Automatic Support for Multi-Module Parallelism from Computational Patterns

Nithin George*, HyoukJoong Lee†, David Novo*, Muhsen Owaida*, David Andrews‡, Kunle Olukotun† and Paolo Ienne*

*Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences.
†Stanford University, Pervasive Parallelism Laboratory.
‡University of Arkansas, Department of Computer Science and Computer Engineering.
{nithin.george, david.novo, mohsen.ewaida, paolo.ienne}@epfl.ch, {hyouklee, kunle}@stanford.edu and dandrews@uark.edu

***Abstract*—*Field Programmable Gate Arrays* (FPGAs) can be customized into application-specific architectures to achieve high performance and energy-efficiency. Unfortunately, they are yet to gain significant adoption by application developers due to their low-level programming model. Moreover, to obtain good performance in an FPGA design, one often needs to correctly parallelize computation and balance the computational throughput with the available data access bandwidth. To address the programming model problem, recent efforts have focused on composing applications out of parallel computational patterns, such as `map`, `reduce`, `zipWith` and `foreach`, and leveraging the properties of these patterns to generate highly parallel hardware modules capable of high performance. In this work, we focus on the problem of further improving the performance and show that we can utilize the knowledge of how data is consumed and produced by these computational patterns in conjunction with the information of the system architecture to automatically parallelize computations across multiple hardware modules. To achieve this, we automatically infer synchronization needs arising due to parallelization and generate a complete system that can obtain high performance for a given application. We evaluate our approach using seven applications from different domains and show that our automatically generated designs achieve performance improvements ranging from 1.8 to 9.4 times.**

I. Introduction

Field Programmable Gate Arrays (FPGAs) are extremely versatile and can be programmed into application-specific circuits which, for many applications, outperform processor-based devices, such as CPUs and GPUs [1], [2]. Despite this, application developers have shown limited interest in using FPGAs because programming them requires hardware design knowledge, an uncommon skill, that renders these devices inaccessible. Many *High-Level Synthesis* (HLS) tools try to bridge this gap by generating hardware from high-level languages, such as C, C++ and OpenCL [3]–[5]. While these tools use a high-level language for design specification, they still require the application developer to consider a hardware oriented programming model and demand hardware design knowledge to perform optimizations (e.g., refactoring the code, instantiating buffers, utilizing burst transfers or annotating the code to enable specific optimizations) to obtain good results [6].

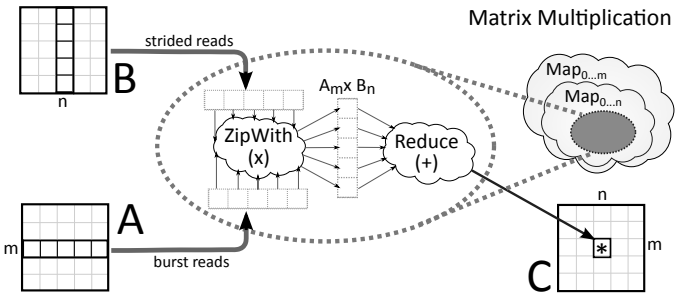

Fig. 1. In the matrix multiplication, two parallel computational patterns (`zipWith` and `reduce`) compute the value of each result cell. Among them, the `zipWith` performs element-wise multiplication between the rows of matrix A and the columns of matrix B. The `reduce` sums all the results from the `zipWith` to produce the final cell value. To implement the complete multiplication, these patterns are nested inside two `map` patterns.

In order to address this problem, recent efforts have focused on composing applications from well understood parallel computational patterns, such as `map`, `reduce`, `zipWith` and `foreach`, and leveraging the properties of these patterns to automatically infer suitable optimizations [7]. To understand how this works, consider the computation $C = A \times B$, shown in Figure 1, where A, B and C are floating point matrices. Here, each cell in C is the dot-product of a row of A and a column of B and this computation can be composed using a `zipWith` pattern, which performs the element-wise multiplication between a row of A and a column of B, followed by a `reduce` pattern that adds the results from the multiplication. To complete the matrix multiplication, the `zipWith` and `reduce` patterns are nested inside two `map` patterns, one iterating along the columns of B and the other along the rows of A. These patterns reveal the parallelism in the computation (e.g., the separate multiplications in `zipWith` can be parallelized) and expose how the data is consumed and produced in the process (i.e., A is read row-wise and B is read column-wise and used to produce a completely new matrix C). By leveraging these properties, a pure functional application description can be automatically translated into a well structured and annotated program that will produce a good quality hardware module using current HLS tools.

Although the computational patterns make it possible to generate a highly parallel hardware module, the execution performance still depends on the data access pattern, dependency

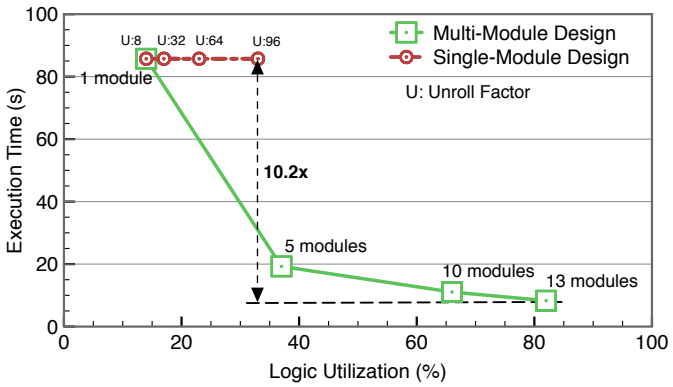

Fig. 2. The design with only a single module performs poorly due to data starvation, and further parallelization will consume additional resources, but not deliver better performance. The design with multiple modules is able to achieve higher aggregate data bandwidth and, therefore, better performance.

