Investigating Performance Losses in High-Level Synthesis for Stencil Computations

Wesson Altoyan
Department of Electrical Engineering
Stanford University
Stanford, CA, U.S.A.
waltoyan@stanford.edu

Juan J. Alonso
Department of Aeronautics and Astronautics
Stanford University
Stanford, CA, U.S.A.
jjalonso@stanford.edu

***Abstract*—With the aid of few directives and canonical forms, high-level synthesis (HLS) tools allow FPGA developers to describe their hardware designs in higher-level languages such as C or C++, thus enabling software engineers to exploit the powerful capabilities of FPGA-based accelerators. Hardware engineers may also benefit from automated RTL generation through HLS, as it can boost their productivity with shorter development cycles and simpler validation processes. However, the introduction of automation with its associated performance losses brings the viability of the current HLS-based approach into question: are we sacrificing the main advantage of FPGA designs, namely their performance, in return for higher productivity? This paper examines the performance-per-power penalties incurred in HLS designs in the context of stencil computations for fluid flow simulations, a prevalent class of applications that are difficult to accelerate because of their low arithmetic intensity. By using Xilinx's Vivado HLS tool to replicate a hand-crafted RTL implementation of a solution of Laplace's equation, this paper evaluates the impact of implicitly expressing parallelism, particularly if specific optimizations are not directly supported by the tool. In addition, by describing scenarios in which the HLS approach does and does not excel for stencil-based computations, this paper offers insights to assist hardware engineers in setting their expectations of the HLS approach and suggests alternative techniques to accomplish common tasks that fail or underperform using typical approaches.**

I. Introduction

Hardware accelerators have emerged as mainstream computing platforms, with leading technology providers such as Microsoft [1], Google [2], and Amazon [3] including GPUs and FPGAs as part of their cloud service offerings. GPUs, in particular, are widely adopted because of their lower entry barrier and superior power/performance rating compared with CPUs. However, they are reaching their power limit [4] and their performance is often capped to conserve energy and prevent overheating, consequently wasting valuable computational resources and maxing out power sources. In addition, GPU performance is often impeded by limited memory bandwidth, especially in applications with low arithmetic intensity [5], where little computation is performed per byte of data retrieved.

With a powerful combination of large on-chip memory and highly-customizable designs, FPGAs hold the potential to provide acceleration where GPUs prove lacking. By leveraging the generous on-chip memory, FPGA clusters can accommodate entire datasets on-chip, eliminating frequent data loads and reloads from off-chip memory and alleviating the memory bandwidth bottleneck that limits GPU performance. Additionally, the flexible design of FPGAs optimizes energy consumption through hand-crafted, highly-customized designs.

However, despite FPGA's promising ability to surmount the two most pressing limitations of GPUs, namely, limited memory bandwidth and high power consumption, the introduction of FPGA-based solutions has been slow owing to the cumbersome implementation efforts involved in FPGA designs. Contrary to GPUs that employ high-level languages, FPGA development leverages hardware description languages (HDL) such as Verilog or VHDL. These languages describe parallel functionality at low levels of abstraction, where timed operations must be meticulously specified and tested for register transfer level (RTL) designs using elaborate verification processes. This slow time to market welcomed the advent of high-level synthesis (HLS) tools [6], which reduce FPGA development cycle times through automated translation of higher-level code, such as C or OpenCL, to RTL. Moreover, by utilizing commonly-used high-level languages, HLS also enables software developers with little to no hardware knowledge to take advantage of the powerful capabilities of FPGAs.

Nevertheless, with the adoption of high-level methodologies such as HLS, the main advantages of FPGA-based solutions may be lost and a healthy dose of skepticism is called for. Two questions in particular arise with the introduction of automation: first, how much performance, power, and hardware resources are we losing in exchange for improved productivity? Second, is this price worth the return?

In the context of stencil computations [7], a class of applications with low arithmetic intensity that requires data accesses from nearest neighbors only, this paper attempts to answer the former question with the objective of empowering researchers and engineers to answer the latter, based on their specific applications and priorities. In this paper, we investigate the ability of Xilinx's Vivado HLS tool to adequately and efficiently describe a hardware design, and the extent to which it can offer developers flexibility and control over the desired optimizations. For situations where the HLS tool proves lacking, we also suggest methods to overcome the limitations of the tool and offer insights to help developers set reasonable expectations. To achieve these goals, a highly-optimized RTL application was

2576-2621/20/$31.00 ©2020 IEEE
DOI 10.1109/FCCM48280.2020.00034

IEEE computer society

developed and then replicated using HLS. Both implementations were then compared in terms of speed, power, and resource utilization.

II. Background and Related Work

FPGA vendors have long recognized the productivity challenge associated with developing RTL designs and have sought to alleviate it by supplementing their development tools with soft processors and IP cores that hide complexity and enable design reuse. HLS tools go a step further towards this goal by using high-level languages to abstract away implementation details, offloading the tedious task of RTL coding to automated tools. With such potential for a breakthrough in FPGA development, the introduction of HLS tools resulted in numerous studies describing their limitations and challenges [8-10].

A. High Level Synthesis

While there is a plethora of academic and commercial HLS tools with the universal goal of automated RTL generation, these tools vary in their approaches and input languages [11]. In this paper, we limited our study to Xilinx's Vivado HLS tool [12] where developers only need to complement their high-level programs with pragmas and Xilinx's canonical forms to guide the tool in its automated translation to low-level parallel descriptors for efficient designs. This approach is not unique to Vivado HLS, and is adopted by other, commonly used HLS tools such as Altera's HLS Complier [13] and Mentor Catapult [14].

