A Soft Coarse-Grained Reconfigurable Array Based High-level Synthesis Methodology: Promoting Design Productivity and Exploring Extreme FPGA Frequency

Liu Cheng
The University of Hong Kong
liucheng@hku.hk

ABSTRACT

Compared to the use of a typical software development flow, the productivity of developing FPGA-based compute applications remains much lower. Although the use of high-level synthesis (HLS) tools may partly alleviate this shortcoming, the lengthy low-level FPGA implementation process remains a major obstacle to high productivity computing, limiting the number of compile-debug-edit cycles per day. Furthermore, high-level application developers often lack the intimate hardware engineering experience that is needed to achieve high performance on FPGAs, therefore undermining their usefulness as accelerators. To address these productivity and performance problems, a high-level synthesis methodology that utilizes soft coarse-grained reconfigurable arrays (SCGRAs) as an intermediate compilation step is presented. Instead of compiling high-level applications directly as circuits implemented on the FPGA, the compilation process is reduced to an operation scheduling task targeting the SCGRA. Furthermore, the softness of the SCGRA allows domain-specific design of the processing elements, while allowing highly optimized SCGRA array be developed by a separate hardware design team. An SCGRA operating at over 400MHz on a commercial FPGA is presented here. When compared to commercial high-level synthesis tools, the proposed design methodology achieved 0.8-21x times speedup in the application run time while application compilation time is reduced by 10-100x.

1. INTRODUCTION

The use of FPGAs as compute accelerators has been demonstrated by numerous researchers as an effective solution to meet the performance requirement across many application domains. However, despite years of research with numerous successful demonstrations, the use of FPGAs as computing devices remains a niche discipline that has yet to receive widespread adoption beyond highly skilled hardware engineers.

Compared to a typical software development environment, developing applications to execute on FPGAs is a lengthy and tedious process. While advancements in high level synthesis (HLS) tools have helped lower the barrier-to-entry for novel users, achieving the desired performance in the final design usually demands detailed low-level design engineering efforts such as retiming and pipelining that are foreign to many high-level application designers. Furthermore, unless the HLS tool can directly synthesize the target FPGA configuration, vendor's backend implementation tools must be employed for tasks such as floorplanning and placing-and-routing of the design. Compared to software compilation, the run-time of such backend implementation tools is at least 2 to 3 orders of magnitude slower. Spending several hours on place-and-route alone is not uncommon for a complex design targeting a state-of-the-art FPGA. As a result, the number of compile-debug-edit cycle per day achievable is greatly limited, hindering the productivity of the designers especially during early development phases.

To address such productivity bottleneck, we propose a high-level synthesis methodology that utilizes a *soft* coarse-grained reconfigurable array as an intermediate step to produce a final configurable bitstream directly from applications written in high-level languages. Instead of synthesizing to circuits on the FPGAs, application compute kernels are extracted as dataflow graphs that are *statically* scheduled to operate on the SCGRA. The lengthy hardware implementation tool flow is thus reduced to an operation scheduling problem.

Although the design and implementation of the SCRGA must rely on the conventional hardware design flow, only one instance of the SCGRA design is required per application or application domain. Subsequent application development may then be accomplished rapidly by executing a simple scheduling algorithm. Furthermore, the performance of the SCGRA can be carefully optimized by a separate experienced hardware engineering team. Since the physical implementation of the SCGRA remains unchanged across applications or design iterations, the physical performance of the design can be guaranteed. In this work, we have demonstrated an optimized SCGRA may execute at over 400 MHz on a Xilinx Virtex 6 FPGA, close to the advertised highest clock rate of the device. When compared to a conventional design methodology using commercial HLS tools, an overall 0.8-21x improvement in end-to-end performance has been achieved across the benchmarked applications.

The main contributions of this work can therefore be summarized as follows:

- A SCGRA based HLS methodology is proposed, which improves design productivity through bypassing the lengthy RTL compilation process.
- A fully pipelined SCGRA is developed that executes at almost full speed of the targeted FPGA device.
- An efficient operation scheduling algorithm is imple-

mented specially targeting the proposed fully pipelined SCGRA.

In Section 3, the proposed methodology will be elaborated. Our current SCGRA implementation will be presented in Section 4 and expreimental results shown in Section 5. Finally, we discuss limitation of current implementation in Section 6 and conclude in Section 7.

2. RELATED WORK

To improve the productivity of FPGA designers, researchers have approached the problem both by increasing the abstraction level and by reducing the compilation time.

In the first case, decades of research in FPGA high-level synthesis have already demonstrated their indispensible role in promoting FPGA design productivity [5]. Numerous design languages and environments [4] have been developed to allow designers to focus on high-level functionality instead of low-level implementation details.

