CSAIL2019 Crypto-Puzzle Solver Architecture

Sergey Gribok
sergey.gribok@intel.com
Intel Corporation
US

Bogdan Pasca
bogdan.pasca@intel.com
Intel Corporation
France

Martin Langhammer
martin.langhammer@intel.com
Intel Corporation
UK

ABSTRACT

The CSAIL2019 time-lock puzzle is an unsolved cryptographic challenge introduced by Ron Rivest in 2019, replacing the solved LCS35 puzzle. Solving these types of puzzles requires large amounts of intrinsically sequential computations (*i.e.* computations which cannot be parallelized), with each iteration performing a very large (3072-bit in the case of CSAIL2019) modular multiplication operation. The complexity of each iteration is several times greater than known FPGA implementations, and the number of iterations has been increased by about 1000x compared to LCS35. Because of the high complexity of this new puzzle, a number of intermediate, or milestone versions of the puzzle have been specified.

In this paper, we present an FPGA architecture for the CSAIL2019 solver, which we implement on a medium-sized Intel Agilex device. We develop a new multi-cycle modular multiplication method, which is flexible and can fit on a wide variety of sizes of current FPGAs. We also demonstrate a new approach for improving the fitting and timing closure of large, chip-filling arithmetic designs. We used the solver to compute the first 21 out of the 28 milestone solutions of the puzzle, which are the first reported results for this problem.

CCS CONCEPTS

• **Hardware → Arithmetic and datapath circuits**; **Hardware accelerators**; *Datapath optimization*; Error detection and error correction.

KEYWORDS

iterative modular multiplier, modular exponentiation, low-latency, FPGA, CSAIL2019 puzzle

ACM Reference Format:
Sergey Gribok, Bogdan Pasca, and Martin Langhammer. 2023. CSAIL2019 Crypto-Puzzle Solver Architecture. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '23), February 12–14, 2023, Monterey, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3543622.3573184

1 INTRODUCTION

This paper presents an FPGA-based architecture for a solver of the CSAIL2019 crypto-puzzle [1]. The CSAIL2019 puzzle is a "refreshed" version of LCS35 crypto-puzzle [2] published in 1999. This area of work is no longer just an academic exercise, but relevant to current industry requirements with the increasing importance of blockchain.

The stated problem is the compute of $2^{2^t} \bmod N$ for specified values of t and N. LCS35 defines $t = 79685186856218$, and N is 2048 bits in length. The original estimate was that LCS35 would need 35 CPU years to solve, assuming that the yearly milestone (i.e. intermediate) value would be transferred to the latest CPU technology available. In 2019, Bernard Fabrot solved LCS35 using 3 years of continuous CPU run-time [3]. Shorter thereafter, the Cryptophage team used an FPGA to solve the puzzle in only 2 months [4].

The CSAIL2019 puzzle defines $t = 2^{56} = 72057594037927936$ which is nearly 1000 times larger than LCS35. The word size of N is now 3072-bit, which is 1.5 times greater than LCS35, so each iteration is about 2.25x more computationally demanding than before. If we attempted to solve CSAIL2019 with a design that had the same compute capability as the Cryptophage approach and hardware, we could expect a runtime of about 375 years. While the full puzzle requires a solution for $t = 2^{56}$, CSAIL is also interested in solutions for $t = 2^k$ for $56/2 \leq k < 56$. These intermediate solutions are called "milestone versions of the puzzle".

These types of puzzles cannot be parallelized. Each modular multiplication follows the completion of the previous one. The only way to accelerate the solution is to run a single instance faster. In this paper we will show a multi-cycle implementation. A larger FPGA, with more DSP Blocks, could potentially reduce the latency by about 10x. The later architectures with newer process technologies that could provide this level of density would likely run faster, although this would be offset by the much increased combinatorial delay through the single core. Likewise, we could not spread this problem out over multiple FPGAs, as the inter-chip communication times would significantly increase the combinatorial delay through one modular multiply.

We make the following contributions in this paper:

- We show some modified approaches to high precision modular multiplication which can be used to implement a more efficient FPGA solution.
- We show a multi-cycle implementation which can also map to more modest sized FPGAs, and also develop a left-to-right calculation where we can perform the multiplication and reduction steps at the same time.
- We introduce a robust error checking mechanism to independently verify running results of a long computation run. We show how this can be used to safely overclock large complex systems such as the CSAIL2019 solver.
- We describe the directed internal pipelining of a large combinatorial circuit to improve fitting and timing closure.

FPGA '23, February 12–14, 2023, Monterey, CA, USA

ACM ISBN 978-1-4503-9417-8/23/02…$15.00
https://doi.org/10.1145/3543622.3573184

In this paper, we use the term latency to denote the total time from the input to output of a circuit, independently of how many clock cycles are taken.

2 PRIOR WORK

The current interest in high-performance large precision modular multiplication in FPGAs is driven by blockchain, with VDF (variable delay functions) as a key motivation. One inflection point was around the VDF Alliance Contest of 2019 [5], which was initiated shortly after the Cryptophage solution of LCS35 was disclosed.

The breakthrough that made this level of modular multiplication performance possible in FPGAs is due to Ozturk [6], which we will explore in more detail later in this section.

The VDF Alliance FPGA contest was the computation of x^{2^t} mod N with $t = 2^{33}$ and N = 1024 bits. Each iteration is therefore about an order of magnitude smaller than the modular multiplication used by CSAIL2019. Moreover, the number of iterations is also significantly smaller. The FPGA target was the Amazon F1 card [7], which used Xilinx Virtex Ultrascale+ VU9P devices [8]. These devices contain 1182K 6-LUTs and 6840 DSP (DSP48E2) blocks [9]. Each DSP Block can support a 27x18 multiplier instance, with the ability to cascade and sum multiple DSP Blocks together. The VDF competition was held in 2 rounds, with the winner [10] of the first round performing each modular squaring in 28.6ns. The second round of the contest was run, with the goal of improving on the latencies of the successful architectures of the first round. Pearson [11] was the winner again, with the latency of each iteration reduced to 25.2 ns.

A third round was run, in order to find the fastest solution that was not based on Ozturk's method. This round was won by Ben Devlin [12] using a Montgomery multiplier based approach [13]. Performance was approximately half the speed of the Ozturk multiplier, at 46ns. Devlin also reported that he investigated and rejected the other non-Ozturk approaches suggested by the VDF contest, such as Chinese Remainder Theorem [14] (no reason given) and Barrett's reduction [15] (because of the impact of the final adjustment subtractor, presumably as it requires a full carry propagation as opposed to the redundant bit value of the Ozturk method). Devlin also investigated alternate multiplier constructions, such as Toom-Cook, Karatsuba [16], and Booth's recoding [17], but stated that these were not suitable, without explanation. (We will analyze the effect of the Karasuba approach in detail later in this section). This round also solicited algorithms in addition to implementation. The winner of this section described nested reductions in a RNS (Residue Number System) [18].

