Hardware System Synthesis from Domain-Specific Languages

Nithin George*, HyoukJoong Lee†, David Novo*, Tiark Rompf*, Kevin J. Brown†, Arvind K. Sujeeth†, Martin Odersky*, Kunle Olukotun† and Paolo Ienne*

*Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
Email: {nithin.george, david.novo, tiark.rompf, martin.odersky, paolo.ienne}@epfl.ch

†Stanford University
Pervasive Parallelism Laboratory
Email: {hyouklee, kjbrown, asujeeth, kunle}@stanford.edu

***Abstract*—*Field Programmable Gate Arrays* (FPGAs) are very versatile devices, but their complicated programming model has stymied their widespread usage. While modern *High-Level Synthesis* (HLS) tools provide better programming models, the interface they offer is still too low-level. In order to produce good quality hardware designs with these tools, the users are forced to manually perform optimizations that demand detailed knowledge of both the application and the implementation platform. Additionally, many HLS tools only generate isolated hardware modules that the user still needs to integrate into a system design before generating the FPGA bitstream. These problems make HLS tools difficult to use for application developers who have little hardware design knowledge. To address these problems, we propose an automated methodology to generate FPGA bitstreams from high-level programs written in *Domain-Specific Languages* (DSLs). We leverage the domain-knowledge conveyed by the DSL and its domain-specific semantics to extract application parallelism, perform optimizations and also identify a suitable system-architecture for the implementation, thereby, relieving the user from most of the hardware-level details. We demonstrate the high productivity and high design quality this approach offers by automatically generating hardware systems from applications written in OptiML, a machine-learning DSL. To evaluate our methodology, we use four OptiML applications and show that we can easily generate different solutions which achieve different trade-offs between performance and area. More importantly, the results reveal that our generated hardware achieves much better performance compared to the one obtained from using the HLS tool without platform-specific optimizations.**

I. Introduction

Field Programmable Gate Arrays (FPGAs) can be configured into highly parallel, application-specific architectures that, for some applications, can offer high performance and a better energy efficiency compared to processor-oriented architectures, such as multi-core CPUs or GPUs [1]–[3]. But, their complicated programming model, the most popular one today still being to start from a *Register Transfer Level* (RTL) specification, has restricted their accessibility to application developers who have little hardware design knowledge. Today, *High-Level Synthesis* (HLS) tools provide a more convenient programming model by starting from specifications in a high-level language, like C, C++ or SystemC. However, to develop high-quality designs using these tools, the user still needs to manually perform optimizations that require a detailed knowledge of the tool, the generated hardware design and the implementation target, which prevents the adoption of these tools by application developers.

```
void(int* mem){
  mem[512] = 0;
  for(int i=0; i<512; i++)
     mem[512] += mem[i];
}
```

(a) Unoptimized HLS Program; Execution Time = 27,236 clock cycles

```
// Width of MPort = 16 * sizeof(int)
#define ChunkSize (sizeof(MPort)/sizeof(int))
#define LoopCount (512/ChunkSize)
// Maximize data width from memory
void(MPort* mem){
   // Use a local buffer and burst access
   MPort buff[LoopCount];
   memcpy(buff, mem, LoopCount);
   // Use a local variable for accumulation
   int sum=0;
   for(int i=1; i<LoopCount; i++){
   // Use additional directives where useful
   // e.g. pipeline and unroll for parallel exec.
   #pragma PIPELINE
      for(int j=0; j<ChunkSize; j++){
      #pragma UNROLL
         sum+=(int)(buff[i]>>j*sizeof(int)*8);}}
    mem[512]=sum;
}
```

(b) Optimized HLS Program; Execution Time = 302 clock cycles

```
// Here, data_array is an array of 512 integers.
// sum adds its elements and the stores back the result
val result = data_array.sum()
```

(c) DSL Program; Execution Time = 368 clock cycles

Fig. 1. Comparing optimized HLS, unoptimized HLS and DSL specifications. All three programs produce hardware to perform the same computation. The optimized specification leverages on a detailed knowledge of the HLS tool, the resulting hardware and the implementation platform while performing optimizations to improve the performance. The DSL code is much simpler to express and yet provides comparable performance.

To illustrate this, consider designing a simple hardware unit to add 512 integers held in an external memory and store back the result. This can be synthesized using an HLS tool from the C++ program shown in Figure 1a. But, by not specifying the details of the underlying system architecture, such as the maximum data-width of the memory interface, the communication modes on this interface (burst mode vs. individual accesses) and the available parallelism in each data word from memory, and by not using the features in the HLS tool to exploit this parallelism, the generated hardware is extremely inefficient. Hardware generated from this code with the highest level of optimization using Vivado HLS (2013.4) [4] needed 27,236 clock cycles on our test platform to complete the computation. To achieve good performance, the developer needs to consider all these aspects and write the more complex program shown in Figure 1b, which generates a hardware to complete the same task in 302 clock cycles. Moreover, many HLS tools will only synthesize these programs into IP modules and not handle their external connections to board-level interfaces and peripherals, like the instantiation of the memory controller and the connection to the external memory in our example. This forces application developers to make these connections manually and sometimes even generate the additional clock and control signals needed for this IP module to obtain a complete design. So, while HLS tools are capable of generating good quality designs, in practice they are difficult to use for application developers who often lack the necessary hardware design skills.