Studies that tackled HLS as a means of exposing hardware accelerators to software developers demonstrated tremendous performance gains that could easily justify the additional learning curve. For example, compared to software implementations, speedups of 67×, 126×, and an astounding 8500×, were achieved in stereo matching [15] embedded benchmark kernels [16], and error correction codes [17], respectively. Applications gaining modest speedups of three to nine times the software performance [18-20] were also considered satisfactory, particularly when the reference software implementations are the highest-performing implementations available. For hardware engineers, however, the trade-off is reversed; RTL designs can be extremely efficient, and their main drawback is prolonged development time. Although hardware engineers are undoubtedly interested in improving their productivity, they may not be willing to give up the high performance they are accustomed to, and therefore recommend confining the use of HLS to design-space explorations [21], and not production-grade FPGA designs.

Nevertheless, numerous efforts have been made to evaluate the performance of HLS implementations compared to RTL designs in various domains, and results depended heavily on the application. For example, a Memcached server application [22] benefited from the accelerated development cycle of HLS, reaping tangible gains in both performance and resource usage at 35% and 30%, respectively. On the other hand, applications in nuclear science [23] suffered drastically as their resource usage doubled and their performance halved. More common in HLS implementations, however, is a detachment of resource savings from performance gains. Under HLS, applications such as image processing [24] and arithmetic units achieved roughly the same performance as the corresponding RTL implementations, at the expense of increased resource usage of up to 100%. Also, an HLS implementation of a RADAR signal processing algorithm [25] achieved a speedup of two, at the cost of a fourfold increase in resource usage, compared to RTL. Resource usage does not always suffer under HLS, however. In the cryptography domain [26], for example, throughput dropped by 47% in the HLS implementation compared to RTL, while resource saving reached 22%. An HLS implementation of K-means clustering [27] also suffered a degraded performance of 50% at the same resource usage level as the RTL. It must be noted, however, that increased usage of one resource may come in tandem with a decrease in another. For example, logic consumption may be accompanied with increased use of BRAM [28-29], and increased lookup-table usage may occur with a decrease in flip-flop usage [30], making it difficult reach a final verdict on the effect of HLS on the application's resource utilization.

Another factor that goes into the evaluation of the HLS approach is the productivity improvements it offers over the traditional RTL approach. Yet, measuring productivity in this context can be quite challenging [30]. Often, comparing two approaches involves a learning curve in one approach but not the other, and the first approach pursed in the comparison incurs the cost of additional activities such as initial design decisions and optimization details. Having said that, several works have cited HLS development time to be about one- to two-thirds that of RTL [19,22,24-25], although it can be five times lower [31].

The third factor in evaluating the HLS approach is power consumption, which is seldom discussed in such studies despite its direct impact on the total cost of ownership of the final product.

In summary, with productivity being difficult to measure, and the performance and resource utilization depending on the application, drawing conclusions on the general cost-benefit analysis of the HLS approach can be quite challenging. In fact, even for the same domain and application, HLS designs can vary in terms of performance and resource usage depending on the adopted architecture [32]. Moreover, the choice of HLS tool directly affects the efficiency of the generated RTL description [33-34], a topic that is outside the scope of this paper.

B. Stencil Computation

In this study, we focus our attention on stencil computation, a class of applications that process multi-dimensional arrays, updating the value of each array cell by using the values of its nearest neighbors. These neighboring cells are determined by the stencil, a fixed pattern that defines the neighborhood of any given cell. For example, in a two-dimensional matrix, a five-point stencil includes the center cell, in addition to the four orthogonal neighbors of a cell, namely the top-, bottom-, left-, and right-neighbor cells. A nine-point stencil, on the other hand, also includes the four diagonal neighbors. Thus, the fundamental idea of stencil computations can be extended to neighbors of neighbors. Because several neighboring cells are required to compute a single cell value, and the number of operations is not

very significant, stencil computations have low arithmetic intensity and consequently high bandwidth requirements. On the other hand, stencil computations are inherently parallel, given that each cell can be updated concurrently, and that the access of only immediate neighbors can lead to high data locality.

We choose to solve Laplace's equation ($\nabla^2\phi = 0$) using a five-point stencil and a Jacobi iteration scheme, a common algorithm that is similar to stencil-based operations used in many scientific computing applications. Implemented in a two-dimensional uniform grid, the algorithm marches through a series of iterations, producing an output matrix from an input one by averaging the neighbors of each inner cell, as shown in figure 1. The boundary cells of the matrix remain unchanged and can be updated using a simple copy. However, by initializing input and output matrices to the same values, the copying step can be eliminated, and boundary cells are simply skipped. Input and output matrices are swapped at the end of each timestep, and the loop goes on until the two matrices converge in an appropriate error norm. Computation is performed using double precision floating-point variables, and convergence is determined by comparing the mean squared error to a user-set tolerance.

