High Level Synthesis: Where Are We? A Case Study on Matrix Multiplication

Sam Skalicky, Christopher Wood, Marcin Łukowiak, Matthew Ryan
Rochester Institute of Technology, Rochester, NY
{sxs5464,caw4567,mxleec,mvr4997}@rit.edu

***Abstract*—One of the pitfalls of FPGA design is the relatively long implementation time when compared to alternative architectures, such as CPU, GPU or DSP. This time can be greatly reduced however by using tools that can generate hardware systems in the form of a hardware description language (HDL) from high-level languages such as C, C++, or Python. Such implementations can be optimized by applying special directives that focus the high-level synthesis (HLS) effort on particular objectives, such as performance, area, throughput, or power consumption. In this paper we examine the benefits of this approach by comparing the performance and design times of HLS generated systems versus custom systems for matrix multiplication. We investigate matrix multiplication using a standard algorithm, Strassen algorithm, and a sparse algorithm to provide a comprehensive analysis of the capabilities and usability of the Xilinx Vivado HLS tool. In our experience, a hardware-oriented electrical engineering student can achieve up to 61% of the performance of custom designs with $1/3$ the effort, thus enabling faster hardware acceleration of many compute-bound algorithms.**

I. Introduction

Compute intensive applications can be time consuming to the point that implementation in a traditional CPU becomes impractical. In such cases, alternative implementations, such as hardware accelerators, need to be considered. FPGAs offer a highly parallelizable platform that makes it an ideal candidate for running such accelerators. However, FPGAs have a long design time driven by factors such as choosing a particular architecture, optimizing the standard design as needed for the specific application, while simultaneously handling constraints such as hardware area and available memory bandwidth. Recently, new tools have been developed that aim to bridge the design time gap between hardware and software. This design methodology, called high-level synthesis (HLS), generates the hardware description language (HDL) from high-level programming language source code, such as C, C++, or Python [1][2][3]. Existing HLS tools are now sophisticated enough at compiling high-level source code down to HDL models that the resulting applications can run an order of magnitude faster [4]. Furthermore, they provide designers with the ability to optimize for more than just performance; power consumption or resource utilization can also be optimization goal during the synthesis process by specifying flags (e.g. compiler pragmas) in the high-level source code or changing options of the compiler.

To date, an area yet to be fully explored is the difference in performance, design time, and resource utilization between HLS and custom FPGA implementations. In order to obtain realistic results, it is necessary to choose a medium for comparison between the techniques. Linear algebra computations constitute core of many compute intensive applications and could benefit from FPGA acceleration. Among these, matrix-matrix multiplication stands out as an ideal candidate for examination due to its exploitable parallelism and variety of different calculation algorithms. The inherent parallelism within the computation gives incentive to implement it on an FPGA rather than on a CPU. Analyzing the architectures generated from different computational algorithms with various optimization goals provides additional information on how the HLS tools compile different types of high-level implementations.

In this work, a cost-benefit analysis of utilizing HLS tools for development of FPGA accelerators versus a custom design is investigated. This is accomplished through the implementation of three distinct matrix multiplication algorithms: the standard algorithm, Strassen algorithm, and a sparse matrix algorithm. For each of these algorithms an existing, well researched custom design is compared to a generated and optimized design using the Vivado HLS tool from Xilinx. Each algorithm was implemented by a hardware-oriented electrical engineering student with a general familiarity of HLS tools. We present resource utilization and performance results for each algorithm using both HLS tools and custom designs, and follow up with tabulated design times for each as a means of cost-benefit analysis to compare the utility gained when using HLS tools versus designing custom architectures.

II. Related Work

A large variety of HLS tools now exist. These range from commercial products such as Vivado HLS from Xilinx to open source tools developed from academic research initiatives, such as LegUp [5], among others. In [6], Economakos *et al.* studied aiding automation experts with no FPGA experience to design programmable logic using the Catapult C HLS tool to prepare programmable logic controllers (PLCs) for use in factory automation and industrial control. The authors experimented with several optimizations available with Catapult C and compared the resulting design latency, throughput, and system resource utilization (e.g. BRAMs, LUTS, etc.) for a variety of different evaluation boards. Their results enabled them to conclude that, while HLS tools are often good substitutes for custom FPGA solutions when exploring the potential design space and available optimizations, it is often necessary to support design decisions with the knowledge of an experienced hardware engineer.

Denolf *et al.* utilized the Synfora PICO Extreme HLS tool to investigate the usefulness of HLS in lowering design time and increasing performance for vision processing kernels [7]. They found that the HLS implementations had similar resource

978-1-4799-2079-2/13/$31.00 ©2013 IEEE

consumption to their handwritten HDL counterparts with a much shorter design time. Furthermore, the authors note the efficacy of using PICO to experiment with various throughput, operating frequency, and resource consumption tradeoffs for different designs as a result of high-level HLS optimizations.