between accesses and even latency of the operations. For instance, in the matrix multiplication, if matrices A and B are stored in row-major order, the columns of B need separate strided accesses that will underutilize the bus-bandwidth and significantly diminish the performance of the module[1]. So, despite the parallelism available in the computation, performance of the module will remain fixed due to data starvation. Therefore, while the pattern-based approach can vary the amount of parallelism (e.g., loop unrolling) exploited to generate multiple implementations of the hardware module (*variants*), the performance does not improve; Figure 2 (single module design) provides experimental evidence of this effect of data starvation where further parallelization consumes additional resources but does not deliver better performance.

In such counterintuitive situations, we may still improve performance by parallelizing computations across multiple hardware modules, similar to parallelizing a computation across multiple CPUs to improve the aggregate throughput. Since parallelizing computations results in the need for synchronization, to achieve this on FPGA, the designer must identify synchronization requirements in the application and build custom synchronization schemes. Sometimes, there are even less obvious needs for synchronization, such as false sharing [8] due to mismatches between the widths of the system bus and the datatype in the computation. Moreover, to efficiently parallelize computation across modules, there is also a need for good work partitioning schemes. All these issues make the task of manually parallelizing the computation tedious, error prone and, above all, hard to accomplish without hardware design expertise.

This work presents a new approach to automate the parallelization of computational kernels composed using computational patterns. The approach includes, (1) automatic extraction of synchronization requirements in the kernel when parallelized across multiple hardware modules, (2) design space exploration using *Integer Linear Programming* (ILP) to find the set of module variants that obtain the best performance for a given application, and (3) automatic generation of the complete system

[1]We are using a simplistic implementation of matrix multiplication here to illustrate the idea. An optimized implementation is also considered later.

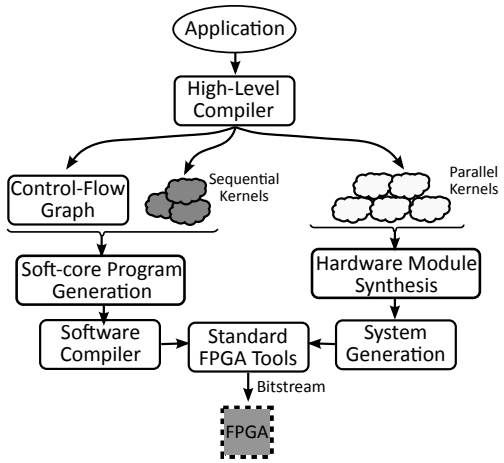

Fig. 3. The Delite compiler decomposes the high-level application into parallel and sequential patterns, and an associated control-flow graph. The parallel patterns undergo hardware synthesis to generate hardware modules that become part of the final system design. The sequential patterns and the execution schedule generated from the control-flow graph are generated into software for the soft-core processor in the system.

using the selected module variants along with all the essential synchronization hardware. We perform this automation by leveraging the properties of the computational patterns, specifically how data is consumed and produced, and the properties of the system architecture, such as the data-allocation strategy and width of the system-bus. By employing this approach on the matrix multiplication kernel, we are able to exploit the parallelism in the outer-most level `map` pattern to parallelize the computation across multiple modules and achieve better performance, as shown in Figure 2.

In the rest of this paper, Section II takes a closer look at the computational patterns and how kernels composed out of these patterns are generated into complete hardware designs. Section III discusses the methodology to parallelize kernels composed from patterns across multiple hardware modules and explains some of the specific optimizations we perform. We demonstrate the benefit of the approach in Section IV by presenting the performance improvements we achieve on seven different applications. Section V examines some related work and Section VI concludes this paper.

II. Computational Patterns to Hardware Systems

In order to understand how kernels should be parallelized across multiple modules, we need to first know how high-level applications are decomposed into these parallel kernels and how they are generated into complete hardware systems.

Our toolchain, shown in Figure 3, is an extension of our prior work [7] and it compiles high-level applications into hardware systems that can be programmed on an FPGA. In this flow, the application programs are first compiled with the Delite [9] compiler and decomposed into parallel and sequential kernels. The parallel kernels contain one or more parallel computational patterns, namely `map`, `reduce`, `zipWith` and `foreach`, that have well understood properties, such as the nature in which it produces or consumes data or the parallelism in its operation;

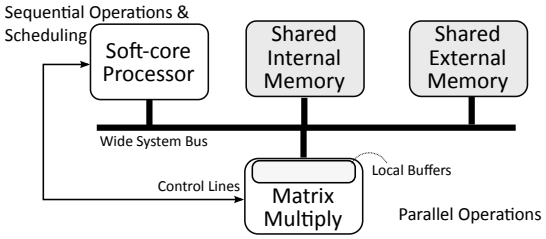

Fig. 4. In the hardware system generated for the matrix multiplication, a hardware module implements the multiplication kernel. The soft-core processor schedules the execution of this module and executes the sequential kernels, if any. The data for the computation is stored in the shared memory that is accessed over the wide system bus. Each hardware module can have a local memory for buffering input data and holding intermediate results.