While high-level abstraction may help a designers express the desired functionality, the low-level compilation time spent on synthesis, map, and place-and-route for FPGAs remains a major hindrances to designs' productivity. Researchers have approached the problem from many angles, such as through the use of pre-compiled hard macros [11] in the tool flow, the use of a partial reconfiguration, modular design flow [8], and the use of coarse-grained reconfigurable fabrics upgrading the configurability from bit-level to word-level [6] [7].

Finally, the use of parameterizable VLIW processor array [10] and even the many-core processors [12] as a template to FPGA design has also been proposed demonstrating improve productivity with moderate performance degradation.

Building on top of many of the above ideas, we have opted to utilize a fully synchronous soft coarse-grained reconfigurable array as an intermediate step to compiling high-level compute intensive application. Productivity is improved both from the vastly reduced compilation time, as well as from the high-level abstraction provided by the generic LLVM compiler framework we utilized as front-end.

3. PROPOSED DESIGN METHODOLOGY

Figure 1 depicts an overview of the proposed high-level synthesis methodology. As shown in the diagram, the proposed methodology can be divided into two distinct parts. The first part, shown in the top half of the figure, is expected to execute frequently. It should be executed every time a new design iteration is required, a new debug cycle is started, or simply when a new application within the same application domain is implemented. On the other hand, the second part of the design flow, shown in the bottom half of the figure, is expected to execute infrequently, perhaps on a per-application domain basis. Towards the end of the top half of the flow, the scheduling result is merged with the pre-built bitstream from the bottom half to produce the final downloadable bitstream for the target FPGA.

3.1 Per Application Domain Steps

The goal of the bottom half is to produce a highly optimized SCGRA for the target application domain. The resulting SCGRA should capture key computational characteristics common to the target application domain. For example, depending on the application domain, either fixed point number or floating number system may be employed. The kinds of supported operations in the processing element (PE) may also be fine-tuned at this stage. For instance, complex mathematical operations may be useful in one application domain while they may be omit to conserve hardware resources in other cases. Finally, system-wide parameters, such as the number of PE employed, the PE connection topology, I/O bandwidth and data memory size should also be considered.

Note that since the design of the CGRA is soft, it is always possible to implement a different SCGRA as deemed appropriate. Therefore, the design for the bottom half should be considered a best effort design. It represents a tradeoff between generality and efficiency – A generic solution helps to avoid executing the lengthy bottom half, saving compilation time, but will inadvertently consume more hardware resources, impacting the maximum clock frequency of the implementation. In Section 4, one class of such SCGRA suitable for our targeted benchmark application will be used to demonstrate how such array can been optimized to execute on extreme frequency of the target FPGA.

3.2 Per Application Steps

The goal of the top half of the design flow is to compile the specific user application to execute on the pre-compiled SCGRA. Since no low-level FPGA implementation tool is involved, the runtime is comparable to common software compilation time of large systems. This enables significant reduction in application development time so long as the SCGRA has already been implemented. The reduced compilation time has net effect on increasing number of achievable debug cycles per day, greatly enhancing the productivity of the designers.

There are three sub-steps in the top half. First, application developed in high-level languages are compiled into an intermediate representation (IR), which in our case, is the data flow graph (DFG) of the compute kernel. Subsequently, a scheduler is invoked to schedule the DFG onto the SCGRA, taking into account the architectural features. Finally, based on the scheduling results, the cycle-by-cycle control words for each PE within the SCGRA is generated and merged into the pre-built bitstream from the bottom half, producing the final updated bitstream for download.

3.2.1 Applications To IR

The first step of our compilation process is to process the user application described in high-level languages into a common intermediate representation (IR) for the subsequent scheduling step. In our current implementation, we have opted to make use of the open source LLVM compiler infrastructure [2] for this task. Apart from having a wide community support, one of the advantages of utilizing LLVM rests on its many readily available front-end for different languages such as C/C++, Java, .NET, Python, etc. This allows easy extension of our work in the future across many different application domains. Currently our applications are written in C.

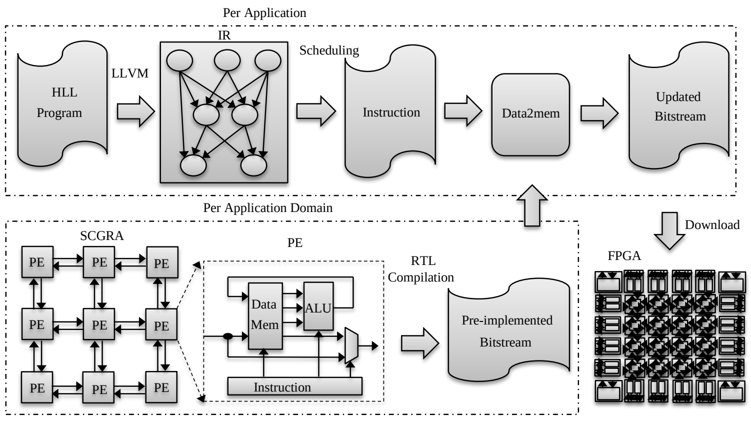

Figure 1: Overview of the proposed soft coarse-grained reconfigurable array based high-level synthesis design methodology.