The Handel C-language was used by Loo *et al.* [1] to develop embedded systems in environments that incorporate FPGA based co-processor logic. Using the Handel C language they were able to develop designs that were comparable in performance to handwritten designs but larger in size. However, there was also evidence that hardware designers would need to be retrained if they were to use the language. Impulse C is another popular HLS tool for prototyping FPGA designs of existing algorithms written in C. Xu *et al.* [4] explored the cost-gain benefits of using Impulse C to create hardware accelerators for image reconstruction algorithms. Their experimental data indicated that Impulse C was sophisticated enough generate HDL that paralleled the custom HDL model in performance (throughput) by achieving an approximate 155x speedup over the original software implementation with very minimal design effort, though the amount of effort is not formally quantified.

Similar to this work, Cong *et al.* [8] studied the tradeoffs between implementation efficiency and designer productivity using AutoESL's AutoPilot HLS tool, now the Xilinx Vivado suite, in the context of a video DSP application and wireless encoding algorithm. They found that the cost per design throughput between an AutoESL-generated FPGA design for an optimal flow algorithm implementation was 30x superior to an analogous pure-software DSP design. In a similar experiment they observed that the chip resource utilization of an AutoESL-generated FPGA design for the same optical flow algorithm implementation subsumed the resource utilization of a hand-written RTL design. The design time for this particular application is not discussed. They do, however, study the time required to design a sphere decoder for a multi-input multi-output wireless communication system between two RTL and AutoESL experts (i.e. engineers fluent in the respective implementation technology). The engineer design time, which was extracted from source code revision history information and work logs, indicated that the AutoESL expert surpassed the RTL expert in development time and design resource (e.g. LUTs, registers, DSP cores, etc) utilization. In fact, for a relative comparison, the design time of the AutoESL expert was 9% less than the the RTL engineer, which was the *smallest* percent difference out of all metrics calculated, e.g. the design from the AutoESL expert required 31% less registers than the RTL expert's design.

MyHDL is another emerging HLS tool that breaks away from the typical C-based languages available on the market. Written in Python, MyHDL benefits from its ease of development; the Python language barrier is much less steep than that of C or C++. Inggs *et al.* [9] report that they were able to generate a custom discrete Fourier transform (DFT) core with 14% greater throughput than the FFT core provided by Xilinx using MyHDL. The design effort, which is a major contributing factor to the selection of HLS tools, was not discussed in this work.

Monson *et al.* [10] compared the performance of CPU and FPGA based implementations of a complex optical-flow algorithm. For the FPGA based implementation, the Vivado HDL synthesis tools were utilized. Using these tools, the designers were able to develop an implementation of the algorithm with comparable performance to the CPU implementation that operated at a fraction of the energy cost. The authors came to several important conclusions, which we have already seen with the previously mentioned tools, regarding the use of the Vivado HLS tools: (1) little modification is necessary to prepare existing C language designs for conversion using the HLS tools, (2) it is possible quickly optimize a design for different goals, and (3) it is easy to compare different versions of the algorithm in C and determine resource consumption in performance.

This work improves upon previous work by analyzing the practicality of utilizing HLS tools and their effectiveness. In addition to directly comparing the performance of several different HLS implementations to custom implementations, a significant portion of this work is devoted to quantifying the amount of effort required to develop HLS designs and custom designs for the same computation.

III. Matrix Multiplication Algorithms

Matrix multiplication is a fundamental operation of linear algebra. As such, many algorithms have been created to optimize and further improve the performance for specific objectives. Compared to the standard algorithm, shown in Algorithm 1, the Strassen algorithm attempts to reduce the number of multiplications in exchange for more additions. A sparse algorithm, on the other hand, attempts to only handle the non-zero elements and thus remove any multiplications or additions with zero to improve performance.

Algorithm 1 Standard Matrix Multiply $C = A \times B$

```
for i = 0 → rows(A) do                    ▷ Rows
    for j = 0 → cols(B) do                ▷ Cols
        for k = 0 → rows(B) do            ▷ Product
            C_{i,j} = C_{i,j} + A_{i,k} × B_{k,j}    ▷ Calculation
        end for
    end for
end for
```

A. Strassen Algorithm

The Strassen algorithm operates on 2×2 matrices and is designed to reduce the number of multiplications operations at the expense of requiring more additions as shown in Algorithm 2. This algorithm requires 7 multiplications and 18 additions to complete a 2×2 matrix. In contrast, the standard algorithm would require $N^3 = 8$ multiplications and additions. So effectively 1 multiplication is exchanged for 10 additions. To handle the larger matrix sizes, a block based approach is used. We assume the size of input matrices are multiples of 2. The algorithm below has been designed to operate on 4×4 matrices. This size was chosen since it used a reasonable amount of resources on the FPGA while still providing high performance.

