Soft CGRA Overlay Library Assisted Rapid FPGA Loop Accedlerator Generation

Cheng Liu, Ho-Cheung and Hayden Kwok-Hay So

***Abstract*—The design productivity of FPGA development which remains magnitudes lower compared to typical software development severely hinders the widespread adoption of FPGAs. Particularly, the lengthy low-level FPGA implementation process including synthesis, placing and routing dramatically limits the number of compile-debug-edit cycles per day and lowers the FPGA design productivity. To address this design productivity problem, we have developed a rapid FPGA loop accelerator generation framework called QuickDough. Instead of trying to reduce the implementation time, it reuses a pre-built accelerator library to avoid the lengthy implementation process during design iterations. By utilizing a soft coarse-grained reconfigurable array (SCGRA) overlay built on top of off-the-shelf FPGAs as the backbone of the accelerators in the library, it compiles a high-level loop to the FPGA through a rapid operation scheduling first and then generates the FPGA accelerator bitstream through a rapid integration of the scheduling result and a pre-built accelerator bitstream selected from the library. According to the experiments, QuickDough is able to produce accelerators in the order of seconds while achieving up to 9X performance speedup over the execution of the same software running on a hard ARM processor.**

***Index Terms*—Soft CGRA Overlay, Loop Accelerator Generation, Design Productivity, Accelerator Library**

I. Introduction

Recent years have witnessed a tremendous growth in the use of accelerators in computer systems to improve the systems' performance and energy efficiency [16, 22, 25, 26, 29]. Among these accelerators, GPUs and Xeon Phi accelerators have stood out as two of the most popular choices in spite of their relatively short history as accelerators–5 of the top 10 systems on the top500 list take advantage of them [32]. On the other hand, despite the long and successful track record of FPGA accelerators with promising performance speedup and energy efficiency [1, 6, 26, 29, 31], the use of FPGA accelerators in main-stream systems remains limited and has yet to receive widespread adoption beyond highly skilled hardware engineers [8]. When compared to the wide adoption of GPUs and Xeon Phi accelerators, it is believed that the much lower design productivity in developing FPGA applications, which is generally caused by both the lengthy FPGA implementation and notorious inaccessibility to software programmers, has become one of the major obstacles that hinder the software developers using FPGA as compute accelerators.

In order to address the FPGA accessibility problem, recent advances in high level synthesis tools have significantly raised the abstraction level of FPGA design, allowing users to effectively express hardware designs using familiar software programming languages such as C/C++, Java, Python and Scala. However, despite the continuously evolved FPGA compilation algorithms and design methods in the past decades, the lengthy FPGA implementation process including synthesis, placing and routing which typically takes a few minutes for the smallest design and upwards to hours or even days for some of the largest design remains a major productivity hurdle to most software developers.

The focus of this work is therefore to address this FPGA implementation challenge. In particular, we are interested in significantly improving the speed of generating FPGA accelerators for compute intensive loops expressed in high level languages while maintaining a competitive overall performance of the resulting FPGA accelerated computing system.

To that end, we have developed QuickDough, a design framework that rapidly generates loop accelerators and their associated software-hardware communication interfaces. By utilizing a soft coarse-grained reconfigurable array (SCGRA) overlay as an intermediate architecture implemented on top of the physical FPGA, we pre-build an accelerator library with a group of various configurations on top of the SCGRA overlay. With the pre-built library, QuickDough generates the loop accelerator by selecting an accelerator configuration from it, schedules the compute operations from the user-provided loop onto the accelerator, and finally updates the pre-built accelerator configuring the target FPGA, which essentially avoids the FPGA implementation process during the accelerator design iterations. By employing different configuration selection algorithms, QuickDough allows users to perform trade-off between performance and compilation time. At the end of the selection process, optimized communication interfaces will be produced accordingly. Meanwhile, QuickDough also helps the users to automatically pre-build an overlay based accelerator library targeting a single application or a group of similar applications. To expedite the library generation process, a small representative set of accelerator configurations are chosen as the library and generated automatically using a template based system.

