2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

An Architectural Framework for Accelerating Dynamic Parallel Algorithms on Reconfigurable Hardware

Tao Chen, Shreesha Srinath, Christopher Batten and G. Edward Suh
Cornell University
Ithaca, NY 14850, USA
{tc466, ss2783, cbatten, gs272}@cornell.edu

***Abstract*—In this paper, we propose ParallelXL, an architectural framework for building application-specific parallel accelerators with low manual effort. The framework introduces a task-based computation model with explicit continuation passing to support dynamic parallelism in addition to static parallelism. In contrast, today's high-level design frameworks for accelerators focus on static data-level or thread-level parallelism that can be identified and scheduled at design time. To realize the new computation model, we develop an accelerator architecture that efficiently handles dynamic task generation and scheduling as well as load balancing through work stealing. The architecture is general enough to support many dynamic parallel constructs such as fork-join, data-dependent task spawning, and arbitrary nesting and recursion of tasks, as well as static parallel patterns. We also introduce a design methodology that includes an architectural template that allows easily creating parallel accelerators from high-level descriptions. The proposed framework is studied through an FPGA prototype as well as detailed simulations. Evaluation results show that the framework can generate high-performance accelerators targeting FPGAs for a wide range of parallel algorithms and achieve an average of 4.0x speedup over an eight-core out-of-order processor (24.1x over a single core), while being 11.8x more energy efficient.**

I. Introduction

As the technology scaling slows down, computing systems need to rely increasingly on hardware accelerators to improve performance and energy efficiency. In particular, field-programmable gate-arrays (FPGAs) are starting to be deployed as a general-purpose acceleration platform, and have been shown to improve performance and/or energy efficiency for many applications. FPGAs are also becoming more widely available (e.g., through the cloud [1]), and increasingly integrated with general-purpose cores either through inter-socket interconnect (e.g., Intel HARP [2], IBM CAPI [3]), or directly on-chip (e.g., Xilinx Zynq SoCs [4], Intel Stratix SoCs [5]). These trends indicate that many applications that traditionally run on general-purpose processors (GPPs) can potentially benefit from FPGA acceleration.

To achieve high performance either on GPPs or FPGAs, applications need to exploit parallelism. In particular, *dynamic parallelism*, where work is generated at run-time rather than statically at compile time, is inherent in many modern applications and algorithms, and is widely used to write parallel software for GPPs. For example, hierarchical data structures such as trees, graphs, or adaptive grids often have data-dependent execution behavior, where the computation to be performed is determined at run-time. Recursive algorithms such as many divide-and-conquer algorithms have dynamic parallelism for each level of recursion. Algorithms that adaptively explore space for optimization or process data as in physics simulation also generate work dynamically.

Unfortunately, today's high-level design frameworks for FPGA accelerators do not provide adequate support for dynamic work generation or dynamic work scheduling. For example, C/C++-based *high-level synthesis (HLS)* [6], [7] and *OpenCL* [8] are mostly designed to exploit static data-level or thread-level parallelism that can be determined and scheduled at compile time and mapped to a fixed pipeline. Domain-specific languages such as Liquid Metal [9] and Delite [10] raise the level of abstraction but also only support static parallel patterns. A recent study explored dynamically extracting parallelism from irregular applications on FPGAs [11], but still only supports a limited form of pipeline parallelism and does not provide efficient scheduling of dynamically generated work on multiple processing elements. Low-level *register-transfer-level (RTL)* designs, on the other hand, provide flexibility to implement arbitrary features, but require long design cycles and significant manual effort, making them unattractive especially when targeting a diverse range of applications. To realize the potential of FPGA acceleration for a wide range of applications, we need a design framework that is capable of exploiting *both static and dynamic parallelism* and producing *high-performance* accelerators with *low manual design effort*.

In this paper, we propose ParallelXL, an architectural framework for accelerating both static and dynamic parallel algorithms on reconfigurable hardware. ParallelXL takes a high-level description of a parallel algorithm and outputs the RTL of an accelerator, which can be mapped to an FPGA using standard tools. The framework aims to enable accelerating dynamic parallel algorithms on FPGAs without manually writing RTL, and efficiently support a wide range of parallel patterns with one unified framework. To achieve this goal, we need to address three major technical challenges; the framework needs (1) a new parallel computation model that is general enough while suitable for hardware, (2) an architecture that efficiently realizes the new computation model in hardware, and (3) a productive design methodology to automatically generate RTL.