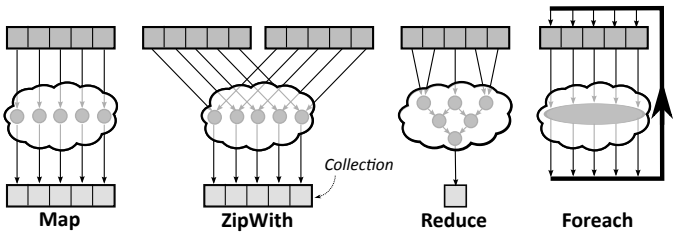

Fig. 5. In the `map` and `zipWith` patterns, a pure function is used to create a new collection from either one or more collections. The `reduce` uses a binary operation that is associative and commutative to compute a single value from a collection. The `foreach` uses an impure function to update the values of an existing collection.

matrix multiplication is an example of such a parallel kernel. The toolchain leverages these properties to automatically infer the suitable optimizations for each pattern in the kernel and then utilizes an HLS tool, in our case Vivado HLS [3], to generate a highly parallel hardware module. Our toolchain allows us to vary the amount of parallelism (e.g., loop-unrolling and pipelining) exploited in the generated hardware to create multiple implementations (*variants*) for each kernel, as shown for single module design in Figure 2.

After the variants for the parallel kernels in the application are generated, one variant is selected per kernel and then connected within a system architecture template using wide, high-bandwidth buses to complete the system design. This template provides shared memories, clock and control circuitry, and a soft-core processor. Figure 4 shows the system design for the matrix multiplication application. The soft-core processor in this system design is used to execute the sequential kernels in the application and for orchestrating the execution of the hardware modules. Our toolchain automatically generates software for this processor from the sequential kernels and the application's control-flow information. Our prior work [7] covers the details of this tool-chain in further depth.

Parallel Computational Patterns. The parallel kernels extracted by the Delite compiler contain one or more computational patterns, such as `map`, `zipWith`, `reduce` and `foreach`. Within each computational pattern, we can have operations on scalars (e.g., `bool`, `int`, `float`, `double`), and *collections*, which include `array` and more complex datatypes such as `vector`, `matrix` and other user-defined types.

More importantly, these patterns provide certain guarantees regarding the nature of their computation and how they produce or consume data. As illustrated in Figure 5, the `map` and `zipWith` patterns always use a *pure* (side-effect free) function to create a fresh collection. The difference between them is that `map` has a single input collection while the `zipWith` has multiple input collections. So, squaring each element in a `vector` uses a `map` while adding two `vectors` requires a `zipWith`. The `reduce` pattern computes a single element by applying a binary function that is both associative and commutative to all the elements in a collection; so, finding the minimum or maximum value in an array are examples for this pattern. The `foreach` pattern is typically used to modify values in an existing collection by applying an *impure* (with side-effects) function to each element, such as to set all the negative numbers in an existing array to zero. Additionally, our programming model restricts `foreach` to guarantee that this pattern can be executed in parallel without data races.

These computational patterns make it easy to identify the parallelism in the application and expose it to the HLS tool. But, as illustrated in Section I with the matrix multiplication, the performance of the resulting hardware module still depends on aspects such as the data access pattern, interdependencies between accesses and latency of the operations.

III. Parallelizing Computation Across Multiple Modules

The key focus of this work is to overcome the problem of low performance from a single hardware module by parallelizing computation across multiple modules. In this section, we discuss this approach and detail how we leverage the properties of the computational patterns and that of the system architecture to automate this process.

A. Analyzing Synchronization Needs for Parallelization

To parallelize kernels across multiple modules, we identify how each kernel uses and produces data and utilize synchronization schemes to guarantee correct use of shared data structures. Since these kernels are composed from computational patterns, we utilize the properties of these patterns to infer how data is consumed and produced in the kernel. We use this knowledge in conjunction with the properties of the system, such as data allocation strategy and width of system-bus, to correctly identify the synchronization requirements between the modules.

Kernels With Single Pattern. In the case of kernels with a single pattern, if this is a `reduce`, it operates on a collection to compute a single new result. When parallelized, multiple modules update this result, as shown in Figure 6(a); therefore, we need to use a mutex to avoid data races. If, however, the kernel has a `map`, `zipWith` or a `foreach` pattern, each elemental operation uses distinct elements from the input collection(s) to compute independent elements in the output collection. Hence, one might naïvely assume that there is no need for any synchronization. However, this would be incorrect if there is a mismatch between the size of datatype used and the width of system bus; since the latter is typically much larger in order to maximize data bandwidth, this can create false sharing problems [8]. False sharing occurs because each