In order to overcome these limitations, we propose an automated methodology to generate complete hardware systems from programs written in a high-level *Domain-Specific Language* (DSL). In this methodology, we leverage the awareness of the application domain obtained through the DSL and its domain-specific semantics to perform optimizations as well as to map the domain operations into a set of structured computation patterns, such as `map`, `reduce`, `foreach` and `zipwith`. Since these computation patterns are few and well understood, we can have premeditated strategies to optimize them and generate high-quality hardware modules to implement them. For instance, the DSL code in Figure 1c is simple to express for the application developer and it will be mapped to a `reduce` pattern. This enables us to automatically generate a program that is similar to Figure 1b, and, thereby, obtain a hardware module to perform the same computation in 368 clock cycles[1]. Furthermore, since the scope of each DSL is limited to a specific application domain, we can have a set of predefined system-architecture templates that are suitable for applications in that domain. By utilizing these templates, we can autonomously interconnect the different hardware modules in the application, generate necessary control signals, and obtain a complete design that is ready for FPGA bitstream generation, thereby, saving the application developers from having to meddle with hardware level details.

The rest of the paper will discuss the details of this proposed methodology, starting with a general overview in Section II. We will take a look at the compiler infrastructure we use in Section III and then discuss our hardware generation process in Section IV along with some of the optimizations we perform to obtain high-performance designs. In Section V, we will first evaluate the quality of our generated computation patterns using a set of micro-benchmarks and then assess the overall effectiveness of our flow using four applications written in OptiML [5], a high-level DSL for machine learning. The results reveal that our optimizations significantly improve the performance of the generated hardware designs. Furthermore, comparison with a laptop CPU shows that the generated hardware achieves reasonable performance and a better energy efficiency. We discuss related work in Section VI and Section VII concludes the paper.

[1]The performance of the automatically generated module is slightly lower compared to the one obtained from Figure 1b because it is more generic and designed for handling reductions of larger sizes.

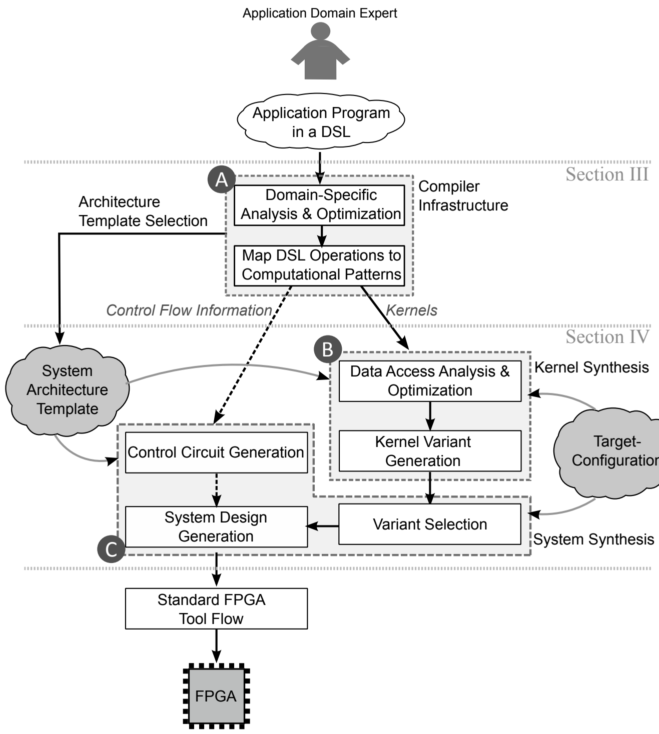

Application Domain Expert
Application Program in a DSL
Section III
A
Domain-Specific Analysis & Optimization
Compiler Infrastructure
Architecture Template Selection
Map DSL Operations to Computational Patterns
Control Flow Information
Kernels
Section IV
B
System Architecture Template
Data Access Analysis & Optimization
Kernel Synthesis
Kernel Variant Generation
Target-Configuration
Control Circuit Generation
System Design Generation
Variant Selection
System Synthesis
C
Standard FPGA Tool Flow
FPGA

Fig. 2. Overview of the methodology. This figure illustrates how the high-level specifications in a DSL is compiled and then automatically transformed into a hardware system that can be implemented on an FPGA.

II. Overview of the Methodology

Our automated methodology accepts an application program written in a high-level DSL and generates a complete hardware design that can be directly programmed on the target FPGA. Figure 2 illustrates the different steps in this process. In this methodology, the high-level DSL used by the application developer offers both high productivity and shields the developer from the hardware-level details. Our *compiler infrastructure*, indicated as *A* in the figure, optimizes programs written in this DSL by performing both general purpose optimizations, such as common subexpression elimination and dead-code elimination, and domain-specific optimizations, such as applying linear algebra simplification rules, based on the domain of the DSL. In our approach, this domain-knowledge is also utilized to select a suitable *system-architecture template* for the final hardware implementation. This template, which takes inspiration from the platform designs used by Cong et al. [6], defines how the various hardware components in the final design will be interconnected. After optimization, the compiler maps the operations in the application program into serial and parallel computational patterns, called *kernels*, and represents this program as a dependency-graph between these kernels.

The hardware generation starts with *kernel-synthesis*, marked as *B* in the figure, that takes the kernels in the application program and generates isolated hardware units to perform the equivalent computation. As shown in the figure, this step gets information about the interfaces to the kernels and the shared components in the system from the system-architecture template and the *target-configuration* provides it with additional details specific to the FPGA used as the implementation target. For each kernel, the kernel-synthesis step produces multiple hardware implementations, which we call *variants*, that achieve different trade-offs between area and performance. Generating multiple variants is essential in order to enable the subsequent system-synthesis step to compose a system design that will achieve good performance and fit within the resource constraints of the chosen FPGA. *System-synthesis*, indicated as *C* in the figure, uses the information from the system-architecture template to assemble the complete system design. During this step, the program's dependency-graph from the compiler is used to generate the control circuitry for the system, and the target information is used to know the capabilities of the chosen FPGA to ensure that the generated design can be implemented on it. After the system-synthesis generates the final system design, a standard FPGA tool flow is used to generate the bitstream and program the FPGA.