As a parallel computation model, we propose to adopt a tasked-based programming model with explicit continuation passing. Task-based parallel programming is becoming increasingly popular for parallel software development (e.g.,

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00014

Intel Cilk Plus [12], [13], Intel Threading Building Blocks (TBB) [14], and OpenMP [15], [16]). The task-based frameworks allow diverse types of parallelism to be expressed using a unified *task* abstraction, which represents an arbitrary piece of computation. They support dynamic work generation by allowing a task to generate child tasks at run-time. To support a wide range of communication patterns among tasks and enable efficient hardware implementations, we use *explicit continuation passing* to encode inter-task synchronization.

Then, we propose a novel architecture that can execute an arbitrary computation described using the explicit continuation passing model. The architecture works as a configurable template that provides a platform to dynamically create and schedule tasks, and supports a trade-off between generality and efficiency. For irregular workloads, the architecture can adaptively schedule independent tasks to a pool of processing elements using *work-stealing* [17]–[19], and supports fine-grained load balancing. When the computation exhibits a simple static parallel pattern (e.g., only data-parallel), the architecture can use static scheduling for efficiency. The architecture separates the logical parallelism of the computation from the physical parallelism of the hardware, and enables a programmer to express the computation as tasks without worrying about the low-level details of how these tasks are mapped to the underlying hardware.

To minimize the manual effort for accelerator designers, we propose a new design methodology that combines HLS with the proposed computation model and architecture template. The design methodology uses (1) HLS to generate the application-specific worker from a C++-based description, and (2) a parameterized RTL implementation of the architecture template to generate the final accelerator RTL with the desired architecture features and configuration. The designer does not need to write any RTL in order to use the framework.

We implemented a number of parallel accelerators using ParallelXL and prototyped them on the Xilinx Zynq FPGA to demonstrate that the framework can indeed handle a wide range of applications and provide performance improvements on FPGAs today. We further evaluated the framework in the context of a future SoC with multiple general-purpose cores and an integrated reconfigurable fabric with a cache-coherent memory system using detailed simulations. The results suggest that ParallelXL can generate scalable FPGA-based parallel accelerators that achieve significant speedup compared to an optimized parallel software implementation using Intel Cilk Plus.

The main contributions of this paper include: (1) a general accelerator architecture based on explicit continuation passing model, capable of supporting dynamic work generation and dynamic work scheduling; (2) a design methodology for such accelerators that leverages the benefits of both HLS (for the worker implementation) and RTL (for the template implementation); (3) a prototype of the architecture on a current-generation FPGA; and (4) a detailed simulation-based evaluation of the performance, area, and energy of accelerators built using the framework for future FPGA platforms.

II. Computation Model for Dynamic Parallelism

In this section, we introduce the computation model that we use in ParallelXL. The model is based on explicit continuation passing, inspired by task-based parallel programming languages such as MIT Cilk [19], [20], and allows diverse types of parallelisms to be expressed and scheduled under a common framework.

A. Primitives

A *task* is a piece of computation that takes as input a number of arguments, as well as a *continuation*. More formally, a task is a tuple $(f, args, k)$, where f is the function, $args$ is a list of arguments to f, and k is the continuation which points to another task that should continue after the current task finishes. Intuitively, a task is analogous to a function call in software, where f and $args$ are a function pointer and the arguments to the function, and k points to the caller, which receives the function's return value and continues execution.

A task can *spawn* new tasks while it is executing. The spawned tasks are called *child* tasks. Spawning tasks is similar to function calls except that the parent and child tasks are allowed to run concurrently. The spawned tasks eventually need to be *joined*, so we know that they finished and subsequent computation (that potentially depends on the output of the child tasks) can proceed. Today's software frameworks use special join commands that call into a sophisticated runtime to perform synchronization, which is difficult to implement in hardware. Instead, our model uses explicit continuation passing that leads to a simpler hardware architecture.

A task can be either *ready* or *pending*. A task is ready if it has received all of its arguments, and thus is ready to execute. A task is pending if some of its arguments are still missing, for example, because the tasks that produce them have not completed yet. Each pending task is associated with a *join counter* j, whose value is the number of missing arguments. A task returns a value by sending it to the pending task pointed by its continuation. Upon receiving the value, the join counter j of the pending task is decremented. When the join counter reaches zero, the pending task becomes ready.

B. Continuation Passing

Today's parallel programming frameworks for software uses a runtime system to manage synchronizations among tasks. This approach is challenging to implement in hardware. First, software can perform control transfers between the user code and runtime with function calls or `setjmp`/`longjmp`, and easily save and restore the program state using stack frames. These capabilities are not present in hardware accelerators. Second, the runtime logic is often quite complex, which would incur high overhead if implemented in hardware. We address these challenges by using *explicit continuation passing*.