Algorithm 2 Strassen Matrix Multiply $C = A \times B$

```
for i = 0 → 1) do                                ▷ Outer
    for j = 0 → 1 do                             ▷ Mid
        for k = 0 → 1 do                         ▷ Inner
            A' = A_{2i:2i+1,k:k+1}
            B' = B_{2k:2k+1,j:j+1}

            S1 = (A'11 + A'22) × (B'11 + B'22)
            S2 = (A'21 + A'22) × B'11
            S3 = A'11 × (B12 − B'22)
            S4 = A'22 × (B21 − B'12)
            S5 = (A'11 + A'12) × B'22
            S6 = (A'21 − A'11) × (B'11 + B'12)
            S7 = (A'12 − A'22) × (B'21 + B'22)

            C'11 = C'11 + S1 + S4 − S5 + S7
            C'12 = C'12 + S3 + S5
            C'21 = C'21 + S2 + S4
            C'22 = C'22 + S1 − S2 + S3 + S6
        end for
        C_{2i:2i+1,2j:2j+1} = C_{2i:2i+1,2j:2j+1} + C'
    end for
end for
```

$$\begin{bmatrix} 0 & s_{1,2} & 0 & 0 \\ 0 & s_{2,2} & s_{2,3} & 0 \\ s_{3,1} & 0 & 0 & s_{3,4} \\ s_{4,1} & 0 & 0 & 0 \end{bmatrix}$$

val	$s_{1,2}$	$s_{2,2}$	$s_{2,3}$	$s_{3,1}$	$s_{3,4}$	$s_{4,1}$
col	1	1	2	0	3	1
row	0	1	3	5		

val	$s_{3,1}$	$s_{4,1}$	$s_{1,2}$	$s_{2,2}$	$s_{2,3}$	$s_{3,4}$
row	2	3	0	1	1	2
col	0	2	4	5		

Fig. 1: Sparse matrix (top) in compressed sparse row (CSR) (mid) and compressed sparse column (CSC) formats (bottom).

B. Sparse Algorithm

When matrices consist largely of zero value elements it is possible to compact the sparse matrix into a form in which its sparsity can be easily exploited. In this work, sparse matrices are stored in the compressed sparse row (CSR) and compressed sparse column (CSC) formats. A sparse matrix displayed in CSR format is comprised of three vectors as shown in Figure 1. The first vector, val, consists of the values of the non-zero elements of the sparse matrix. The second, col, contains the column index of each of the non-zero elements of the sparse matrix. Finally, row stores the index in val of the first non-zero element of row i. Conversely, CSC format stores the row index of each non-zero element in the row vector and the index of the first non-zero element of each column in the col vector. The sparse algorithm operates by multiplying each non-zero element in a row of A with every non-zero element in a column of B and then repeats that process for every row and column of the matrix as shown in Algorithm 3. For this algorithm, we assume that the matrix A is stored in CSR format, and matrix B is stored in CSC format.

Algorithm 3 Sparse Matrix Multiply $C = A \times B$

```
for i = 0 → rows(A) do                                ▷ Top
    for j = row_A[i] → row_A[i+1] do                  ▷ Mid1
        for k = 0 → cols(B) do                        ▷ Mid2
            for m = col_B[k] → col_B[k+1] do          ▷ Bottom
                if col_A[j] == row_B[m] then
                    C_{i,k} = C_{i,k} + val_A[j] × val_B[m]
                end if
            end for
        end for
    end for
end for
```

IV. An Overview of High Level Synthesis Techniques

The overall goal of high level synthesis is to take an algorithm specified in a high level language, extract the control logic (i.e. loops, conditionals, etc.) and operations (add, multiply, divide, etc.) contained therein, and generate an equivalent hardware accelerator specified in the designer's choice of HDL. HLS tools are composed of multiple stages that iteratively modify and then translate the input source code towards an equivalent HDL model. Normally, the front-end compiler tasks consist of a pre-processing and analysis stage, initial optimization stage, loop unrolling stage, and secondary optimization stage responsible for applying pipeline directives. The result of this is modified source code and a set of internal compiler information necessary to generate the equivalent HDL.