III. Compiler Infrastructure

We implemented our compiler infrastructure, indicated as *A* in Figure 2, by extending the Delite [7] compiler framework. Delite is an extensible compiler framework to easily develop DSLs targeting heterogeneous systems. The main idea of Delite is to provide DSL developers with a set of structured computation patterns and data structures that can be extended to implement domain operations and data structures. Delite currently supports computation patterns such as `map`, `reduce`, `zipwith`, `foreach`, `filter`, `group-by`, `sort` and `serial`[2] and data structures such as scalar datatypes, `array`, `struct` and `hashmap`. The structured nature of Delite components enables parallelizing and optimizing DSL programs for different architectures such as multi-core CPUs or GPUs. Additionally, DSL developers using Delite can easily add domain-specific optimizations that leverage the domain-knowledge and automatically get generic optimizations such as loop fusion and data structure transformations that further improve the performance. The output of compiling a DSL program with Delite are a set of computation pattens (kernels) generated for the target architecture and a dependency-graph between these kernels.

In order to generate hardware from this DSL program, we generated hardware units to implement the kernels found in the program and used the dependency-graph to generate a controller that guarantees that these kernels are executed in a valid order. We will discuss this in detail in Section IV. In order to demonstrate our approach, we use applications written in OptiML [5], a machine learning DSL implemented using Delite. Among the Delite components OptiML supports, we currently limit ourselves to the most widely used computation patterns (`map`, `reduce`, `foreach` and `zipwith`) and data structures (scalar datatypes and `array`). Since Delite allows composing the components in different ways, interesting applications can still be written with the limited set we currently support for FPGA. One thing to note, however, is that our approach is not limited to OptiML and can be applied to other DSLs using Delite or a similar compiler infrastructure.

IV. Hardware Generation

In this section, we will discuss how the kernels extracted from the application and its associated dependency-graph are automatically transformed into a hardware design.

[2]`serial` is for non-structured computations that cannot be parallelized.

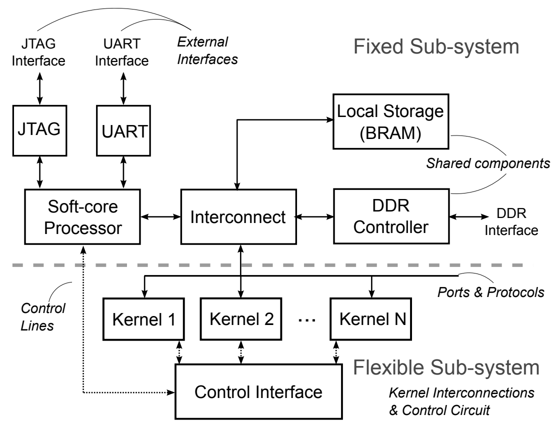


Fig. 3. This figure shows the system-architecture template used for hardware systems generated from OptiML applications. The annotations in the figure are some of the specific details this template provides to the hardware generation process.

A. System Architecture Template

The hardware design generation starts with the selection of a system-architecture template by the compiler. This template delineates the various components and interfaces available in the final system as well as their interconnections. Hence, it provides a general outline based on which the hardware for the application will be generated. Each template is composed of two parts: the fixed subsystem, which remains constant for every application using that template, and the flexible subsystem, which changes from one application to another. Figure 3 shows the system-architecture template that is used for all OptiML applications. Here, the fixed subsystem, is composed of components that include a soft-core processor, on-chip memory, external memory controller, and connections to the DRAM, UART and JTAG interfaces. The flexible subsystem of this template defines how the kernels in each application will be connected to the rest of the system and how the control infrastructure will be generated. Although we currently use this single system-architecture template for all OptiML applications, the methodology supports having multiple templates and then the selection of the specific template is up to the DSL developer.

B. Kernel Synthesis

The kernel-synthesis step generates hardware modules that implement the kernels (computation patterns) in the application; it is marked as *B* in Figure 2. As shown in the figure, the inputs to this step are the list of kernels from the compiler, and the information about the hardware system to design from the system-architecture template and the target-configuration. The system-architecture template provides details, such as the number of data-ports to each kernel, communication protocols on these ports, shared memories in the design and address range for these memories, that are necessary to ensure that the synthesized kernels will function correctly and can be easily integrated into the final design. The target-configuration specifies additional details, such as the sizes of the available memories, the width of the port, that help to tailor the generated kernels to the specific FPGA used in the implementation.

OptiML applications contain multiple serial and parallel kernels. Among them, the serial kernels offer only limited opportunities for acceleration using custom hardware. Therefore, we map all the serial kernels in the application to the soft-core processor in order to share implementation resources among them.

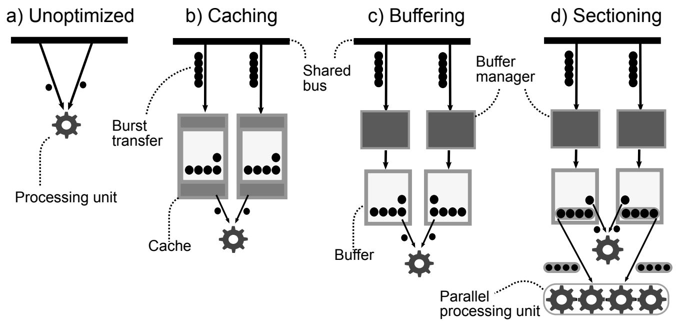


Fig. 4. Optimizations applied to the kernels. The unoptimized kernel access each data element separately and, therefore, has a low effective bandwidth to the data (a). When accesses are sequential, we can improve this bandwidth using burst transfers by adding a local cache (b) or by using local buffer and a buffer manager (c). To benefit from this improved bandwidth, we need to generate the kernel differently to correctly leverage the available data parallelism (d).

