Mission Control: A Performance Metric and Analysis of Control Logic for Pipelined Architectures on FPGAs

Sam Skalicky, Sonia Lopez, Marcin Lukowiak
Rochester Institute of Technology, Rochester, NY USA
{sxs5464,slaeec,mxleec}@rit.edu

Christopher Wood
University of California Irvine, Irvine, CA USA
woodc1@uci.edu

***Abstract*—The performance of a pipelined architecture is often limited by incorrectly designed or poorly implemented control logic. Once a design is implemented and meets timing constraints, the mission is to evaluate if it is achieving optimum performance. At this stage, the number of pipelines and functional units are fixed and the amount of resources and memory bandwidth are finalized. If a design is performing suboptimally the only recourse is to improve the control logic. In this paper we present a metric to quantify the achievable performance of a design and use it to analyze performance degradation due to control logic. We analyze the control logic of existing architectures and present improvements that achieve speedups of up to 10.7x.**

I. Introduction

Custom accelerators are often designed to speed up computations when performance is critical for an application. The design of these computation-specific hardware accelerators is typically pipelined to achieve high utilization and faster clock frequencies. In such architectures, the flow of data through the pipeline stages is regulated by the control logic. The implementation's control logic is designed to follow the data dependencies between operations. An optimally designed hardware accelerator will achieve the best possible performance, which we define as the *achievable performance*, and is not easily determined. FPGAs are ideally suited for the process of implementing, testing, and modifying a design thanks to their reconfigurability. Although designs can be progressively tuned to increase performance, it is difficult to tell when the limit has been reached or how far we are from it without a quantification of the *achievable performance*.

We often know how a design operates, but not how it should be operating to achieve the best performance. Designers are very capable of formulating new modifications to improve performance. In fact, this is a standard approach to improving a design. First the designer implements an initial architecture and then evaluates its performance. If the design is found to be underperforming, improvements are formulated and the design is modified. This iterative cycle continues until the further improvements are no longer feasible. Can this process be enhanced to give designers more information about how close they are to the achievable performance?

One way to do this is to compute the minimal-length schedule that maps operations from the dataflow graph (DFG) of a computation onto the functional units in the architecture. Figure 1 shows how this *graph analysis* can be incorporated into the *standard design process*. Generating an optimal schedule provides the following information:

(1) it uniquely identifies each operation,
(2) specifies operation-to-functional unit assignments, and
(3) dictates specifically when to execute an operation.

978-1-4799-5944-0/14/$31.00 ©2014 IEEE

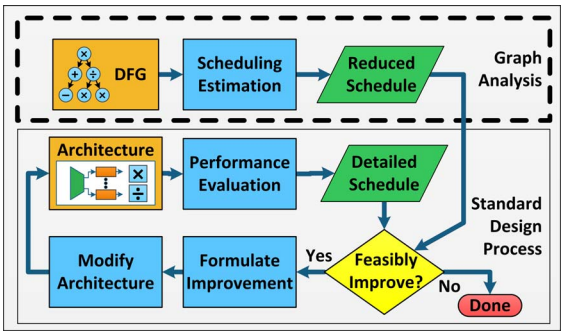


Fig. 1: High level flow of the methodology.

Therefore, optimally scheduling these operations will provide the detailed recipe for exactly how the design should operate to get the best performance. However optimal scheduling is NP-hard [1] and thus this approach is infeasible for non-trivial real world architectures and computations.

To improve this approach, we reevaluate what benefit each piece of information provides. Since the underlying objective when modifying a design is to achieve better performance, we need to know when an operation should be executed. Hence, item (3) is the most important piece of information. We relax our requirements on items (1) and (2) to more easily calculate the schedule. We define this variant as the *reduced schedule* where only the number of operations of a particular type (add, multiply, etc.) executed at each time step are specified. We exploit this relaxation to construct a polynomial-time algorithm that approximates the optimal schedule and is used to guide the designer to make better modifications. This methodology was applied to a series of benchmark computations. We show that the performance of some of these designs matches the metric exactly, validating its good design. For others —those with *more complex control flow*— the performance of the original hardware architecture differs significantly from the metric. For those, the control logic is improved achieving speedups of up to 10.7x compared to the original architectures.

This paper is organized as follows. Section II presents related work. Section III describes our graph analysis. Then benchmark results and our conclusions in Sections IV and V.