In Section III, the proposed FPGA acceleration design framework QuickDough including both the SCGRA overlay implementation and compilation will be elaborated. Then the automatic SCGRA overlay based accelerator library pre-building will be detailed in Section **??**. Experimental results are shown in Section V. Finally, we will discuss the limitations of current implementation in Section VI and conclude in Section VII.

II. Related Work

FPGA design productivity remains a major obstacle that hinders the use of FPGA as computing devices from widespread adoption. Many researchers attempt to approach this problem by raising the abstraction level of hardware design. Particularly, decades of research in FPGA high-level synthesis have already demonstrated their indispensible role in promoting FPGA design productivity [8]. Numerous design languages and environments [5] have been developed to allow designers to focus on high-level functionality instead of low-level implementation details. While the high-level abstraction helps to express the desired functionality, the low-level implementation time spent in synthesis, placing and routing is equally crucial to the design productivity.

In order to address the challenge of lengthy hardware implementation tool run time, researchers have proposed a number of approaches from various angles. Some of the researchers focus on the low-level FPGA EDA tools and have tried to decrease the run-time by improving the implementation algorithms, by making quality-runtime trade-offs [24] and by parallelizing the tools themselves [10, 11, 13, 23]. Other researchers take advantages of the dynamic partial reconfiguration capabilities of modern FPGAs to shorten run time by effectively reducing the user design size [**?**]. Yet another group of researchers approach the problem from a higher level, innovating on how these tools are being used from a design methodology's point of view. The use of modular design flow and by using pre-built hard macros [19, 20] have thus been explored. While these approaches have significantly reduced the hardware implementation time, they remain at least 2 orders of magnitude slower when compared to the software compilation experience.

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. Overlays with different granularity ranging from virtual FPGAs [9, 14**? ?**], CGRA overlays [4, 12, 18, 28**?**] to many-core processor arrays [3, 15**?**] and GPU-like overlays [17] have been developed.

Among these overlays, CGRA overlays are particularly suitable for compute intensive loop acceleration as demonstrated by numerous prior works [7, 30]. A large number of CGRAs with different features have been developed and prototyped on FPGAs. A VLIW architecture based CGRA overlay was developed [18], which support dynamic topology customization. A heterogeneous CGRA overlay was proposed [12] that utilized a global multi-stage interconnection to achieve topology customization to adapt to different applications. A customized CGRA overlay called QUKU [28] was developed for DSP algorithms and it supported fast configuration for similarly to this work applications and slow configuration for distinct applications. Finally High-speed CGRA overlays were built in [4] and [**?**] by using the elastic pipeline technique and smart DSP reuse respectively to achieve better performance and higher throughput.

Our proposed fully pipelined synchronous coarse-grained reconfigurable array overlay continues this trend of exploiting coarse-grain reconfigurability to improve both design productivity and the resulting accelerator performance. We particularly focus on using the overlay as the backbone of FPGA loop accelerators targeting a hybrid CPU-FPGA computing system. With the pre-built library, the loop accelerator can be generated in seconds. In addition, QuickDough pays particular attention to the communication between the host processor and the accelerator and provides communication optimization automatically, creating a seamless hardware-software co-design experience for the user. Finally, our overlay was designed to be *soft* from the beginning, featuring a template system to allow for rapid overlay generation for either a single application or a domain of applications.

III. QuickDough Framework

QuickDough is a development framework for FPGA-accelerated applications. It generates FPGA accelerators for compute intensive loop kernels rapidly through the use of a pre-built soft coarse-grained reconfigurable array (SCGRA) overlay. It also generates the communication infrastructure between the CPU host and the accelerator automatically, integrating both software and hardware generation in a unified framework. The overall design goal of QuickDough is to enhance the designer's productivity by greatly reducing the hardware generation time. Instead of spending hours on conventional hardware implementation tools, QuickDough is capable of producing the targeted hardware-software system in the order of seconds. By doing so, it provides a rapid development experience that is compatible with that expected by most software programmers.

