer afford the hardware overhead for intensive loop unrolling, limiting the performance of the resulting accelerators. As shown in Figure 18, the DSP resource consumption increases significantly with additional unrolling in return for higher performance as the matrix size increases. While it is possible for an expert to start manually time-sharing the resources, we did not explore such sophisticated implementation technique in attempt to maintain a comparable design effort.

On the other hand, the QuickDough overlay can naturally accommodate intensive loop unrolling through time-sharing of hardware resources. It is therefore capable of accelerating much larger portion of computation with relative ease. Its performance is mainly limited by the on-chip memory serving as instruction ROMs.

In our experiments with MM that target the larger `zc706` device, MM-8x8 is the crossover point that QuickDough begins to outperform. As a comparison, the crossover point on the originally targeted Zedboard was much smaller because of the limited hardware resource.

7.6.2. Overlay Processing Array Size. To study the scalability of our SCGRA overlay, the performance of processing array with increasing size was experimented using MM-20x20 as an example.

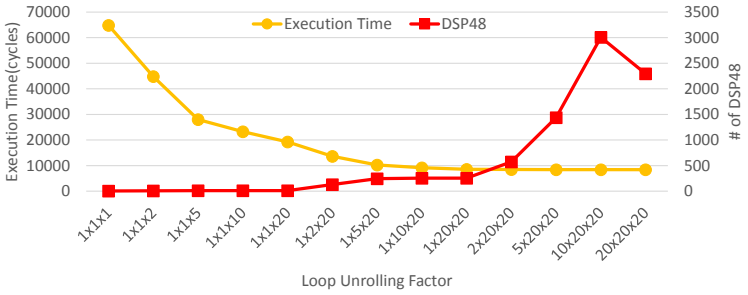
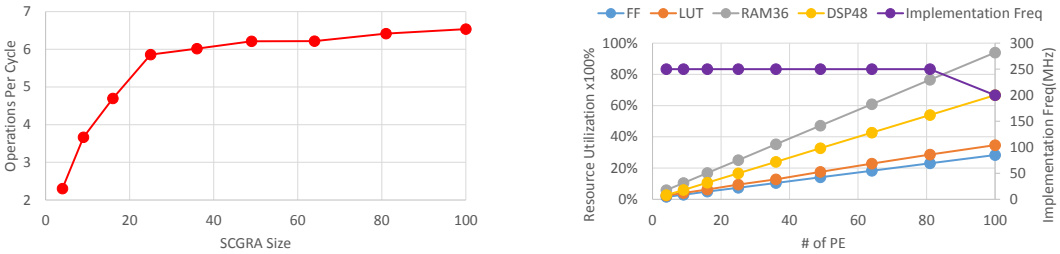


Fig. 18. MM 20x20 Implemented Using Direct HLS Based Design Framework With Diverse Loop Unrolling



(a) Performance (b) Overhead & frequency

Fig. 19. Simulated performance, hardware overhead and implementation frequency of MM-20 implemented with increasing SCGRA overlay size.

Figure 19a shows that in general, the additional processing power provided by the larger arrays results in better performance as expected. The performance gain reduces as the array size increases to a point when there simply is not enough available compute operations to be scheduled. This point of reflection depends on the nature of the user application. In the case of MM, the 5×5 array provided the near-optimal performance with reasonable amount of PEs.

As an overlay design, it is important that its design should allow flexible scaling of its computing power to address the needs from the application. As a simple, fully synchronous and highly regular reconfigurable array, the QuickDough overlay is very much scalable in that dimension as shown in Figure 19b.

From the figure, it can be seen that the hardware overhead associated with the QuickDough overlay increases linearly with the number of processing elements in the array. Most importantly, except for the largest array when the FPGA is almost full, it the QuickDough overlay is able to run at a constant high clock frequency.

8. 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 high-level synthesis on random logic.

Secondly, 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. In particular, the generation of data flow graphs from the user design currently

involves several manual conversion steps to match the user design with the target overlay. It is anticipated that an automated process that is able to analyze the user application and generate the corresponding DFG suitable for the overlay will be developed.

Thirdly, even with as simple an overlay architecture as presented in this work, there remains a vast design space with a labyrinth of parameters to adjust that will affect the power-performance of the resulting accelerators. In this work, we have chosen only 2 specific configurations in the experiments as representations, a fully automatic overlay customization framework is being developed to assist designers in making design choices.

Finally, the capacity of the address buffers used in the accelerator is currently limiting the DFG grouping factor that can be adopted to the FPGA in a many cases. In the future, optimized address encoding and compression scheme will be adopted to further improve the resulting accelerator performance.

9. CONCLUSIONS

In this work, we have presented the QuickDough compilation framework for high productivity development of FPGA-based accelerators. QuickDough makes use of a soft coarse-grained reconfigurable array as an overlay architecture to greatly improve the designer's compilation experience. The QuickDough overlay is simple, regular, deterministic, and is highly scalable to future devices. Taking advantage of the overlay, the lengthy low-level implementation tool flow is reduced to a rapid operation scheduling problem. Compared to a typical design methodology based on off-the-shelf high-level synthesis tools, the compilation time of QuickDough is reduced by 2 orders of magnitude, which contributes directly into higher application designers' productivity.

Despite the use of an additional layer of overlay architecture on the target FPGA, the overall application performance remains competitive in many cases. 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 both contribute to the competitive performance of QuickDough.

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.