Both Montgomery and Barrett's approaches have been previously used in building high-precision FPGA modular multipliers. The smaller resources counts of earlier FPGA devices is the reason that most of the previously reported designs are based on folded (multi-cycle or resource shared) architectures. Devlin's architecture is also multi-cycle, but uses predictive branching to improve throughput (i.e. reduce the overall number of cycles required). Before the advent of VDF, where latency is the most important metric, resource reduction, especially of DSP Blocks, appeared to be the key goal. As an example, an earlier Montgomery multiplier [19] used 3 iterations of a 4 level Karatusba decomposition to perform a 256-bit modular multiply. This design used 81 18x18 multipliers - we can compare this to Devlin's design, which has a 16x higher arithmetic complexity (1024-bit v.s. 256-bit), but a 28x increase in multiplier count. Iterative Mongomery-style modular multipliers of up to 1024 bits are also studied in [20] for FPGA devices. Barrett's method requires even deeper pipelining. One recent design [21] distributed six 48-bit multipliers over the calculation of a 512-bit modular multiplication. This calculation needed 79 cycles.

A large modular multiplier based on an updated Barrett's approach was recently reported [22]. We will analyse this design, because it reports details on different combinations of multi-radix Karatsuba decomposition, FPGA mapping, and analysis of Barrett's algorithm; all of these were mentioned by Devlin, so we can now examine why the latest approaches used for VDF-type calculations (long iterations of modular exponentiations) are better.

If we look at the history of large multiplier implementations in FPGAs [23] [24] [25] [26] we can see that Karatsuba methods feature prominently. Latency (or more usually, pipeline depth) will generally not be a consideration, but rather the reduction in the number of resources, especially DSP Blocks.

Barrett's algorithm requires three chained multiplications: an initial full multiplication, followed by two multiplications to create an approximate estimate of the modular value. A fine adjustment then takes place to calculate the exact result. The two multipliers in the estimation phase do not have to be full multipliers; we need the MSB bits from one of them and the LSB bits from the other, which means that we do not have to construct the full parallel multipliers for this. The reported method used a looser error bound to reduce the required accuracy of two of the three multiplier operations (*i.e.* allowing a multi-bit error in both of the subset multipliers) in the algorithm, thereby reducing DSP and logic resource requirements. Although the Karatsuba decomposition still requires that we need to generate all of the partial products, we do not need to sum all of the combinations of partial products together, which saves a significant amount of soft logic. Of perhaps greater impact is the simplification of the place and route problem.

The aggregate cost of the modified Barretts approach is 255K ALMs, 1183 DSP Blocks, and requires 143 cycles per modular multiplication. A key point is that this is the depth of the modular multiplier - once the pipeline is full, a new result can be output per clock cycle. However, a VDF function depends on round-trip latency, not the number of modular multiplications per unit time. The 500MHz performance is realized on the older Stratix 10 [27] FPGA. It is likely that there would be a 35% increase in performance if recompiled into the newer Agilex [28] family. We can therefore estimate the expected 1Kb squaring time at 212ns. We can now compare the three approaches: Ozturk, predictive Montgomery, and modified Barretts, as shown in Table 1.

We need to look at this data with many qualifiers. We have estimated the performance improvement of Barrett's design when ported to a newer and faster FPGA. We treat the cost and capability of the Xilinx/AMD and Altera/Intel devices for soft logic and DSP resources the same, although there are differences. While the Ozturk and Montgomery architectures have been optimized for squaring (which cuts the resource usage approximately in half compared to a full multiplication), Barrett's architecture uses a full AxB multiplication. If Barrett's design were optimized for squaring,

Table 1: Performance of 1024-bit Modular Multiplication Methods.

Architecture	LUTs	DSP	latency (ns)
Ozturk	464K	2212	25.2
Montgomery	201K	2272	46.0
Barrett	255K	1183	*est. 212*

the area reduction would not be as high, as the squaring tuning would only apply to the first multiplier. A 20% area reduction of the three-multiplier system would be more likely in this case.

One of Devlin's goals was to implement a design that fits into one of the SLRs [29] of the VU9P to improve timing closure. A SLR is an FPGA die, and multiple SLRs are combined using an interposer to create a larger FPGA device. Crossing SLR boundaries reduces performance because of the interface latency. In contrast, the Pearson design is spread out over the entire FPGA. This may become more significant as we try to support the much larger (3072-bit vs. 1024-bit) word size for CSAIL2019. None of these designs report routing stress, which will become more pronounced for this new puzzle, as the arithmetic complexity of each iteration increases by roughly an order of magnitude.

At the 1024-bit word size, we can see that the Ozturk method is clearly the winner when the important metric is latency. Efficiency only becomes an issue if it prevents the design from running in a single FPGA, although with the approximately 9x increase in multiplier size, we can see this is a possibility. We have a number of ways to manage this. First, we will explore high performance multi-cycle approaches, which may fit into current FPGAs more readily. Secondly, we need to find more efficient ways of implementing the Ozturk algorithm. DSP blocks are intrinsically more efficient than soft logic, as the functionality is already in ASIC form. This gives us a strong motivation to restate the Ozturk approach from a table based to arithmetic based reduction operation.

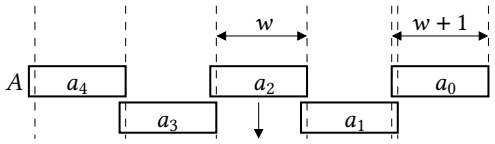
Finally, it is important to remember that although the Ozturk approach is the best for round-trip latency, a modified Barrett's design is still both the most efficient (least number of resources per computation) and fastest for throughput, for example if many independent RSA cryptographic operations we run simultaneously, but the latency is many times that of the Ozturk or predictive Montgomery methods.

3 BACKGROUND

In this section we briefly present the techniques behind current FPGA modular multiplication methods. We start with a review of the Ozturk approach, and then present a newer method [30] that uses the embedded FPGA DSP resources for a more efficient and higher performance result. In the next section, we will further develop the DSP based approach into an efficient multi-cycle version.

3.1 Integer Multiplication

The large integer multiplication is implemented as a polynomial multiplication. The inputs A and B are unsigned integers represented in polynomial form as $d+1$ radix $R = 2^{w+1}$ digits. The polynomial is constructed with a 1 bit overlap between consecutive



$$A = a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_0x^0$$

Figure 1: Integer viewed as a degree-d polynomial with overlapping coefficients leading to a redundant representation.

digits. This feature allows reducing modular reduction latency - as it will be made clear in this section.

$$A = \sum_{i=0}^{d} b_i 2^{wi}, B = \sum_{i=0}^{d} b_i 2^{wi}$$

From the radix-R digit notation the polynomial notation ($x = 2^w$) follows naturally:

$$A(x) = \sum_{i=0}^{d} a_i x^i, B(x) = \sum_{i=0}^{d} b_i x^i.$$

Here a_i, b_i are the coefficients of the polynomials, and correspond to the radix-R digits from the original representation. This is highlighted in Figure 1.

The product P of two degree d polynomials A and B is a degree $2d$ polynomial. Figure 2 highlights the partial-products for $d = 3$. The subproducts a_ib_j are $2w+2$ bits wide values, and can be written in terms of two w-bit values and one 2-bit value ($P^{\mathrm{H}}_{i,j}$):

$$\begin{aligned} a_ib_j &= Pij \\ &= P^{\mathrm{H}}_{i,j}2^{2w} + P^{\mathrm{M}}_{i,j}2^{w} + P^{\mathrm{L}}_{i,j}. \end{aligned}$$

Knowing that $x = 2^w$, the subproduct alignments are such that:

- $P^{\mathrm{H}}_{i,j}$ overlaps over $P^{\mathrm{M}}_{k,l}$ where $k + l = i + j + 1$,
- $P^{\mathrm{M}}_{i,j}$ overlaps over $P^{\mathrm{L}}_{k,l}$ where $k + l = i + j + 1$,
- $P^{\mathrm{H}}_{i,j}$ overlaps over $P^{\mathrm{L}}_{k,l}$ where $k + l = i + j + 2$.