III. Design

To exploit the algorithm's locality, we designed our solution to employ multiple parallel processing elements (PEs), each of which processes a submatrix of 16 rows and 256 columns. PEs operate independently within a timestep, but not across timesteps because the convergence check performed at the end of each iteration is needed before proceeding to the next timestep, and it cannot be performed until all PEs have finished processing the current timestep. In addition, the border cells at the edges of each submatrix will have to obtain one of their four neighbors from an adjacent PE, and synchronization is necessary to ensure the retrieval of the updated value. Such cells are referred to as "halos" or "halo cells", and their efficient and timely exchange is a crucial factor in high-performing implementations. For a simple and efficient exchange of halo cells between PEs, PEs are stacked vertically to limit halos to top and bottom PE rows. PEs are wrapped in a top-level Engine module that controls the time-stepping through convergence checks.

```
while (error > tolerance) {
  running_diff = 0
  Loop from i=1 to i=rows{
    Loop from j=1 to j=columns{
      if ( in_matrix(j,i) is boundary cell){
        out_matrix(j,i)=in_matrix(j,i)
      }
      else{
        sum = in_matrix(j+1,i) + in_matrix(j,i+1)
            + in_matrix(j-1,i) + in_matrix(j,i-1)
        out_matrix(j,i) = sum/4
      }
      diff = (out_matrix(j,i) - in_matrix(j,i))^2
      diff_sum +=diff
    }
  }

  error = sqrt(diff_sum / size);
  swap (in_matrix, out_matrix);
}
```

Fig. 1. Pseudo-code listing of the implemented algorithm.

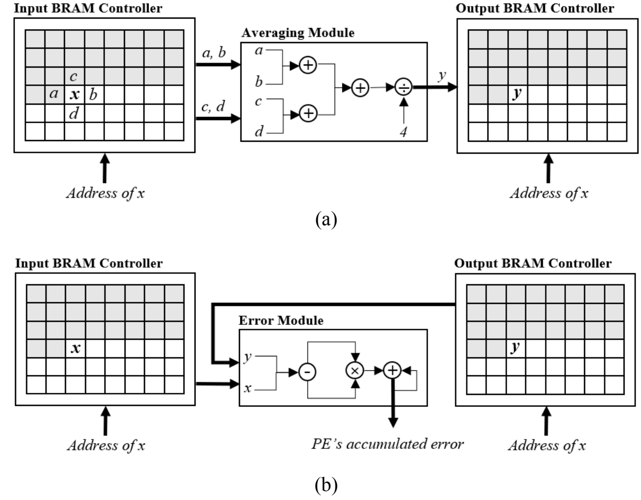

Fig. 2. Basic design uses two passes for a throughput of three cycles. (a) First pass takes two cycles to pass the four neighbors from input BRAM Controller to averaging module. (b) Second pass computes error every cycle

Each PE is composed of a BRAM controller module that manages the input and output matrices, an averaging module that computes the new value of each cell, and an error module that computes the PE's accumulated error. As shown in figure 2, the basic operation of a PE involves traversing the input matrix twice in each timestep. In the first pass, the four neighbors of the target cell are retrieved from the input BRAM controller and passed to the averaging module. Once the computed value is ready, it is passed from the averaging module to the output BRAM controller for storage. In the second pass, the original value residing in the input matrix and the newly-computed value stored in the output matrix are passed to the error module and the final accumulated error is then forwarded to the Engine module for the convergence check, based on the total error computed by all PEs.

A. HDL Implementation

The pipelined HDL design achieves high performance by pursuing several optimization directions. First, an ideal PE throughput of one cell update per cycle is achieved by removing memory bottlenecks. Next, runtime is reduced by eliminating control overheads in the management of both timesteps and halos. Finally, through efficient use of hardware resources, the FPGA can accommodate more PEs for higher computational power. Table 1 lists the six optimization techniques carried out in the HDL implementation, along with their resulting impact on performance. Moreover, a description of each optimization technique is detailed below.

BRAM Partitioning: In the first pass, the averaging module needs to access four neighbors while BRAM blocks support only two ports, leading to a module throughput of two cycles per cell update. By observing the predefined access pattern, however, we notice that the four accesses performed by the averaging function require only two cells of the same parity. As such, by partitioning the PE's BRAM memory into two blocks

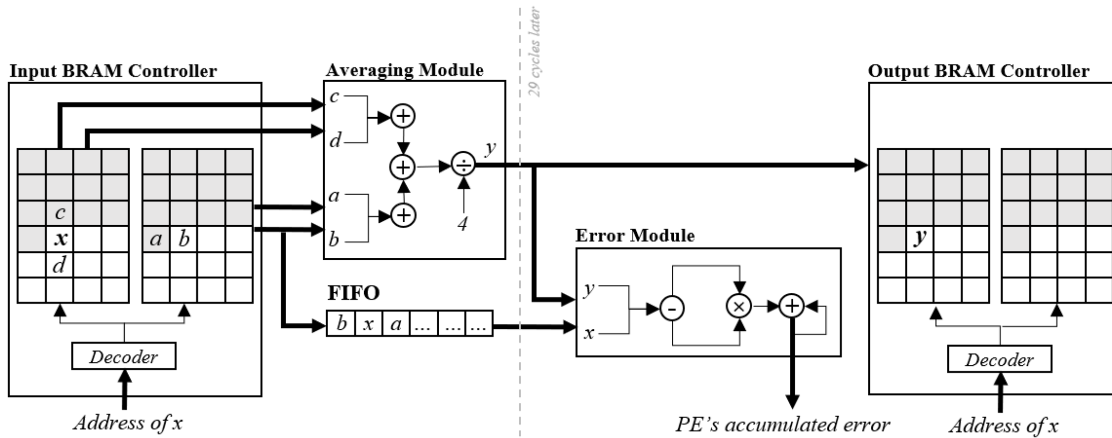

Fig. 3. The BRAM Controller in the optimized HDL implementation uses two BRAM blocks. In the current cycle, neighbors *a,b,c*, and *d* of the target cell *x* are simultaneously passed to the averaging module. The right neighbor, *b*, is also buffered into a FIFO. Similarly, in the previous cycle, *x* was buffered when the neighbors of *a* were retrieved. After 29 cycles, the averaging module produces *y*, the updated value of *x*. Both *x* and *y* are then passed to error module.