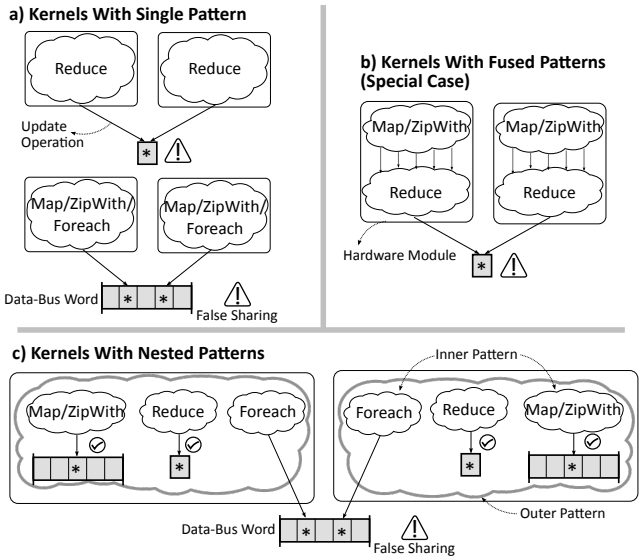

Fig. 6. We identify rules for correctly synchronizing between patterns that are parallelized across multiple modules: a) Kernels with a single pattern need to use mutex for all writes. b) In kernels where `map` or `zipWith` is "fused" with a `reduce` that directly consumes the output from the former, only the `reduce` needs to use a mutex. c) In kernels with nested patterns, the outermost pattern needs to use mutex for all writes; among the inner-level patterns, only `foreach` patterns need to use mutex.

elemental update to the output collection will need to perform a read-modify-write operation which will give rise to data races if multiple modules update the same bus-word simultaneously. This is shown in Figure 6(a). Some bus protocols provide data masks to selectively update specific bytes in a bus-word, however, we can have collections of datatypes that are smaller than a byte making such schemes insufficient. To illustrate with an example, if the output from a `foreach` is a collection of boolean values, the modules will need to update single bits in this collection and this is only possible with a read-modify-write operation. If another module is simultaneously updating another bit in the same data-bus word, they can inadvertently overwrite each other's results. Therefore, a mutex is necessary to make this read-modify-write operation atomic.

Kernels With Fused Patterns. While generating parallel kernels, the Delite compiler sometimes *fuses* multiple patterns together so that they can execute in parallel. For instance, if we compute the minimum and maximum from the same collection, the two `reduce` patterns might get fused into a single kernel to compute both the minimum and maximum values in parallel. If the fused patterns are completely independent, they each retain the synchronization requirements they had in the simple case. The exception to this is when a `map` or `zipWith` is fused with `reduce` and the latter directly consumes the data from the former. In this case, as shown in Figure 6(b), if the output of the `map` or `zipWith` never write into the shared memory, only the output from the `reduce` would need to use a mutex.

Kernels With Nested Patterns. When computational patterns are nested, such as in matrix multiplication where the dot-product is nested inside `map` patterns, the outermost pattern retains the same synchronization requirements as they would have had in the single pattern case. If the inner-level pattern is either a `map`, `zipWith` or a `reduce`, the new data it produces is visible only within the execution context of a single computation of the outer-level pattern and hence disjoint. However, to guarantee this, the data allocation strategy we use ensures that data created by the inner-level patterns from different modules never share the same data-bus word. Thus, in the matrix multiplication, the `zipWith` and `reduce` that calculate the dot-product at the inner-level do not need any synchronization. However, when the result matrix is updated by the outer-level `map` pattern, the different modules need to synchronize using a mutex. This is advantageous since it enables the inner-level computation that executes more often to progress in parallel without any synchronization overheads. If, however, the inner-level pattern is a `foreach`, it can update any collection that was allocated before and, therefore, the potential exists for data races between the `foreach` patterns in different modules that write to same collection. Hence, in the case `foreach`, we need to use a mutex even when it is nested inside other patterns.

These synchronization rules, summarized on Figure 6, are utilized in our compiler analysis to identify synchronization requirements and to correctly parallelize kernels across multiple modules.

B. Reducing Synchronization Requirements

Synchronization, while needed for correct execution, serializes operations across the modules and, therefore, diminishes performance benefits of parallelization. Hence, it is better to reduce synchronization requirements as much as possible.

Optimizing the `reduce`. When a kernel with a `reduce` pattern at the outermost level is parallelized across multiple modules, a straightforward optimization is to provide each module with a local data structure to hold partial results. This permits the different modules to operate in parallel and synchronization is only needed at the end of the computation when the different partial results are combined and written to the shared data structure. This optimization is possible because the elemental computation in `reduce` is both associative and commutative; the order in which the operations are performed and later combined does not matter.

Optimizing `map`, `zipWith` and `foreach`. If the parallel operation is a `map` or a `zipWith` pattern, we know that the i^{th} elemental operation will utilize the i^{th} element(s) of the input collection(s) to produce the i^{th} result in the output collection. We utilize this knowledge while partitioning the kernel's computation so that each module is given a contiguous range (from i^{th} to $(i+n)^{th}$) of the computation, thereby, ensuring that the same module writes n sequential values in the output collection. Therefore, the writes from different modules can interfere with each other only on the edges of their respective sequential ranges, avoiding synchronization for the non-edge writes. Unlike the `map` and `zipWith`, the `foreach` can update a preallocated collection in any order and is, therefore, not compatible for this optimization. We overcome this problem with additional compiler analysis to identify if the collection written to by the `foreach` is updated sequentially and selectively apply this optimization. Figure 7(a) illustrates