II. Related Work

One of the main components of this work is scheduling operations from a dataflow graph onto a set of pipelines, called *pipelined scheduling* [1]. These pipelined architectures are designed to operate on a stream of datasets. This scheduling problem of *rigid* and *monolithic* tasks scheduled onto architectures with multiple types of pipelines has been studied extensively [2][3][4]. Compared to the previous work, we

present a novel use case of pipelined scheduling in which the entire detailed schedule is not required. Given this we simplify the graph storage representation and scheduling algorithm.

Performance models have been designed for specific computations [5] and for general FPGA modeling [6][7]. A model for pipelined architectures was designed by Skalicky *et al.* [8] that incorporated the number of pipelines, memory bandwidth, and other design factors to accurately predict the performance of a computation. However, none of these models attempted to estimate the achievable performance of an architecture. Instead their focus was on calculating performance of the current design without requiring execution in hardware.

Scheduling has been used in previous works to map DFGs to hardware, assisting the designer in various ways. Capalija *et al.* [9] presented a method to map dataflow graphs onto a given mesh of functional units. Their dynamic scheduling approach is not portable to our problem where the pipelines are reused for a single computation. Fowers *et al.* [10] presented a dependency analysis for pipelined designs to improve the architecture by optimizing data locality. Reardon *et al.* [11] presented a modeling language to evaluate a design before functional implementation, but ignores the delay due to control logic. To improve upon the previous work, we present a metric to quantify the achievable performance of a design and use it to analyze performance degradation due to control logic

III. Graph-Based Methodology

In this work we present a performance metric and methodology for improving the control logic of a pipelined design. Scheduling the operations from the graph onto the functional units in the pipelines results in a detailed schedule providing all three items of information identified in the introduction. The length of this schedule is proportional to the number of clock cycles required for execution. This metric provides a reference to compare the performance of the design against and gauge how close the design is to the achievable performance of the architecture.

We define three types of schedules: the *optimal schedule* is the minimal length schedule of a DFG for a given pipelined architecture, the *execution schedule* is the actual order operations were executed by a given design, and the *reduced schedule* is produced from the DFG and only provides the number of operations of a particular type that are are executed at each time step. Both the *optimal schedule* and *execution schedule* provide all three pieces of information: (1) uniquely identifying each operation, (2) specifying the exact functional unit, and (3) dictating the specific time in which an operation is executed.

In the *Graph Analysis* shown in Figure 1, the DFG from the computation is scheduled onto the functional units in the pipelines. To do this, a pipeline representation is needed to describe the organization of functional units within a particular architecture. Hardware systems that have pipelined architectures contain one or more different types of pipelines. In general each pipeline contains one or more stages, where each stage contains at least one type of functional unit (adder, subtractor, etc). Figure 2 shows the functional units of an example pipelined architecture design. One way to represent this architecture is as a single-stage pipeline $\{+\}$ and a three-stage pipeline $\{\times, \div, -\}$. The representation of this architecture contains the quantity and type of functional units at each stage of the pipelines. In total, set S combines the quantity of

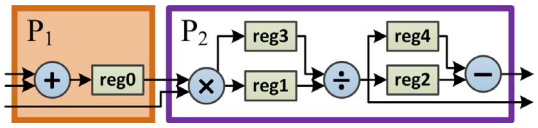


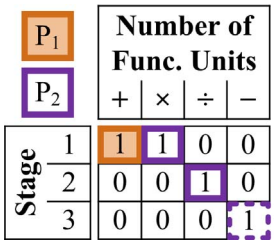
Fig. 2: Example pipelined architecture design showing only the functional units, control logic not shown, broken into one single-stage $\{+\}$ and one three-stage pipeline $\{\times, \div, -\}$.

each type of functional unit at each stage of all pipelines, as shown in Figure 3a. In the three-stage pipeline, the last stage can be bypassed and is marked with a dotted outline. We define a *use* of a pipeline as the set of operations from the graph that corresponds to the functional units present in the stages of that pipeline.