To achieve this compilation speed, while maintaining a reasonable accelerator performance, QuickDough avoids the creation of custom hardware directly for each application. Instead, the compute kernel loop bodies are scheduled to execute on an SCGRA overlay based accelerator, which is selected from a pre-built library. By sidestepping the time-consuming low-level hardware implementation tool flow, the time to implementing an accelerator in QuickDough is reduced to essentially just the time spent on accelerator selection and scheduling compute operations on the resulting overlay based accelerator.

A. QuickDough Overview

Figure 1 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 while the rest part of the program is compiled to the processor through a conventional software compilation.

The focus of QuickDough is to rapidly generate the FPGA loop accelerator. As QuickDough takes an SCGRA overlay as the backbone of the resulting accelerator, the loop accelerator generation is essentially mapping the high level loop kernel to the SCGRA overlay. To that end, the loop kernel is statically transformed to the corresponding data flow graph (DFG)

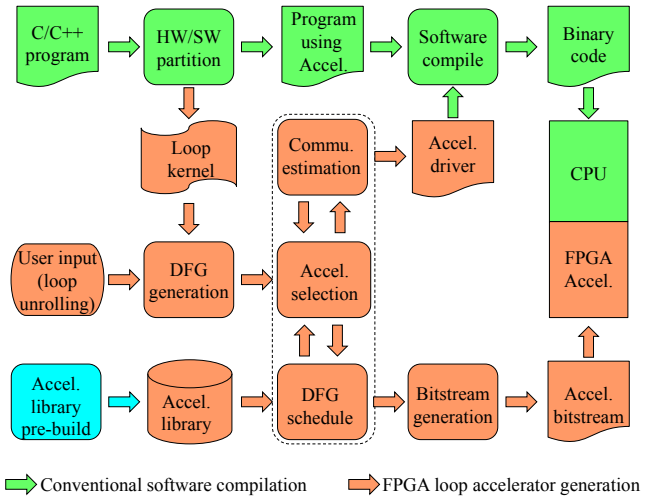

Fig. 1. QuickDough: FPGA loop accelerator design framework using pre-built accelerator library. The compute intensive loop kernel of an application is compiled to the FPGA accelerator while the rest is compiled to the host processor.

with user-specified loop unrolling factor. Then the accelerator selection process selects an accelerator from a pre-built accelerator library based on the scheduling performance and the estimated communication cost. The scheduling performance is obtained from the DFG scheduling process which schedules the generated DFG to the SCGRA overlay included in the selected accelerator while the communication cost is obtained through a CPU-FPGA communication estimation model. After the accelerator selection process, the accelerator drivers can be generated accordingly based on the on-chip buffer size of the selected accelerator. Meanwhile, the selected empty pre-built accelerator bitstream and the corresponding scheduling result are integrated to create the final FPGA accelerator configuration bitstream. This bitstream, in combination with the binary code created in the software compilation process, forms the final application that will be executed on the target CPU-FPGA system.

As shown in Figure 1, QuickDough still needs a pre-built SCGRA overlay based accelerator library, which is also a challenge to the designers. To address this challenge, the accelerator library pre-building process is also done automatically for the target high-level loop kernels. Detailed accelerator library pre-building process will be elaborated in next section.

B. SCGRA overlay based FPGA accelerator

QuickDough utilizes an SCGRA overlay as the backbone of the resulting accelerators. Figure 2 shows the typical structure of an SCGRA overlay based accelerator. It consists of an array of homogeneous simple processing elements (PEs) connected by a direct network executing synchronously. 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 design framework. 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

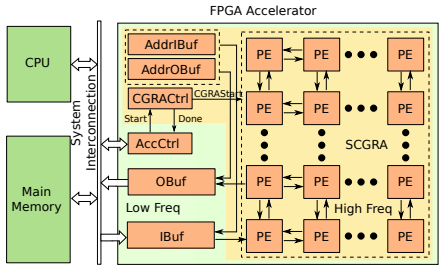

Fig. 2. SCGRA overlay based FPGA accelerator

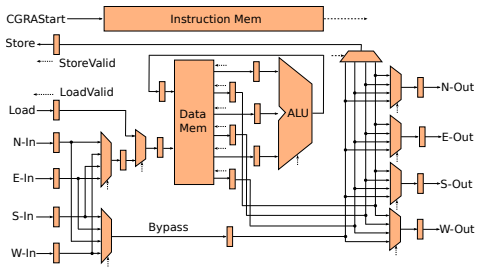

Fig. 3. Fully pipelined PE structure.

