Evaluating The Highly-Pipelined Intel® Stratix® 10 FPGA Architecture Using Open-Source Benchmarks

Tian Tan*, Eriko Nurvitadhi†, David Shih†, Derek Chiou‡
*Department of Electrical and Computer Engineering
University of Texas at Austin, USA
Email: tan.tian@utexas.edu
†Intel Corporation, USA
Email: {eriko.nurvitadhi, david.shih}@intel.com
‡Microsoft Corporation and University of Texas at Austin, USA
Email: dechiou@microsoft.com,derek@ece.utexas.edu

***Abstract*—Intel® Stratix® 10 FPGAs offer a novel architectural feature called HyperFlex™ that enables an extreme degree of pipelining resulting in up to 1GHz clock frequencies. Prior work has evaluated HyperFlex on pre-production Stratix 10 FPGAs using internal designs not accessible to the general public. This paper presents an updated evaluation of HyperFlex on the latest publicly-available production Stratix 10 FPGA using open-source benchmarks. In particular, our evaluation started with seven RTL designs from existing open-source projects, carefully chosen to capture a variety of architectures (simple pipeline to pipeline with loop/M20Ks/DSPs) implementing well-known functions such as crypto, math, and image processing. An FPGA developer who was not an expert in HyperFlex then spent around 250 engineering hours to develop 24 optimized versions of these designs, following the Intel Stratix 10 FPGA HyperFlex optimization guide. Those optimized designs run at 400MHz to 850MHz. In this paper, we describe the optimizations, efforts, and results. Upon publication, those optimized designs will be open-sourced and published in GitHub.**

I. Introduction

Pipelining is a well-known technique to improve the frequency of digital circuits. Intel's latest Stratix 10 FPGAs [1] includes *HyperFlex*, a novel architectural feature that incorporates registers in many of the routing wires to enable additional pipelining that can potentially double the maximum frequency.

Previous studies [2], [3] have described and evaluated HyperFlex. These studies, however, use internal designs in their evaluation, without providing details or source code of those designs, eliminating the possibility of others to reproduce and learn from those results. In addition, those studies were done on pre-production versions of Stratix 10. Stratix 10 is now generally available but, to the best of our knowledge, there are no publications that provide an updated HyperFlex study.

In this paper, we present Stratix 10 results using seven standard open-source RTL benchmarks across multiple domains, specifically crypto, image processing, and math. The benchmarks were carefully chosen to capture a variety of architectures from simple pipeline, to pipeline with tight dependency loops, to using hard blocks such as M20Ks and/or DSPs. We then developed 24 optimized versions of these designs to take full advantage of HyperFlex. All versions were developed in approximately 250 engineering hours. These optimized designs are publicly available [4].

We evaluated the latest publicly-available production Stratix 10 using these baseline and optimized designs. For each version, we report the frequency, throughput, power, FPGA resource utilization, and an estimate of the number of engineering hours needed to implement that version. Our results show that HyperFlex supports 400-800MHz designs at the cost of additional resources, power, and effort. Even a design with a tight, single-cycle loop achieved 733MHz. Overall, HyperFlex offers a richer design space, allowing tradeoff across wider range of performance, resources, and effort.

The rest of the paper is organized as follows. The next section provides the background on HyperFlex and summary of its optimization guidelines. Section III details the benchmarks we used. Section IV describes our evaluation methodology. Section V presents the evaluation results. Finally, Section VI and VII offers related work and concluding remarks.

II. Background

A. Overview of HyperFlex Architecture and Design Flow

Figure 1a shows a high-level overview of the Intel HyperFlex architecture. The new HyperFlex architecture includes bypassable registers (referred as Hyper-Registers) on every input block and routing segment. These Hyper-Registers can be used to pipeline long routing paths or balance the delay of two consecutive pipeline states without occupying dedicated registers in the logic elements.

The new architecture comes with an updated design flow, referred to as the Hyper-Aware Design Flow, as shown in Figure 1b. The new flow includes a Retime stage in the forward compilation pass. During Retime stage, registers in the logic elements are "moved" to registers on routing segments (or vice versa) to balance the delay of all pipeline stages. In the Hyper-Aware Design Flow, Retime stage is always turned on. The new flow also includes a Fast Forward Compiler than analyzes performance bottlenecks and provides optimization recommendations.

B. *Review of HyperFlex Optimizations*

To achieve the best performance with HyperFlex architecture, developers generally follow three optimizations, namely Hyper-Retiming, Hyper-Pipelining, and Hyper-Optimization.

1) Hyper-Retiming: Hyper-Registers do not support general register capabilities such as asynchronous reset, power-up conditions, and so on. Thus, to make a register a candidate for placement into a Hyper-Register, use of such features must be removed from their Verilog/VHDL description after which the compiler can move registers around automatically, without any RTL modification.

2) Hyper-Pipelining: Developers can add pipeline stages wherever convenient, then let the compiler retime the design by automatically moving registers to minimize the critical path delay. Thus, though the developer must add registers manually, the fact that the registers can be placed anywhere, such as the inputs to the component, reduces the effort to do so. In the latest Quartus compiler, pipeline registers can be freely moved across any combinational logic, but not dedicated blocks such as RAM blocks and DSP blocks. For designs that contain dedicated blocks, developers need to identify each combinational piece and insert pipeline stages accordingly.