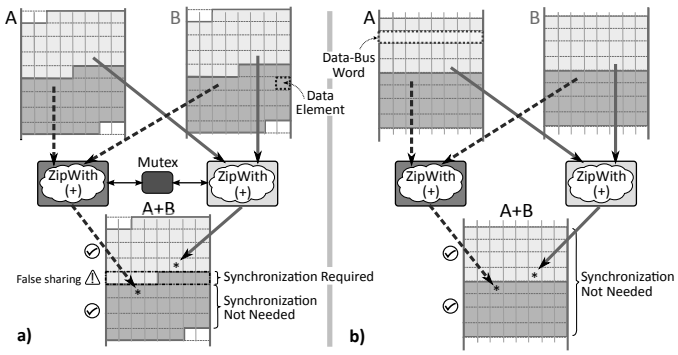

Fig. 7. Vector addition is used as an example to illustrate the opportunities to reduce synchronization overheads. a) The write from each hardware module updates sequential locations; therefore, the modules can only have false sharing at the edges of their respective sequential ranges. A hardware mutex is used to make updates to these edge elements atomic. b) The access range of each module is aligned to data-bus width and the work unit size is chosen appropriately to eliminate the need to use synchronization hardware.

the benefit of this optimization by considering the case of vector addition implemented with a `zipWith` pattern. In this case, since each module updates a sequential range of locations, the only possibility for false sharing arises at the boundaries of these ranges when different modules need to write to the same data-bus word. Therefore, the writes to the other locations do not need any synchronization.

A special situation arises when the writes are sequential and we can statically determine that an output collection is accessed starting from a memory address that is aligned to the data-bus width; this is sometimes possible for `array` and `vector` datatypes when they do not have runtime determined variables used in their access. In this case, by controlling the work assigned to each module, we can ensure that last memory address written to by the different modules align with the end of the data-bus word. In such a case, the results from different modules never overlap, thereby, avoiding the need for any synchronization between them. Figure 7(b) illustrates this case, again, using vector addition.

C. Managing the Multiple Modules

Having addressed synchronization problems, we now need a work sharing scheme to partition the computation among multiple hardware modules. In our toolchain, the Delite compiler represents the work for each computational pattern as iterations over a sequential index range. But, the processing times can vary widely due to conditional and data-dependent operations within the kernel. Additionally, for each kernel, we can have hardware modules with different processing performance. To tolerate this variability and still produce good performance, we employ a dynamic load balancing scheme with a central task-pool [10] to distribute the work. To implement this, we store the iteration index range for the outermost level pattern, which is parallelized across the modules, in a shared data structure (task-pool). During execution, each module dynamically updates this index range, taking away a small portion of the work for execution. Since the modules that finish faster take away portions more frequently, we achieve the load balancing. However, since

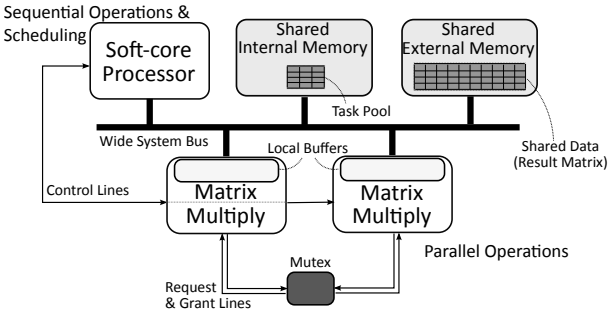

Fig. 8. Multiple hardware modules are used to improve the performance of matrix multiplication. The system uses a mutex to synchronize the modules while updating the shared task pool or the result matrix in the shared memory. Each module connects to the mutex using separate request and grant lines.

multiple modules access and update the task-pool, accesses must be synchronized using a mutex.

D. Generating the Complete System

The knowledge of the synchronization requirements derived by our compiler is used while generating the hardware modules; these modules correctly acquire mutex locks while updating shared data structures and, therefore, can execute in parallel. Furthermore, the toolchain described in Section II produces multiple variants (hardware implementations) for each parallel kernel by varying the degree of parallelism (i.e., loop-unrolling and pipelining) exploited in the variant. Additionally, the ability to parallelize computations across multiple modules further widens the design space making it hard to find the optimum *configuration*, which is the number and types of variants used per parallel kernel, in the final system design.

Design Space Exploration. We utilize the performance and resource estimates from the HLS tool to guide the design space exploration. To find the optimum configuration, we model this as an *Integer Linear Programming* (ILP) problem and use an ILP solver, as done by Graf et al. [11]. In the ILP formulation, we try to maximize the performance of the application while ensuring that at least one variant is selected for each parallel kernel and the design fits on the FPGA device. However, this modeling is approximate since it is based only on static analysis and it assumes that each extra hardware module provides the performance improvement as indicated in the HLS estimates; the latter implies that external factors, such as maximum system bandwidth and contention between modules over shared data, do not significantly affect performance. To make this assumption reasonable, we address the bandwidth problem with an ILP constraint to ensure that the total bandwidth used by all modules of a given kernel is less than the maximum bandwidth of the system bus For the contention problem, we perform compiler analysis to identify kernels where the computation at the innermost level of nesting performs frequent 'locked' updates to shared data structures and add constraints to ensure that these modules are not parallelized. With these additional constraints, the ILP solver finds the optimal configuration of the system design for the application based on the model.