Loop unrolling will fully (or partially) unfold an iteration block into a sequence of *finite* statements that can be later joined together and executed in parallel. It is important to note that non-deterministic loops cannot be unrolled, as the number of iterations cannot be determined at compile-time. Given the flexibility in how much a loop is unrolled, the user has direct control over the area consumption of each unrolled block of code. Unrolling an outer loop completely unrolls any inner loops. Even though the control logic may vanish as loops are unfolded and their internal statements are synthesized into parallel statements, the amount of FPGA resources increases linearly with respect to the number of iterations of the loop. Similarly, pipelining is an optimization that inserts registers between combinational logic for a sequence of code blocks to enable higher clock speeds for the resulting design. This technique may be applied to nested loops with the caveat that doing so will completely unroll the inner loop to a single piece of combinational logic with corresponding pipeline registers. This is particularly useful for applications that stream a great deal of data through the accelerator. Furthermore, since the pipeline optimization can be selectively applied to any loop, nested or otherwise, the user is given more control over the consumption of the FPGA resources.

Control over the memory interface is not as liberal in modern HLS tools. For example, Vivado HLS provides an extensive capability for the user to define the memory interface, however it lacks the complete control that writing custom HDL provides.

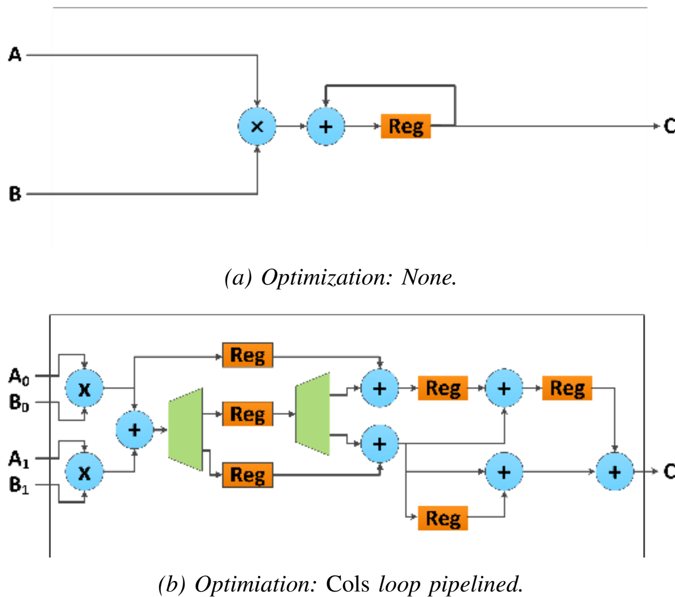
V. RESULTS

The FPGA device chosen for this research was the Xilinx Virtex 6-475T device due to its large number of DSP slices that would be able to accommodate all of the multipliers needed for matrix-matrix multiplication. The three algorithms discussed in Section III were implemented using the Vivado HLS tool. Additionally, a well researched custom design from previous work was also implemented for comparison [11][12][13]. To evaluate the performance improvement of the HLS generated designs over software, each algorithm was implemented in C++ and executed in an Intel Core i7 Sandy Bridge 3.4GHz processor. Each design operated on integers for simplicity. The adders and multipliers were implemented as distinct functional units so that they could be swapped out for any other type such as floating point. For each algorithm, we will discuss the architecture that was generated from the HLS tool and compare its performance to the software implementation. In Section VI we compare the design times for the HLS and custom designs.

For all of the HLS generated designs, each matrix input into the compute logic was a standard BRAM interface. The matrix inputs and outputs for each design were then manually connected to memory interfaces for off-chip DDR storage. Then, each design was synthesized and post place and route statistics such as clock speed and resource utilization were collected. A number of optimizations were applied to each algorithm. For the sake of brevity, we only present two architecture diagrams for the standard and Strassen algorithms and just one for the sparse algorithm.

A. Standard Algorithm

The architecture diagram for the HLS generated design for the standard algorithm operating on a 8x8 matrix with no optimizations is shown in Figure 2a. Table I shows the percentage of resources utilized in the FPGA and the speedup



(a) Optimization: None.

(b) Optimiation: Cols *loop pipelined.*

Fig. 2: Architectures generated from HLS tool for standard algorithm.

TABLE I: HLS and custom results for the standard algorithm. The utilization is shown as a percentage of the total, and the speedup is that of the HLS optimized design versus software.

Optimization	**Resources** [Total]			Speedup
	LUTs [297760]	**FFs** [595520]	**DSPs** [2016]	
None	1%	1%	1%	0.2x
Product Pipelined	1%	1%	1%	0.6x
Product Unrolled	1%	1%	1%	3.2x
Cols Pipelined	1%	1%	1%	3.2x
Cols Unrolled	1%	1%	5%	1.5x
Rows Pipelined	1%	1%	2%	2.7x
Rows Unrolled - 2	3%	2%	27%	3.1x
Rows Unrolled - 4	7%	5%	75%	4.8x
Custom	1%	1%	13%	50.6x

that was achieved over the software implementation for a variety of optimizations applied in the HLS tool as well as for the custom implementation.