3) Hyper-Optimization: RTL modifications beyond adding pipeline stages can greatly improve the maximum frequency of a Stratix 10 design. One such RTL modification is to increase number of cycles for block RAM access and multiplication operations. These added cycles can be used by the compiler to fully register dedicated RAM and DSP blocks and allow them to be run at up to 1 GHz. Another type of RTL modification is to pipeline or unroll a feedback loop structure. Unlike with Hyper-Pipelining, where the tool does the work, developers must explicitly manage the data dependencies when pipelining a loop or replicate shared resource when unrolling a loop. Loop pipelining and unrolling is design-specific. A concrete example will be discussed in Section V-B3.

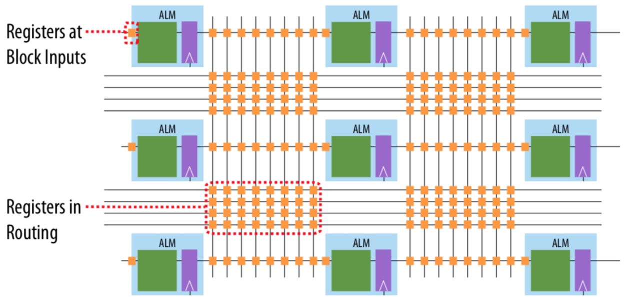
III. Applications

We gathered a set of RTL applications from existing open-source projects to evaluate the Stratix 10 HyperFlex capability. We carefully chose the designs to include varying architectural properties, from simple feedforward pipeline to designs with loops, RAMs, and/or DSPs, and from various application domains. Table I shows these application benchmarks in the collection. We describe each design in more detail below.

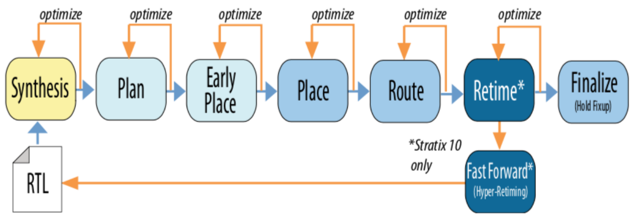
A. **DES**, **MD5**, *and* **DotProduct***: pipeline without loops or hard blocks*

DES [5], **MD5** [5] and **DotProduct** [6] are straightforward pipelines without feedback loops. They also use no dedicated blocks (i.e. M20K RAM block, DSP block) on the Stratix 10. They are selected to evaluate Stratix 10 and HyperFlex architecture for most straightforward scenarios.

DES implements the Data Encryption Standard cryptography algorithm. It performs both encryption and decryption operations. Figure 2a illustrates the high-level architecture of the DES benchmark. The input first goes through an initial



(a) High-level Overview of HyperFlex Architecture



(b) Hyper-Aware Design Flow

Fig. 1. HyperFlex Architecture and Design Flow for Intel Stratix 10 FPGAs. Figures are taken from [1].

TABLE I
RTL Designs Used in Evaluation

Name	Architecture	# pipeline stages	# Lines of Code (LoC)	Origin
DES	Feed forward pipeline	16	1548	OpenCores [5]
MD5	Imbalance feed forward pipeline	64	393	OpenCores
Dot Product	Feed forward pipeline	8	347	Intel [6]
AES	Feed forward pipeline with RAM	29	769	OpenCores
SHA1	Single-cycle loop	1	496	OpenCores
CAVLC	Multi-cycle loop	9	2641	OpenCores
DFT	Feed forward pipeline with RAMs and DSPs	253	8371	SPIRAL[7]

permutation (IP) stage, followed by 16 rounds of identical operations, after which the results go through a final permutation (FP) stages to produce the final output. Each round is essentially eight parallel table-lookup operations. Because the size of the lookup tables is small, the Quartus compiler implements these lookup tables using registers. The baseline version of **DES** encrypts or decrypts one data block in 16 cycles.

MD5 implements the md5 checksum. There are 64 rounds in total to compute a MD5 checksum for a 512-bit word. Unlike DES, not all rounds in MD5 are identical. There are four variants of round operations, each variant being used in 16 consecutive rounds. The baseline version takes 64 cycles to compute a checksum for a 512-bit data block. Compared with **DES**, **MD5** has a longer pipeline and more skewed data path.

DotProduct implements 8-bit dot product operation for vectors of size 16. It is used to evaluate Stratix 10 for emerging machine learning workloads that benefit from low precision operations. Because it is inefficient to implement 8-bit operations using 18-bit hard DSP blocks, in **DotProduct**, low precision arithmetic operations are implemented using pure logic elements. **DotProduct** uses an array of multipliers to compute individual product in parallel. Then, individual products are summed using an adder tree. Figure 2b illustrates the high-level architecture of **DotProduct**. Both Multiplier Array and Adder Tree have 4-cycle latency, resulting 8-cycle total latency for baseline **DotProduct**.

B. **AES** *and* **DFT***: pipelines without loop, with M20K/DSP*

AES [5] and **DFT** [7] represent more complicated designs that utilize dedicated Stratix 10 hard blocks. Both benchmarks use feed forward pipeline structures. They were selected to evaluate how HyperFlex optimizations work with dedicated blocks on Stratix 10.