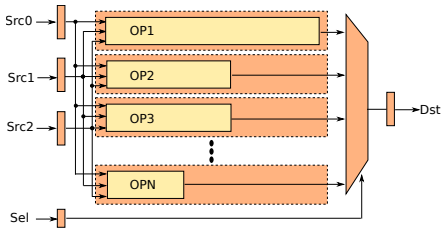

Fig. 4. QuickDough ALU supporting up to 16 pipelined 3-input operations.

system. The exact buffer location to be accessed is controlled by the AddrIBuf and AddrOBuf blocks. Both of them are ROM populated with address information generated from the QuickDough compiler.

1) PE template: Figure 3 shows the current implementation of a QuickDough PE template 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 multiplexers 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 neighboring PEs, as well as from the optional IBuf loading path. The action of the PE is controlled by the control words that are read from the instruction memory. Finally, a global signal (*CGRAStart*) from the *SCGRACtrl* block controls the start/stop of all PEs in the array.

2) ALU template: At the heart of the proposed PE is the ALU and it can easily be customized to support different operations specifically for any given user applications. Figure 4 shows the ALU template used in the QuickDough overlay. These operators in the ALU may execute concurrently in a pipelined fashion and must complete in a deterministic number of cycles. Given the deterministic nature of the operators, the QuickDough scheduler will ensure that there is no conflict at the output multiplexer.

*3) Accelerator Controller (*AccCtrl*):* As the accelerator is eventually attached to a processor, it also has a standard accelerator controller AccCtrl to facilitate the HW/SW co-design. Basically, it generates the corresponding triggering signal when the host processor starts the computing and acknowledges the host processor when the computing is done. Most importantly, it also generates cyclic controlling signal that repeats the computing array multiple times which helps to make good use of the communication bandwidth between the main memory and the on-chip buffer. This will be detailed in next subsection. In addition, part of the accelerator controller that is connected to the host processor must handle complex system interconnection protocol and work at lower clock frequency. The other part of the controller that is coupled with the highly pipelined processing array typically work at higher clock frequency.

C. Loop execution on the accelerator

The loop kernels are mostly partially unrolled, transformed to DFGs and scheduled to the SCGRA overlays of the accelerators. A straightforward way to perform the whole loop computation on the overlay is to repeat the same DFG computation until the end of the loop. Nevertheless, this may require data transfer between host processor and I/O buffer for each DFG computation. As a result, the communication cost increases dramatically especially when the amount of each data transfer is small. Worse still, input data of the consecutive DFGs may be reused and the straightforward data transfer strategy may greatly increase the total amount of data transfer through out the loop computation.

To alleviate this problem, we have proposed to batch data transfers for multiple executions of the same DFG into groups as shown in Figure 5. Specifically, after the loop is unrolled U times, G of them are grouped together for each data transfer. This group strategy helps to amortize the initial communication cost between host processor and the accelerator. In addition, it allows input data to be reused for different DFG computation in the same group and the group size is mainly limited by the I/O buffer depth. Meanwhile, the accelerator communicates with host processor for each group execution. The group strategy is also supported by the proposed accelerator micro-architecture as mentioned in previous subsection. Basically, the number of cycles for each DFG execution and the number of the DFG to be iterated in a group should be configured in *AccCtrl*. Finally, the accelerator driver that handles the communication depends on the I/O buffer depth as well. Clearly, accelerator with larger I/O buffer is preferable when the rest part of the accelerator configuration fulfills the requirements.

D. FPGA loop accelerator generation

The aim of QuickDough is to produce FPGA loop accelerator rapidly and thus the loop accelerator generation is the most critical part of QuickDough. It consists of five dependent processes including DFG generation, accelerator selection, DFG scheduling, communication estimation and bitstream integration. They will be detailed in the following sub sections.