The parallel kernels, however, can benefit greatly from custom hardware that can aptly exploit the available parallelism. Moreover, in typical OptiML applications, these parallel kernels dominate the overall execution time. Since these parallel kernels are generated from a limited number of computational patterns used in the compiler, we can have premeditated strategies to generate high-quality hardware implementation for them. Although the set of computational patterns are small, actual computation and data access properties in the kernels will still vary significantly from one application to another. Additionally, in the kernel, these patterns can be nested within one another or fused together after the optimizations done by Delite. Due to this variability, we cannot use fixed templates for each pattern. Instead, we perform compiler analysis and optimizations to produce code for an HLS tool that will generate a high-quality design.

Kernel Optimization. For the parallel kernels we synthesize, we can apply generic optimizations, such as loop-unrolling and loop-pipelining, to generate parallel hardware. But, to generate high-performance designs, we need to analyze the properties of the individual kernels and perform additional optimizations. For instance, consider this single line of an OptiML program `z = (x + y).sum()`. In this program, `x` and `y` are large `array` data structures stored in the external memory and the hardware generated from this line will read these data-structures sequentially, sum their elements and store the result as `z`. Since the data in `x` and `y` are accessed over a shared bus, each access entails overhead due to the bus protocol and the latency of the memory. However, since the kernel accesses these data structures sequentially, we can use burst communication to reduce this overhead. To implement this, we first analyze the data access pattern of each data structure in the program and add a local cache to those that are accessed sequentially, `x` and `y` in this example. Now, data requests from/to these data-structures are served from the local cache and, in the event of cache-miss, the cache is filled/flushed using burst-transfers from/to the memory; this is depicted in Figure 4b.

However, having to check for cache hit/miss on every access will incur some performance overheads. To avoid this cache lookup overhead, we can perform a more detailed analysis of the access patterns, and use this information to replace the cache with a simple local buffer and an associated buffer manager, as shown in Figure 4c. The difference between the two is that buffer manager *knows* the access pattern to the data structure and uses this information to move the data from/to the buffer without checking on each access. In our example, the buffer manager systematically transfers chunks of data from `x` and `y` into the local buffers and makes it available for the computation.

We will still not get the most out of this higher data bandwidth by only depending on optimization directives in the HLS, like loop-unrolling or loop-pipelining. To obtain better throughputs, in addition to using these directives, we need refactor the kernel computation into multiple units that are specialized to exploit different amounts of parallelism in the data; this is the same as the loop-sectioning optimization done for SIMD processors. After applying this optimization, as shown in Figure 4d, the generated hardware uses dedicated parallel processing units when the input data has sufficient parallelism to improve the overall processing throughput. In our example, the parallel processing unit will read small blocks of data from the local buffers of `x` and `y`, and use a balanced reduction tree to compute the result.

All the discussed optimizations serve to improve the performance of the hardware kernels, but they also consume more resources. To implement an application on the FPGA, we need to ensure that all its constituent kernels will fit within the limited resources on the device. To achieve this, during kernel-synthesis, we generate multiple hardware variants for each kernel in the application by applying these optimizations in different combinations. The selection of the specific variant to use in the final design is deferred until the system-synthesis step.

Data management. Hardware kernels generated from OptiML use scalar datatypes or `array` data structures to exchange data with each other. Among them, the scalar datatypes have well known sizes and the space needed to store them is much lesser than the available on-chip memory capacity. So, they are statically allocated in a shared on-chip memory to reduce the data-access latency and make it easy to access this data from the different kernels. In the case of the `array` data structure, it is internally composed of a small amount of metadata and a contiguous block of raw-data. The size of this metadata is fixed and known at compile-time and, therefore, it is statically allocated in the on-chip memory. The raw-data, however, is typically very large and its size is known only during the runtime. Therefore, it needs to be dynamically allocated in the larger shared external memory during the application execution. To perform dynamic allocation, this external memory is managed as a single circular buffer with fresh allocations happening from the tail of this buffer and deallocation from its head. To dynamically allocate memory from the kernels, the *tail-pointer*, which is the current address of the tail of this buffer and always points to first free location in the memory, is stored in the shared on-chip memory. During application run, each kernel allocates memory starting from the current value of this tail-pointer and updates its value. Memory deallocation is handled by the control circuit that is aware of the full context of the executing application and, therefore, knows when the data structures can be safely deallocated. Dynamic memory management is covered in more detail in the following subsection, when discussing the control circuit generation.

C. System Synthesis

The system-synthesis step, indicated as *C* in Figure 2, has three responsibilities: (1) selecting the kernel variants to use in the design, (2) interconnecting them to the other system components and interfaces, and (3) generating the control circuitry to complete the hardware design. The system-architecture template, which for OptiML applications is shown in Figure 3, provides this step with the information needed to generate the fixed subsystem and the strategy for generating the flexible subsystem. The target-configuration contains information about the resources on the target (e.g. LUTs, DSP blocks, BRAMs), the target-specific IP modules (e.g. DRAM controller, on-chip memory, soft-core processor) to use as well as their individual configuration parameters. As noted earlier, the kernel-synthesis generates multiple variants for each kernel and the system-synthesis selects the specific variant to use in the generated hardware system. Ideally, this variant selection is perfomed using a cost model that considers the estimate of performance and area overhead for each variant. However, our current implementation still lacks this feature and always selects the highest performing variants. As an interim solution, we provide an option to override this selection by supplying additional parameters to this step.