These alignments can be observed in Figure 2. The columns of sub-products sections aligned at weights 2^{wi} correspond to non-evaluated sums that, when evaluated, correspond to coefficients of the output polynomial. Therefore, each column is summed together (columns have between 1 and 10 subproduct components), to generate intermediary coefficients D_i, with widths ranging from w bits (for D_0) to $w + 4$ bits (for D_4).

$$D_k = \sum_{i+j+2=k} P^{H}_{i,j} + \sum_{i+j+1=k} P^{M}_{i,j} + \sum_{i+j=k} P^{L}_{i,j}$$

A set of w-bit wide additions aligned on the column outputs, are used for creating modified polynomial coefficients such that their maximum widths does not exceed $w + 1$ bits. These short adders sum the lower w bits of D_i ($D_i \bmod 2^w$) with the bits having weights larger than 2^w from D_{i-1} ($D_{i-1} \gg w$). This propagation is only required for $i \geq 2$, as for $i = 0$ the subproduct column has only one term.

$$C_i = (D_i \bmod 2^w) + (D_{i-1} \gg w), i \in [2, 2d + 2]$$

This level of adders is depicted on the the bottom of Figure 2.

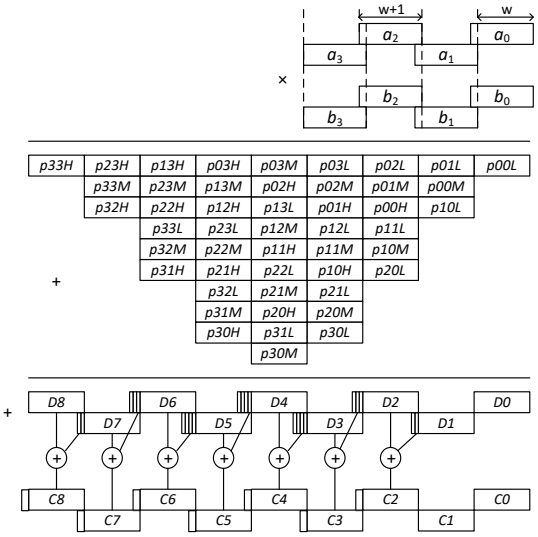


Figure 2: Subproduct alignments, column-based summations and short carry-propagation stage for a degree-3 polynomial multiplication corresponding to 4-digit radix R inputs

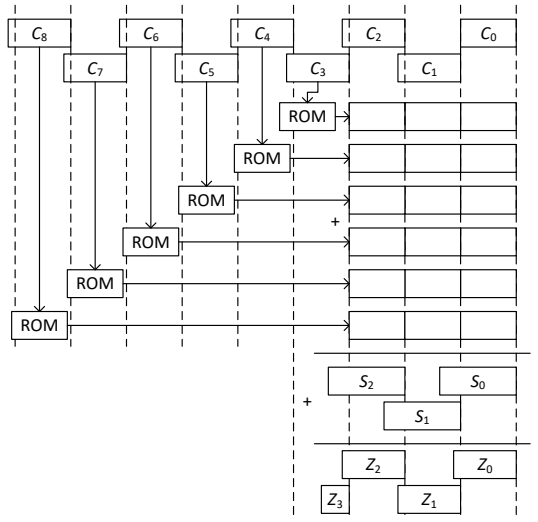


Figure 3: Polynomial reduction, resulting in a $d+1$-degree polynomial

Finally, the product P can be written in polynomial form:

$$P = \sum_{i=0}^{2d+2} C_i x^i,$$

with C_i holding on 2^{w+1} bits.

3.2 Modular Reduction

The second part of the modular multiplication involves reducing $P \pmod N$:

$$M = P \bmod N,$$

where P is the polynomial previously generated on the output of the multiplier.

In the context of modular exponentiation we do not require an exact (non-reducible) M, but rather any equivalent M is sufficient, as long as it meets a number of properties. One of these is that it's easy to obtain, another is that the output has the same form as the inputs of the polynomial multiplication.

The following identity is used for obtaining M:

$$\begin{aligned} A + B \bmod N &\equiv (A \bmod N) + (B \bmod N) \\ &\equiv ((A \bmod N) + B) \bmod N \end{aligned}$$

Note that with respect to the degree d of the input polynomials A and B, the degree of the modulus N is $d-1$.

We split P in two parts:

$$P = \sum_{i=d}^{2d+2} C_i x^i + \sum_{i=0}^{d-1} C_i x^i$$

Next, the high part is composed of $d+3$ radix 2^{w+1} coefficients. For each coefficient and for a constant value of N, the reduced value mod N can be pre-computed and tabulated:

$$M_i = C_i x^i \bmod N, i \in [d, 2d+2].$$

Additionally, each M_i can be viewed as a degree-d polynomial, with coefficients $M_{i,j}$ radix 2^w digits. This allows for the following rewrite:

$$M = \sum_{i=0}^{d-1} \left(C_i + \sum_{j=d}^{2d+2} M_{j,i} \right) x^i \bmod N.$$

This results again in column-based summations, as shown in Figure 3.

A final reduction consisting of $d-2$ parallel w-bit adders (similar to the structure at the bottom of Figure 2 for the case of integer multiplier) is implemented in order to obtain $w+1$-bit wide coefficients for the output polynomial. This is shown on the bottom of Figure 3.

Note that the previous reduction phase produces an output that is in redundant form. An alternative implementation based on a full-width adder would produce an output in standard form (non-overlapping coefficients). This would have a higher area (associated to pipelining) and longer latency ($\approx (d-1)$x higher for a ripple-carry adder implementation). Fortunately, this is not required since the polynomial multiplier described in Section 3.1 is designed to accept coefficients in this redundant form. Moreover, the $w+1$ bitwidth used for coefficient radix R selected to match the DSP Block multiplier size. For instance Intel FPGA devices contain 27-bit multipliers making $w = 26$ a good choice.

3.3 Multiplicative Reduction

The lookup-based reduction combined with the redundant polynomial-based multiplication is a significant improvement for low-latency modular multiplication in FPGAs. Nonetheless, tackling very large word sizes exposes a significant limitation of the approach – as the partial reduction values M are stored in LUTs, the amount of logic required to store the tables can become prohibitively large (Section 4.2 presents a resource estimation in the context of the

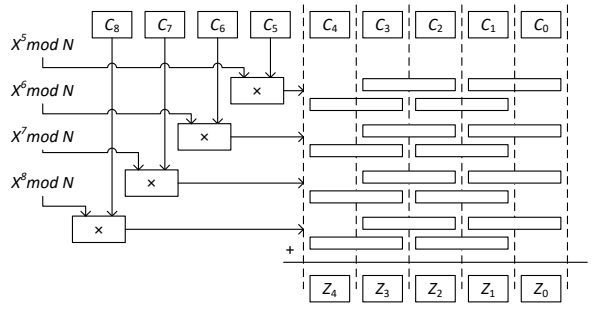


Figure 4: Simplified diagram showing the use DSP blocks for performing modular reduction in polynomial form.