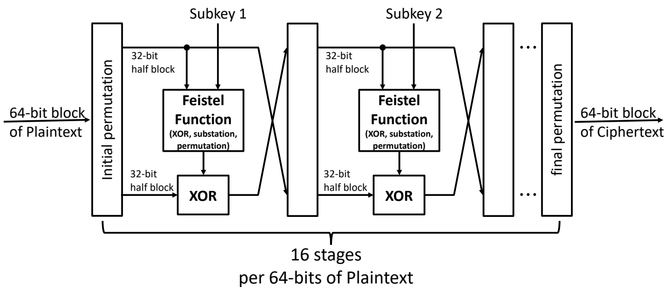
AES implements the AES-256 encryption algorithm. There are 13 regular rounds and 1 final round. Each regular round is paired with a key expansion block to generate the key for next round. A round and key expansion block each requires the same latency (2 cycles in baseline version.) Inside each round and key expansion, the major operations are table lookup operations and XOR operations. Lookup tables in **AES** are larger than those in **DES** and are implemented using dedicated Stratix 10 M20Ks. As a result, the baseline version takes 29 cycles to encrypt one data block.

DFT implements a 64-point discrete Fourier transform. It uses floating-point data. The architecture of **DFT** is a deep pipeline. There are 13 stages in total. Each stage is a small pipeline, with latency ranging from 10 cycles to 31 cycles. Stages communicate with a single "valid" bit. No back pressure is implemented since stages will never block. M20Ks are used to buffer and permute intermediate values. DSP blocks perform floating point multiplications. **DFT** is the most complex benchmark in the collection. It uses 144 dedicated RAM blocks and 52 dedicated DSP blocks on Stratix 10. The **DFT** implementation is generated using SPIRAL [7].

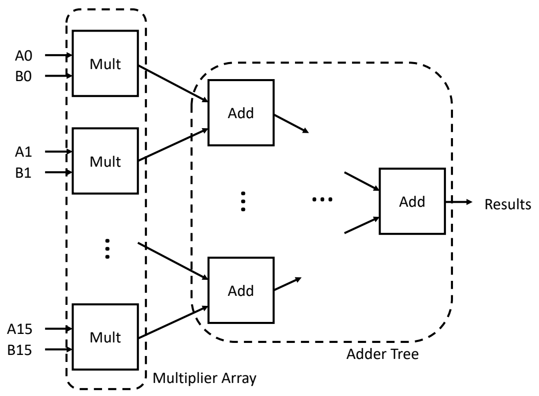
C. **SHA1***,* **CAVLC***: designs with feedback loops*

SHA1 [5] and **CAVLC** [5] are designs with feedback loops. **SHA1** implements the secure hash algorithm 1. Figure 2c illustrates the high-level architecture of **SHA1**. It features a compact design with feedback loops to reduce resource usage. The architecture follows a typical finite state machine with data (FSMD). The data path consists of a set of registers that hold the intermediate values, and a single cycle feedback loop that update the intermediate value every cycle. It takes 80 cycles to process one 512-bit data block.

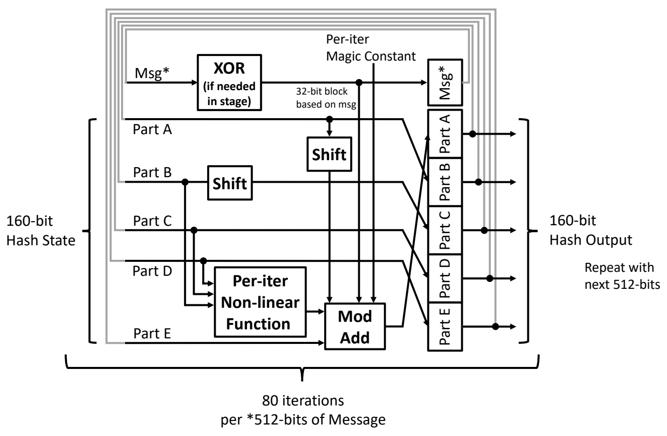
CAVLC implements the decoding algorithm for context-adaptive variable-length coding used in H.264 video encoding. **CAVLC** consists of several submodules running in sequence. An FSM is used to control the firing order of each submodule. Such an architecture leads to a low-power design since most logic can be clock gated. Due to the fact that the encode is variable length, the alignment of next codeword is dependent on the decoding of current codeword, which leads to a multi-cycle feedback loop in CAVLC implementation.


Subkey 1
Subkey 2
64-bit block of Plaintext
Initial permutation
32-bit half block
Feistel Function (XOR, substation, permutation)
XOR
final permutation
64-bit block of Ciphertext
16 stages per 64-bits of Plaintext

(a) **DES**



(b) **DotProduct**



(c) **SHA1**

Fig. 2. Architecture overview of **DES**, **DotProduct**, and **SHA1**

IV. Evaluation Methodology

Intel Quartus Prime Pro 18.1 is used to compile all benchmarks to a Stratix 10 (part number 1SG280LN3F43E1VG), one of the largest, fastest speed grade Stratix 10s. We compile with mostly default settings with the exception of the following.

- Automatic shift register recognition is disabled to increase the opportunities to do retiming. By default, Quartus recognizes shift register chains and maps them into on-chip memory. Disabling that feature forces shift register chains to be implemented using register circuitry, enabling them to be retimed in the later steps. Based on our empirical experience, the resource overhead of disabling shift register recognition is minor.
- The Register power-up condition is set to "don't care" to relax the constraint on using Hyper-Registers. The power-up condition is the state of a design before reset. It is dependent on devices and should not be used as a functional state [1]. Setting power-up condition to "don't care" should have no impact on practical designs.
- The length of synchronization register chains is set to 1 for more retiming opportunities.
- All input and output pins are connected to virtual pins to simulate IP submodules without having to provide the logic that would otherwise wrap those modules.