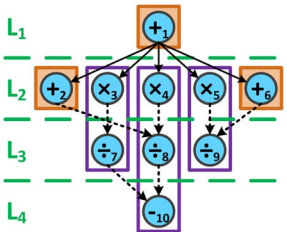
For this pipelined architecture design, the computation DFG is shown in Figure 3b. The graph is broken down into levels where the nodes in a level are independent of each other and all have incoming dependencies from nodes in the previous level (or none at all in the first level). The set of all levels in the graph is denoted as L and thus the number of levels is $|L|$. We define the *minimized configuration* of a graph to be such that all edges must go from a node in level L_i to another node in a later level L_{i+j}, and all nodes are arranged such that a node in level L_i cannot be placed in an earlier level L_{i-1} due to dependencies. Figure 3b shows the *uses* marked on the graph given the two example pipelines from Figure 2. There are three $uses$ of the single-stage pipeline (orange boxes) and three $uses$ of the three-stage pipeline (purple boxes). Additionally, only one use of the three-stage pipeline contains a subtract operation.

We assume that the operations in the graph can be evenly grouped into $uses$ with none left over. We define a graph that meets these assumptions as *well formed*. Given the simplicity of this example architecture and computation's DFG, the optimal schedule was produced and is shown in Figure 4a with 5 $epochs$ denoted E_1 to E_5. An $epoch$ is defined as a set of nodes that are executed concurrently during the same period in time (ie. same clock cycle). Notice that in this schedule each operation is uniquely identified, by its operation type and node index, and specifies the exact functional unit that the operation is assigned to.

Two problems that prevent real world applicability for this approach are the amount of memory required to store large graphs, and that the complexity of computing the optimal schedule is NP-hard [1]. As mentioned previously the optimal schedule (1) uniquely identifies each operation, (2) specifies the exact functional unit that an operation is assigned to, and (3) dictates the specific time in which that operation should

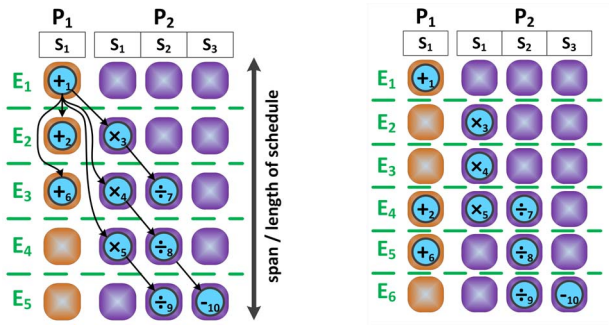


(a) Pipelines and stages for an example architecture. Note stage S_3 of pipeline P_2 is optional.



(b) Example DFG with four levels L_1 to L_4, 10 operations, 3 $uses$ of P_1, and 3 $uses$ of P_2.

Fig. 3: Pipeline representation (a) and computation DFG (b).



(a) Optimal schedule with 5 *epochs* E_1 to E_5, and *span* of 5. (b) Execution schedule with 6 *epochs* E_1 to E_6, *span* of 6.

Fig. 4: The optimal schedule (a) and execution schedule from initial architecture design (b).

be executed. Of these, we are only interested in item (3) thus simplifications are made to both the graph representation and scheduling algorithm. In addition to when operations should be executed, the length of the schedule or *span* represents the number of cycles required to complete a computation. This *span* is the metric used to gauge the performance of the design. To find this *span*, we do not need to know exactly which operation will be executed on which functional unit, but instead we just need to determine how many operations of each type are being executed during every *epoch*. Given this we simplify the graph representation and construct a *reduced graph* by only storing the number of each type of operation in a level as shown in Figure 5. In this reduced graph the operations from level L_2 are stored as a vector containing the values $\{2, 3, 0, 0\}$ that corresponds to the set of operation types $T = \{+, \times, \div, -\}$.

In this reduced representation, data dependency information is not explicitly stored. However the data dependencies can be inferred from the level ordering. For example, if an operation is in the first level, L_1, that implies that it has no