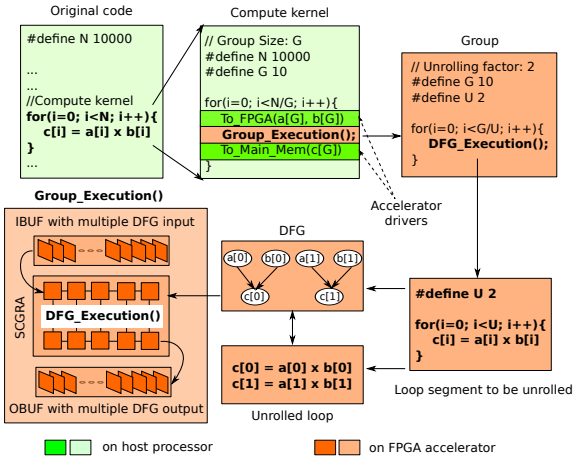

Fig. 5. Loop execution on an SCGRA overlay based FPGA accelerator

1) DFG generation: In order to produce an FPGA loop accelerator using SCGRA overlay, DFGs are extracted from the kernel that is often expressed as inner loop body. In order to transform the loop body to a DFG, data dependency in the loop body must be known at compilation time. Similar to LLVM intermediate representation, simple branches in the loop body can be removed by using PHI instruction which can further be mapped to PHI operation in ALU. The users may further unroll the loops multiple times to increase the amount of operation parallelism in the generated DFG. In this work, we have developed a C++ library to help automate the DFG generation with specified loop unrolling factor.

2) DFG Scheduling: 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 [27] 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 [21], 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