and using the cells' parity bit to index into the BRAM blocks, all four neighbors can be retrieved in one cycle, without conflict and with minimal overhead. Thus, by effectively doubling the memory ports, the throughput of the averaging module is improved to one cell update per cycle.

One-pass Processing: To achieve a PE throughput of one cell update per cycle, the two passes must be fused such that a PE traverses its matrix only once per timestep. Consequently, the PE must be able to perform up to five memory reads in any given cycle, because the pipelined nature of the RTL design must allow for the averaging and error computation steps to occur simultaneously as part of different pipeline stages. Further BRAM partitioning does not offer an efficient solution for the fifth access needed for the error computation, as it would introduce complex access management to avoid conflicting accesses. Alternatively, the data access pattern is exploited by observing that the right neighbor of the current cell will be the target cell in the next cycle, which is the same value to retrieve for computing the error of the next cell once its output is available. As shown in figure 3, by buffering the right neighbor of every cell until the output of its next cell is computed, we eliminate the need for a fifth memory access altogether. Because the averaging module has a latency of 29 cycles, a small, register-based FIFO of 32 entries is used to hold the right neighbors until the corresponding output is available.

Seamless Halo Exchange: By exploiting the lockstep nature of PE execution, PEs can access halos as if they were local neighbors instead of exchanging halos in a separate, prior step in each iteration. This is possible because cells at the PE border use only three out of four available BRAM ports, so the fourth port can be used to service a halo to an adjacent PE, as illustrated in figure 4.

Back-to-back Iterations: The total runtime can be further reduced by running timesteps back to back, assuming that the matrices have not converged yet. Thus, instead of waiting for the convergence test results after a timestep completes, the next timestep starts immediately in parallel with the error computation for the convergence check. If convergence is achieved, the timestep is aborted and the input matrix, rather than the output one, is returned.

Optimizing FP Division: The number of hardware resources can be reduced by substituting the expensive division by 4.0 operation with the cheaper multiplication by 0.25 operation. However, a greater resource saving would take advantage of the FPGA's ability to manipulate data at the bit level and transform floating-point operations into integer operations. By observing that a floating-point division by 4.0 simply reduces the floating-point number's mantissa by 2, the floating-point division is replaced with a decimal subtraction operation, thereby releasing 10 DSP slices per PE.

Optimizing FP Accumulation: Taking advantage of the small data size of each PE, we decreased resource utilization by 87% through configuring the floating-point accumulator to minimal resource usage. This does not affect the accuracy of the results because the accumulator is only used for the convergence test, and in the worst-case scenario the accumulator would introduce a rounding error of 2^{-53} for each of the submatrix's 4,096 cells, totaling to a negligible sum for a typical tolerance of 10^{-6}.

TABLE I. OPTIMIZATION TECHNIQUES USED IN HDL

Optimization	Benefit Compared to HDL Baseline
BRAM partitioning	Speedup of 1.33×
One-pass Processing	Speedup of 1.33×
Seamless halo exchange	Speedup of 1.11×
Back-to-back iterations	Speedup of 1.03×
Optimizing FP ivision	Reduction of DSP utilization by 26%
Optimizing FP accumulation	Reduction of logic utilization by 87%

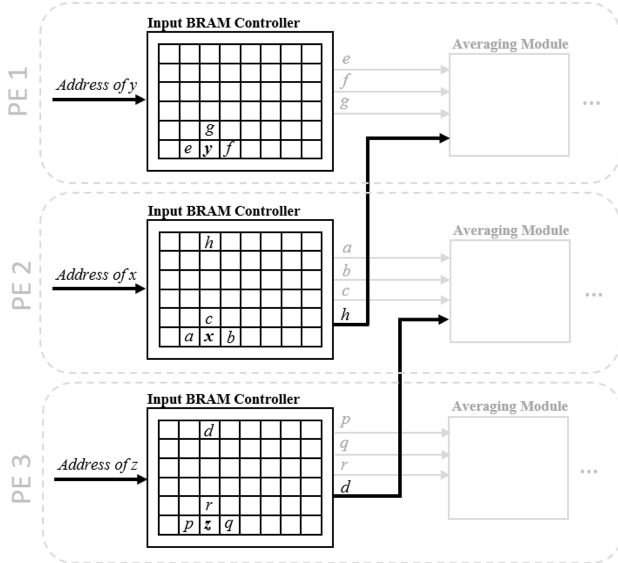

Fig. 4. Border cells retrieve their halo values from adjacent PEs as if they were local. PEs operate in lockstep, so the addresses of *x, y,* and *z,* are identical. *PE 2* receives its bottom halo, *d*, from *PE 3*, as it passes *h* to *PE 1*.