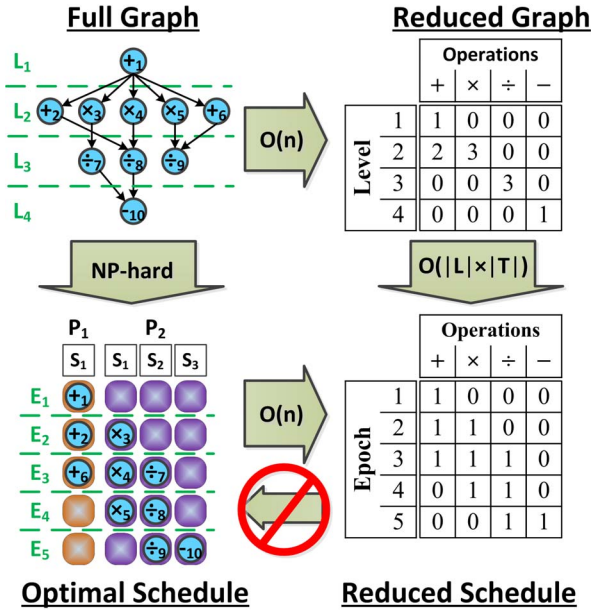


Fig. 5: Comparing graph representations and schedules from standard scheduling to our reduced approach.

Algorithm 1 EstimateSchedule(L, T, Ops, S, p)

```
1:  span := 0
2:  sch[][] := {{}}
3:  for i := 1 to |L| do
4:      uses[] := matching(T, Ops, p, i)
5:      epochs := max{uses[]}
6:      sch[][] := addToSchedule(Sch, T, Ops, S, p, epochs, i)
7:      Ops := removeNodes(T, Ops, S, p, epochs, i)
8:      span := span + max{1, epochs}
9:  end for
10: return sch, span
```

incoming dependencies. And if an operation is in the second level, L_2, that implies that there is *at least* one incoming dependency from an operation in the first level. No guarantees are given regarding the number of outgoing dependencies from an operation in level L_i to any operation in level L_{i+j}. The implications of using this reduced representation may result in a loss of edge information. This effect can be seen in the example graph for nodes in levels L_2 and L_3 as shown with the dotted arrows in Figure 3b.

The reduced representation can be constructed using a depth-first-search requiring minimal memory by only storing each node and its level, until all of its successor nodes have been labeled. Once all of its successors are labeled, the memory used to store the information for that node would no longer be needed. This conversion from the full graph to the reduced graph has a computational complexity of $\mathcal{O}(n)$. This reduced representation is necessary to store large graphs such as matrix-matrix multiplication graphs on data sizes larger than 2048x2048 that contain more than 10^{10} nodes. To illustrate the savings of our reduced representation, consider the graph for matrix-matrix multiplication of 8192x8192 sized matrices. This graph contains more than one trillion operations and more than one trillion edges. Storing this graph using an adjacency list requiring $\mathcal{O}(|V| + |E|)$ storage would equate to almost 8TB of memory. However, the reduced representation of this graph can be stored using only a 13x2 array (104 bytes) since there are 13 levels in the graph containing only addition and multiplication operations.

Since the edge information is stored implicitly in the reduced representation, none of the the current state-of-the-art scheduling algorithms are capable of processing its structure. Algorithm 1 presents a simple scheduling algorithm capable of handling this representation. This algorithm calculates the number of operations that can be executed at each epoch, producing the reduced schedule. It operates using the set of levels in the graph, L, the set of operation types, T, and the reduced graph, a 2D-vector containing the number of operations of each type in each level of the graph, Ops. It also requires the set of stages in the pipelines, S, and the pipeline representation, a 2D-vector containing the number of functional units of each type in each pipelines stage, p.

This scheduling algorithm employs an iterative procedure where the operations in the DFG are assigned to the functional units in the pipelines, grouping operations into $uses$, via a $matching$ function as shown on line 4. Once the $uses$ are identified then we calculate the number of $epochs$ that will be required to complete these $uses$ by taking the max of all the $uses$ as shown on line 5. Then, the operations that can

be executed in each pipeline stage are added to the schedule (line 6) and removed from the DFG (line 7). Lastly, we add the number of $epochs$ needed for this level with the number required for previous levels on line 8. If the number of $epochs$ required for a particular level is zero, then we add 1 to the $span$ to account for pipeline latency. This process continues for each level in the graph.

For the example graph, the presented scheduling algorithm produced the reduced schedule shown in Figure 5. In comparing the optimal schedule to the reduced schedule, notice that there are the same number of epochs (thus same schedule length) and the epochs in which operations are executed match up with those in the optimal schedule. Using this reduced schedule, the designer can still understand when the operations should be executed. This algorithm has a polynomial runtime on the order of the number of levels in the graph and the number of operation types $\mathcal{O}(|L| \times |T|)$. This runtime is much less than the runtime for existing algorithms that are on the order of the number of operations and/or edges in the graph [2][3][4]. Compared to current minimal storage techniques such as adjacency list or incidence list, which require $\mathcal{O}(|V| + |E|)$ memory for storage, our reduced representation requires storage on the order of the number of levels in the graph and the number of operation types, $\mathcal{O}(|L| \times |T|)$. Figure 5 shows the complexity of computing the optimal schedule and our solution using the reduced schedule.