The performance of the HLS tool over the software was generally a few times faster. Overall, the pipelining optimizations generally performed better than the unrolling. Especially, when moving from unrolling an inner loop to pipelining its outer loop since pipelining a loop unrolls all inner loops. Initially, the performance of the first two HLS designs didnt achieve a speedup since they ran at a lower clock speed than on the CPU in software yet only had a single multiplier and divider. Once the designs began to become parallelized extra performance was achieved. Unrolling the $Product$ loop only produced two multipliers sharing a single adder. However, we expected that 8 multipliers and adders would have been generated. Due to the design of the algorithms used within the HLS tool only two multipliers with a shared adder were generated. Pipelining the $Cols$ loop also unrolled the $Product$ loop, but added registers between the various components as shown in Figure 3b. After unrolling the $Cols$ loop however, 8 of the two multipliers with shared adders were generated, as we originally expected. The best performance of the HLS designs was achieved by unrolling the outer *Rows* loop by a factor of 4 which consumed 2x the resources of unrolling by a factor of 2, yet was only 1.7x faster. Perhaps the most notable discrepancy between this design and the custom was the fact that the HLS design only reads 2 elements from each input matrix while the ping-pong buffers used in the custom design were capable of 8 simultaneous reads. Compared to the custom design, the fastest HLS design used almost 8x the resources but only achieved 1/10th the performance.

B. Strassen Algorithm

The Strassen algorithm from Algorithm 2 was analyzed by the HLS tool and a design generated with no optimizations as shown in Figure 3a. The basic element (BE) shown in the architecture diagram represents the adders and multipliers required for the calculations within the *Inner* loop of the algorithm for brevity. Table II shows the percentage of resources utilized in the FPGA and the speedup that was achieved over the software implementation.

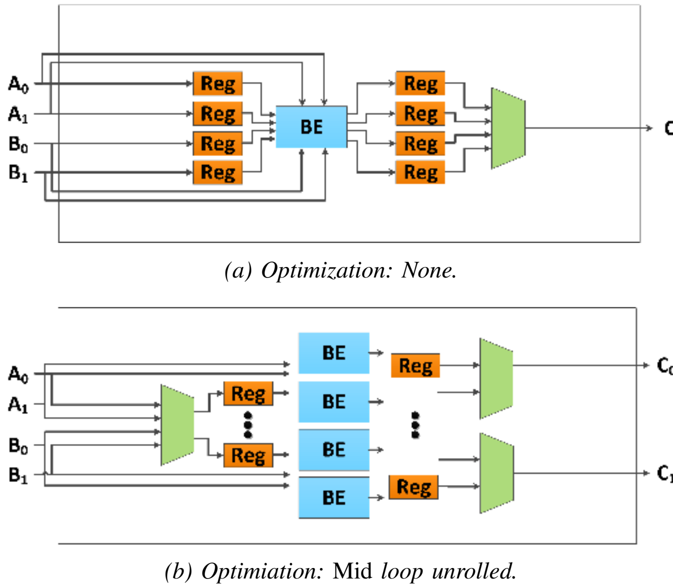
The custom Strassen design consisted of 4 2x2 matrix multipliers. Disregarding the *Outer* loop, the trip count of

the *Inner* loop is equivalent to 4. Thus it is easy to see that when these two loops are fully unrolled the design should be equivalent (at least in terms of hardware components) to the custom design. However, the HLS design has the advantage of being able to unroll the *Outer* loop.

This algorithm wasnt able to achieve the same level of performance improvement as the standard algorithm. Overall, the HLS designs achieved a few times speedup over the software implementation with very little extra effort. Just as with the standard algorithm, once this algorithm's loops were unrolled a speedup was achieved. As mentioned above, the BE computes the matrix multiplication for a 2x2 matrices. The loop bounds were set for a 8x8 matrix, and so unrolling the *Inner* loop produced two BEs and unrolling the *Mid* loop produced four BEs as shown in Figure 3b. Additionally, since in this design more results are being calculated in parallel, the HLS tool generated a design with a second output port to allow for this increased bandwidth to be written out to memory. Pipelining the *Outer* loop just added pipelining to the unrolled *Mid* and *Inner* loops. This version is architecturally equivalent to the custom design, yet it only achieved a speedup of 2.0x compared to the custom's 4.8x speedup. Further unrolling produced a larger speedup at the expense of using more resources than the custom design.