B. HLS Implementation

A naïve HLS implementation was also written in C. A top-level Engine function includes a main loop that iterates through timesteps and places several calls to the PE function in each iteration, before checking for convergence using the running difference sums returned by the PE function calls. Because PEs are implemented as functions, submatrices are placed within the Engine, not within each respective PE. Each PE receives pointers to the input and output submatrices, in addition to pointers to the submatrices of other PEs where halos reside.

Using this naïve implementation, two problems arise. First, a simple pointer swap to alternate between the two submatrices is not possible, because pointers-to-pointer are not allowed, and pointers must be statically assigned. Second, having two functions access the same array is not permissible, even if the two functions do not alter the array's data. This is the case when a PE is reading its input submatrix and another PE is reading the same submatrix to access its halos. To resolve this issue, dedicated halo arrays are created to host duplicate copies of the halo cells, and these arrays are passed to PE functions instead of adjacent submatrices, as shown in figure 5. Similar to the double-matrix approach, each PE has two halo arrays, one for the input and another for the output, and a new function is introduced for initial loading of halos. The performance cost of this function, however, is negligible as it is only run once. Although the halo-array approach incurs additional memory usage, it is unavoidable for successful code generation. As for matrix swapping, PE calls are duplicated and hardcoded with the two possible combinations of pointer arguments, and a conditional statement is used to choose the appropriate call based on the iteration's parity, also shown in figure 5.

By solving these issues, the baseline HLS implementation successfully generated RTL code, albeit showing very poor performance. This degraded performance was mainly caused by

```
load_halo(matrix_in, halos_in);

iteration_count = 0;
acc_error = 0;

while (error > tolerance) {
  if (iter_count %2 == 0){
    err = PE(matrix_in, matrix_out, halos_in, halos_out);
    acc_error += err;
  } else {
    err = PE(matrix_out, matrix_in, halos_out, halos_in);
    acc_error += err;
  }

  iteration_count++;
}
```

Fig. 5. Pseudo-code for hardcoded PE function calls in HLS implementation.

the absence of floating-point accumulators in HLS, and the use of floating-point binary adders instead. The problem with using adders for accumulation is the self-dependency of the adder's inputs on its outputs: the output of the current addition is an input for the next one and, therefore, each addition operation must be completed before starting the next. This means that pipelining is not possible and given the adder's eleven-cycle latency, processing a submatrix will take at least eleven times its size (in clock cycles) to complete one timestep. To improve performance, Xilinx recommends accumulating the values by cyclically distributing them over several adders, allowing a minimum of eleven cycles to elapse before any adder receives its next accumulation input [35]. As shown in figure 6, each adder maintains a partial sum, and these are also added up using a tree adder once the submatrix processing is complete.

As listed in table 2, optimization techniques used in the HDL version were adopted in the HLS implementation, where applicable. BRAM blocks were partitioned to increase memory ports by using pragmas, and reuse of the right neighbor through buffering was also implemented but without the explicit use of a FIFO. Instead, as the PE iterates through its cells, the right neighbor is cached in a variable and used in the next iteration by the next cell. To facilitate pipelining, HLS internally instantiates a FIFO to hold this value until the output of the next cell is available.

Resource reduction techniques used in the HDL version were not applicable to the HLS implementation and could not be replicated. In particular, converting floating point division to integer addition was not performed because bit manipulation is not easily accessible using high-level languages, and defeats the purpose of a using high-level language. As for accumulator

```
double add_all(double x[32]) {
  double acc_part[4] = {0.0, 0.0, 0.0, 0.0};
  for (int i = 0; i < 32; i += 4){// Manually unroll by 4
    for (int j = 0; j < 4; j++){// Partial accumulations
      acc_part[j] += x[i + j];
    }
  }
  for (int i = 1; i < 4; i++) {// Final accumulation
     acc_part[0] += acc_part[i];
  }
  return acc_part[0];
}
```

Fig. 6. Example code describing the implementation of accumulators using four partial adders for higher throughput in Vivado HLS. Adapted from [38].

optimizations, these were not applicable because adders were used instead. In fact, excessive use of resources to implement partial sums was necessary to overcome the adders' latency when used as accumulators.

Finally, runtime could not be further reduced through back-to-back execution of timesteps because loop-pipelining requires unrolling of all internal loops, which is not feasible. As for optimizing halo management, the design was refined to utilize custom PE implementations that process halos depending on the PE's position: First PE, which uses a bottom halo, Last PE, which uses a top halo, or Middle PE, which uses both halos. Consequently, the latency overhead of halo management was minimized, but the double-buffering of halos and the associated memory usage could not be improved.

TABLE II. APPLIED OPTIMIZATIONS

Optimization	HDL	HLS
BRAM partitioning	✓	✓
One-pass processing	✓	✓
Seamless halo exchange	✓	×
Back-to-back iterations	✓	×
Optimizing FP division	✓	×
Optimizing FP accumulation	✓	×

IV. EVALUATION

The performance of FPGA designs largely depends on the resources available on the FPGA. Thus, we proceed with a review of the experimental setup before evaluating the two designs.