Given an input application, we begin with manually identifying compute kernels that will be accelerated by FPGAs. Once identified, the compute kernels undergo a series of reshaping to increase the amount of available parallelism. For this purpose, we have initially opted to fully unroll the inner loops of the compute kernels. We note that fully unrolling loops may not always be feasible and may not result in an optimal design. This is left as future extension to this work while we focus on the overall design methodology here.

The identified compute kernel is subsequently compiled to the machine-independent assembly language of LLVM through its Clang frontend for C/C++ programs. It is then processed within the LLVM framework in preparation for intermediate DFG generation. Several LLVM passes, including dead code elimination, loop simplification, and function inline are employed as optimization. Moreover, branch instructions within the kernel body are merged into simple basic blocks by the introduction of PHI instructions. Finally, once the code is transformed into static single assignment (SSA) style, it goes through our in-house pass that transforms the LLVM assemble code into a DFG ready for SCGRA scheduling.

3.2.2 SCGRA Scheduling

In the SCGRA scheduling step, a list scheduling algorithm similar to [13] is adopted to schedule the DFG to the SCGRA infrastructure. It statically schedules the entire DFG on the target SCGRA. Such statically scheduled architecture is crucial in keeping the target SCGRA simple and efficient. To adapt to the proposed SCGRA structure, the scheduling metric is delicately adjusted to compromise the communication cost and load balance.

3.2.3 Bitstream Integration

The final step in our proposed HSL methodology is to incorporate the instruction for each PE obtained from the scheduling stage with the pre-compiled SCGRA design. By design, our SCGRA do not have mechanism to load in instruction streams from external memory. Instead, we take advantage of the reconfigurability of SRAM based FPGAs and stored the cycle-by-cycle configuration words using on-chip ROM. The content of these instruction ROMs are embedded in the configuration bitstream. In particular, the organization of the instruction ROM in the place-and-routed SCGRA design is obtained from its XDL file [3], which in turn allows us to create the corresponding BMM file. With this BMM file, the encoded instructions collected from the DFG scheduling may then be incorporated into the pre-implemented bitstream using the data2mem tool from Xilinx [1]. While original SCGRA design needs hours to implement, the bitstream updating scheme only costs a few seconds.

4. SCGRA DESIGN

One key idea of the proposed design methodology is to rely on an intermediate SCGRA layer to improve compilation time of the high-level user application. While the exact design of this SCGRA does not affect the compilation methodology, its implementation does have a significant impact on the performance of the generated gateware. In this section, an instance of such SCGRA design is thus presented to demonstrate the feasibility of producing high performance gateware without incurring long compilation time. As shown in Figure 2, the PE of this SCGRA is highly optimized for FPGA implementation, centering its design around a hard DSP block with the addition of an instruction ROM and a multi-port data memory. In addition, a load/store path is implemented on the PEs that are responsible for data I/O beyond the FPGA. Using this design as a template, it is envisioned that a separate hardware design team, or the high-level compiler may be able to produce similarly high-performance SCGRA that is optimized for the targeted application domain.

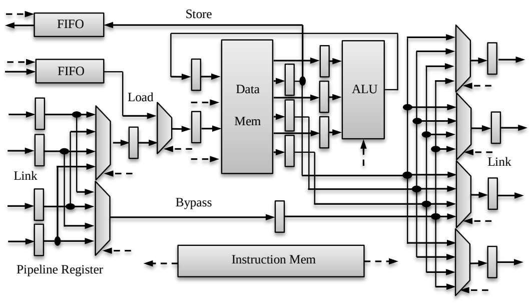

Figure 2: PE structure

4.1 Instruction Memory and Data Memory

There are two types of memory in each PE. The instruction memory stores all the control words of the PE in each cycle.

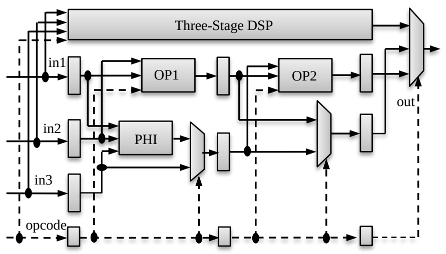

Figure 3: ALU Template

Since its content does not change during runtime, a ROM is used to implement this instruction memory. The content of the ROM is loaded together with the configuration bitstream.