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
```

produced. These control words will subsequently be used for bitstream generation in the following compilation step.

3) Accelerator selection: Accelerator selection process selects an accelerator from the accelerator library based on the resulting accelerator performance which mainly includes the computation latency and communication latency. The computation latency of the loop kernel can be calculated using (1). $DFG\_Lat(SCGRA\_Size)$ stands for the number of cycles needed to complete the DFG scheduling and mostly depends on the SCGRA overlay size while $Freq$ stands for the pre-built accelerator implementation frequency. The communication latency can be calculated using (2) where $Trans()$ represents the data transfer latency function of the target platform and $GpIn$ and $GpOut$ represent the amount of data transfer of a group.

$$CompLat = DFG\_per\_Loop \times DFG\_Lat(SCGRA\_Size)/Freq \quad (1)$$

$$CommLat = Gp\_per\_Loop \times (Trans(GpIn) + Trans(GpOut)) \quad (2)$$

On top of the SCGRA size, the rest design parameters also affect the accelerator selection. However, the influence can be immediately analyzed given the DFG scheduling result. For instance, the instruction memory depth must be larger than or equal to DFG_Lat which represents the number of control words in each PE. Similarly, data memory capacity requirement of each is also decided via the DFG scheduling. The input/output buffer depth must be larger than or equal to $GpIn$ / $GpOut$. In other words, the maximum grouping factor is decided by the capacity of the I/O buffers. Thus the communication latency is essentially determined by the input/output buffer depth according to (2).

In summary, the performance of the accelerator can be estimated with the analytical models when the scheduling performance is obtained through the DFG scheduling while the scheduling performance is mostly determined by the SCGRA overlay size. The analytical estimation is fast while the scheduling process is relatively slow. Therefore, the accelerator selection process essentially centers the SCGRA overlay size selection and then explores all the accelerator configurations with the same SCGRA overlay size.

To compromise the loop accelerator generation time and performance, three different levels of accelerator selection optimization levels are provided in this framework namely O0, O1 and O2 centering the SCGRA overlay size selection. O0 doesn't provide any optimization, and it selects an accelerator with the smallest SCGRA overlay. O1 estimates three typical accelerators with the smallest SCGRA overlay, a medium one and the largest SCGRA overlay. Then the one that provides the best performance will be adopted. O3 explores all the accelerators in the library and searches for the best accelerator configuration. With the increase of the optimization level, the accelerator selection process spends more efforts in searching the accelerator library for better performance and thus results in longer acceleration generation time.

4) Accelerator bitstream generation: The final step of the accelerator generation is to generate the instructions for each PE and the address sequences for the I/O buffers based on the scheduling result, which will subsequently be incorporated into the configuration bitstream of the overlay produced from previous steps. Then 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 the `data2mem` tool from Xilinx [33] 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 [2]. This bitstream integration process costs only a few seconds of the compilation time.

IV. Automatic Accelerator Library Pre-building

Accelerator library pre-building is essentially to pre-implement a group of SCGRA overlay based FPGA accelerators upon users' request. It is the basis for the proposed rapid FPGA loop accelerator generation framework. As QuickDough aims to enhance the designers' productivity and make FPGA accelerator design accessible to high-level application designers, the library pre-building process which involves low-level circuit design and optimization is thus automated so that it will not become a new barrier to the application developers. In this section, we will illustrate how the accelerator library is pre-built given the hardware resource budget and target loop kernels.

Figure 6 presents the proposed 4-step automatic accelerator library pre-building flow. Given a group of high-level loop kernels to be accelerated using QuickDough, it extracts DFGs from the loop kernels first by using the DFG generator. Afterwards, operation sets that are needed for each loop kernel can be decided. By utilizing an operation set similarity metric, the loop kernels are divided into different clusters in the second step. Basically, loop kernels that require a similar operation set are considered as the same cluster and therefore corresponding accelerator library will be pre-built for each cluster of applications in next steps. After the loop kernel classification, a representative accelerator configurations will be generated for each cluster of loop kernels based on the SCGRA overlay template and the resource constraints in the

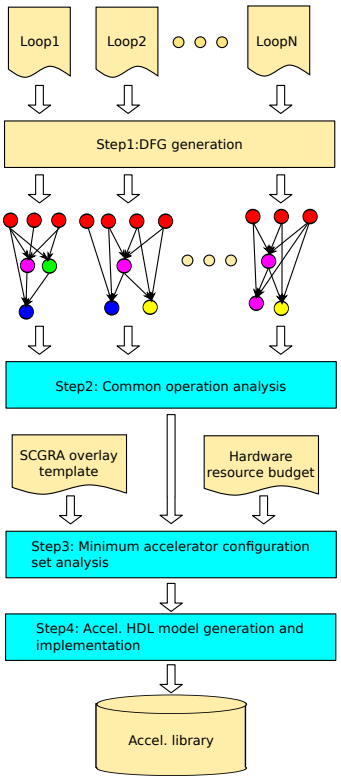

Fig. 6. Automatic SCGRA overlay based FPGA accelerator library update

third step. Finally, the accelerator libraries to be built will be implemented on a parallel computing machine. Since DFG generation has been discussed in previous section, we will mainly detail the rest three steps in this section.