Quartus TimeQuest Timing Analyzer generates the static timing information for all benchmarks. A 1 GHz clock is used as the timing constraint during the compilation. TimeQuest Timing Analyzer reports the maximum frequency for multiple corners (e.g. 0°C and 100°C). The lowest maximum frequency across all corners is selected as the frequency to be reported. For each benchmark, we explore the design space with six different placement seeds. The one with highest frequency, as defined above, is selected as the one used when reporting resource usage and power.

Resource usage is obtained from the Quartus Fitter report. Because virtual pins use ALMs, we subtract ALMs used for virtual pins from the total ALM usage to obtain the ALM usage of actual designs. The number of M20K blocks, DSP blocks, and registers is obtained from the report directly.

Power consumption is measured with the Quartus Power Analyzer. A toggle rate of 12.5% is selected to determine power. The power consumption is measured at 25°C with typical power characteristics. For each benchmark, the operation frequency is set to its maximum obtained in previous step.

We also tracked the effort to implement the various optimized versions of the original RTL benchmarks. We recorded estimated engineering hours spent. We exclude compilation time, as an engineer is free to spend the time doing other tasks while waiting for compilation to finish. Aside from simulation, all relevant tasks, including design, modifying RTL, validation, and debugging, are included in the recorded hours. We report the number of engineering hours and accumulative Line of Code (LoC) changes between one version and the next version.

V. Results

Results are shown in Table II. For each benchmark, the results for the original RTL version (Ver 0), as well as modified versions (Ver 1 and above) of the original RTL with optimization(s) applied are provided. The major columns show the benchmark, the Hyperflex optimizations applied, FPGA resource usage (ALMs, M20Ks, DSPs, Registers), performance metrics (frequency, throughput, est. dynamic power), as well as estimated effort needed to apply the optimization (engineering hours, lines-of-code changes).

A. HyperFlex Optimizations and Effort

The optimizations applied to the baseline RTL designs (Ver 0 in Table II) along with the effort to implement them are described below.

1) Initial effort and automated optimizations: We first used Quartus to automatically apply retiming optimizations to the Ver 0 designs when possible. Quartus did especially well for balanced pipelined designs without feedback loops. For example, as Table II shows, Ver 0 of **DES**, **DotProduct**, **AES**, **DFT** achieved 539MHz, 802MHz, 496MHz, and 598MHz, respectively. In these cases, there is no additional engineering effort required as no changes were made to the original RTL. There was some typical effort needed to bring-up the designs (e.g., setting up testbenchs), as recorded in estimated engineering hours for Ver 0 in Table II.

B. Hyper-Pipelining by adding registers at inputs/outputs

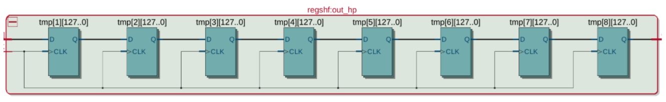
From the initial Ver 0 designs, we explicitly added pipeline stages to the design, without changing the internals of the designs. Manually adding pipeline stages to any design consumes engineering hours, especially when working with designs that the engineer is not familiar with. For fully pipelined designs without loops, one way to apply Hyper-Pipelining is by adding registers at the input and/or output of the design, then allowing Quartus to automatically retime the modified design by moving the registers.

Figure 3 shows an illustration of adding pipeline registers at the front and back of **AES**. The inputs (state, key) and output (out) of the original **AES** design (aes_256:core in Figure 3b) are connected to pipeline registers. In this case, effort was expended to add these pipeline registers. After we made an initial effort to implement pipeline register(s) as a parameterized shift register module (regshf, illustrated in Figure 3a), additional pipelining can be added with minor additional effort by changing parameter values to increase pipeline depths to find the optimal depth. The process of attaching pipeline registers to the inputs (front) and outputs (back) of a pipelined design with no loops can also potentially be automated. As an example of effort involved, for **DES** the effort to change number pipeline registers was an hour (as shown in Table II, **DES** Ver 1 to Ver 2), with 6 lines of code changed, to modify the regshf parameters to enable deeper pipelines to be instantiated. Note that this approach works only if the design is already fully pipelined; otherwise, the design must be converted into a fully pipeline design. As an example, the original **DES** design was mostly pipelined, except for one signal, specifically the encrypt/decrypt mode input bit that gets broadcasted to all the pipeline stages. In this case, some effort was needed to understand the internal design in