On the other hand, data memory stores intermediate data that can either be forwarded to the PE downstream or be sent to the ALU for calculation. For fully parallelized operation, *at least* four read ports are needed – three for the ALU and one for data forwarding. Similarly, at least two write ports are needed to store input data from upstream memory and to store the result of the ALU in the same cycle. Although a pair of true dual port memories may seems to be able to satisfy this port requirement, conflicts may arise if the ALU needs to read the data while the data path needs to be written. As a result, a third dual port memory is replicated in the data memory.

4.2 ALU

At the heart of the proposed PE is an ALU designed around the DSP core in the target Xilinx FPGA as shown in Figure 3. The DSP core is responsible for all basic arithmetic operations such as multiply-add. In addition, operations that are not provided by the DSP blocks such as add, sub and xor are handled by the 3-stage pipeline that takes the form of $in1$ $<OP1>$ $in2$ $<OP2>$ $in3$. Finally, a special PHI operation was embedded in the template to handle the applications that have branches merged. The PHI operation was implemented simply as a multiplexer, which has little influence on the final PE timing.

4.3 Load/Store Interface

For the PEs that also serve as I/O interface to the SCGRA, an additional load/store path is implemented. In this work, the input data are assumed to be available in the scheduled order via an input FIFO. As such, only one additional signal bit is needed to control the popping of the FIFO. Similarly, another single bit signal is used to decide whether data should be pushed into the output FIFO. These control bits are similarly stored using the reserved bits in the proposed instruction format.

5. EXPERIMENTS

In this section, we take 5 computation kernels including matrix multiply (MM), fast Fourier transform (FFT), discrete convolution (CONV), advanced encryption standard (AES) and Viterbi decoder (VD) as our benchmark. The benchmark is implemented using the proposed HLS methodology and a direct mapping methodology respectively. After that, compilation time, hardware implementation efficiency and performance of the two distinct design methodologies are compared.

5.1 Benchmark

MM has two input matrices and an output matrix. In the experiment, we set the matrix dimension to be 20×20. And there are 8000 multiply-add operations.

FFT initially takes complex vector $x(i)$ as input, and complex vector $X(i)$ as output where $i = 1, 2, 3, ..., N, N = 2^M, M \in \mathbb{Z}$. In order to escape Sine and Cosine computation on FPGA, roots of unity vector are also considered to be input data. In the experiment, we set $N = 256, M = 8$ and all the complex operations are further transformed to real operations. Finally, there are 8192 real multiply-add operations.

Input signal vectors of CONV are $u(i)$ and $h(i)$ where $i = 1, 2, 3, ..., N$. The output signal vector can be expressed as follow:

$$y(k) = \sum_{i=1}^{N} u(i) \times h(k-i), k = N/2, N/2+1, ..., 3N/2$$

In the experiment, $N = 100$ and there are around $3N^2/4$ multiply-add operations.

AES adopts 128-bit key and takes 7 128-bit blocks as input. There are 8864 operations mainly including circular shift and bitwise exclusive or.

VD adopts the NASA standard in which the decoding rate is 1/2 and the decoding length is 7. Input message length is 150 bit and the total operation number is around 2989 including shift, add, or and compare.

5.2 Experiment Setup

All runtimes were obtained on a Linux workstation with an Intel(R) Xeon(R) CPU E5345 and 8GB of RAM. All the implementations targeted Xilinx Virtex6 FPGA (xc6vlx240tff784-1). The proposed HLS methodology employed LLVM v3.2 and clang v3.2 for C program compiling and PlanAhead v13.4 for the SCGRA implementations. As for the direct mapping methodology, AutoESL 2011.4.2 was taken as a representative. Additionally, both design methodologies provided three implementations with different trade-off between performance and hardware overhead for a comprehensive comparison.

AutoESL achieves the trade-off between hardware overhead and performance mostly through altering the loop unrolling factor. Three sets of implementations including no unrolling, medium unrolling and large unrolling were experimented. Configurations of the unrolling for each benchmark are summarized in Table 1. Also, note that all the implementations were set to be fully pipelined using AutoESL directive. In addition, we assumed that the FPGA had direct shared access to the main system memory with the CPU and only a single memory port was presented. As AutoESL transforms each input/output array/vector to a separate input/output memory port, all the input/output arrays/vectors were reorganized as a single input/output vector to ensure that the implementations meet the IO requirement.