CSAIL2019 3072-bit multiplier size). In addition to high logic utilization, place and route will also become an increasingly difficult problem as multiplier sizes increase. The partial reduction terms $M_{j,i}$ need to be summed with the result of the multiplicative expansion (C_i), with many long wires inevitably being necessary for routing operands to the corresponding adder trees.

A recent solution presented in [30] reduces the lookup table reduction cost (and alleviates some of the routing problems) by implementing a multiplier-based reduction (therefore replacing the LUTs in Ozturk's algorithm with DSP Blocks).

The DSP-based solution is based on the following rewrite:

$$M_i = C_i(x^i \bmod N), i \in [d+2, 2d+2].$$

Here ($x^i \bmod N$) is a constant (for constant N), and can therefore be pre-computed. Then M_i is calculated by simply multiplying C_i by that constant. That could be done by using a layer of DSP blocks as shown on Figure 4. Note that M_i is now $w+1$ bits wider than N. To account for this, the input polynomial degree needs to be increased by 1.

The multiplicative modular-reduction scheme presented here significantly reduces the number adder tree terms compared to the lookup-based solution, therefore reducing the overall latency of the implementation as showed in [30].

4 CSAIL2019 SOLVER ARCHITECTURE

4.1 Iterative Modular Multiplication

CSAIL2019 uses a much larger value of N(3072-bit) compared to VDF (1024-bit) or LCS35 (2048-bit). A fully parallel [6] [30] N-bit modular squaring operation does not fit into even the largest FPGAs available today.

An iterative approach saves FPGA area, but it also increases latency, and therefore reduces overall design performance. A straightforward iterative modular multiplication mapping uses an iterative multiplication block followed by an iterative modular reduction block. The overall operation latency is therefore a sum of the multiplication latency and the modular reduction latency.

In order to reduce latency, we have developed an optimized iterative modular multiplication algorithm where the iterative multiplication and the iterative modular reduction operate in parallel, with a 1-cycle offset. The first modular reduction iteration operating on a partial multiplication result starts immediately after the first multiplication iteration. Consequently, the overall modular multiplication latency is only one modular reduction iteration latency more than latency of a regular (non-modular) iterative multiplication.

Algorithm 1: Iterative Multiplication mod N

Input: $A = \{A_{n-1}, \cdots, A_0\}$
Input: $B = \{B_{n-1}, \cdots, B_0\}$
Output: $Z = \{Z_{n-1}, \cdots, Z_0\}$
Variable: $M = \{M_{n-1}, \cdots, M_0\} = 0$
Variable: $S = \{S_{n-1}, \cdots, S_0\} = 0$
Variable: $P = \{P_n, \cdots, P_0\}$
for i from $n-1$ to 0 **do**
  $P = AB_i + (M \ll (W/n))$
  $M = \{P_{n-1}, \cdots, P_0\}$
  $S = S + (P_n 2^{W+iW/n} \bmod N)$
end for
$Z = S + (M \bmod N)$;

The multi-cycle high-level modular multiplication approach is shown in Algorithm 1. The W-bit wide inputs A and B are subdivided into n limbs each. On every iteration (loop index i is decremented from $n-1$ down to 0) A is multiplied by B_i by means of a rectangular multiplier, to produce an $n+1$ limb product. The lower n limbs of the product are stored in variable M to use in the next iteration. The upper limb of that product (P_n) is sent to the multiplier-based modular reduction circuit where it reduced modulo N. The reduced value is then fed into the running accumulator S. Upon completion of the loop one modular reduction is required in order to reduce M mod N, before constructing the final result Z.

Using this algorithm we can calculate (AB mod N) in $n+1$ cycles (assuming multiplication and reduction each take 1 cycle) using n times fewer resources when compared with a fully parallel implementation. Note that in order to be able to start the modular reduction on cycle 2 we need to perform the multiplication starting with the most significant limb of B – which corresponds to a left-to-right approach – as opposed to the classical (pen-and-paper) approach which is right-to-left.

4.2 Hardware Architecture and FPGA Mapping

The hardware implementation of this algorithm contains two main modules: the iterative modular multiplier and a customized modular reduction, as shown in Figure 5.

The iterative multiplier operates on polynomials A and B of degree 120 (121 terms) - with x corresponding to 2^{26} and coefficient radix $R = 2^{27}$ (denoted by `[121x27]` in Figure 5). This allows for integer inputs of up to $3146 = 26 \cdot 121$ bits to be represented. This bit-width is sufficient to handle the $3072 + 32 = 3104$ bit N' (the additional 32 bits are required for the error detection mechanism described in Section 4.3). Note that due to the redundant polynomial representation (1-bit overlap between consecutive coefficients), the total number of bits used to manipulate the polynomials is $27 \cdot 121 = 3267$.

The polynomial B is split into 8 limbs, with each limb having 16 coefficients (most significant limb has only the 9 least significant coefficients populated, with the rest tied to zero). The *Polynomial Multiplication* component multiplies iteratively A by the limbs of

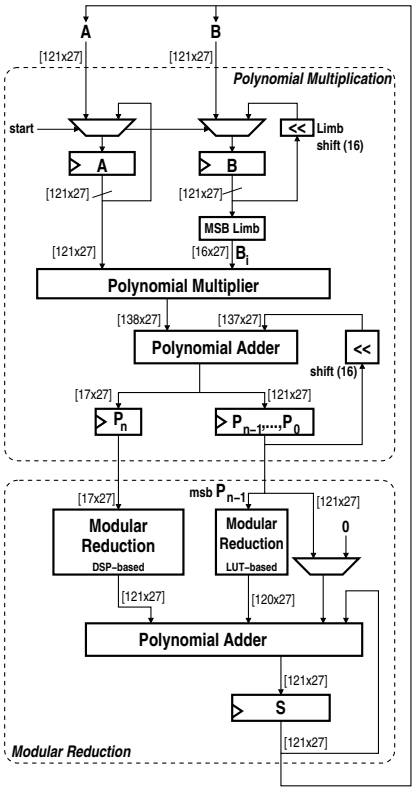


Figure 5: High-level architecture of the iterative modular multiplier used for the CSAIL2019 puzzle

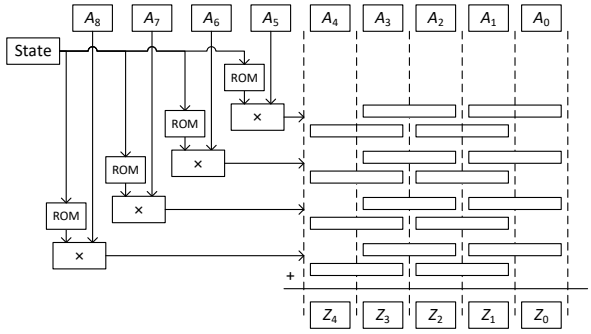


Figure 6: Iterative modular reduction using ROMs

B, starting from the most significant one, as previously explained in Algorithm 1. For the first iteration the product flows through the polynomial adder unaltered and gets split into an upper part P_n and a lower part $\{P_{n-1}, ..., P_0\}$. Subsequent iterations recirculate the aligned low part ($\{P_{n-1}, ..., P_0\} \ll 16$) back on the second input of the adder, to be summed with the next partial product AB_{i-1}.