A. Experimental Setup

Both the HDL and HLS implementations were developed using Xilinx technologies, which were selected because of their support of double-precision floating-point operations. The board used was Nexys 4, a low-cost FPGA evaluation board equipped with Xilinx Artix-7 100T [36] with limited resources, as shown in table 3. The HDL model was developed using Xilinx Vivado 2018 and Floating-Point Operator v7.1 LogiCORE IP for double-precision floating-point operations, and Clocking Wizard v5.4 LogiCORE IP for automated clock creation. The HLS version used HLS Vivado 2018, and the packaged IP was wrapped in a top-level module in Xilinx Vivado 2018 for implementation on the actual FPGA. Power measurements for both implementations were provided by the same tool.

TABLE III. FPGA RESOURCES

Resource	Count
Lookup-tables (LUTs)	63,400
Flip-flops (FFs)	126,800
DSP Slices	240
BRAM Blocks (18 Kbits)	270

B. Results and Analysis

Inadequate support of floating-point accumulators proved to be particularly problematic for the HLS implementation. The use of adders and partial sums introduced higher latencies to tally up the partial sums, and increased resources because of the array that holds the partial sums, the adders for each partial sum, and the tree adder used for the final summation. Excessive use of BRAM is another concern, with partial sums using eight blocks for storage and the HLS tool instantiating four blocks per PE to facilitate efficient implementations.

To implement pre-fetching of right neighbors, the tool used a BRAM-based FIFO internally to buffer the values. This raised BRAM usage by either six or ten blocks per PE, depending on the PE's position, whether it had one or two halo edges, respectively. Compared with the 32 registers used in the HDL implementation regardless of PE position, this excessive use of BRAMs suggests the tool's poor efficiency with regard to complex control logic. This conclusion is in alignment with the findings of Sharafeddin et al., in their evaluation of HLS's effectiveness in accelerating MapReduce functions [37]. They observed that while improvements in either data flow or control flow designs are possible using HLS, combining both in the same design can be challenging for the tool and is likely to result in degraded performance, particularly when floating-point operations are involved.

As shown in table 4, this inflated use of resources allowed the design to accommodate merely four PEs in the HLS implementation, as opposed to eight PEs in the HDL version. Although this lower PE count led to smaller resource consumption in total, the HLS implementation suffered from a slower clock, suggesting a poor underlying RTL description. In addition, the HLS version endured long iteration delays given its inability to pipeline the main loop, leading to low PE utilization. With a combination of faster clock, larger PE count, and higher PE utilization, the HDL implementation performed over 47,000 iterations per second at 2.1 watts, compared with the HLS version with only 33,000 iterations per second and merely 0.1 watts less power. As such, the HDL version achieved more than double the *cell updates per watt* compared with HLS. Results in both implementations were bit-wise identical to that of a reference software solution developed in MATLAB.

TABLE IV. PERFORMANCE RESULTS

Version	*Usage %*				*Freq.* *MHz*	*PEs*	*PE Utilization*	*Million Cell Updates / Watt*
	LUT	*FF*	*DSP*	*BRAM*				
HDL	89	70	96	95	200	8	98%	~ 740
HLS	73	48	89	68	166	4	82%	~ 270

C. Observations

Some studies [33-34] have found Xilinx's Vivado HLS to require more hardware knowledge than other HLS tools, which suggests that it offers higher control over the generated RTL design. Yet, the difficulty in describing a parallel system using a sequential language is inescapable. Not only does it not come naturally, but it also introduces design limitations that prevent optimal performance.

In theory, pragmas facilitate parallel design descriptions and allow for flexible implementations and improved performance. But in reality, they are of limited benefit given their many use-case restrictions. For example, the dataflow directive could be used to facilitate a loop pipeline, reducing runtime by allowing timesteps to run back-to-back. However, because halo arrays are written to and from by more than one PE function call, the main loop does not conform to the canonical form required by the tool to infer pipelined behavior. Even if the functions were transformed into the canonical form, pipelining would not be allowed because the PE functions return their running difference sum, which is not permissible when using the directive unless it is a top-level function.

Without explicit means to define parallelism, the tool must adopt a conservative approach to safely detect parallel operations without jeopardizing correctness. Often, the tool falsely detects data dependencies and assumes serial execution, without responding to pragmas or offering other tips for improving performance as it cannot recognize the parallelism to begin with. This inability to specify parallelism explicitly, and failure to detect it implicitly, may come at a high cost. For instance, the tool's inability to run PEs in parallel when reading from the same submatrix led to the introduction of halo arrays, a steep price in terms of memory usage.

Additionally, the reliance on the tool to infer desired behavior may produce an unexpected outcome that is difficult to rationalize. For instance, before the main loop, halos load in parallel if there are no PE calls in the main loop, and PEs execute in parallel if there are no halo loading calls. However, if both calls are placed, halo loading is serialized, and yet, dedicated resources are allocated for each call. The serialization of halo loading is puzzling because all halos should and do complete loading before running the PEs, and their execution is independent of each other. If the tool rightfully serialized the calls, then how was it able to run them in parallel when PE calls were not present? And if it wrongfully did so, then why would it duplicate resources for parallel execution, and then run the calls sequentially? Although there might be a good explanation that escapes the developer, it is not immediately obvious, and it takes time and effort to experiment with various scenarios to elicit behavioral patterns that could help rationalize such behavior. This point bears emphasis, as hardware engineers adopting HLS as a time-saving tool may end up spending a significant amount of time, up to several days if not weeks, trying to understand the tool's refusal to behave as desired. Developers may ultimately succeed in conjuring up a maneuver that steers the tool into desired behavior, but this is not always a trivial task, and it is likely to render the code harder to understand and maintain.