System Generation. The configuration found by the ILP solver is used to select the hardware modules and integrate them

TABLE I
PARALLEL OPS. AND COMPUTATIONAL PATTERNS IN EACH APPLICATION

Application	Parallel Ops.	Map	Zipwith	Reduce	Foreach
MMuopt	3	3	1	1	0
MMopt	3	3	1	1	1
ACorr	5	6	1	5	0
PRank	6	5	0	3	0
PFrnd	6	4	0	0	5
TCount	3	6	0	1	0
BFS	9	6	0	1	3

into a system-level template that connects them to the other shared components (as done in toolchain described in Section II). For instance, Figure 8 shows the automatically generated system for matrix multiplication with the kernel parallelized across two modules. To automatically add hardware mutexes and connect them to the different modules, we utilize the synchronization information for each kernel. While connecting the hardware mutexes, we use the performance estimates from the HLS tool to provide a higher priority to kernel variants that achieve higher performance; this ensures that the high performance variants are given priority when multiple modules contend to acquire mutex locks.

We augmented the toolchain described in Section II using these concepts to automatically generate designs that parallelize computation across multiple modules.

IV. RESULTS AND DISCUSSION

In order to illustrate the benefits of parallelizing computation across multiple modules, we select seven applications from linear algebra, signal processing and graph processing domains.

Evaluation Setup and Methodology. All the applications were written in OptiML [12]. Although OptiML is primarily a language for machine learning, it supports a rich set of datatypes and operations that are sufficient to develop these benchmark applications. The applications were compiled using our toolchain that was augmented with the ideas presented in Section III and generated into hardware designs with system clock frequency of 100MHz. The toolchain used Vivado HLS 2013.4 [3] to synthesize parallel kernels in the application into hardware modules and Vivado Design Suite [13] to connect these hardware modules within a system design template and generate the FPGA bitstream. The generated bitstreams were executed on the Xilinx VC707 development board that houses a XC7VX485T device and has 1GB of DRAM. Each application's performance was measured with hardware counters during execution and the resource consumption values are from the post-implementation reports generated by Vivado Design Suite.

Applications Benchmarks. We used the following applications to evaluate our approach:

1) Matrix Multiplication–unoptimized (**MMuopt**) is a linear algebra application that multiplies two square floating-point matrices with 250,000 elements each. This is the running example we have used throughout this paper.
2) Matrix Multiplication–optimized (**MMopt**) is a more optimized version of the matrix multiplication that buffers the columns of the second matrix and uses the buffered data to compute multiple cells of the result matrix.

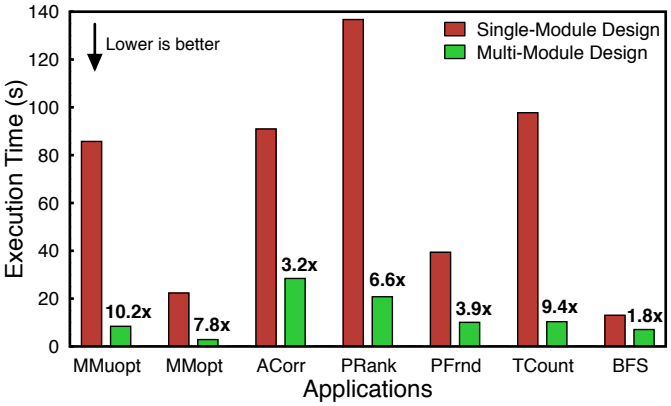

Fig. 9. The figure shows the relative performance of the designs with a single module per kernel and those with multiple modules per kernel. Since the HLS generated modules in these applications do not fully utilize the bandwidth of the system bus, parallelizing kernels across multiple modules always achieves higher performance.

TABLE II
PERCENTAGE UTILIZATION OF RESOURCE IN EACH APPLICATION

Apps.	Single Module Design (%)				Multi-Module Design (%)			
	LUTs	FFs	BRAMs	DSPs	LUTs	FFs	BRAMs	DSPs
MMuopt	13.60	6.65	14.56	1.79	82.10	37.88	59.42	21.07
MMopt	15.26	7.49	14.56	1.79	71.60	3.70	48.35	16.25
ACorr	29.62	15.97	18.45	4.07	69.42	36.77	46.50	8.00
PRank	44.36	22.35	23.79	7.57	67.08	30.56	53.20	3.39
PFrnd	63.27	31.40	18.54	14.71	80.62	34.48	51.84	4.32
TCount	51.92	28.04	13.40	14.25	89.07	34.56	58.06	2.36
BFS	65.65	29.82	32.04	0.32	73.45	33.64	63.50	1.18

3) 1-D Autocorrelation (**ACorr**) is a signal processing application that computes the autocorrelation of a 20,000-element floating-point vector.
4) PageRank (**PRank**) is a popular graph algorithm used in search engines that iteratively computes the weights of each node in the graph based on the weights of nodes in its in-neighbor set (nodes with edges leading to it). We used a graph with 1,000,000 nodes.
5) Potential Friends (**PFrnd**) uses the principle of triangle completion in graphs to recommend new connections (friends) for each node. We used a 15,000-node graph.
6) Triangle Counter (**TCount**) counts the number of triangles in a graph with 1,000,000 nodes.
7) Breadth First Search (**BFS**) computes the distance of every node in a 1,000,000-node graph from a given source node.

Since we did not have a mechanism to read data from an external source, the generation of test data (e.g., input matrices, vector and arbitrary graphs) was made a part of the application. But, this did not affect the results since the time needed for data generation is insignificant compared to total execution time. Table I lists the number of parallel kernels and the computational patterns in each application.