A. Common operation analysis

As mentioned in previous sub section, the common operation analysis step first divides the loop kernels into different clusters based on the operation set similarity between the loop kernels and then generates a common operation set for each cluster of applications.

In this work, we use Jaccard coefficient as the similarity metric. Suppose there are N loop kernels to be analyzed, C_i and C_j are the operation sets of two loop kernels where $i, j = 1, 2, ..., N$. Then the Jaccard coefficient of any two loop kernels S_{ij} can be calculated using (3).

$$S_{ij} = \frac{C_i \cap C_j}{C_i \cup C_j} \tag{3}$$

As some of the operations can be complex and consume a lot of hardware resource, it may be better to build separate library for loop kernels that only differ on such an operation. To that end, we may also take the overhead of the operations into consideration while calculating the similarity coefficient. Assume IW_{ij} and UW_{ij} as weight vectors of operations in $C_i \cap C_j$ and $C_j \cup C_j$ respectively, and then the similarity matrix of the loop kernels can be calculated using (4) instead.

$$S_{ij} = \frac{\sum(IW_{ij})}{\sum(UW_{ij})} \tag{4}$$

When the similarity matrix of the target loop kernels are obtained, a user specified threshold T is used to decide whether the two loops can be considered as a cluster. According to the definition, S_{ij} is close to 1 when two operation sets are similar. Thus when $S_{ij} \geq T$, the two loop kernels can be put in the same cluster. Otherwise, they will be divided into separate clusters.

When the loop kernels are divided into different clusters, we can further obtain the minimum common operation set of each cluster of loop kernels, which is essentially a union of the operation sets included in the loop kernels of the same cluster. The analysis is trivial, but the minimum operation set can be decided automatically and rapidly.

B. Minimum accelerator configuration set analysis

Although the library can be implemented off-line, it does take a long time to complete. Therefore, we try to find out the minimum set of accelerator configurations that need to be pre-implemented as the library for each cluster of loop kernels and maintain the application coverage of the library at the same time.

The proposed SCGRA overlay based FPGA accelerator utilizes block RAM to implement the instruction memory, data memory, on-chip buffer as well as the address buffer, and block RAM is the hardware resource bottleneck. As a result, the library basically depends on how the block RAM budget is allocated to different components of the accelerators. Therefore, the minimum library can be obtained using equation (5). Row and Col stand for the SCGRA overlay size and they are integers. IM, DM, $AIOB$ and $DIOB$ stand for the instruction memory capacity, data memory capacity, address IO buffer capacity and IO buffer capacity. In addition, they can only increase with the granularity of a primitive block RAM. B stands for the user specified block RAM budget.

$$Row \times Col \times (IM + DM) + AIOB + IOB \leq B \tag{5}$$

Moreover, empirical settings such as limiting data memory in each PE to a single primitive block RAM (i.e. $DM = 1$), constraining the difference between SCGRA row size and column size (i.e. $Col \leq Row \leq (Col + Gap)$, Gap is an integer) and setting $AIOB = IOB$ are employed to further reduce the number of accelerators pre-built in the library. Meanwhile, the accelerator configurations to be built are also determined.

C. Accelerator HDL model generation and implementation

With the proposed SCGRA overlay template and the required operation set, ALU data path will be updated first. Currently, the data paths of the ALU are manually created, but it is possible to rely on the high level synthesis tools in future. Then HDL models of the accelerators can be generated based on the configurations decided in the third step. This part is done with a python script.

When all the HDL codes of the accelerators are generated, they can be further implemented using the classical hardware implementation tools. As the implementation jobs are completely independent, they can be done in parallel on either a multi-core processor or a cluster conveniently. Moreover, the regular tiling structure even allows the implementations to be accelerated using macro based implementation techniques as

presented in [**?**], which can be up to 20X faster than a standard HDL implementation with negligible timing and overhead penalty. After the implementation, implementation frequency is added to the corresponding accelerator configuration, which completes the whole library generation process.

V. Experiments

With an objective to improve designers' productivity in developing FPGA accelerators, the key goal of QuickDough is to reduce FPGA loop accelerator development time for a hybrid CPU-FPGA system. By using 32 typical loop kernels as the benchmark, we have evaluated the FPGA accelerator generation time with QuickDough. Meanwhile, to warrant the merit of such framework, the performance of the generated acceleration system should remain competitive. For that purpose, the performance is then compared against to that of software executed on an ARM processor. Finally, the pre-built accelerator library that affects both the design productivity and overhead of the resulting accelerators is also discussed.