Control Circuit Generation. The control circuit for the hardware design is automatically generated from the dependency-graph of the application obtained from the compiler. This circuit performs two essential tasks: scheduling the kernel execution and freeing the dynamically allocated memory. To schedule the kernel execution, the dependency-graph of the application is analyzed to determine the control and data dependencies among the kernels. Based on this information, the generated schedule tries to maximize the performance by executing multiple kernels in parallel, when possible. In order to free the dynamically allocated memory, during the application execution, the control circuit keeps track of the amount of memory used by each kernel. To achieve this, since the dynamically allocated memory is managed as a circular buffer, the control-circuit keeps track of the updates to the *tail-pointer*, which is the shared pointer that points to the first free location in the memory. By knowing how the value of the *tail-pointer* changes after each kernel execution, the control circuit can determine the block of memory used for storing that kernel's data. Additionally, by analyzing the data dependencies in the application's dependency-graph, the lifetime of this data can be determined. With this information, the schedule for freeing memory blocks is statically determined and generated into the control circuit. Moreover, since the control circuit is always aware of the amount of memory used by the application, it can detect memory overflows and terminate the application execution. For OptiML applications, this control circuit is implemented using the soft-core processor that executes the sequential kernels. Hence, the complete program for this soft-core processor is generated during the system generation step. To control the kernel execution from this processor, the system-architecture for OptiML applications, shown in Figure 3, includes a control interface module in its flexible subsystem.

V. Results and Discussion

In this section, we will first evaluate the benefits of the optimizations discussed in Section IV. Then, we will use four OptiML applications to illustrate the range of solutions, with different performance and resource utilizations, we can generate easily from our tool. Additionally, to provide a broader perspective on the quality of the generated hardware systems, we will compare their performance and energy efficiency with running the same applications on a laptop CPU.

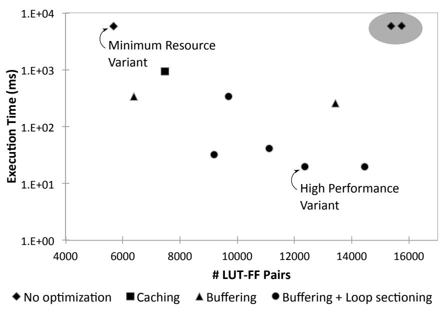
Evaluation Setup and Methodology. All the hardware designs generated using the proposed methodology were written as OptiML applications. These were compiled using the Delite compiler which we modified for the purpose of generating hardware. During the compilation, the kernel-synthesis step generates multiple variants for each parallel pattern in the application that are then synthesized into hardware using Vivado HLS (2013.4) [4]. The system-synthesis step then automatically generates the final hardware design using specific kernels variants that we select. This step also generates the code for the soft-core processor in the design that executes the sequential kernels and controls the overall hardware execution flow. To measure the performance of the generated designs, they were synthesized into a bitstream using Xilinx Vivado Design Suite [8] and implemented on a Xilinx VC709 development board that houses a XC7VX690T FPGA that was fabricated with a 28nm process. It is worth noting that due to addressing limitations of the 32-bit soft-core processor in the design, only 2GB out of the total DRAM on the board was available to the hardware design. The hardware energy consumption values reported are based on the worst-case estimates obtained from the Vivado Design Suite. It includes both the static and dynamic energy consumed in all the components implemented on the FPGA, like the kernels in the application, the soft-core processor, local memory and the DRAM controller.

To measure the performance on the CPU, we manually implemented each application in C++ and used OpenMP API [9] for multi-threaded execution. The execution time and energy consumption were then measured by running these applications on an Intel Sandy Bridge Core i7-2620M laptop CPU running at 2.7GHz and fabricated with a 32nm process. The energy consumption was measured using LIKWID performance tool [10] and it includes the energy expended in the CPU package, which also includes the on-chip DRAM controller.

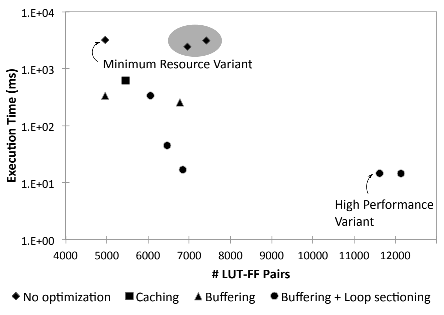
Micro-benchmarks. To understand the benefit of the proposed optimizations, we consider three patterns (`map`, `reduce` and `foreach`) separately and study how their performance and resource usage are affected by these optimizations[3]. For each case, we additionally consider the different alternatives that can be generated by applying generic optimizations, like loop-unrolling and loop-pipelining, to see how this would impact the results. Since the proposed optimizations only target sequentially accessed data structures, we use an array of 10 million integers which is accessed sequentially from each of these patterns. In the `map` and `foreach` patterns, we increment the elements in this array by a constant value and store them back into the memory[4]. In the case of the reduce pattern, we add up all the elements in the array to produce a single value which is then stored into the memory. Figure 5 reports the results from this evaluation. In all three cases, the hardware generated without applying any optimization needs the least resources, but also has lowest performance. More importantly, the data-points in the gray regions of the figure show that using generic optimizations like loop-unrolling and loop-pipelining on this unoptimized version has little effect on the performance. This is because applying these optimizations creates parallel processing resources, but without additional information the HLS tool performs each memory accesses sequentially providing no performance improvement in the `map` and the `foreach` and only minor improvement in the `reduce`. This demonstrates that blindly applying generic optimizations without considering the hardware-level details may not produce better results. The results further show that adding a local cache to each sequentially accessed data structure and using burst transfers helps to improve the performance significantly.