For each iteration, the high part of the sum P_n is propagated to the *Modular Reduction* component. The DSP-based modular reduction outputs a 121-coefficient result that is fed into the polynomial accumulator S. On the last iteration, all but the most significant bit of the most significant limb of $\{P_{n-1}, ..., P_0\}$ also gets added into S. The most significant bit of P_{n-1} is passed through a LUT-based modular reduction, and gets added in to S as well. The 3120-bit range offered by the 121-coefficient polynomials ensures that at the output of the DSP-based modular reduction no overflow can happen in the most significant coefficient by summing up 17 3104-bit terms. Even considering the 8 iterations required for performing the full modular multiplication would not grow the most significant limb contribution above 3111 bits, which is lower than 3120.

Focusing on the modular reduction component, the standard Ozturk modular multiplication [6] implements the modular reduction using an array of ROMs, as shown in Figure 3. The multiplicative reduction method [30] (Figure 4) instead uses an array of DSP blocks configured as multipliers by constants. However, neither of these approaches can be directly applied to the iterative modular reduction design because the relative weight of the P_n term changes with every iteration - thus changing the value that needs to be reduced.

The Ozturk multiplier therefore offers minimal savings in an iterative implementation, because all values outside of the N bit range still need to be tabulated. Both the expansion in multiplier and reduction tree are smaller, but most of the logic is in the ROM tables. The total amount of storage is unchanged. The ROM cost for the 3072-bit case (CSAIL2019 bitwidth) can be calculated easily. We split the out of band expansion into 6 bit segments, with each segment containing a 3072-bit modulus value. We therefore have 512 ROMs of 3K LUTs each, or over 1.5M 6LUTs. This is as much logic as some of the largest FPGAs available, and in itself will create a structure that will be difficult to place and route. As a comparison, this calculation shows that the 1024-bit modulus of the VDF Alliance competition would require about 175K 6 LUTs, which appears correct in the context of the reported solution size of 464K 6 LUTs [11].

We have a similar problem for the multiplicative reduction method - every iteration needs a different constant. The difference is that the cost of tabulation is much less, and in one case can also be absorbed by the FPGA DSP Blocks themselves. The read address for the table is the index of the current iteration (see Figure 6).

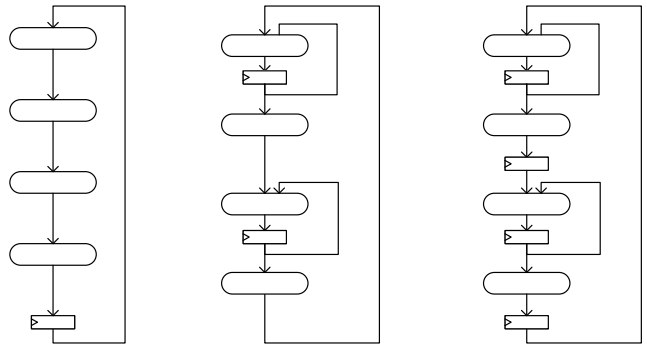
Here the "State" register contains the current iteration index and is used to select the correct constant from the ROMs for every iteration. In some FPGAs [31], a built-in coefficient storage (which was originally designed to support multi-channel FIR filters [32]) can be repurposed for the C_i storage. These internal ROMs are 8 elements deep; as long as the number of iterations per modular multiplication does not exceed 8, we are able to absorb the entire coefficient storage into these embedded blocks. The cost of soft logic for this method is therefore zero - as opposed to 1.5M LUT6 using the tabular reduction case. In contrast, it is also possible to map our iterative ROM tables to soft logic, at the cost of 14 half-ALMs per DSP block. One table would be required per DSP Block in the reduction section, or about an additional 27K ALMs.

4.3 Error detection

For very long running computations - such as those required to unlock this puzzle - it is very important to be able to detect errors early on. If an error goes undetected then all computations performed after the error (which may amount to many years worth

Algorithm 2: Error detection

Const: P // prime that generates maximum cycle
Const: $L = (P-3)/2$ // P cycle length
Input: X // Current result modulo $N' = NP$
Input: K // Current index
Variable: $X' = X \bmod P$
Variable: $K' = (K \bmod L) + L$
Variable: $T = 2$
for i from 1 to K' **do**
  $T = T^2 \bmod P$
end for
Return $T == X'$;



(a) Simple Loop (b) Loop with inner loops (c) Pipelined loop with inner loops

Figure 7: Loop Construction Development

of compute) will be useless. The error-detection mechanism needs to be combined with a checkpointing mechanism. At regular time intervals intermediary results are checked using the error-detection mechanism. If no errors are detected then the results are stored offline as a valid system state (checkpoint).

Having an error-detection mechanism in place, we can safely overclock the hardware (run it using a clock frequency larger than what is reported by the Quartus Timing Analyzer), and rely on the error detection mechanism to catch errors caused by overclocking. In the unlikely event of an error, the system will only need to revert to a starting point (checkpoint state) several minutes old. This is insignificant in the case of what can amount to multi-month or multi-years runs.

To be able to detect an error we use the following approach: instead of performing calculations modulo N, we perform calculations modulo $N' = NP$, where $P = 4294963787$ is a 32-bit prime that produces the longest possible cycle (see [33]) $L = (P-3)/2 = 2147481892$. Converting of a value modulo N' into a value modulo N simply requires taking the remainder modulo N of that value. Operating mod N' provides a way to check for errors in the calculations at any moment in time. The process involves comparing the result modulo P with the expected value as shown in Algorithm 2. Note that K in Algorithm 2 represents the total number of modular multiplications (squarings) done so far. We reported our method to the CSAIL team to check that it was correct [34].

Running the error detection algorithm takes just a couple of seconds on a CPU. A probability of an undetected error is $1/2147481892$ which is extremely small.

4.4 Directed Pipelining

Recent designs for large modular multiplication [6][30] contain a datapath organized as a "simple loop" as shown on Figure 7(a). Adding additional clock stages into a simple loop architecture does not improve the overall speed of computations. Even though the additional pipeline stage allows the computations to run at a higher clock frequency, the trade-off is that it also increases the number of clock cycles to go through the loop, marginally reducing the overall performance - in our case loop completion latency. We see this trade-off in recently reported designs. Some of these designs [3][4] use just 1 or 2 clock cycles per iteration, and therefore contain very deep un-pipelined datapaths with very slow clock frequencies (20-40 MHz). If the designs are pipelined slightly deeper, the clock frequency increases, but this is almost perfectly offset by the increase in number of cycles per iteration, again with a iteration rate of 40MHz [11].

We now describe an improved way to introduce pipelining into a selected class of very deep combinatorial designs for FPGAs. We use the iterative modular multiplier architecture from Figure 5 as a working example. A high-level view of the loop organization of this design is shown in Figure 7(b). The design is organized as a nested two-level loop structure. Both inner loops iterate 8 times (synchronously) to produce a modular multiplication result while the outer loop iterates over the modular multiplication $t = 2^{56}$ times to complete the modular exponentiation. The total time to finalize the modular exponentiation therefore equals $2^{56}T$, where T denotes the time to complete one modular multiplication.

Denote by X the inner loop iteration count (the execution stays in the inner loop for $X = 8$ clock cycles), while completing one iteration of the outer loop takes an additional Y iterations. The total number of clock cycles per iteration is therefore $X + Y$. Adding an extra pipeline stage into the outer loop ($Y \rightarrow Y + 1$) increases the total number of clock cycles per iteration by 1 ($X + Y \rightarrow X + Y + 1$). The relative increase in clock cycles required to compute one modular multiplication (outer loop) can be expressed as $C1 = (X + Y + 1)/(X + Y)$.