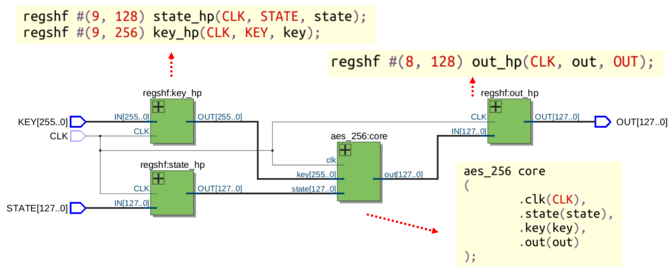
TABLE II
EVALUATION RESULTS

Benchmark		HyperFlex Optimization Applied		FPGA Resource Usage				Performance Metrics			Effort Estimate	
Name	Ver	Hyper Pipelining	Hyper Opt	ALM	M20K	DSP	Reg	Freq (MHz)	Thpt. (Mop/s)	Power (mW)	Engr Hours	LoC change
DES	0			1632	0	0	2239	539.37	539.37	41.15	5	0
DES	1	13 pipeline stages at front		1806	0	0	4628	739.10	739.10	77.32	15	895
DES	2	19 pipeline stages at front		1946	0	0	5757	843.88	843.88	109.37	16	901
DES	3	19 pipeline stages at front	duplicate critical registers	1948	0	0	5796	855.43	855.43	101.57	26	976
MD5	0			12443	0	0	38690	264.76	264.76	226.85	5	0
MD5	1	43 pipeline stages at front		19062	0	0	66345	370.37	370.37	657.98	10	148
MD5	2	63 pipeline stages at front		19361	0	0	80144	408.16	408.16	1101.35	15	153
Dot Product	0			1134	0	0	2202	801.92	24.86G	33.43	5	0
Dot Product†	1	1 pipeline stage at front		1214	0	0	1752	851.79	26.41G	51.56	10	206
Dot Product†	2	4 pipeline stages at front		1366	0	0	3506	851.79	26.41G	67.31	11	207
AES	0			4514	276	0	12491	495.79	495.79	1096.24	5	0
AES	1		fully register RAMs	4965	276	0	15360	638.57	638.57	2261.69	25	635
AES	2	1 stage pipeline register in between RAMs (13 stages in total)	fully register RAMs	6139	276	0	18700	747.38	747.38	2703.25	35	648
AES	3	2 stage pipeline register in between RAMs (26 stages in total)	fully register RAMs	7319	276	0	23548	783.09	783.09	2988.30	36	661
AES	4	3 stage pipeline register in between RAMs (39 stages in total)	fully register RAMs	8469	276	0	27893	786.78	786.78	3017.02	37	674
AES	5	4 stage pipeline register in between RAMs (52 stages in total)	fully register RAMs	9664	276	0	32695	786.16	786.16	3206.91	38	687
AES	6	5 stage pipeline register in between RAMs (65 stages in total)	fully register RAMs	10850	276	0	37406	808.41	808.41	3371.34	39	700
AES	7	6 stage pipeline register in between RAMs (78 stages in total)	fully register RAMs	11984	276	0	41940	811.03	811.03	3526.80	40	713
DFT	0			49888	144	48	130391	598.44	74.81	3745.49	5	0
DFT	1	1 pipeline stage at front, 1 pipeline stage in between RAMs (14 stages in total)		52777	144	48	140453	608.27	76.03	4229.46	15	1289
DFT	2	3 pipeline stage at front, 2 pipeline stage in between RAMs (29 stages in total)	hard FP multiplier	50929	144	52	150288	651.47	81.43	4901.11	30	1312
DFT	3	4 pipeline stage at front, 3 pipeline stage in between RAMs (43 stages in total)	hard FP multiplier	54429	144	52	161780	680.74	85.09	5014.15	31	1341
SHA1	0			1361	0	0	983	320.31	4.00	7.64	5	0
SHA1†	1	1 stage pipeline register at front		1438	0	0	1531	338.64	4.23	29.82	25	56
SHA1†	2	1 stage pipeline register at front	pipeline loop	1805	0	0	4599	733.14	1.5(9.2)	128.69	65	155
SHA1†	3	1 stage pipeline register at front	unroll loop	52764	0	0	200433	698.32	1.4(698)	3608.36	105	510
CAVLC	0			855	0	0	620	361.53	59.97	20.99	8	0
CAVLC	1	1 pipeline stage at front		869	0	0	779	382.85	54.88	20.27	18	139
CAVLC	2	2 pipeline stages at front		877	0	0	1183	394.79	49.48	34.45	19	145
CAVLC	3	2 pipeline stages at front, 1 pipeline stage at back		1012	0	0	1803	483.79	53.94	46.91	20	165
CAVLC	4	3 pipeline stages at front, 1 pipeline stage at back	duplicate critical registers	895	0	0	2334	568.83	57.08	47.87	26	512

†Changed from asynchronous reset to synchronous reset



(a) Shift Registers (regshf)



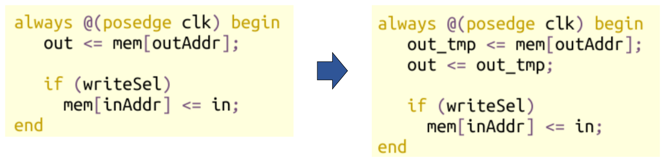
```
regshf #(9, 128) state_hp(CLK, STATE, state);
regshf #(9, 256) key_hp(CLK, KEY, key);
```

```
regshf #(8, 128) out_hp(CLK, out, OUT);
```

```
aes_256 core
(
        .clk(CLK),
        .state(state),
        .key(key),
        .out(out)
);
```

(b) Adding pipeline stages at the front and back of AES design (aes_256)

Fig. 3. Applying Hyper-Pipelining optimization to **AES**.