Using this reduced representation, each operation is not stored individually thus the edge information between two operations is no longer needed. The accuracy of this estimation is sufficient for use in this methodology to improve the performance of pipelined architectures. In the next section we present an example architecture and show how this methodology can be applied to aid the designer in their formulation of modifications for improved performance.

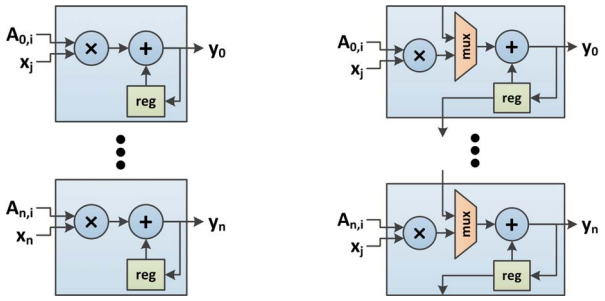
IV. Results

In this section, we present the results of analyzing a series of benchmark computations on five well researched pipelined architectures [12][13][14][15] for dot product, matrix-vector and matrix-matrix multiplication, Cholesky decomposition, and matrix inversion. These were chosen as representative computations constrained by different factors, having various levels of control flow complexity. Since these architectures were designed with scalability in mind, they were evaluated using from 1 to 256 pipelines. We varied data sizes from 4x1 to 8192x1 for vectors, and from 4x4 to 8192x8192 for square matrices. We used square matrices without loss of generality by using a block-based approach for matrix computation.

TABLE I: Summary of Results

Comp.	Performance difference between metric and		Achieved Speedup
	Original [% diff.]	Improved [% diff.]	
Dot Prod.	0	0	1.0x
M-V Mult.	0 to 84	0	6.4x
M-M Mult.	0 to 91	0	10.7x
Cholesky	0 to 55	0 to 10	2.2x
Inverse	50 to 97	0 to 93	3.0x

Note: the percent difference is calculated with respect to the architecture such that % diff. = $(Arch - Metric)/Arch$



(a) Original from [12] (b) Improved

Fig. 6: Matrix-vector multiply designs.

Using our methodology we compare the performance of the current design to the achievable performance for the architecture to determine how much more performance can be extracted out of the architecture by making improvements to the control logic. Table I shows the collected results for each of the five architectures evaluated. The first two columns show the range of difference in performance between the metric and the original and improved designs. The third column shows the best speedup achieved for that computation across the range of data sizes evaluated.

Given the simplicity of the dot product computation very little control logic was required. We found that the performance of the design was not degraded by the control logic: the performance metric equaled the number of clock cycles required by the design for every data point. These results validate that our methodology correctly identifies a good design that already achieves high performance.

Similarly, the performance of both **matrix-vector** and **matrix-matrix multiplication** designs matched that of the metric when the number of pipelines was less than or equal to the dimension of the matrices. However, the designs for these computations have an upper limit to the number of pipelines that can be used for a given data size, and when a larger number than needed is available, the extra pipelines are unused. The graph analysis however, is not restricted by this particular design constraint. Comparing the metric to the performance of the original design for data sizes from 4 to 128, there was potential to achieve increased performance ranging from 50% to 91%. No performance improvement was possible for data sizes from 256 to 8192 given that the maximum number of pipelines (256) was not greater than the dimension