Adding the additional pipelining stage in the outer loop, would decrease the maximum logic depth by a coefficient $C2 = (Y + 1)/Y$ (assuming that the pipeline stages are distributed evenly). Since $C1 < C2$, the overall design performance will increase if the logic depth in the inner loop is smaller than the outer loop logic depth. Therefore, we can improve the overall performance by adding pipeline stages into an outer loop until the outer loop maximum logic depth will match the inner loop logic depth as shown on Figure 7(c).

Figure 8 shows an example for $X = 8$ where by increasing the pipeline depth of the outer loop from $Y = 2$ to $Y = 3$ the critical path delay (which was assumed to be in the outer loop) has decreased by one third, from 3 period units to 2 period units. Consequently, the total delay of the deeper pipelined design has decreased from 30 to 22 period units.

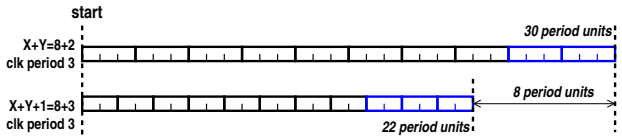


Figure 8: Directed Pipelining Example

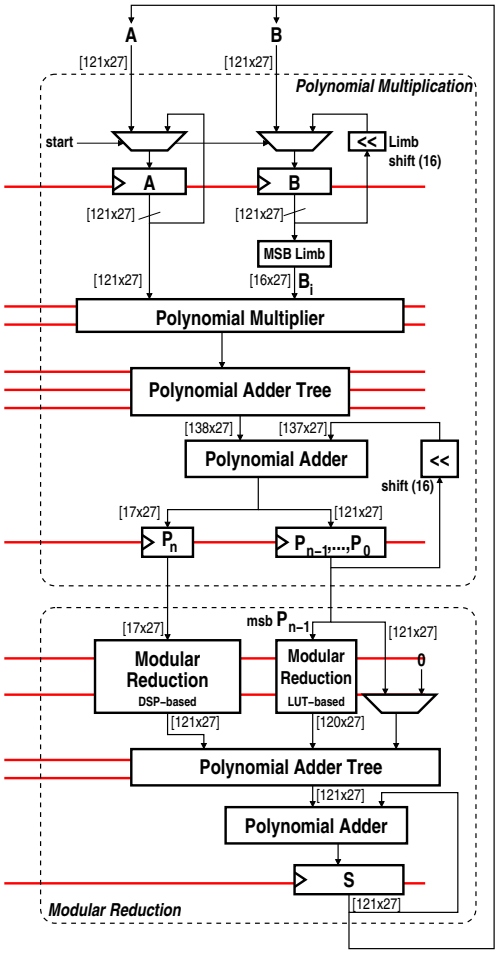


Figure 9: Pipelined iterative modular multiplier

We applied this approach to the two-level loop datapath shown in Figure 9. A total of 9 additional pipeline stages were added to the outer loop, bringing the overall number of pipeline stages to 12. Since the accumulation loops (inner loops) run for 8 cycles, the overall number of clock cycles per iteration (of the outer loop) is 12+8-1 = 19. The maximum logic depth of the pipelined design corresponds to 2 consecutive adders which is the minimum possible depth of the accumulation loop.

Table 2: Resource Report

Hierarchy	ALM	ALM (%)	DSP	DSPs (%)
Solver	161269	33	3891	86
Multiplcation	72256	15	1936	43
Reduction	82240	18	1955	43

5 RESULTS AND DISCUSSION

5.1 Implementation

We implemented the solver using an Intel Agilex F-Series FPGA Development Kit [35] based on AGFB014R24A2E3VR0 FPGA device, as this was the most recent FPGA available to us in a development board form. This is a mid-size, slowest-speedgrade (-3) Agilex device and therefore motivates our development of an area-efficient solution. In terms of frequency, we are confident that switching to a faster-speedgrade device (-1) can improve frequency by another 20%. Although this device has less than half the logic of the Xilinx VU9 devices on the Amazon F1 instances (as used by the VDF Alliance competition and the Cryptophage LCS35 solution), our DSP based method is very logic efficient as we do not need any soft logic for the coefficient storage. The number of DSP resources of our selected Agilex device is also much lower than the VU9 device (4510 vs. 6840), but the Agilex DSP Blocks can support 27x27 multipliers directly, which is 50% more arithmetically dense than the individual Xilinx VU9 DSP Blocks (which support 27x18 multipliers) for this application.

Our goal was to create a regularly placed design. In this version, we achieved this by a combination of explicit (placing DSP blocks in column groups) and implicit (the directed pipelining method introduced in the earlier section) methods. We did not use any logic floorplanning, but rather let the DSP placement drive the place and route of the solver. Table 2 presents the total resource utilization of our proposed solver, combined with a resource breakdown for our two main units. An additional 6573 ALMs and 14 M20K memory blocks - not reported in the table are needed to construct the entire solver implementation. The presented architecture is using a 350MHz clock. As it can be observed from the table, our proposed solver balances well the resources between the multiplication and reduction components (both ALMs and DSPs). This can also be observed from the floorplan shown in Figure 10, where the multiplier is shown in purple and the modular reduction in red.

5.2 Power Analysis and Performance Comparison

We measured the actual dynamic power of our FPGA to be 32W. We also ran the Agilex Power Analyzer [36] tool for comparison. The default settings estimated the dynamic power consumption at 13.1W. In our experience, the default setting are not correct for arithmetically dense designs, as the default toggle rate of 12.5% is too low. At a 50% toggle rate, the tool returned 28.7W, which is in line with our measured value. This understanding is important, as the reported (but only simulated) VDF Alliance contest result power numbers are likely incorrect. Devlin reports [12] that his design consumes 4.9W and the Pearson design 18.3W (both numbers from the Xilinx Power Estimator Tool). We ran the reported area for

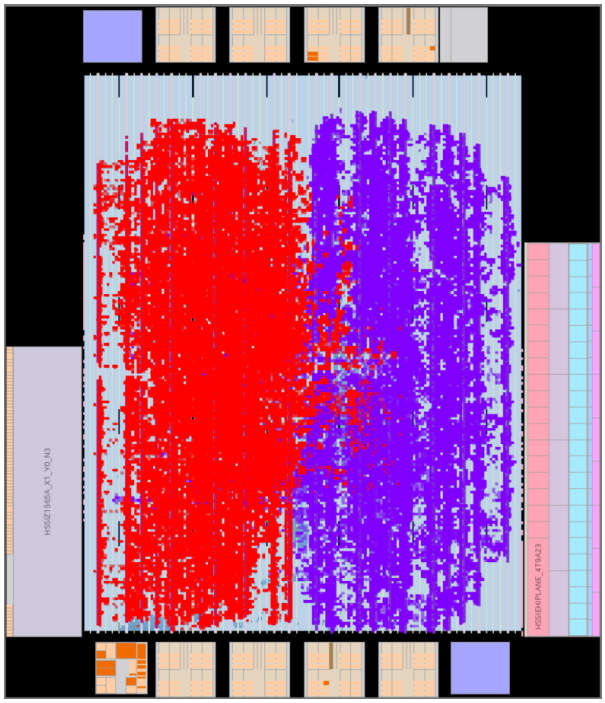


Figure 10: Multiplier and Reduction Modules

Table 3: Power Efficiency Analysis

Parameter	Ours	Devlin	Pearson
Bits	3072	1024	1024
Latency	54ns	46ns	25.2ns
Normalized Latency	6ns	46ns	25.2ns
Normalized Performance	7.7	1	1.8
Power *est.*	28.7W	12.0W	65W
Power Efficiency	3.2	1	0.32

Devlin's design through the Power Estimator [37] and obtained similar numbers. Like the Intel tool, Xilinx default toggle rates are 12.5%. Again, we believe that the correct toggle rates for these applications is 50%, which would approximately quadruple the estimated power consumption reported.