[3]We do not consider the `zipwith` pattern separately because it is also implemented in hardware using the `map` pattern

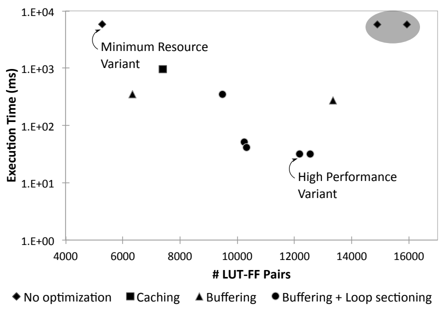
[4]The difference between these patterns this is that `map` creates a new array as output while `foreach` overwrites an already existing one



(a) Map



(b) Reduce



(c) Foreach

Fig. 5. Performance-area trade-off between the different variants. The proposed optimizations progressively improve the performance of the generated kernels, but they also consume additional resources. In all cases, the local buffer + loop-sectioning variant has the highest performance and the unoptimized variant uses the lowest area. The data-points in the gray region reveal that using general-purpose optimizations (loop-unrolling, loop-pipelining) alone does not yield substantial improvement.

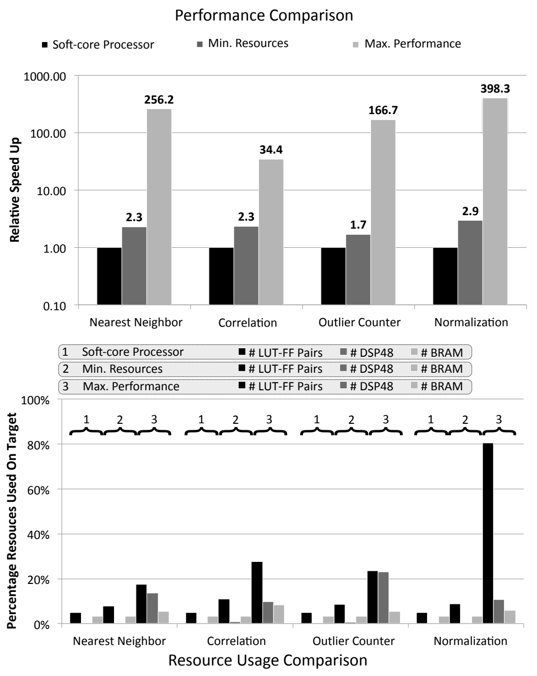


Fig. 6. The performance and resource utilization of the minimum area design and the highest performing design are compared to illustrate the range of designs satisfying different performance-area trade-offs the tool can produce. The designs employing only the soft-core processor is used as reference to understand the relative speeds and area overheads of the automatically generated designs.

But, by using a local buffer along with a buffer manager instead of the cache, we can achieve even better performance with less resources. The performance improves from avoiding the cache lookup overhead on each access, and the resource usage is lowered because the buffer manager is simpler to implement, due to its statically determined memory transfers, compared to the dynamic cache control logic. The highest performance in all three cases is obtained when the local buffer and buffer manager are used in conjunction with the loop-sectioning optimization, which achieves between 180 to 290 times speedup over the unoptimized case. This is because the loop-sectioning creates blocks with fixed amounts of parallelism which the HLS tools can schedule more optimally and, thereby, obtain higher performance.

Application Benchmarks. We performed the full system evaluations using the following four OptiML applications that utilize the features we currently support for hardware generation.

- Nearest Neighbor application finds from a set of data-points the one that is closest to a given point.

TABLE I. DATA-TYPE, COMPUTATION PATTERNS AND HARDWARE KERNEL COUNT IN EACH APPLICATION

Application	Datatype	Map/ Zipwith[3]	Reduce	Foreach	Hardware Kernels
Nearest Neighbor	integer	3	1	0	2
1-D Correlation	float	8	5	0	4
Outlier Counter	integer	4	2	0	2
1-D Normalization	integer	1	2	1	3

- Outlier Counter application uses the criterion proposed by Knorr and Ng [11] to count outliers in a given data-set.
- 1-D Correlation computes the cross-correlation between two large data-sets.
- 1-D Normalization applies a linear transforms to a given data-set to fit it within a provided upper and lower bounds.

Table I lists the datatype, the parallel patterns and number of separate hardware kernels in each of these applications. The number of hardware kernels is always lower than the number of computational patterns since Delite's optimizations fuse multiple independent patterns into a single more complex kernel.

To illustrate the range of solutions we can generate, for each application we generate two specific hardware implementations: one that uses the least resources and another that achieves the highest performance. In order to pick the kernel variants for these implementation, we use the insights obtained from Figure 5 and provide parameters to the system-generation step to select variants with the lowest resource usage for one and the those producing the highest performance for the other. In the case of the high-performance design, however, for some applications we had to use a lower performance variants when the HLS tool failed to generate the high performance variants meeting the timing/resource constraints of the target FPGA. Figure 6 compares these two implementations. To serve as reference, we used a manually developed design that executes the same application only using the soft-core processor. The results show that the design using the lowest resources is between 1.7 to 2.9 times faster than the processor-based implementation while using slightly less than twice the amount of resources. Additionally, from Figure 5, we see that all the minimum area variants are generated without any additional optimizations. Therefore, this design also represents the performance one can expect by naïvely using the HLS tool without any additional optimizations. The high-performance solution, on the other hand, can achieve between 34 to 398 times the soft-core processor's performance, albeit by using more resources. Comparing across applications, the highest speed-up is obtained in the nearest-neighbor and normalization applications which uses fixed point representation and have simple kernels that enabled us fit all the highest performing kernel variants on