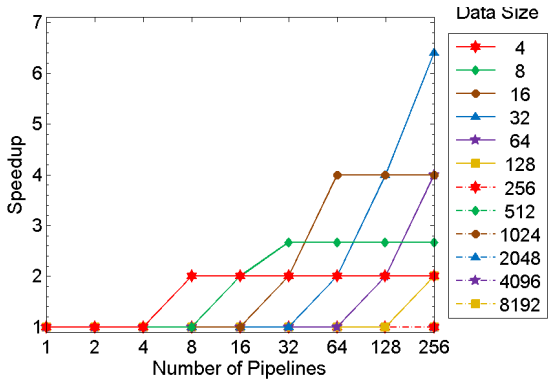
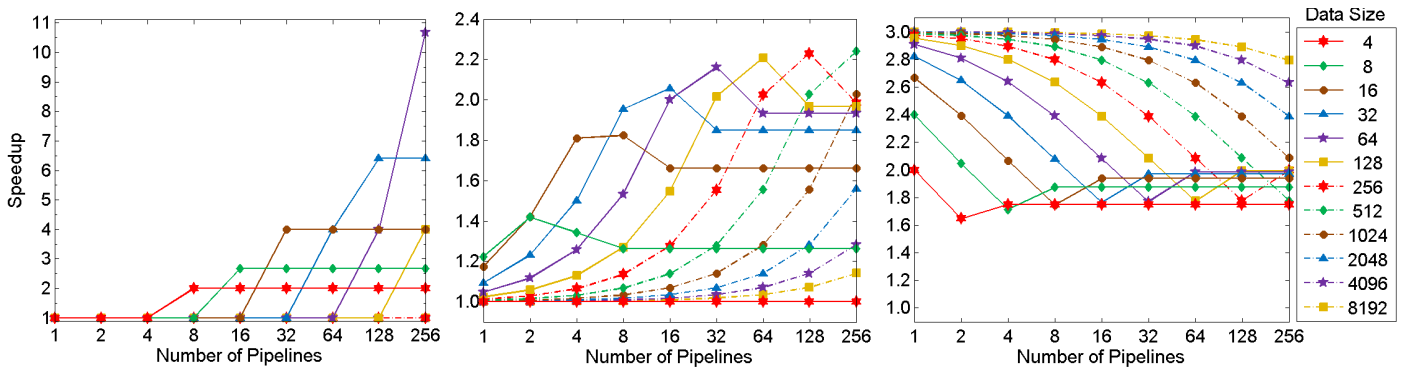


Fig. 7: Speedup of the improved matrix-vector multiplication design versus the original design. Note that results overlap at 1x for data sizes from 256 to 8192.

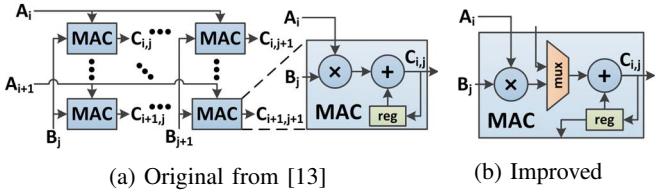


(a) Matrix-matrix multiplication (b) Cholesky decomposition (c) Matrix inverse

Fig. 8: Speedup of the improved designs versus the original designs. Note that results overlap at 1x for data sizes from 256 to 8192 for matrix multiply.

of the matrices. Figure 6a shows the original design for matrix-vector multiplication. When there are more pipelines than the dimension of the data, the additional pipelines can be used in a fashion similar to the dot product architecture where pipelines cascade their multiplication results to be summed by other pipelines as shown in the improved design in Figure 6b. Thus, instead of multiplying the vector element against each element in the row of a matrix sequentially and summing using a multiply-accumulator, the better approach would multiply all elements in the row of the matrix against the vector element simultaneously. This approach would take advantage of the additional pipelines available and achieve speedups of up to 6.4x and 10.7x for matrix-vector and matrix-matrix multiplication respectively as shown in Figures 7 & 8a. The original and improved designs for matrix-matrix multiplication are shown in Figure 9.

Compared to the previous computations, the calculations for Cholesky decomposition and matrix inverse are much more complex. Using our approach, we found very large differences between the metric and the performance of the original designs. In general, this means that the designs for both computations are capable of achieving better performance. The **Cholesky decomposition** architecture operates on a column-by-column basis to expose more parallelism than in the traditional row-by-row method [14]. In the original architecture design, shown in Figure 10a, each processing element (PE) pipeline executes the calculations for each element in the column of the input matrix. Since the linear equations contain more variables for each subsequent row, a single divider pipeline is shared among each of the pipelines. We found that the difference between the metric and actual performance ranged from 0.04% to 55%. Based on these results, we investigated the source of the differences and present improvements for the architecture.