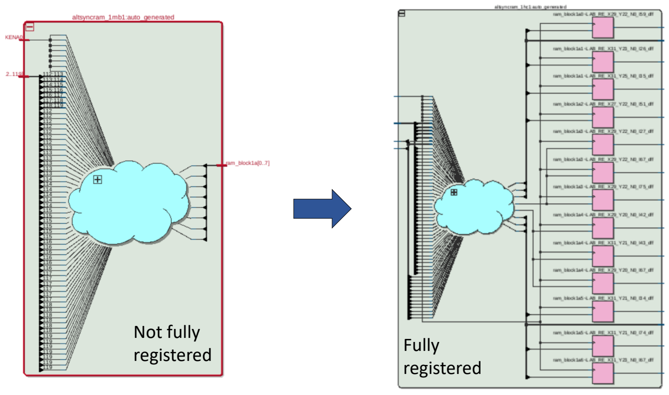
```
always @(posedge clk) begin
   out <= mem[outAddr];

   if (writeSel)
     mem[inAddr] <= in;
end
```

```
always @(posedge clk) begin
   out_tmp <= mem[outAddr];
   out <= out_tmp;

   if (writeSel)
     mem[inAddr] <= in;
end
```

(a) RTL coding style to infer fully registered RAMs



(b) inferred RAMs

Fig. 4. Inferring fully configured RAMs to allow for high frequency designs.

order to modify it to be fully pipelined. Hence, **DES** Ver 1 took 10 engineering hours with 895 lines of code changes.

1) Hyper-Pipelining between hard blocks (M20Ks, DSPs): Special consideration is needed for pipelined designs when hard blocks (M20Ks, DSPs) are used between pipeline stages. Currently, Quartus is not capable to perform retiming between such blocks[1]. As such, pipelining is done by manually inserting pipeline registers between such stages. The engineer must understand the internal design to do so. In addition, validation is needed to ensure functionality is correct. **AES** and **DFT** are examples of designs that utilize M20Ks and DSPs. As previously described, in **AES** each “round” block is based on RAMs (M20Ks). Hence, pipelining between rounds would require manual effort to insert pipeline registers. As shown in Table II, **AES** Ver 2 consumed ten engineering hours to understand the design and correctly add pipeline registers between rounds. Once that was done, optimizing for deeper pipelines was simpler. **AES** Ver 3 to Ver 7 took much less effort (approximately one hour each) as the modification was to add additional pipeline registers between rounds, a simple incremental addition to Ver 2.

2) Hyper-Optimizations on designs with hard blocks (M20Ks, DSPs): One would also need to ensure that the hard blocks are configured to accommodate high frequency pipelined implementations. RAMs (M20Ks), must be configured as “fully registered”. One can do this by following the recommended RTL coding style that describes the RAM as two stages at the RAM output. The resulting fully registered M20K inferred by Quartus is illustrated in Figure 4.

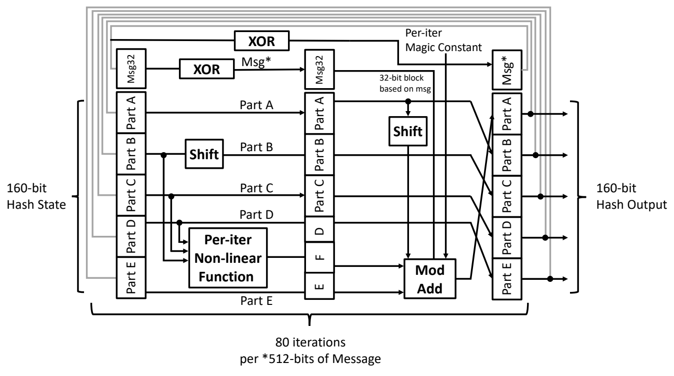
DSPs need to be configured to multi-stage mode. A four-stage DSP configuration is able to run at high frequency (max 1GHz).

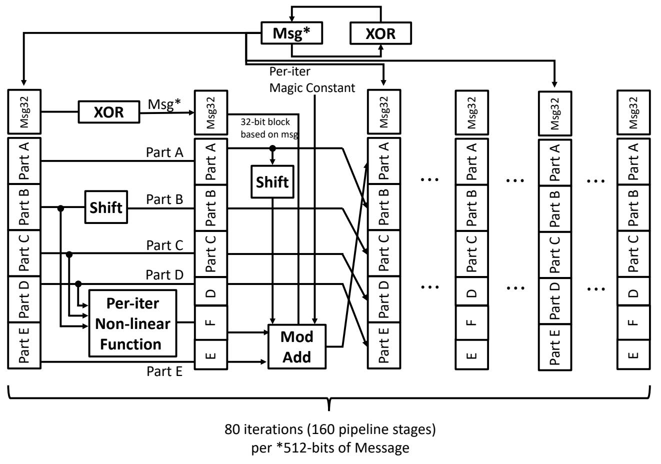
3) Hyper-Optimizations on designs with feedback loops: For pipelined designs with feedback loops, such as **SHA1**, Quartus was not automatically able to obtain as high of a frequency as designs with no loop. **SHA1** Ver 0 achieved 320 MHz. There are hyper optimizations that can be done to optimize such loops. First, one can pipeline the loop. To ensure correctness, appropriate hazard detection and resolution logic are needed to accommodate the data dependency on the feedback loop. There are multiple ways to handle such dependency, such as by stalling (i.e., introduce pipeline bubbles), forwarding, speculation, and/or interleaving [8], [9], [10].