Having said that, the tool seems to manage resources quite efficiently when pragmas, not manual manipulation, are used to achieve desired behavior. For example, inlining halo loading functions used fewer resources than hand-crafted functions, designed to reuse idle buffers to cut memory usage in half.

V. Recommended Practices

Development time using Vivado HLS can be considerably reduced with improved understanding of the tool's limitations, and recognition of common tasks and usage scenarios that are affected by these limitations. To this end, following are some design techniques that proved of repeated benefit.

Dynamic Arguments: HLS prohibits dynamic assignment of pointers because hardware designs often have several memory spaces, and the tool must know which memory space the function intends to access so it can place wire connections accordingly. One way to pass function arguments dynamically is to introduce a layer of indirection by statically defining all possible function calls and hardcoding the arguments for each, then dynamically selecting the correct call using a conditional statement. An example is illustrated in the pseudo-code listing in figure 5.

Parallel Execution: To ensure correct execution, HLS prevents functions accessing the same array from running in parallel. With the absence of support for constant parameters, this includes read-only functions as there is no way to classify them as such. Arrays must be split or duplicated, or functions must be merged to enable parallel execution.

Memory Over-allocation: There are two ways to synthesize large arrays into a chain of multiple BRAM blocks. First is width expansion, which distributes word bits across multiple BRAM blocks, activating several blocks simultaneously with each memory access which uses up power. Second is depth expansion, which stores full words in each block and employs a multiplexer to activate one block per memory access at the expense of extra logic and delay. To enable high-performing designs, HLS seems to favor width expansion which can lead to prohibitive BRAM usage, particularly when words are very wide and blocks are not fully utilized. As such, splitting large arrays into smaller ones that fully utilize BRAM blocks can drastically reduce memory usage.

Resource Reuse: The most efficient way to serialize execution and reuse logic is by extracting common logic in an inlined sub-function. Conversely, turning the inlining off duplicates resources and enables parallel execution, although this doesn't always succeed. A sure-but-less-convenient way to ensure parallel execution is to replicate the functions and rename them differently. Granted, this approach makes the code harder to maintain, but in some cases it may be inevitable.

FP Accumulators: With the absence of floating-point accumulation support, the most efficient way to implement accumulation is using partial sums and tree adders, as described in the pseudo-code listing in figure 6.

VI. Conclusions

In this paper, we evaluated the efficiency of Xilinx's Vivado HLS tool compared to the traditional HDL approach in the context of stencil computations, a class of applications that are difficult to parallelize given their low arithmetic intensity and high memory bandwidth requirements. For each approach, we described a parallel, highly-optimized FPGA design solving Laplace's equation, using a five-point stencil and a Jacobi iteration scheme. Analyzing the results, we found the HLS implementation lacking in terms of performance and resource usage, with performance-per-power reaching merely 36.4% that of the HDL implementation.

While the Vivado HLS tool demonstrated excellent reuse of logic resources with minimal overhead, it underdelivered with respect to complex control and memory allocation. Therefore, it may be concluded that an ideal approach would combine the use of HLS code for computationally intensive portions, with hand-crafted HDL implementations for logic control and memory management. However, when memory management and control logic are the most complex and time-consuming portions of the application's implementation process, as is the case in this study, a hybrid method might not add much value compared with a pure HDL design. Furthermore, in such method, the HDL modules would be treated as a black box within the HLS design, hindering the tool's efforts to reuse resource consumed by the HDL modules.

Either way, HLS introduces its own set of challenges as it shifts the developer's attention and effort from design and testing to pondering the tool's unexpected behavior, speculating why the tool is allocating so many resources or incurring so much latency. As such, developers are advised to set their expectations accordingly, and incorporate ample time for optimizations in their project planning.

ACKNOWLEDGMENT

The first author gratefully acknowledges the generous support of the King Abdulaziz City for Science and Technology (KACST), Riyadh, Saudi Arabia, for the conduct of this work, and thanks Dr. Abdulaziz Alhussien for his valuable feedback on the first draft of this manuscript.

REFERENCES

[1] "Microsoft Azure Virtual Machines," https://azure.microsoft.com/en-us/services/virtual-machines/.

[2] "GPUs on Compute Engine," https://cloud.google.com/compute/docs/gpus.

[3] "Amazon EC2 F1 instances," https://aws.amazon.com/ec2/instance-types.

[4] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in 2011 38th Annual International Symposium on Computer Architecture (ISCA), 2011, pp. 365–376.

[5] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," Commun. ACM, vol. 52, no. 4, pp. 65–76, Apr. 2009.

[6] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," Des. Autom. Emb. Sys., vol. 16, pp. 31–51, Sep. 2012.

[7] A. Dubey, "Stencils in Scientific Computations," in Proceedings of the Second Workshop on Optimizing Stencil Computations, 2014, p. 57.

[8] S. A. Edwards, "The Challenges of Synthesizing Hardware from C-Like Languages," IEEE Des. Test Comput., vol. 23, no. 5, pp. 375–386, 2006.

[9] G. Martin and G. Smith, "High-Level Synthesis: Past, Present, and Future," IEEE Des. Test Comput., vol. 26, no. 4, pp. 18–25, 2009.

[10] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," IEEE Trans. Comput. Des. Integr. Circuits Syst., vol. 30, no. 4, pp. 473–491, 2011.