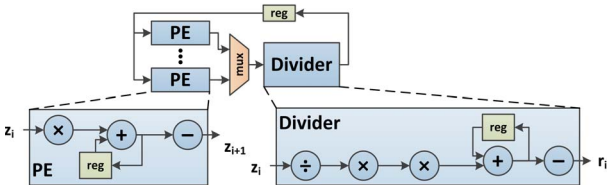
(a) Original from [13] (b) Improved

Fig. 9: Matrix-matrix multiply designs.

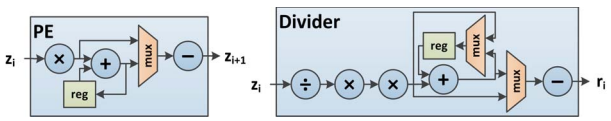
First, we observed that the graph scheduling did not require the first PE pipeline *use* to execute any add operations, thus bypassing the adder. In comparison, the architecture control logic forced data to pass through the adder (effectively adding zero to the result from the multiplier) for every *use*. This can be avoided for each column of the matrix and the savings increase as the matrix size increases. Similarly, the adder in the divider pipeline can also be bypassed for the first *use* in each column. Our improved design added multiplexers to allow the adders in both the multiplier and divider pipelines to be bypassed as needed and is shown in Figure 10b. Second, we also found that the control logic in the architecture design restricts each column to be executed separately from other columns. After making the improvements to address these problems, we analyzed the improved Cholesky decomposition design and found that the performance was now within 10% of the calculated achievable performance.

Our improvements achieved from 1.0x (no improvement) to 2.2x speedup over the original design as shown in Figure 8b. There was no improvement for the 4x4 data size. The data sizes that gained the most performance ranged from 64x64 to 512x512. For the original design, the performance at the larger data sizes was very close to the calculated achievable performance, thus the improved design was not able to achieve the same level of improvement as with the smaller data sizes.

The **matrix inverse** architecture was designed from an original systolic array design [15]. By collapsing the triangular systolic array into a linear array, their design achieves improved performance thanks to a larger array size. Each processing element (PE) of the array can either perform a

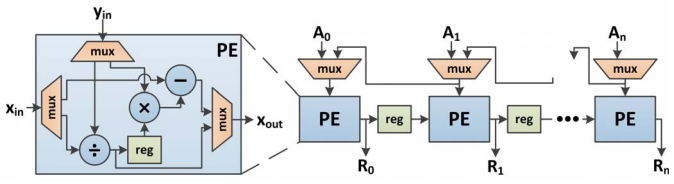


(a) Original from [14]

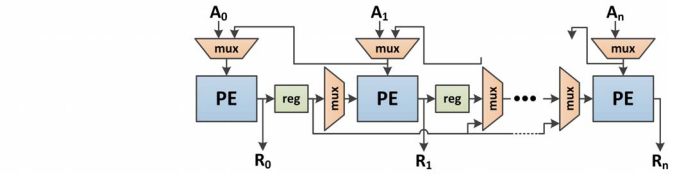


(b) Improved

Fig. 10: Cholesky decomposition designs.



(a) Original from [15]



(b) Improved

Fig. 11: Matrix inverse designs.

single division operation or a multiply and subtract sequence of operations as shown in Figure 11a. By evaluating the architecture using our methodology, we found that the difference between the metric and the actual performance of the design ranged from 50% to 97% for all data sizes and number of pipelines. In comparing the execution of the architecture design against that of the scheduling approach, we found that the main bottleneck was distributing the divider result to the other elements. The division is normally calculated in the left-most element, so multiplexers can be added to allow the result of the division to be distributed in a single clock cycle as shown in Figure 11b.

After making these improvements, we compared the new inverse design to the metric and found that the differences ranged from 0% to 93% with the majority of the results within 31% of the calculated achievable performance. The range of results for the 8192x8192 data size went from 67% to 83% for the original architecture, down to 0% to 12% using the improved design. For data sizes from 8x8 to 256x256 the majority of the improvement was achieved using 1 to 16 pipelines. The improved design was not able to achieve the same level of increased performance for smaller data sizes using a larger number of pipelines. For these smaller sizes improvements ranged from only 4% to 20%. Our improvements achieved speedups of 1.6x to 3.0x better performance than the original design as shown in Figure 8c. The larger data sizes gained the most performance for all numbers of pipelines.

V. Conclusion

In this work we presented a methodology for improving the performance of pipelined architectures using a high-level graph-based approach. The dataflow graph of the computation is correlated to the functional units in the pipelines via scheduling to quantify the achievable performance of the design. This is then compared to the actual performance from execution to determine how much improvement can be achieved by making modifications to the design. Thus validating the performance of a good design and aiding in the formulation of modifications to the design.