VI. 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.

VII. Conclusions

In this paper, we have proposed QuickDough, an SCGRA overlay based FPGA accelerator design method, to compile compute intensive applications to a CPU+FPGA system. With the SCGRA overlay, the lengthy low-level implementation tool flow is reduced to a rapid operation scheduling problem. The compilation time from high level language application to the CPU+FPGA system is reduced by around two magnitudes, which contributes directly into higher application designers' productivity.

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

References

[1] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 126–131. IEEE, 2009.

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

[3] Srinivas Boppu, Frank Hannig, and Jürgen Teich. Compact code generation for tightly-coupled processor arrays. *Journal of Signal Processing Systems*, 77(1-2):5–29, 2014.

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

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

[6] Shuai Che, Jie Li, Jeremy W Sheaffer, Kevin Skadron, and John Lach. Accelerating compute-intensive applications with gpus and fpgas. In *Application Specific Processors, 2008. SASP 2008. Symposium on*, pages 101–107. IEEE, 2008.

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

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

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

[10] Altera Corporation. Quartus II 14.0 handbook. https://www.altera.com/en_US/pdfs/literature/hb/qts/quartusii_handbook.pdf, 2015. [Online; accessed 18-March-2015].

[11] Xilinx Corporation. Command line tools user guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/devref.pdf, 2015. [Online; accessed 18-March-2015].

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

[13] Jeffrey B Goeders, Guy GF Lemieux, and Steven JE Wilton. Deterministic timing-driven parallel placement by simulated annealing using half-box window decomposition. In *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pages 41–48. IEEE, 2011.

[14] David Grant, Chris Wang, and Guy G.F. Lemieux. A CAD framework for Malibu: An FPGA with time-multiplexed coarse-grained elements. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 123–132, New York, NY, USA, 2011. ACM.

[15] Frank Hannig, Vahid Lari, Srinivas Boppu, Alexandru Tanase, and Oliver Reiche. Invasive tightly-coupled processor arrays: A domain-specific architecture/compiler co-design approach. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):133, 2014.

[16] SangDon Kim, SeongMo Lee, SangMuk Lee, JiHoon Jang, Jae-Gi Son, YoungHwan Kim, and SeungEun Lee. Compression accelerator for hadoop appliance. In RobertC.-H. Hsu and Shangguang Wang, editors, *Internet of Vehicles - Technologies and Services*, volume 8662, pages 416–423. Springer International Publishing, 2014.

[17] Jeffrey Kingyens and J. Gregory Steffan. The potential for a GPU-Like overlay architecture for FPGAs. *Int. J. Reconfig. Comp.*, 2011, 2011.

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

[19] Sebastian Korf, Dario Cozzi, Markus Koester, Jens Hagemeyer, Mario Porrmann, U Ruckert, and Marco D Santambrogio. Automatic HDL-based generation of homogeneous hard macros for FPGAs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 125–132. IEEE, 2011.

[20] Christopher Lavin, Brent Nelson, and Brad Hutchings. Improving clock-rate of hard-macro designs. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 246–253. IEEE, 2013.

[21] Colin Yu Lin and Hayden Kwok-Hay So. Energy-efficient dataflow computations on FPGAs using application-specific coarse-grain architecture synthesis. *SIGARCH Comput. Archit. News*, 40(5):58–63, March 2012.

[22] T. Majumder, P.P. Pande, and A. Kalyanaraman. Hardware accelerators in computational biology: Application, potential, and challenges. *Design Test, IEEE*, 31(1):8–18, Feb 2014.

[23] Yehdhih Ould Mohammed Moctar and Philip Brisk. Parallel fpga routing based on the operator formulation. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, pages 1–6. ACM, 2014.

[24] Chandra Mulpuri and Scott Hauck. Runtime and quality tradeoffs in fpga placement and routing. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 29–36. ACM, 2001.

[25] A. Putnam, A.M. Caulfield, E.S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G.P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P.Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24, June 2014.

[26] Souradip Sarkar, Turbo Majumder, Ananth Kalyanaraman, and Partha Pratim Pande. Hardware accelerators for biocomputing: A survey. In *IEEE International Symposium on Circuits and Systems*, pages 3789–3792, 2010.

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

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

[29] Iouliia Skliarova and Antonio De Brito Ferrari. Reconfigurable Hardware SAT Solvers: A Survey of Systems. *IEEE Transactions on Computers*, 53:1449–1461, 2004.

[30] Russell Tessier and Wayne Burleson. Reconfigurable computing for digital signal processing: A survey. *Journal of VLSI signal processing systems for signal, image and video technology*, 28(1-2):7–27, 2001.

[31] David Barrie Thomas, Lee Howes, and Wayne Luk. A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 63–72. ACM, 2009.

[32] TOP500. Top500 list june 2015. http://top500.org/lists/2015/06, June 2015. [Online; accessed 18-October-2015].

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