We normalized the performance (latency/(bits^2/1024)) to obtain a comparison of the arithmetic efficiency of all three approaches and divided by the estimated power (all at 50% toggle rate) for the efficiency. The results are shown in Table 3. Our approach has several times the arithmetic efficiency (normalized latency) of the next nearest method, and also significantly better power efficiency. As with our earlier latency comparison, we need to apply some caution. We need to discount our efficiency somewhat, as the Agilex devices are on 10nm FinFET, and the VU9P devices use 16nm FinFET. Our multiplier reduction also requires more DSP Blocks, although for a VDF application the latency metric greatly outweighs cost or power considerations.

Table 4: Solver Milestone Status

Milestone	Runtime	Obtained	Status
$t = 2^{28}$	14.57s	Feb 23, 2022	**Done**
$t = 2^{29}$	29.14s	Feb 23, 2022	**Done**
$t = 2^{30}$	58.29s	Feb 23, 2022	**Done**
$t = 2^{31}$	1.94m	Feb 23, 2022	**Done**
$t = 2^{32}$	3.89m	Feb 23, 2022	**Done**
$t = 2^{33}$	7.77m	Feb 23, 2022	**Done**
$t = 2^{34}$	15.54m	Feb 23, 2022	**Done**
$t = 2^{35}$	31.09m	Feb 23, 2022	**Done**
$t = 2^{36}$	1.04h	Feb 23, 2022	**Done**
$t = 2^{37}$	2.07h	Feb 23, 2022	**Done**
$t = 2^{38}$	4.14h	Feb 24, 2022	**Done**
$t = 2^{39}$	8.29h	Feb 24, 2022	**Done**
$t = 2^{40}$	16.58h	Feb 27, 2022	**Done**
$t = 2^{41}$	1.38d	Feb 28, 2022	**Done**
$t = 2^{42}$	2.76d	Mar 2, 2022	**Done**
$t = 2^{43}$	5.53d	Mar 7, 2022	**Done**
$t = 2^{44}$	11.05d	Mar 14, 2022	**Done**
$t = 2^{45}$	22.11d	Mar 25, 2022	**Done**
$t = 2^{46}$	44.21d	Apr 15, 2022	**Done**
$t = 2^{47}$	88.43d	May 31, 2022	**Done**
$t = 2^{48}$	176.85d	Aug 26, 2022	**Done**
$t = 2^{49}$	*est.* 353.71d		**In progress**
$t = 2^{50}$	*est.* 1.94y		**not started**
$t = 2^{51}$	*est.* 3.87y		**not started**
$t = 2^{52}$	*est.* 7.75y		**not started**
$t = 2^{53}$	*est.* 15.49y		**not started**
$t = 2^{54}$	*est.* 30.99y		**not started**
$t = 2^{55}$	*est.* 61.98y		**not started**
$t = 2^{56}$	*est.* 123.95y		**not started**

5.3 Solver

The solver started running in February 2022, and in the first 6 months found 21 milestone solutions (from $t = 2^{28}$ to $t = 2^{48}$) of the puzzle. The solver is using a 350MHz clock with one modular squaring operation taking 19 clock cycles, which gives 54 nanoseconds per squaring operation. The actual runtimes for completed milestones and estimated run times for future milestones, together with the completion date of the achieved milestones are given in Table 4.

We have reported our results to the MIT CSAIL team. Our milestone solutions have been recorded and are at the top of the leaderboard [34] [38].

As it can be observed from Table 4, the performance of the presented solver is not sufficient to fully solve the puzzle before the deadline in 2033. Solving the puzzle requires roughly one order of magnitude improvement in performance (loop latency) to meet the deadline. Nonetheless, we hope that our work will inspire more FPGA researchers to the problem, and will serve as a base architecture to solve other time-locked puzzles with future generations of FPGAs and more architectural improvements.

To meet the deadline of the CSAIL2019 puzzle, we need a 10x improvement - in other words, our 54ns iteration time must reduce to

5.4ns. Our current architecture uses an 8-iteration modular multiplication, which is first driven by the number of resources (principally the 4150 DSP Blocks) on the mid-size device we are using. Larger FPGAs are currently available [39], with over 12K DSP Blocks on some of the larger devices. This same FPGA family (Agilex) also has members with 3x the DSP Blocks. Although this would not evenly divide into our iteration granularity, we can also investigate a mixed (multiplicative and table based) approach. As both types of reduction calculations remap portions of values that are outside the N modulus width back into that space, they will be compatible with each other. We are using a relatively small amount of soft logic compared to the amount of DSP, so it is likely that a reasonably routable solution could be realized. One caveat is that reducing the number of iterations would increase the critical path, especially in the summation of the partial product columns (comprising both the multiplication and reduction portions of the implementation), which may impact the operating frequency negatively.

6 FUTURE WORK

We believe that the clock rate of our solver implementation can be improved. We are currently overclocking the FPGA for this application by 10% and we are confident that it is possible to push this even further. Given that the design has a low logic usage and almost no memory blocks, the power consumption is lower than it is typically for a full-chip design of this size. We know that we are not near the thermal limits of this device, and we also have a robust error checking method to continuously verify our results. There are several different possibilities to further develop overclocking: both tuning the existing methods as well as developing new ones. We believe that we can boost the operating frequency by 25% over the Quartus reported value by monitoring power (which will increase with increased frequency, thereby increasing temperature, and in turn reducing our thermal margin). The power based performance optimization will be a slowly varying parameter. The continued correct operation of the circuit can be monitored by our error checking methodology.

We have also identified possible data-based clocking improvements, which we may be able to apply on an iteration by iteration basis.

The combination of using larger existing devices and both the slowly and quickly varying clock rates will likely not achieve a 10x increase in performance, although a 5x-6x improvement is possible.

7 CONCLUSIONS

In this paper we have analysed the CSAIL2019 problem and compared the computational scale of the problem with known approaches of solving it. It appears that a standard CPU is well out of running, and the current FPGA methods will have difficulty scaling.

We have described a new FPGA based algorithm that shifts the computation from tables to DSP Blocks, which allows for a higher density solution, with higher throughput and more predicable timing closure. This also reduces the routing problem for soft logic, allowing more scalability with higher performance. We then developed a multi-cycle implementation using FIR filter coefficient storage, which lets us directly calculate milestone solutions of the puzzle even using modest-sized FPGAs. We have also introduced a new way of pipelining large systems, which when combined with a modest level of embedded feature placement, creates a regular placement for a complex design, with repeatable and predictable performance. This allows for modifications to the design to be made with a high degree of confidence in the new FPGA fitting.

As part of our discussion on FPGA performance, we have described how we can confidently run the design somewhat faster than the reported timing closure, and how this might be applicable to many other types of designs.