Table 1: Detailed unrolling configurations of the benchmark

MM	It is a three-level nested loop and each loop iterates 20. Unrolling factors of three implementations are $1 \times 1 \times 1$, $1 \times 5 \times 20$ and $1 \times 10 \times 20$.
FFT	It is a three-level nested loop. The outmost loop iteration number is 8 while the iteration number of the rest two loops varies with the outmost loop. Unrolling configurations of the three implementations are $1 \times 1 \times 1$, $8 \times 1 \times 1$ and $8 \times m \times n$, $m \times n$ is fixed to be 32.
CONV	It is a two-level nested loop and each loop iterates 100. Unrolling configurations of the three implementations are 1×1, 1×100, 5×100.
AES	AES kernel procedure iterates 9. Unrolling configurations of the three implementations are 1, 3, and 9.
VD	VD kernel iterates 150. Unrolling configurations of the three implementations are 1, 2 and 5.

The proposed HLS methodology obtains the trade-off between hardware overhead and performance by altering the number of PEs in the SCGRA. Three SCGRA designs, with the PEs connected as a 4×4 Torus, a 3×3 Torus and a 2×2 Torus were developed. Each PE contained a 256×16-bit data memory. The ALU within each PE supported 8 types of operations and thus required a 3-bit opcode. In total, PEs without I/O port required 65-bit instruction and PE with I/O required one additional bit for load/store FIFO operations. Since the size of a primitive BRAM on the device was 18K-bit or 36K-bit, we built instruction ROMs with 72-bit width. Furthermore, to cater for different benchmark sizes, 7 instruction ROM sizes – 1K×72-bit, 2K×72-bit, 4K×72-bit, 6K×72-bit, 8K×72-bit, 10K×72-bit and 12K×72-bit – were considered. Combining with the 3 different SCGRA sizes, a total of 21 different SCGRA designs were implemented as target SCGRA platforms.

5.3 Experiment Results

In this section, comparisons of compilation time, hardware implementation efficiency and performance using both design methodologies are presented in detail.

5.3.1 *Compilation Time*

Given a HLL program, the HLS tools should be responsible for the entire process transforming the program all the way to bitstream. Therefore the compilation time that the process consumes is considered to be the metric of the design tools' efficiency.

Figure 4 shows the compilation time of the benchmark using both AutoESL and the proposed HLS methodology. AutoESL includes two steps: AutoESL synthesis and AutoESL implementation. Generally speaking, the time spent on AutoESL synthesis is relatively short, ranging from seconds to dozens of seconds. However, the resulting design must go through the lengthy back-end implementation tools such as floor planning, mapping and placing-and-routing. As a result, the compilation time of a single computation kernel with moderate loop unrolling costs around 20 minutes. Larger loop unrolling that requires more hardware components further slows down the AutoESL implementation process. As shown in the Figure 4, the compilation time of FFT and VD with large loop unrolling even exceeds 2 hours.

On the other hand, the proposed HLS methodology bypasses

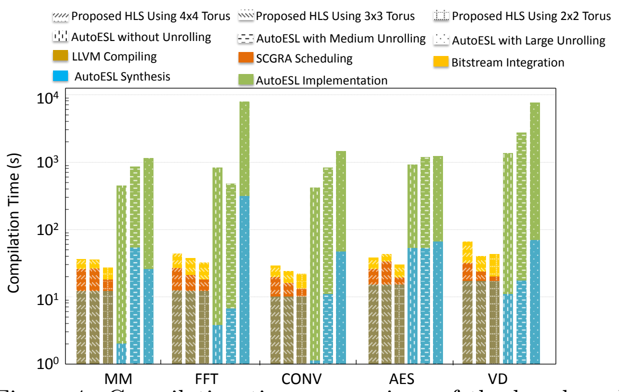

Figure 4: Compilation time comparison of the benchmark using both AutoESL and the proposed HLS methodology

the lengthy low-level implementation steps and simply needs three high-level steps: LLVM compiling, SCGRA scheduling and bitstream integration. Each of these steps could be completed in a few seconds and implementation using this methodology is generally 10X-100X faster than even the smallest implementation using AutoESL.

5.3.2 *Hardware Implementation Efficiency*

In this section, hardware implementation including both hardware resource overhead and implementation frequency using the two design methodologies are compared.

Table 2 presents the hardware overhead of the benchmark using the proposed HLS methodology with 4×4 Torus(P44), 3×3 Torus (P33), and 2×2 Torus (P22), and AutoESL with no unrolling (NUR), medium unrolling (MUR), and large unrolling (LUR). From the table, it can be seen that the proposed HLS methodology tends to use more BRAM than AutoESL does in all the occasions. The prime reason is that the proposed HLS methodology employs SCGRA as the hardware infrastructure and the SCGRA typically requires a number of large instruction memories to store all the control words (instructions). Actually, the BRAM consumption using the proposed HLS methodology roughly equals to the SCGRA scale multiplied by the scheduling result of the target application. It is also the resource bottleneck of the proposed HLS methodology. Hopefully, the context compression techniques [9] used in CGRA design may help remove the instruction redundancy and alleviate the BRAM requirements. While AutoESL mainly uses BRAM for buffering between different sub blocks and IO, the BRAM requirement is much lower because there are few sub blocks in the benchmark.