In order to evaluate the performance benefits of our proposed approach, we compared the performance obtained from our prior work [7] which utilized only a single hardware module for each parallel kernel (single module design) with that of our automatically generated multi-module design. We used the ILP solver discussed in Section III-D to perform design space exploration for each application and determined the

hardware configurations that maximized the performance while utilizing up to 80% of the FPGA resources. To find the optimal configurations of the single module designs, we added an extra constraint to the solver to ensure that only one variant was selected for each parallel kernel. For each application, the designs with these configurations where automatically generated using our toolchain. Performance results for these designs are shown in Figure 9 and their resource consumptions are reported in Table II.

Across all applications, as expected, the performance obtained by exploiting multi-module parallelism was better than that of the single module design. This is because the hardware modules in these single module designs had low effective parallelism, either due to irregular or strided access patterns, or due to the long latency of the computation; hence, they did not fully utilize the system-bus bandwidth. The highest performance improvement was for **MMuopt** where the performance of the single module design was limited due to the strided access pattern, as discussed in the Section I. The multi-module parallelization improved the aggregate data bandwidth to this kernel and, thereby, its performance, as shown in Figure 2; the optimizations discussed in Section III-B reduced the synchronization needed in the outer-level `map` of the matrix multiplication kernel and aided to deliver this increased performance. In **MMopt**, the overall performance of the matrix multiplication improved because buffering the columns of the second matrix reduced the total data read for the computation; yet, the multi-module design achieved better performance due to the parallelization and the resulting improvement in bandwidth utilization.

In the **ACorr**, the kernels have very regular access patterns. However, due to the long latency of the floating point operations, the data bandwidth used by the single module design was lower than the available bus bandwidth. Multi-modules designs improved the bandwidth utilization, and the optimization discussed in Section III-B enabled the most critical kernels, which contained fused `zipWith–reduce` operations, to operate in parallel with very little synchronization overhead.

In the graph applications, **PRank**, **PFrnd**, **TCount** and **BFS**, the performance improvements were again due to improved data access bandwidth to critical kernels with irregular access patterns. Additionally, the optimizations discussed in Section III-B were able to remove all the synchronization requirements for the most critical kernel in **PRank** and significantly reduced the synchronization requirements for **TCount**, improving their performances. In **PFrnd** and **BFS**, the ILP solver correctly allocated more resources to the most critical kernels in these applications and achieved $8\times$ and $3.7\times$ improvements, respectively, for these kernels. But, as a side-effect, the resources allocated to the less critical kernels reduced and their performance degraded, diminishing the overall improvement to $3.9\times$ and $1.8\times$, respectively, as seen in Figure 9. These results demonstrate the performance benefits of the multi-module parallelization approach proposed in this work.

Considering resource consumption, the exact values of resources used varied significantly due to the timing and resource optimizations done by the FPGA tool. However, as seen in Table II, the single kernel designs were not able to achieve better performance, in spite of having unused resources. In comparison, the ability to use multiple modules in our approach enabled the ILP solver to find better design points that made better use of the FPGA resources.

V. Related Work

In recent years, there has been widespread interest in providing high-level tools to make FPGAs more accessible to application developers. The most popular approach has been to develop tools that use functional specifications in a C-like language, such as C, C++, SystemC, OpenCL and CUDA, to synthesize custom hardware. Among them, tools such as Vivado HLS [3], LegUp [4] and ROCCC [14] use C, C++ or SystemC. However, extracting coarse-grain parallelism from a C program is difficult [15], hence tools such as OpenCL-to-FPGA [5] and FCUDA [16] use explicitly parallel languages, namely OpenCL [17] and CUDA [16], to design hardware. But, all these tools rely on the programmer to correctly parallelize the application and perform optimizations which often needs hardware design knowledge. In contrast, the parallel computational patterns we use are extracted from the high-level application, and their properties can be exploited to automatically infer the suitable optimizations for the abovementioned HLS tools [7], [9]. More importantly, the contribution of this work is utilizing these computational pattens to identify the parallelism and synchronization requirements as well as automating the generation of multi-module hardware designs that achieve high performance, which is not addressed by any of these efforts.

In order to correctly map the parallelism and synchronization requirements of applications on FPGA, researchers have proposed using parallel programming models, such as OpenMP [18] and Pthreads [19]. Among them, Leow et al. [20] automated the generation of hardware from OpenMP programs and Choi et al. [21] achieved the same from programs using both OpenMP and Pthreads. Efforts such as hthreads [22], Fuse [23] and ReconOS [24] use Pthreads and provide generalized operating system services to systems that support hardware and software threads. SPREAD [25] utilizes PThreads and provides an integrated solution for streaming applications. All these efforts place the tedious and error-prone task of identifying synchronization requirements and correctly parallelizing the application on the programmer. Furthermore, synchronization requirements, such as false sharing, is often not visible in the application and hard to detect without intimate knowledge of the hardware architecture. Using the properties of the computational patterns in conjunction with the system architecture, we completely automate this task and also perform optimizations to reduce synchronization requirements.

As we have seen in Section IV, our parallelization technique benefits applications that have irregular data access patterns. There has been interesting work done on improving the performance of such applications by generating hardware using deep pipelining [26] and context switching [27] techniques. The contribution of our work is automatic identification of the synchronization requirements and parallelization of computations

across multiple modules, which is not addressed by these efforts. Additionally, the techniques discussed in this work can be used in conjunction with all of these efforts to further improve the performance of applications with irregular data access patterns. Moreover, the underutilization of system data bandwidth also effects applications with regular data access patterns [28] and the parallelization technique developed in this work is also useful for such cases.