C. Sparse Algorithm

The sparse algorithm from Algorithm 3 is very different from the other two algorithms in that the bounds on the loops are non-deterministic. The bounds depend on the sparsity and distribution of non-zero elements in the matrix. However, the *Top* loop is bounded to the number of rows in the matrix, and so this loop is able to be optimized in the HLS tool. The design with no optimizations is shown in Figure 4. The generated design has at its core a multiplier and an adder just like the standard algorithm but with extra inputs for the rows and columns to determine which elements to operate on.



(a) Optimization: None.

(b) Optimiation: Mid *loop unrolled.*

Fig. 3: Architectures generated from HLS tool for Strassen algorithm.

TABLE II: HLS and custom results for the Strassen algorithm. The utilization is shown as a percentage of the total, and the speedup is that of the HLS optimized design versus software.

Optimization	**Resources** [Total]			**Speedup**
	LUTs [297760]	**FFs** [595520]	**DSPs** [2016]	
None	1%	1%	1%	0.4x
Inner Pipelined	1%	1%	1%	0.5x
Inner Unrolled	1%	1%	2%	1.0x
Mid Pipelined	1%	1%	2%	2.0x
Mid Unrolled	1%	1%	4%	1.6x
Outer Pipelined	1%	1%	4%	2.0x
Outer Unrolled	1%	1%	8%	2.9x
Custom	1%	1%	6%	4.8x

Unfortunately, due to the complexity of this design the HLS tool was not able to parallelize the algorithm in a beneficial way. It generated so much control logic that the achievable clock speeds were so low the performance actually decreased as shown in Table III. In fact, for the unrolling optimization of the *Top* loop the design used 12% of all the LUTs in the device, much more than all of the other designs. However, the design with no optimizations did achieve a 1.2x speedup over the software for matrices with 30% density. Overall, as the matrix density decreases these implementations becomes less efficient. The custom design was orders of magnitude faster than the HLS designs and the 8 processing element (PE) design was able to parallelize the architecture and use more DSPs for higher performance. Given these results, HLS tools are unsuitable for algorithms with this type of non-deterministic loop bounds.

VI. HLS Utility Through Design Time and Performance

The design time and performance results of implementing three matrix multiplication algorithms in software, HLS, and custom were collected and plotted in Figure 5. These design times are the result of one of the authors to implement each particular design as recorded in their own work log. The HLS design times reflect the time required to achieve the best performance among all optimizations evaluated. The performance of each implementation was evaluated in terms of the number of integer operations completed per second (IOPS). The custom implementations had significantly longer design

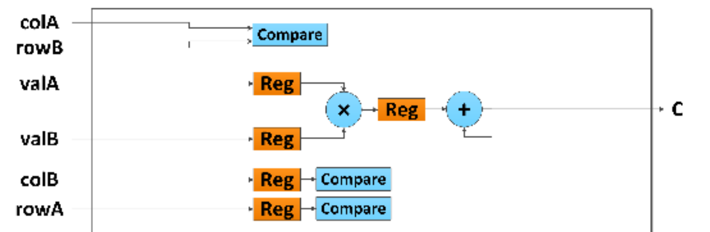


Fig. 4: Architecture generated from HLS tool for Sparse algorithm with no optimization.

TABLE III: HLS and custom results for the sparse algorithm. The utilization is shown as a percentage of the total, and the speedup is that of the HLS optimized design versus software using the same sparse matrix density.

Optimization	**Resources** [Total]			**Speedup** [Density]		
	LUTs [297760]	**FFs** [595520]	**DSPs** [2016]	[30%]	[20%]	[10%]
None	1%	1%	1%	1.2x	1.0x	0.6x
Top Pipelined	1%	1%	1%	0.9x	0.8x	0.5x
Top Unrolled	12%	4%	1%	0.5x	0.4x	0.2x
Custom PE - 4	1%	1%	1%	223.9x	140.4x	56.7x
Custom PE - 8	1%	1%	2%	240.0x	132.0x	43.8x

times when compared to their HLS counterparts. This holds true particularly for the standard and sparse implementations, where the HLS source codes were ported directly from established software implementations of the algorithms.