As for SLICE, LUT, FF and DSP, the consumption using the proposed HLS methodology mainly depends on the SCGRA scale, and it fluctuates within a narrow range when the instruction memory capacity varies. While the consumption of these components using AutoESL increases dramatically with the loop unrolling factor and it will soon eat up all the FPGA resource. When the computation kernels are not unrolled at all, AutoESL provides quite compact implementations and it consumes only a small number of SLICE, LUT, FF and DSP. When the medium loop unrolling is adopted, AutoESL and the proposed HLS methodology cost comparable number of the primitive components. When a large loop

Table 2: Hardware overhead of the benchmark.

		SLICE	LUT	FF	DSP	RAM
MM	T44	2273	4303	11129	16	176
	T33	1029	3229	6958	9	99
	T22	619	1045	2805	4	76
	NUR	158	296	320	1	2
	MUR	1151	3517	4224	78	2
	LUR	1958	6711	8006	182	2
FFT	T44	2273	4303	11129	16	176
	T33	1519	2574	6976	9	171
	T22	619	1045	2805	4	76
	NUR	286	648	823	5	6
	MUR	595	1901	2024	32	20
	LUR	17190	49099	37283	654	20
CONV	T44	2273	4303	11129	16	176
	T33	1029	3229	6958	9	99
	T22	619	1045	2805	4	76
	NUR	87	217	205	1	2
	MUR	1149	3763	4558	98	2
	LUR	4213	13633	19940	497	2
AES	T44	2273	4303	11129	16	176
	T33	1029	3229	6958	9	99
	T22	620	1395	2817	4	108
	NUR	501	1251	1660	0	8
	MUR	705	1799	1947	0	8
	LUR	708	1767	1872	0	8
VD	T44	2782	4811	11209	16	806
	T33	1668	2544	6325	9	387
	T22	725	1298	2825	4	172
	NUR	1911	4883	6684	0	12
	MUR	3306	10033	12863	0	12
	LUR	8833	24693	31886	0	12

unrolling is employed, the overhead of these components using AutoESL turns to be larger than that using the proposed HLS methodology for most applications in the benchmark. The overhead comparison of AES is a bit different, because the SCGRA has more computation capability than that is required. Pre-built more light-weight SCGRA will decrease the SCGRA overhead.

Figure 5 shows the implementation frequency using both tools. It is clear that the SCGRAs employed in the proposed HLS methodology for all the kernels except VD are generally able to work around 400 MHz, which is close to the extreme frequency of the three-stage-pipelined DSP core implemented on the target FPGA. AutoESL has specific implementation for each benchmark, but the highest implementation frequency is only around 300MHz. When large loop unrolling is adopted, the implementation frequency deteriorates radically. The worst implementation frequency as presented in Figure 5 is only around 50 MHz, which offsets the performance improvement brought by the loop unrolling. Essentially, a large number of irregular blocks that randomly scatter around the FPGA using AutoESL increase the difficulty for place-and-route, causing a reduced implementation frequency. In contrast, the SCGRA is regular and well pipelined, therefore the implementation frequency is much higher. Figure 6 displays the frequency of all the 21 SCGRA implementations. It demonstrates that the implementation frequency degrades gracefully with the increase of the SCGRA scale and instruction memory.

5.3.3 *Performance*

In this section, the execution time of the benchmark is taken as the metric of performance. The execution time is computed by multiplying the number of execution cycles with

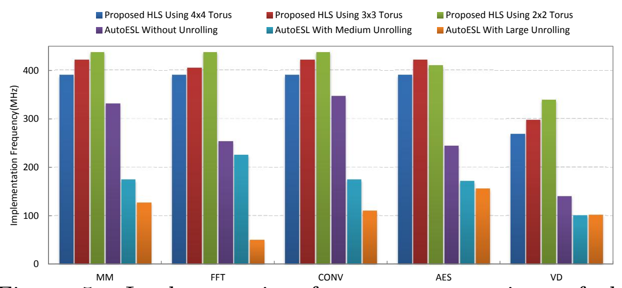

Figure 5: Implementation frequency comparison of the benchmark using both AutoESL and the proposed HLS methodology

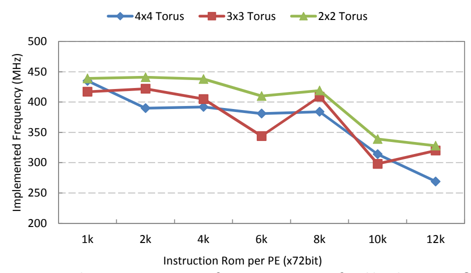

Figure 6: Implementation frequency of all the 21 SCGRAs

the implemented clock period. The number of execution cycles for AutoESL was obtained from the synthesis tools, while that of the proposed HLS methodology was obtained from the SCGRA scheduler. All clock periods of the implementations were obtained from timing reports of the final FPGA implementations.