Given our unique requirements on scheduling, we have presented a reduced graph representation that allows large graphs to be stored using considerably less memory. We presented a simple polynomial-time scheduling algorithm that operates on this reduced representation. To the best knowledge of the authors, this is the first effort to schedule a graph and produce a subset of information about the detailed schedule. This approach greatly simplifies the scheduling algorithm.

We evaluated five benchmark computations using our methodology and presented improvements to existing well researched architectures from the literature. Our results showed speedups of up to 10.7x were achieved over the original designs. Our method can be used to quickly estimate performance without any implementation or detailed design description. The same method can also be used with different design goals to improve the power consumption or other factors. This manual process could also be automated and integrated with existing synthesis or simulation tools. In the future, we will investigate how close this scheduling approach is to the optimal schedule. We will also apply a similar methodology to CPU and GPU implementations to improve the performance of relevant computations on those platforms as well.

References

[1] A. Benoit, ́U. Çatalyürek, *et al.*, "A Survey of Pipelined Workflow Scheduling: Models and Algorithms," *ACM Computing Surveys*, vol. 45, no. 4.

[2] M. Lee, W. Liu, *et al.*, "A Mapping Methodology for Designing Software Task Pipelines for Embedded Signal Processing," *Intl. Parallel Processing Sym.*, 1998.

[3] A. Benoit and Y. Robert, "Mapping Pipeline Skeletons onto Heterogeneous Platforms," *Journal of Parallel and Distributed Computing*, vol. 68, no. 6, 2008.

[4] M. Spencer, R. Ferreira, *et al.*, "Executing Multiple Pipelined Data Analysis Operations in the Grid," *ACM/IEEE Conf. on Supercomputing*, 2002.

[5] C.Y. Lin, H.K. So, *et al.*, "A Model for Matrix Multiplication Performance on FPGAs," *Intl. Conf. on Field Programmable Logic and Applications*, 2011.

[6] B. Holland, K. Nagarajan, Karthik, *et al.*, "RAT: RC Amenability Test for Rapid Performance Prediction," *ACM Trans. on Reconfigurable Technology and Systems*, vol. 1, no. 4, 2009.

[7] B. Holland, A. George, *et al.*, "An Analytical Model for Multilevel Performance Prediction of Multi-FPGA Systems," *ACM Trans. on Reconfigurable Technology and Systems*, vol. 4, no. 3, 2011.

[8] S. Skalicky, S. Lopez, *et al.*, "Performance Modeling of Pipelined Linear Algebra Architectures on FPGAs," *Intl. Sym. on Applied Reconfigurable Computing*, 2013.

[9] D. Capalija and T. Abdelrahman, "A High-Performance Overlay Architecture for Pipelined Execution of Data Flow Graphs," *Intl. Conf. on Field Programmable Logic and Applications*, 2013.

[10] J. Fowers and G. Stitt, "Dynafuse: Dynamic Dependence Analysis for FPGA Pipeline Fusion and Locality Optimizations," *Intl. Sym. on Field Programmable Gate Arrays*, 2013.

[11] C. Reardon, B. Holland, *et al.*, "RCML: An Environment for Estimation Modeling of Reconfigurable Computing Systems," *ACM Trans. on Embedded Computing Systems*, vol. 11, no. S2, 2012.

[12] L. Zhuo and V. K. Prasanna, "High-Performance Designs for Linear Algebra Operations on Reconfigurable Hardware," *IEEE Trans. on Computers*, vol. 57, no. 8, 2008.

[13] I. Sotiropoulos and I. Papaefstathiou, "A Fast Parallel Matrix Multiplication Reconfigurable Unit Utilized in Face Recognitions Systems," *Intl. Conf. on Field Programmable Logic and Applications*, 2009.

[14] D. Yang, G. Peterson, *et al.*, "Compressed Sensing and Cholesky Decomposition on FPGAs and GPUs," *Parallel Computing*, vol. 38, no. 8, 2012.

[15] F. Edman and V. Owall, "Implementation of a Highly Scalable Architecture for Fast Inversion of Triangular Matrices," *IEEE Intl. Conf. on Electronics, Circuits and Systems*, 2003.