The Strassen HLS implementation, which was designed to mimic the developed custom Strassen design, took significantly longer to design. However, when comparing the results of the different implementations it is clear that the Strassen HLS design performs closest to its custom implementation in terms of run time. Thus the longer design time of the Strassen HLS implementation yielded a comparatively shorter run time. The Strassen custom design had a longer design time than the standard custom design due to the more complex nature of the design. While the standard custom implementation consisted almost entirely of multiply accumulators in parallel, the Strassen custom implementation required long elementary operation chains (to form the intermediary matrices) and multiplexers on the input buses in order to switch between different source matrices. The sparse custom implementation fell in between the standard and Strassen designs in terms of design complexity, as it required a systolic array architecture that could easily toggle between different numbers of processing elements. This led to the sparse custom design time being substantially longer than that of the standard algorithm, but not as long as the Strassen implementation.

The utility of a particular HLS tool can be perceived as the efficiencies it lends its user. It is well established that measuring the efficiency or productivity of an engineer or developer is an arduous, if not impossible, task due to the lack of available metrics with which to quantify such

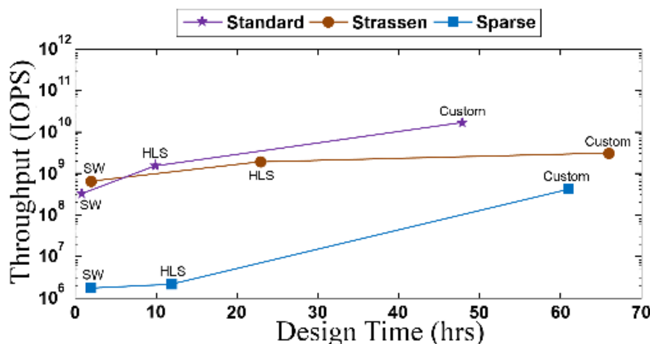


Fig. 5: Performance (in integer operations per second, IOPS) of each of the three implementations: SW, HLS, and custom and the design time (in hours) required to design each.

TABLE IV: Utility calculations for each of the algorithms.

Algorithm	$\mathbf{U_S}$ $(\frac{\mathbf{IOPS}}{\mathbf{hrs}})$	$\mathbf{U_C}$ $(\frac{\mathbf{IOPS}}{\mathbf{hrs}})$	$\mathbf{U_S/U_C}$
Standard	1.57×10^8	3.43×10^8	0.46
Strassen	8.27×10^7	4.73×10^7	1.75
Sparse	1.71×10^5	6.83×10^6	0.03

performance - lines of code, feature points, and the like to not accurately capture the context in which an application is developed. However, given the primary use case for HLS tools in optimizing the performance of a particular algorithm, we can directly measure and quantify this performance P_S with respect to any metrics (e.g. throughput, power consumption, area, etc.). Then, using the amount of time T_S invested in the design to achieve this performance, we may quantify the utility, or gain, of the tool as follows:

$$U_S = P_S/T_S.$$

We may perform a similar measurement and computation to derive the baseline performance for the custom component with which we are comparing against as

$$U_C = P_C/T_C.$$

If $U_S > U_C$ then the HLS tool proved to be more effective than designing a custom accelerator, and vice versa.

The corresponding values of U_S and U_C for the HLS and custom designs are shown in Table IV for each algorithm. The utility is represented as the number of integer operations per second (IOPS) divided by the amount of design time required in hours. From the utility calculations it is clear that software to be compiled into corresponding HDL by HLS tools needs to be worth the time investment. The Strassen algorithm, which was specifically optimized using directives in the HLS tool to parallel the custom component design, had more utility than the other algorithms, which were not modified in any way from their original form before being compiled. Therefore, if time and expertise are invested to design the software, then HLS tools can be greatly beneficial in practice. In fact, as these tools continue to be developed and improved and engineers are trained to use them, such utility gains will only increase, perhaps even enabling software engineers without a rigorous hardware background to develop high performance designs.

TABLE V: Modified utility calculations for each of the algorithms.

Algorithm	$\mathrm{U'_S}\left(\frac{\mathrm{IOPS}}{\mathrm{area\times hrs}}\right)$	$\mathrm{U'_C}\left(\frac{\mathrm{IOPS}}{\mathrm{area\times hrs}}\right)$	$\mathrm{U'_S/U'_C}$
Standard	5.38×10^8	7.09×10^9	0.08
Strassen	2.34×10^9	1.37×10^9	1.35
Sparse	1.45×10^8	1.08×10^9	0.13

In addition to our utility comparison based solely on throughput, we also considered the metric in which throughput and area were both taken into account. In particular, this new utility calculation for HLS and custom designs is defined as $U' = P/T$, where $P = (throughput/area)$. Plots of the modified metric are shown in Figure 6, with the updated utility calculations presented in Table V. Clearly, when area is considered, the performance of the HLS tools decreased due to the additional area required by the designs. For the Strassen, which had the most invested time, the utility ratio was above 1.0, meaning that the HLS design performed better in combined performance and area than the custom component. As a result we conclude that, given the current state of these tools, HLS tools should not be used for critical designs where area and throughput need to be optimized.