VI. Conclusions

Microsoft is introducing FPGAs in data centers and Intel is packaging FPGAs with high-end processors: there is today a unique window of opportunity for FPGAs in the general computing world. Yet, success will critically depend on the ability of application programmers to build efficient accelerators without any hardware design experience. Parallel computational patterns provide a useful design abstraction that alleviates the need for hardware design expertise and make FPGAs more accessible to these developers. But, to achieve high performance from these devices, application developers need to correctly parallelize computation across multiple modules and carefully balance the computational throughput with data bandwidth. Additionally, developers need to identify synchronization requirements in the application and build custom synchronization schemes, which is both tedious and error prone. To address these problems, we exploit the properties of the computational patterns and system architecture to infer synchronization requirements, and automatically parallelize computations across multiple modules. This results in a new ability to generate complete computing systems that can effectively utilize FPGAs to accelerate applications. Most importantly, this completely automated method liberates the application developer from *any* understanding of the hardware platform.

References

[1] P. Cooke, J. Fowers, G. Brown, and G. Stitt, "A Tradeoff Analysis of FPGAs, GPUs, and Multicores for Sliding-Window Applications," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 1, pp. 2:1–2:24, Mar. 2015.

[2] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," in *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 13–24.

[3] Xilinx, "Vivado High-Level Synthesis," http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, [Accessed: 10-Apr-2015].

[4] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, pp. 24:1–24:27, Sep. 2013. [Online]. Available: http://doi.acm.org/10.1145/2514740

[5] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From OpenCL to High-Performance Hardware on FPGAs," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2012, pp. 531–534.

[6] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen, "High-Level Synthesis: Productivity, Performance, and Software Constraints," *Journal of Electrical and Computer Engineering*, vol. 2012, p. 1, 2012.

[7] N. George, H. Lee, D. Novo, T. Rompf, K. J. Brown, A. K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne, "Hardware System Synthesis from Domain-Specific Languages," in *24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2014, pp. 1–8.

[8] W. J. Bolosky and M. L. Scott, "False Sharing and its Effect on Shared Memory Performance," in *USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, 1993, pp. 57–71.

[9] H. Lee, K. J. Brown, A. K. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun, "Implementing Domain-Specific Languages for Heterogeneous Parallel Computing," *IEEE Micro*, vol. 31, no. 5, pp. 42–53, 2011.

[10] M. Korch and T. Rauber, "A Comparison of Task Pools for Dynamic Load Balancing of Irregular Algorithms," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 1, pp. 1–47, 2004.

[11] S. Graf, M. Glas, J. Teich, and C. Lauer, "Multi-Variant-based Design Space Exploration for Automotive Embedded Systems," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE, 2014, pp. 1–6.

[12] A. Sujeeth, H. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, and K. Olukotun, "OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning," in *28th International Conference on Machine Learning (ICML)*, 2011, pp. 609–616.

[13] Xilinx, "Vivado Design Suite," http://www.xilinx.com/products/design-tools/vivado.html, [Accessed: 10-Apr-2015].

[14] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing Modular Hardware Accelerators in C with ROCCC 2.0," in *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2010, pp. 127–134.

[15] 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.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, 2011.

[16] Nvidia, "Nvidia CUDA Programming Guide," 2009.

[17] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science & Engineering*, vol. 12, no. 3, p. 66, 2010.

[18] L. Dagum and R. Menon, "OpenMP: An Industry Standard API for Shared-Memory Programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[19] B. Nichols, D. Buttlar, and J. P. Farrell, *PThreads Programming*. Sebastopol, Calif.: O'Reilly, 1996.

[20] Y. Leow, C. Ng, and W.-F. Wong, "Generating Hardware from OpenMP Programs," in *International Conference on Field-Programmable Technology (FPT)*. Ieee, 2006, pp. 73–80.

[21] J. Choi, S. Brown, and J. Anderson, "From Software Threads to Parallel Hardware in High-Level Synthesis for FPGAs," in *International Conference on Field-Programmable Technology (FPT)*. IEEE, 2013, pp. 270–277.

[22] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp, "Achieving Programming Model Abstractions For Reconfigurable Computing," *IEEE Trans. VLSI Syst.*, vol. 16, no. 1, pp. 34–44, January 2008.

[23] A. Ismail and L. Shannon, "FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators," in *IEEE 19th Annual Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2011, pp. 170–177.

[24] A. Agne, M. Happe, A. Keller, E. Lubbers, B. Plattner, M. Platzner, and C. Plessl, "ReconOS: An Operating System Approach for Reconfigurable Computing," *IEEE Micro*, vol. 34, no. 1, pp. 60–71, 2014.

[25] Y. Wang, X. Zhou, L. Wang, J. Yan, W. Luk, C. Peng, and J. Tong, "SPREAD: A Streaming-Based Partially Reconfigurable Architecture and Programming Model." *IEEE Trans. VLSI Syst.*, vol. 21, no. 12, pp. 2179–2192, 2013.

[26] R. J. Halstead and W. Najjar, "Compiled Multithreaded Data Paths on FPGAs for Dynamic Workloads," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. IEEE Press, 2013, p. 3.

[27] M. Tan, B. Liu, S. Dai, and Z. Zhang, "Multithreaded Pipeline Synthesis for Data-Parallel Kernels," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2014, pp. 718–725.

[28] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, 2015, pp. 161–170.