In **SHA1** Ver 2, we implemented loop pipelining, where hazard resolution is done by introducing bubbles. Figure 5a shows **SHA1** loop pipelining. Though we saw large frequency improvements (339MHz to 733MHz), overall throughput went down due to bubbles that reduced pipeline utilization. It took 40 engineering hours to implement this optimization. To fully utilize the pipeline, we could also avoid introducing bubbles by interleaving. In this case, instead of providing one input at a time and wait for the output to be produced, we could provide a batch of independent inputs and interleave the execution of these inputs across the pipeline. We did not do this since the original design only assumed one input at a time. For designs that accept batches of inputs, such an optimization could be considered. In Table II, we project the expected throughput with batched input for **SHA1** Ver 2-3 with fully utilized pipelines, where the throughput number are in parenthesis.

Aside from **SHA1** Ver 2, **CAVLC** is another benchmark that illustrates loop pipelining. In this case, the original (Ver 0) design was already loop pipelined.

A second way to optimize designs with feedback loops is to unroll the loop. In this case, we could instantiate multiple unrolled iterations in hardware, as shown in Figure 5b. Of course, this would come at the cost of additional resources, depending on how much unrolling is done. In **SHA1** Ver 3, we performed full loop unrolling. As Table II shows, FPGA resources increased dramatically. Nevertheless, the overall throughput is similar to Ver 2 (with loop pipelining). Similar


(a) Loop pipelining for **SHA1**


(b) Loop unrolling for **SHA1**

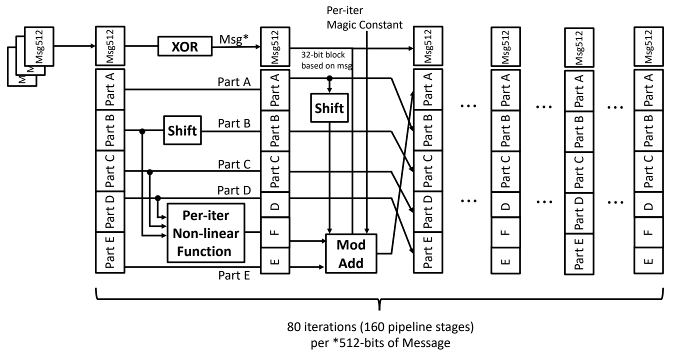

(c) Replicate Msg block to allow interleave

Fig. 5. **SHA1** Loop pipelining and unrolling

to the loop pipelined design, since the use case is to accept an input at a time, only one instance of loop iteration is active at a given time. Again, having a use case that accepts independent batches of inputs could help here as well.

Figure 5c shows an unrolled **SHA1** that allows fully utilize the unrolled pipeline. In Figure 5c, the 512-bit input is replicated for each pipeline stage. Although the resource cost goes even higher than **SHA1** Ver 3, it can interleave a batched input and achieve extremely high throughput (698 Mop/sec).

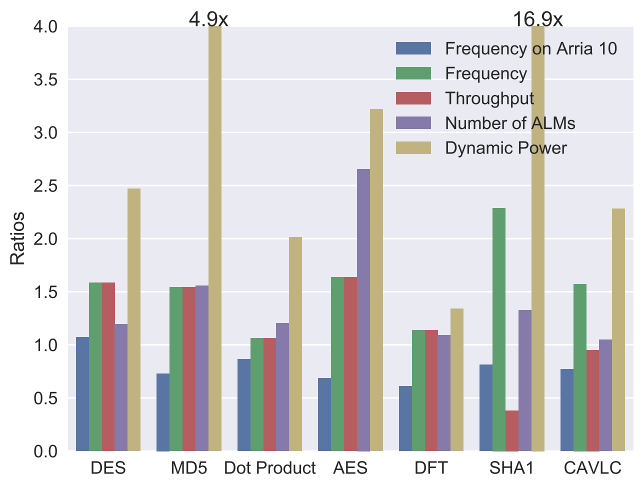


Fig. 6. Relative frequency, throughput, dynamic power, and FPGA resources (ALMs) of the highest frequency optimized designs relative to the baseline designs.

C. HyperFlex Benefits and Costs

A summary of the results is shown in Figure 6, which depicts relative changes in frequency, throughput, dynamic power, and FPGA resources for the optimal highest frequency designs against original baseline designs. For reference, we also include the maximum frequency of baseline designs measure on the previous generation Intel Arria® 10 FPGA. Note that Arria 10 FPGAs do not have HyperFlex architecture, and the HyperFlex optimizations do not apply to Arria 10 directly.

1) Benefits of HyperFlex: Without any modifications, the HyperFlex architecture and Stratix 10 together achieve up to 40% frequency improvement compared with Arria 10. HyperFlex modifications to the RTL further improved the maximum frequency when targeting Stratix 10. For all the pipelined designs without loops, those modifications directly translated into overall throughput improvements. Among such designs, relatively balanced designs (**DES**, **DotProduct**, **AES**, **DFT**) achieved higher frequencies (from 539MHz to 855MHz, 802MHz to 852MHz, 496MHz to 811MHz, 598Mz to 681MHz.) Even for the designs with loops (**SHA1**, **CAVLC**), it is possible to improve frequency from 320MHz to 733MHz and 362MHz to 568MHz, respectively. A design with tight feedback loops, like **SHA1**, would need to be pipelined properly to achieve such high frequencies, as described previously. Overall, as Figure 6 shows, explicit HyperFlex optimizations result in frequency improvements ranging from 1.06x to 2.28x.