VII. CONCLUSION

In this paper we studied the performance improvements and design time reductions possible with the use of HLS tools in lieu of creating custom hardware accelerators for three matrix multiplication algorithms. We analyzed the architecture of each design as generated using the Xilinx Vivado HLS tool with various optimizations applied, such as pipelining and loop unrolling, and compared them against custom designs for the same algorithm. We found that the HLS tools achieved speedups of up to 5x for the standard algorithm and 3x for the Strassen algorithm compared to their software implementations. However the sparse algorithm only achieved a speedup of 1.2x since the loops in the algorithm had non-deterministic bounds.

Then, cost-benefit analyses for each algorithm empirically showed that HLS tools are effective when high throughput is an optimization goal but not so when throughput per unit of area is a goal. We found that given ample design time of a particular computational algorithm and proper use of the optimizations made available by the HLS tool in use, it can be more cost-effective to opt for an HLS solution instead of designing a custom component. We expect this effectiveness to only increase as HLS tools improve and become more usable by a larger number of engineers.

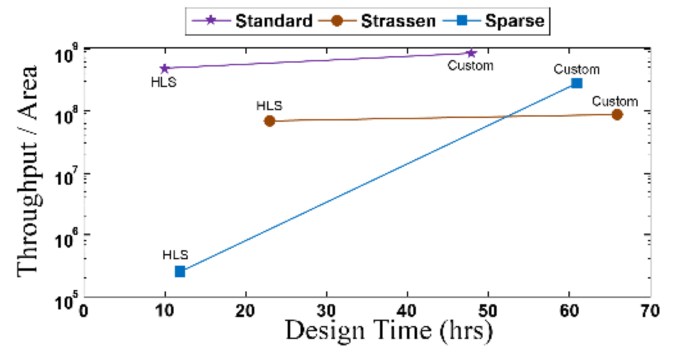


Fig. 6: Comparing the $Throughput/Area$ metric to design time for standard, Strassen, and sparse algorithms.

REFERENCES

[1] S. M. Loo, B. E. Wells, N. Freije, and J. Kulick, "Handel-C for Rapid Prototyping of VLSI Coprocessors for Real Time Systems," *Southeastern Symposium on System Theory*, Mar. 2002.

[2] D. Pellerin and S. Thibault, *Practical FPGA programming in C*. Prentice Hall Press, May 2005.

[3] J. Villar, J. Juan, M. Bellido, J. Viejo, D. Guerrero, and J. Decaluwe, "Python as a Hardware Description Language: A Case Study," *Southern Conference on Programmable Logic*, Apr. 2011.

[4] J. Xu, N. Subramanian, A. Alessio, and S. Hauck, "Impulse C vs. VHDL for Accelerating Tomographic Reconstruction," *IEEE International Sympoisum on Field-Programmable Custom Computing Machines*, May 2010.

[5] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High Level Synthesis for FPGA-based Processor/Accelerator Systems," *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Feb. 2011.

[6] C. Economakos and G. Economakos, "FPGA Implementation of PLC Programs Using Automated High-Level Synthesis Tools," *IEEE International Symposium on Industrial Electronics*, 2008.

[7] K. Denolf, S. Neuendorffer, and K. Vissers, "Using C-To-Gates To Program Streaming Image Processing Kernels Efficiently on FPGAs," *International Conference on Field Programmable Logic and Applications*, Aug. 2009.

[8] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, Dec. 2011.

[9] G. Inggs, D. Thomas, and S. Winberg, "Exploring the Latency-Resource Trade-off for the Discrete Fourier Transform on the FPGA," *International Conference on Field Programmable Logic and Applications*, Aug. 2012.

[10] J. Monson, M. Wirthlin, and B. L. Hutchings, "Implementing High-Performance, Low-Power FPGA-based Optical Flow Accelerators in C," *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, June 2013.

[11] I. Sotiropoulos and I. Papaefstathiou, "A Fast Parallel Matrix Multiplication Reconfigurable Unit Utilized In Face Recognition Systems," *International Conference on Field Programmable Logic and Applications*, Aug. 2009.

[12] I. Bravi, J. Pedro, J. Luis Lazaro, J. de las Heras, and A. Gardel, "Different Proposals to Matrix Multiplication Based on FPGAs," *IEEE International Symposium on Industrial Electronics*, June 2007.

[13] C. Yu Lin, Z. Zhang, N. Wong, and H. Kwok-Hay So, "Design Space Exploration for Sparse Matrix-Matrix Multiplication in FPGAs," Dec. 2010.