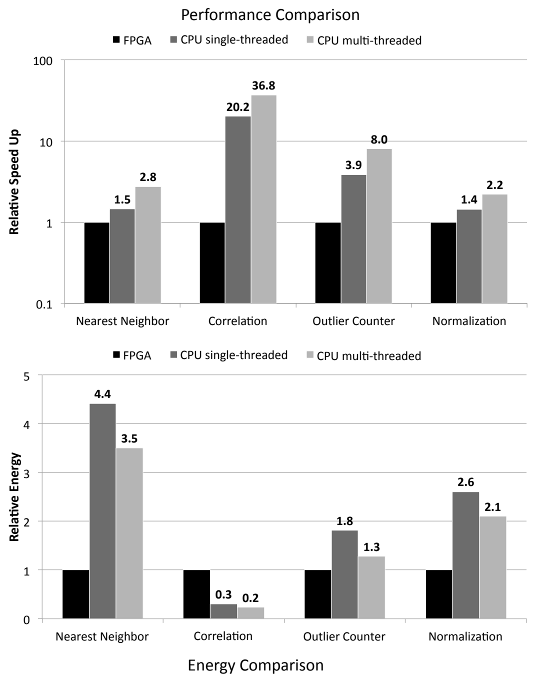


Fig. 7. The CPU achieves better execution performance compared to the FPGA. However, considering energy efficiency, the FPGA outperforms the CPU in all applications dominated by fixed-point computation.

the FPGA. The lowest speed-up is for the correlation application that performs floating-point computations which are not very efficient to implement on an FPGA. These results show that our methodology can easily generate a variety of different solutions from a high-level DSL program that achieve different trade-offs between performance and resource usage. More importantly, it demonstrates that by decomposing the application into a set of well understood computation patterns, we can generate high-performance hardware systems.

To provide an insight into performance and energy efficiency of the generated solutions, we compare the highest performing design with a laptop CPU. The results on Figure 7 shows that when we use the highest-performing variants, like in the case of nearest neighbor and normalization, FPGAs execution performance is quite close to the single-threaded execution on the laptop CPU. This performance gap increases in the outlier counter application because of using lower performing variants, and in the correlation application due to the low performance of floating point core in the FPGA. In terms of energy efficiency, however, the FPGA clearly outperforms the CPU on all applications that use fixed point computation. The figure also shows that multi-threaded execution can improve the execution performance and the energy efficiency of the CPU, but the FPGA is still the more energy-efficient platform for the fixed point computations.

To understand the performance difference, we need to take a closer look at the applications we used for the evaluation. All these applications have a low computational complexity and process large amounts of data that is stored in the external DRAM. Therefore, its execution performance on any platform depends greatly on the platform's effective memory bandwidth. The maximum memory bandwidth available to the CPU is 21.3GB/s, compared to the theoretical maximum of 8.4GB/s available to the hardware design. This along with the highly tuned memory architecture of the CPU, enables it to have a higher effective bandwidth and, thereby, achieve better performance. Moreover, in the hardware designs, the DRAM controller also accounts for between 40% to 70% of the total energy consumption in the system. On the CPU, this controller is implemented using as an ASIC and, therefore, is much more energy-efficient. This suggests that implementing this DRAM controller as an hard IP will enable FPGAs to achieve a better performance and a much higher energy efficiency for such applications.

VI. Related Work

There has been widespread interest in providing high-level tools to program FPGAs. The most popular approach is to take high-level specifications in a C-like language, like C, C++ or SystemC, and synthesize hardware. These include tools such as Vivado HLS [4], Catapult [12], Gaut [13], LegUp [14] and Trident [15], to name just a few. Since extracting coarse-grain parallelism from a C program is often difficult [6], researchers have also used explicitly parallel C-like languages, such as OpenCL [16] and CUDA [17] to design hardware. Notable efforts here include OpenCL-to-FPGA [18], SOpenCL [19] and FCUDA [20]. But, these tools are all too low level and rely on the programmer to explicitly specify details that affect quality and performance of the generated hardware. This makes them difficult to use for application developers who have little hardware design knowledge. Furthermore, many of these tools only produce IP modules that need to be integrated into a system-level design before it can be implemented on an FPGA. In contrast, our approach focuses on providing a high-level DSL that is intuitive to use for the application developers, shields them from much of the hardware-level details and produces complete hardware designs that are ready to implement on the FPGA. Moreover, our approach is complementary to these efforts which is clearly evident since we use Vivado HLS for the low-level hardware synthesis.

Since there are well known deficiencies in using C-like languages for expressing hardware circuits [21], researchers have also proposed using languages such as Lime [22], Bluespec Verilog [23] or Chisel [24]. Among them, Bluespec Verilog and Chisel are essentially hardware design languages and Lime is a general purpose language that can also target FPGAs. These approaches, hence, do not leverage any application domain knowledge to perform optimization that can help to produce better quality hardware designs [25]. Our proposal leverages this knowledge and uses well understood computational patterns to make designing hardware more accessible to application developers.