Figure 7 presents the simulation performance (execution cycles) of the benchmark using both AutoESL and the proposed HLS methodology. Generally, larger loop unrolling using AutoESL leads to better simulation performance. Nevertheless, the performance gain brought by loop unrolling varies in a wide range. CONV with large loop unrolling is almost 20X faster than that without loop unrolling while AES barely benefits from the loop unrolling. The situations of MM, FFT and VD lie between the previous two extreme examples. Basically, there are three reasons for this:

First of all, single input port and single output port constrain the IO bandwidth, and potential parallel operations may be forced to be serial. This is part of the reason for all the unsatisfying performance acceleration.

Second, AutoESL leaves the user to decide the loop unrolling. However, the design space can be extremely large and it is difficult for the user to decide the optimal unrolling factors especially under IO constrain. Take AES as an example. It consists of quite a few sub loops and AutoESL fails to unroll all the sub loops, so we have to leave some of the sub loops unchanged. These serial loops limit not only its own's execution time but also the unrolled loops in the downstream. In this case, randomly unrolling some of the sub loops may have little influence on the performance of the entire application.

Third, source code structure may not be appropriate for AutoESL to synthesize. Take FFT as an example. FFT is a three-level nested loop, and the inner loops depend on the outmost loop. In fact, this is the reason that AutoESL fails to estimate the simulation performance. When the medium loop unrolling is adopted, we manually unroll the outmost loop into 8 stages. And it is natural to connect the 8 stage in serial manner according to the FFT structure. As each stage depends on one after another, the performance acceleration mainly relies on the overlapped computation across the stages. When the large loop unrolling scheme is employed, each stage is further unrolled. Since the neighboring stages communicate through RAM buffers which have limited IO ports, each stage still suffers the same communication bandwidth constrain and loop unrolling in a single stage is almost useless. Declaring multiple smaller vectors to guide AutoESL to synthesize more RAM buffers for communication between neighboring stages and delicately allocating the intermediate data to proper sub vectors to fully reuse the increased communication bandwidth may remove the inner communication bottleneck. Nevertheless, it is challenging for the user to reorganize the source code to fulfill such harsh requirements. In this experiment, two vectors are used to store real part and imaginary part of a complex intermediate data respectively between the neighboring stages. Therefore, it is not surprising that AutoESL fails to accelerate FFT much through the straightforward loop unrolling.

As for the proposed HLS methodology, the performance is more predictable. According to Figure 7, larger SCGRAs with more PEs generally provide higher performance and require more instruction memory while smaller SCGRAs are exactly the opposite. The only exception is VD and there are combined reasons for this. The SCGRA scheduler tends to keep load balance for better performance and operations are scattered across the SCGRA, but the parallel operations in VD are insufficient for scheduling and the communication gets more frequent. While communication in larger SCGRAs is more costly and the deep pipeline of the SCGRA makes the situation even worse, therefore, the smaller SCGRAs achieve even better performance than the larger SCGRAs. This problem can be alleviated by adjusting the scheduling strategy through slightly de-emphasizing the load balance.

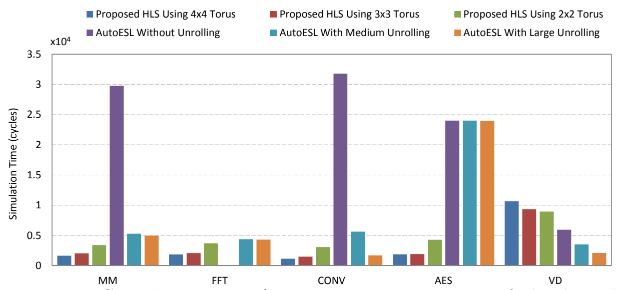

Figure 7: Simulation performance comparison of the benchmark using both AutoESL and the proposed HLS methodology

With the above analysis, we can conclude that the proposed HLS methodology outperforms AutoESL in implementing the benchmark with larger parallel computation but it is not quite effective in handling the benchmark with limited parallelism. Note that we are not denying that AutoESL is able to achieve optimal performance acceleration on FPGA. It is just difficult for the user to figure out proper source code structure and unrolling configuration for the optimal performance acceleration over such a broad design space. The proposed HLS methodology that builds SCGRA over FPGA actually scales down the design space. Meanwhile, the loops are fully unrolled and all the data dependency are completely exposed to the scheduler, so it is getting easier to find an near optimal solution. Particularly, the user simply needs to provide the SCGRA scale and maximum instruction memory depth and doesn't need to make any low level optimization decisions, which makes the design methodology more friendly to the user.