We have compared normalized metrics of our architecture to similar designs used in the VDF Alliance contest, and found that our approach (which uses methods such as the multiplicative reduction, as well as a left-to-right multiplication calculation that allows a simultaneous multiplication expansion and reduction operation), provides significantly better arithmetic and power efficiencies. More importantly for VDF applications, our method will provide the lowest overall latency, and is also scalable to larger bit widths.

Finally, we look to the future. While the performance and computation density of our new work is a significant improvement over previously reported work, the presented design will not meet the full CSAIL2019 puzzle solving deadline. We may, however, be able to get close, assuming that we can continue to develop more ways of increasing performance, either by improving our implementation parallelism, our frequency tuning, or a combination of both. In any case, the CSAIL2019 puzzle is a very challenging problem for FPGAs.

REFERENCES

[1] "Description of the CSAIL2019 Time Capsule Crypto-Puzzle," https://people.csail.mit.edu/rivest/pubs/Riv19f.new-puzzle.txt, accessed: 2022-08-15.

[2] "Description of the LCS35 Time Capsule Crypto-Puzzle," http://people.csail.mit.edu/rivest/pubs/Riv99b.lcs35-puzzle-description.txt, accessed: 2022-08-15.

[3] "Programmers Solve MITs 20-year-old Cryptographic Puzzle," https://www.csail.mit.edu/news/programmers-solve-mits-20-year-old-cryptographic-puzzle, accessed: 2022-09-14.

[4] https://www.supranational.net/, accessed: 2022-09-14.

[5] "VDF Alliance FPGA competition," https://supranational.atlassian.net/wiki/spaces/VA/pages/36569208/FPGA+Competition, accessed: 2021-06-14.

[6] E. Öztürk, "Design and implementation of a low-latency modular multiplication algorithm," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 6, pp. 1902–1911, 2020.

[7] "Amazon EC2 F1 instances," https://aws.amazon.com/ec2/instance-types/f1/, accessed: 2022-09-14.

[8] "Virtex UltraScale+ product tables," https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html, accessed: 2022-09-14.

[9] "UltraScale Architecture DSP Slice," https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp, 2021-08-30.

[10] "VDF Alliance FPGA competition round 1 results and announcements," https://www.vdfalliance.org/news/fpga-competition-round-1-results, 2019-10-31.

[11] "Pearson round 2," https://github.com/supranational/vdf-fpga-round2-results/tree/master/eric_pearson_2, accessed: 2022-09-14.

[12] https://blog.janestreet.com/really_low_latency_multipliers_and_cryptographic_puzzles, 2020-06-22.

[13] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[14] D. J. Bernstein, Jonathan, and P. Sorenson, "Modular exponentiation via the explicit Chinese Remainder Theorem," pp. 443–454, 2007.

[15] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard Digital Signal Processor," in *Advances in Cryptology — CRYPTO' 86*, A. M. Odlyzko, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 311–323.

[16] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *USSR Academy of Sciences*, vol. 145, pp. 293–294, 1962.

[17] A. D. Booth, "A signed binary multiplication technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951. [Online]. Available: +http://dx.doi.org/10.1093/qjmam/4.2.236

[18] https://github.com/supranational/vdf-fpga-round3-results/tree/master/papers, 2020-01-30.
[19] Y. Gong and S. Li, "High-throughput FPGA implementation of 256-bit Montgomery modular multiplier," in *2010 Second International Workshop on Education Technology and Computer Science*, vol. 3, March 2010, pp. 173–176.
[20] M. Morales-Sandoval and A. Diaz-Perez, "Scalable GF(p) Montgomery multiplier based on a digit-digit computation approach," *IET Computers Digital Techniques*, vol. 10, no. 3, pp. 102–109, 2016.
[21] E. Ozcan and S. S. Erdem, "A high performance full-word Barrett multiplier designed for FPGAs with DSP resources," in *2019 15th Conference on Ph.D Research in Microelectronics and Electronics (PRIME)*, July 2019, pp. 73–76.
[22] M. Langhammer and B. Pasca, "Efficient FPGA modular multiplication implementation," in *FPGA '21: The 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Virtual Event, USA, February 28 - March 2, 2021*, L. Shannon and M. Adler, Eds. ACM, 2021, pp. 217–223. [Online]. Available: https://doi.org/10.1145/3431920.3439306
[23] F. de Dinechin and B. Pasca, "Large multipliers with fewer DSP blocks," in *International Conference on Field Programmable Logic and Applications*. IEEE, aug 2009.
[24] M. Kumm, O. Gustafsson, F. de Dinechin, J. Kappauf, and P. Zipf, "Karatsuba with rectangular multipliers for FPGAs," in *25th IEEE Symposium on Computer Arithmetic, ARITH 2018, Amherst, MA, USA, June 25-27, 2018*. IEEE, 2018, pp. 13–20. [Online]. Available: https://doi.org/10.1109/ARITH.2018.8464809
[25] E. Vitali, D. Gadioli, F. Ferrandi, and G. Palermo, "Parametric throughput oriented large integer multipliers for high level synthesis," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 38–41.
[26] M. Langhammer and B. Pasca, "Folded integer multiplication for FPGAs," in *FPGA '21: The 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Virtual Event, USA, February 28 - March 2, 2021*, L. Shannon and M. Adler, Eds. ACM, 2021, pp. 160–170. [Online]. Available: https://doi.org/10.1145/3431920.3439299
[27] *Intel Stratix®10 GX/SX Device Overview*, 2018, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/s10-overview.pdf.
[28] J. Chromczak, M. Wheeler, C. Chiasson, D. How, M. Langhammer, T. Vanderhoek, G. Zgheib, and I. Ganusov, "Architectural enhancements in Intel® Agilex™ FPGAs," in *FPGA '20: The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, February 23-25, 2020*, S. Neuendorffer and L. Shannon, Eds. ACM, 2020, pp. 140–149. [Online]. Available: https://doi.org/10.1145/3373087.3375308
[29] "UltraFast design methodology guide for Xilinx FPGAs and SoCs," https://docs.xilinx.com/r/2021.2-English/ug949-vivado-design-methodology/Super-Logic-Region-SLR, 2021-11-19.
[30] M. Langhammer, S. Gribok, and B. Pasca, "Low-latency modular exponentiation for FPGAs," *IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2022.
[31] *Intel Agilex Variable Precision DSP Blocks User Guide*, 2019, https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/agilex/ug-ag-dsp.pdf.
[32] *Intel® Stratix® 10 Variable Precision DSP Blocks User Guide*, https://www.intel.com/content/dam/support/us/en/programmable/support-resources/bulk-container/pdfs/literature/hb/stratix-10/archives/ug-s10-dsp-18-1.pdf, 2018, 2018-09-24.
[33] "The On-Line Encyclopedia of Integer Sequences: sequence A141305," https://oeis.org/A141305, accessed: 2022-08-16.
[34] R. Rivest, Personal communication - April 2022.
[35] *Intel® Agilex™ F-Series FPGA Development Kit*, https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=142&No=1262, 2022, accessed: 2022-09-14.
[36] *Intel® Agilex™ Power Analyser*, https://www.intel.com/content/www/us/en/support/programmable/support-resources/power/pow-powerplay.html, 2022, accessed: 2022-09-14.
[37] *Xilinx Power Estimator User Guide*, https://docs.xilinx.com/r/en-US/ug440, 2022-04-06.
[38] R. Rivest, personal communication - May 2022.
[39] *Intel Agilex F-Series FPGA and SoC Product Table*, 2019, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/intel-agilex-f-series-product-table.pdf.