[11] R. Nane et al., "A Survey and Evaluation of FPGA High-Level Synthesis Tools," IEEE Trans. Comput. Des. Integr. Circuits Syst., vol. 35, no. 10, pp. 1591–1604, 2016.

[12] Xilinx, "Vivado Design Suite User Guide, High-Level Synthesis." 2018.

[13] Intel Corporation, "Intel High Level Synthesis Compiler Pro Edition User Guide." 2019.

[14] "Catapult," http://www.mentor.com/hls-lp/catapult-high-level-synthesis.

[15] K. Rupnow, Y. Liang, Y. Li, D. Min, M. Do, and D. Chen, "High level synthesis of stereo matching: Productivity, performance, and software constraints," in 2011 International Conference on Field-Programmable Technology, 2011, pp. 1–8.

[16] Y. Liang, K. Rupnow, Y. Li, D. Min, M. Do, and D. Chen, "High-Level Synthesis: Productivity, Performance, and Software Constraints," J. Electr. Comput. Eng., vol. 2012, Feb. 2012.

[17] B. E. Conn, "Exploring High Level Synthesis to Improve the Design of Turbo Code Error Correction in a Software Defined Radio Context," Thesis. Rochester Institute of Technology, 2018.

[18] J. Choi, R. Lian, Z. Li, A. Canis, and J. Anderson, "Accelerating Memcached on AWS Cloud FPGAs," in Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, 2018.

[19] Y. Afsharnejad, A.-A. Yassine, O. Ragheb, P. Chow, and V. Betz, HLS-based FPGA Acceleration of Light Propagation Simulation in Turbid Media. 2018.

[20] K. Georgopoulos et al., "An evaluation of vivado HLS for efficient system design," in 2016 International Symposium ELMAR, 2016, pp. 195–199.

[21] D. Bailey, The advantages and limitations of high level synthesis for FPGA based image processing. 2015.

[22] K. Karras, M. Blott, and K. Vissers, "High-Level Synthesis Case Study: Implementation of a Memcached Server," 1st Int. Work. FPGAs Softw. Program. (FSP 2014), Sept. 1, 2014, Munich, Ger., Aug. 2014.

[23] T. Marc-André, Two FPGA Case Studies Comparing High Level Synthesis and Manual HDL for HEP applications. 2018.

[24] M. D. Zwagerman, "High Level Synthesis , a Use Case Comparison with Hardware Description Language," Thesis. Grand Valley State University, 2015.

[25] S. Luthra, "High Level Synthesis and Evaluation of an Automotive RADAR Signal Processing algorithm for FPGAs," Electronic Theses and Dissertations, 7274, 2017.

[26] E. Homsirikamol and K. Gaj, "Can high-level synthesis compete against a hand-written code in the cryptographic domain? A case study," in 2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14), 2014, pp. 1–8.

[27] F. Winterstein, S. Bayliss, and G. A. Constantinides, "High-level synthesis of dynamic data structures: A case study using Vivado HLS," in 2013 International Conference on Field-Programmable Technology (FPT), 2013, pp. 362–365.

[28] D. O'Loughlin, A. Coffey, F. Callaly, D. Lyons, and F. Morgan, "Xilinx Vivado High Level Synthesis: Case studies," in 25th IET Irish Signals & Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014), 2014, pp. 352–356.

[29] D. J. Warne, N. A. Kelson, and R. F. Hayward, "Comparison of High Level FPGA Hardware Design for Solving Tri-diagonal Linear Systems," Procedia Comput. Sci., vol. 29, pp. 95–101, 2014.

[30] Z. Zhao and J. Hoe, "Using Vivado-HLS for Structural Design: a NoC Case Study," Carnegie Mellon University, ECE Department, Pittsburgh, PA USA, Tech. Rep. 27-Oct-2017.

[31] K. Rupnow, Y. Liang, Y. Li, and D. Chen, "A study of high-level synthesis: Promises and challenges," in 2011 9th IEEE International Conference on ASIC, 2011, pp. 1102–1105.

[32] T. Cenova, "Exploring HLS Coding Techniques to Achieve Desired Turbo Decoder Architectures," Thesis. Rochester Institute of Technology, 2019.

[33] M. B. Shaodong Qin, "A Comparison of High-Level Design Tools for SoC-FPGA on Disparity Map Calculation Example," in Presented at Second International Workshop on FPGAs for Software Programmers (FSP 2015), 2015.

[34] G. Inggs, S. Fleming, D. Thomas, and W. Luk, "Is high level synthesis ready for business? A computational finance case study," in 2014 International Conference on Field-Programmable Technology (FPT), 2014, pp. 12–19.

[35] Xilinx, “Vivado HLS - How to achieve PIPELINE II=1 of floating point accumulation?,” Support Article # 62859, 2914. [Online]. Available: https://www.xilinx.com/support/answers/62859.html.

[36] Xilinx, “7 Series FPGAs Data Sheet: Overview.” 2018.

[37] M. Sharafeddin, M. Saghir, H. Akkary, H. Artail, and H. Hajj, “On the effectiveness of accelerating MapReduce functions using the Xilinx Vivado HLS tool,” Int. J. High Perform. Syst. Archit., vol. 6, p. 1, Jan. 2016.

[38] J. Hrica, “Floating-Point Design with Vivado HLS.” Xilinx, 2012.