With both the implementation frequency in Figure 5 and simulation performance in Figure 7, the real performance of the benchmark using different HLS tools is acquired as shown in Figure 8. It can be seen that the proposed HLS method outperforms AutoESL in all the benchmark except VD. The highest performance of MM, FFT, CONV and AES using the proposed HLS method is around 7X, 4X, 5X and 21X faster than that using AutoESL respectively, while the performance of VD is a bit slower due to the limited parallelism and the performance acceleration is around 0.8X.

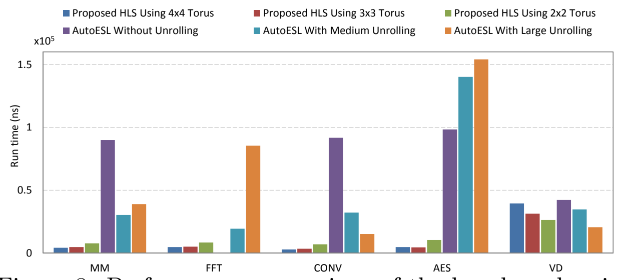

Figure 8: Performance comparison of the benchmark using both AutoESL and the proposed HLS methodology

6. LIMITATIONS AND FUTURE WORK

While the current implementation of our proposed HLS methodology 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 methodology is designed to synthesize parallel computing kernels to execute on FPGAs only. As such, it is not a generic methodology to perform HLS on random logic. Furthermore, 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.

To maximize the amount of parallelism, loops are fully unrolled in the current implementation and thus the loop iterations in the kernels must be known at compile time. In the future, the degree of unrolling should be automatically determined based on the amount of available on chip resources. Since the scheduler depends on lock step execution, all the input data are assumed to be available whenever they are required and all the output data can always be accommodated by the store FIFO. As a result, the application with a large data set may require an extremely large input/output FIFO. In future, we may allow the SCGRA to be stalled to tolerate load/store latency variation and smaller load/store FIFO will be sufficient.

Finally, on-chip ROM resources for instruction storage is our current resource limitation. We intend to alleviate this bottleneck in the future through the use of better instruction encoding schemes and instruction sequence reuse. Partial loop unrolling instead of fully loop unrolling as mentioned above will also help relieve this problem.

7. CONCLUSIONS

In this paper, we have proposed a SCGRA based high-level synthesis (HLS) methodology for compiling computing kernels on FPGAs.

By using an SCGRA as an intermediate compile step, the lengthy low-level implementation tool flow is reduced to a relatively rapid operation scheduling problem. The number of FPGA application debug cycles achievable per day is thus significantly increased, 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 close to maximum clock frequency resulting from the highly regular structure of the SCGRA, in combination with an in-house scheduler that can effectively schedule operation to overlap with pipeline latencies are both contributing factors to such overall high performance.

Compared to a conventional HLS methodology, experiments have shown that design compilation time is reduced by 10-100x while performance of the resulting application run time is improved by 0.8-21x. The implementations resulting from the proposed HLS methodology consume more BRAMs but fewer SLICEs, LUTs, FFs and DSPs when compared to the conventional HLS implementations with relatively heavy loop unrolling.

8. REFERENCES

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

[2] The LLVM compiler framework. `http://llvm.org`. [Online; accessed 19-September-2012].

[3] 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.

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

[5] 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.

[6] 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. IEEE, 2010.

[7] 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.

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

[9] Y. Kim and R.N. Mahapatra. Dynamic context compression for low-power coarse-grained reconfigurable architecture. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 18(1):15–28, 2010.

[10] D. Kissler, F. Hannig, A. Kupriyanov, and J. Teich. A dynamically reconfigurable weakly programmable processor array architecture template. In *Proceedings of the 2nd International Workshop on Reconfigurable Communication Centric System-on-Chips (ReCoSoC), Montpellier, France*, pages 31–37, 2006.

[11] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. HMFlow: Accelerating FPGA compilation with hard macros for rapid prototyping. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 117 –124, may 2011.

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

[13] Hayden Kwok-Hay. Yu, Colin Lin. So. Energy-efficient dataflow computations on FPGAs using application-specific coarse-grain architecture synthesis. In *Highly Efficient Accelerators and Reconfigurable Technologies, The 4th International Workshop on.* IEEE, 2012.