2) Cost of HyperFlex: The improvement in frequency comes at the cost of additional resources, power, and engineering effort. As depicted in Figure 6 and Table II, resource usage increases across designs, especially in the number of registers as higher frequency designs use Hyper-Registers. There are, however, millions of Hyper-Registers available and that the use of Hyper-Registers does not affect the use of registers in logic elements, which are used for RTL registers (e.g., FSM state). In fact, without HyperFlex architecture, such high frequency designs on conventional FPGAs, if even possible, would consume full logic elements for their registers, leaving the rest of the logic element wasted, and would, therefore, increase power consumption. In terms of effort, simple optimizations such as adding a pipeline stage typically takes around 1 hour, while more sophisticated optimizations could take much longer. For example, **SHA1** loop pipelining took around 40 hours. There is also diminishing return as some optimizations may only increase frequency slightly, or even worsen frequency. For example, **AES** Ver3 to Ver 4 went from 783MHz to 787MHz and unrolling loops in **SHA1** caused the frequency to go from from 733MHz to 698MHz. Thus, optimizations should not be applied blindly but, instead, account for tradeoffs.

3) Tradeoff considerations: If starting from an existing design, the architecture should be considered when applying HyperFlex optimizations. For balanced pipelined designs with no feedback loops, our results indicate that it is generally possible to improve frequency. Even for designs with loops, it is possible to achieve higher frequency, but effort is needed to ensure the optimization provides higher throughput. Also, the use case may need to be tailored to the optimized design (e.g., by batching inputs).

If starting a new implementation, then it is important to properly architect the design to be suitable for HyperFlex. Avoid loops if possible. Optimize data dependencies. When using hard blocks (M20Ks, DSPs) across pipelines, use the correct pipeline register parameters to enable automated derivations of varying pipeline depths.

VI. Related Work

Pipelining has been very well studied. There is myriad of work proposing techniques for automatic pipelining of designs[8], [9], [10].

The novel highly-pipelined FPGA architecture behind Intel's HyperFlex technology has also been detailed previously in both publications[2] and in a tutorial[3]. Nevertheless, such existing studies (1) are not performed on open-source RTL designs, and (2) targeting pre-production Intel Stratix 10 FPGAs. Intel also provides documentation on HyperFlex[1]. This work is an up-to-date evaluation targeting production Intel Stratix 10, using open-source benchmarks, allowing others to independently study and replicate our experiments.

In general, open-source benchmarking studies have been done for FPGAs. For example, recently, benchmarks for evaluating HLS technology for FPGAs has been proposed[11]. Several studies have also benchmarked a set of designs to compare FPGAs against ASICs[12], [13]. To our knowledge, this work is the first to offer open-source optimized high-frequency designs for the Stratix 10 HyperFlex.

VII. Conclusion

This paper evaluates production Intel Stratix 10 FPGA architecture based on HyperFlex technology using open-source benchmarks. The results show that HyperFlex enhances the possible design space, allowing trading off between higher frequency, throughput, resources, power, and effort. HyperFlex enables higher frequencies than what was possible in previous generation FPGAs. In particular, for the balanced feedforward pipelined designs we studied, even those with M20Ks and/or DSPs in the data path, it was relatively straightforward to achieve frequencies in the 500MHz to 800MHz range. Designs with feedback loops require re-architecting the design (e.g., pipeline the loop, unroll the loop, enabling processing of multiple inputs in a batch) to achieve high frequency and, especially, high throughput. The optimized designs we developed in this paper are made available at[4]. Over time, we will continue to add more benchmarks and encourage others to do so as well.

References

[1] I. Corporation, *Intel Stratix 10 High-Performance Design Handbook*, Intel Corporation.

[2] D. Lewis, G. Chiu, J. Chromczak, D. Galloway, B. Gamsa, V. Manohararajah, I. Milton, T. Vanderhoek, and J. Van Dyken, "The stratix 10 highly pipelined fpga architecture," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 159–168.

[3] G. Baeckler, "Hyperpipelining of high-speed interface logic." in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, p. 2.

[4] "Stratix 10 hyperflex optimized benchmarks," https://github.com/doublsky/Stratix10HyperFlex.git.

[5] OpenCores, "The reference community for free and open source gateware ip cores," 2016, [Online; accessed 4-July-2016]. [Online]. Available: https://opencores.org

[6] D. J. Moss, S. Krishnan, E. Nurvitadhi, P. Ratuszniak, C. Johnson, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. Leong, "A customizable matrix multiplication framework for the intel harpv2 xeon+ fpga platform: A deep learning case study," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2018, pp. 107–116.

[7] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005.

[8] S. Hassoun and C. Ebeling, "Architectural retiming: pipelining latency-constrained circuits," in *Proceedings of the 33rd annual Design Automation Conference*. ACM, 1996, pp. 708–713.

[9] E. Nurvitadhi, J. C. Hoe, S.-L. L. Lu, and T. Kam, "Automatic multithreaded pipeline synthesis from transactional datapath specifications," in *Proceedings of the 47th Design Automation Conference*. ACM, 2010, pp. 314–319.

[10] E. Nurvitadhi, J. C. Hoe, T. Kam, and S.-L. L. Lu, "Automatic pipelining from transactional datapath specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 3, pp. 441–454, 2011.

[11] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez *et al.*, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2018, pp. 269–278.

[12] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," *IEEE Transactions on computer-aided design of integrated circuits and systems*, vol. 26, no. 2, pp. 203–215, 2007.

[13] H. Wong, V. Betz, and J. Rose, "Quantifying the gap between fpga and custom cmos to aid microarchitectural design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 10, pp. 2067–2080, 2014.