There are also tools that use a DSL and the domain-knowledge to design hardware, such as PARO [26] which uses loop transformations to implement highly parallel systems; Spiral [27] which synthesizes linear transforms in signal processing applications into hardware; Optimus [28] which focuses on streaming applications; and HDL coder [29] which synthesizes hardware from Matlab/Simulink systems. We present a methodology that is more general and relies on using well known computational patterns that enables us to have premeditated optimization strategies to generate hardware. Although we illustrated this methodology using OptiML applications, it can be applied to other domains as well. But, using computational patterns alone is not a new idea. In the past, researchers have used design patterns to target reconfigurable architectures [30] and also to analyze the amenability of algorithm acceleration on such platforms [31]. Our approach, however, generates a complete hardware system starting from a DSL-program, thereby, sparing the application developer from low-level hardware details. To achieve this, we the extract our computational patterns (kernels) directly from the DSL-program and perform additional optimizations to implement them efficiently on the FPGA. While generating the hardware system, we automatically partition the application logic, using a soft-core processor to execute the serial parts and custom hardware for the parallel parts, to utilize the FPGA resources where they serve best.

VII. Conclusions

FPGAs are very versatile devices, capable of delivering high performance along with high energy efficiency. Therefore, they are bound to play a vital role in modern computational ecosystems that have tight energy budgets. The problem has been to make them more easily usable for application developers who typically have no knowledge about designing hardware. To solve this, we proposed an automated methodology that transforms applications written in high-level DSLs into high-quality hardware designs. The DSL specification is intuitive to express for application developers and also shields them from the hardware details. By representing these applications using well understood computational patterns, we showed that we can perform optimizations to obtain high-performance hardware designs. Our results show that our proposed optimizations are effective and that our approach can produce complete hardware designs to target FPGAs. We also show that for fixed point computations these designs can offer a better energy efficiency compared to a laptop CPU. This methodology can thus make the benefits of reconfigurable technology easily accessible for application developers in different domains.

References

[1] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-Window Applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, 2012, pp. 47–56.

[2] S. Kestur, J. D. Davis, and O. Williams, "Blas Comparison on FPGA, CPU and GPU," in *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*. IEEE, 2010, pp. 288–293.

[3] B. Betkaoui, D. B. Thomas, and W. Luk, "Comparing Performance and Energy Efficiency of FPGAs and GPUs for High Productivity Computing," in *Field-Programmable Technology (FPT), 2010 International Conference on*. IEEE, 2010, pp. 94–101.

[4] Xilinx, "Vivado High-Level Synthesis," http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/index.htm, [Accessed: 26-June-2014].

[5] 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 *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 609–616.

[6] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473–491, 2011.

[7] 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.

[8] Xilinx, "Vivado Design Suite," http://www.xilinx.com/products/design-tools/vivado/index.html, [Accessed: 26-June-2014].

[9] 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.

[10] J. Treibig, G. Hager, and G. Wellein, "LIKWID: Lightweight Performance Tools," in *Competence in High Performance Computing (CiHPC) 2010*. Springer, 2012, pp. 165–175.

[11] E. M. Knorr and R. T. Ng, "A Unified Notion of Outliers: Properties and Computation." in *KDD*, 1997, pp. 219–222.

[12] Calypto, "Catapult," http://calypto.com/en/products/catapult/overview, [Accessed: 26-June-2014].

[13] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin, "GAUT: A High-Level Synthesis Tool for DSP Applications," in *High-Level Synthesis*. Springer, 2008, pp. 147–169.

[14] 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," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 2011, pp. 33–36.

[15] J. L. Tripp, M. B. Gokhale, and K. D. Peterson, "Trident: From High-Level Language to Hardware Circuitry," *Computer*, vol. 40, no. 3, pp. 28–37, 2007.

[16] 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.

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

[18] 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 *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE, 2012, pp. 531–534.

[19] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos, "Synthesis of Platform Architectures from OpenCL Programs," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. IEEE, 2011, pp. 186–193.

[20] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. Hwu, "FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs," in *Application Specific Processors, 2009. SASP'09. IEEE 7th Symposium on*. IEEE, 2009, pp. 35–42.

[21] S. A. Edwards, "The Challenges of Synthesizing Hardware from C-like Languages," *Design & Test of Computers, IEEE*, vol. 23, no. 5, pp. 375–386, 2006.

[22] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: a Java-Compatible and Synthesizable Language for Heterogeneous Architectures," *ACM Sigplan Notices*, vol. 45, no. 10, pp. 89–108, 2010.

[23] R. Nikhil, "Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications," in *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*. IEEE, 2004, pp. 69–70.

[24] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing Hardware in a Scala Embedded Language," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 1216–1225.

[25] N. George, D. Novo, T. Rompf, M. Odersky, and P. Ienne, "Making Domain-Specific Hardware Synthesis Tools Cost-Efficient," in *Field-Programmable Technology (FPT), 2013 International Conference on*. IEEE, 2013, pp. 120–127.

[26] F. Hannig, H. Ruckdeschel, H. Dutta, and J. Teich, "PARO: Synthesis of Hardware Accelerators for Multi-Dimensional Dataflow-Intensive Applications," in *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2008, pp. 287–293.

[27] P. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Computer Generation of Hardware for Linear Digital Signal Processing Transforms," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 17, no. 2, p. 15, 2012.

[28] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah, "Optimus: Efficient Realization of Streaming Applications on FPGAs," in *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM, 2008, pp. 41–50.

[29] MathWorks, "HDL Coder," http://www.mathworks.com/products/hdl-coder/, [Accessed: 26-June-2014].

[30] A. DeHon, J. Adams, M. DeLorimier, N. Kapre, Y. Matsuda, H. Naeimi, M. Vanier, and M. Wrighton, "Design Patterns for Reconfigurable Computing," in *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*. IEEE, 2004, pp. 13–23.

[31] K. Nagarajan, B. Holland, A. D. George, K. C. Slatton, and H. Lam, "Accelerating Machine-Learning Algorithms on FPGAs Using Pattern-Based Decomposition," *Journal of Signal Processing Systems*, vol. 62, no. 1, pp. 43–63, 2011.