Continuation passing style (CPS) is a style of programming where control is passed explicitly in the form of a continuation that represents what should be done with the result that the current procedure generates. Our framework uses continuation passing to express computation as a dynamic task graph with

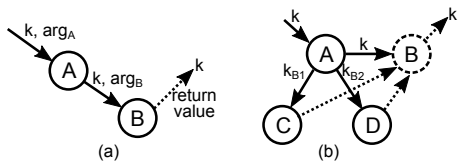

Fig. 1. Continuation passing for (a) sequential composition of tasks, (b) fork-join. Downward arrows represent spawning tasks. Horizontal arrows represent creating successor tasks. Dotted arrows represent returning values (arguments).

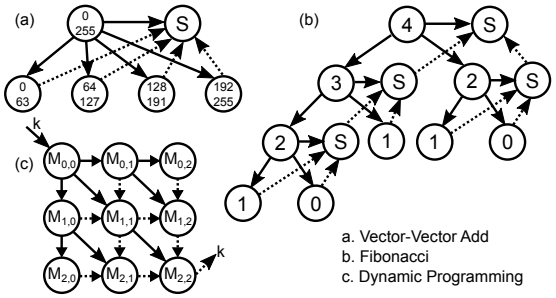

Fig. 2. Task graphs constructed using continuation passing. (a) Vector-vector add. Node labels represent the start and end indices of the sub-vectors. (b) Fibonacci. Each numbered node represents the task for `fib(n)`. Nodes labeled `S` represent the successor (sum) tasks. (c) Dynamic programming. Solid arrows represent spawns along which the continuation is passed.

explicit dependence. The continuation passing serves as the foundation and can be used to construct other abstractions such as data-parallel loops and fork-join patterns.

In our model, the continuation of a task points to a pending task (more precisely, one of the pending task's arguments) that should receive the current task's return value. The simplest use of continuation passing is to implement sequential composition of tasks. Suppose we want to execute tasks A and B sequentially and return the result to continuation k. Using continuation passing, we can invoke task A with k as its continuation. When A finishes, it spawns task B, passing its own continuation k to B. When B finishes, it returns its result to k. Figure 1(a) illustrates this operation.

Continuation can also be used to implement the fork-join pattern, which is a common pattern for dynamically generating parallel tasks and perform synchronization. Suppose we would like to run two parallel tasks and combine their results. In this case, task A creates a pending task B called the *successor*, spawns two child tasks C and D, and points their continuations to B. This completes the fork step. When the child tasks finish, they send their result values to B. B becomes ready once it receives the results of both C and D. This completes the join step. Figure 1(b) illustrates the fork-join operation.

1) Task Graph Examples: Using continuation passing combined with task spawning, both static and dynamic parallel algorithms can be expressed in our model. When the algorithm executes, the tasks form a graph that dynamically unfolds. Here we show three example task graphs from different algorithms.

The first example computes the sum of two vectors of length 256. Suppose we allow the vector to be divided into chunks of length 64. Figure 2(a) shows the task graph. As the computation is data-parallel, the task graph is very regular. In addition, only the child tasks perform actual work, and the parent and successor tasks are used only for synchronization. In case where the source vectors are very long, it is more efficient to use recursive decomposition, where the vectors are recursively divided into smaller and smaller vectors using multiple levels of intermediate tasks, rather than relying only on the root task to perform the decomposition.

Consider another example that calculates the n^{th} Fibonacci number by recursively applying the formula $fib(n) = fib(n-1) + fib(n-2)$, and calculate $fib(n-1)$ and $fib(n-2)$ concurrently. This is a typical fork-join pattern. For example, calculating `fib(4)` results in the graph shown in Figure 2(b).

This example illustrates why it is challenging to map computations with dynamic task-level parallelism to a hardware accelerator. First, tasks are created dynamically during execution, which makes it difficult to enumerate them statically. Second, the computation involves recursion, which is often not supported in current hardware design tools. Third, the task tree is often imbalanced, which makes techniques that partition work statically inefficient.

Now consider a third example, in which an algorithm fills a matrix with values. Each element depends on its left and top neighbor. This pattern is common in many dynamic programming algorithms. Figure 2(c) shows the task graph for filling a simple 3x3 matrix. This type of general task-parallel pattern be expressed using continuation passing, but cannot be easily expressed in frameworks that only support fork-join.

2) Nesting and Composability: It is apparent from the examples above that the computation model naturally supports nested parallelism, where a spawned task can recursively spawn parallel sub-tasks. Nesting is important for achieving good parallelization for many algorithms. For example, in Fibonacci, the degree of parallelism at each task is only two. Without nesting, only the root task can be parallelized, yielding a maximum speedup of two. With nesting, all non-overlapping subtrees of the fib task tree can run in parallel, significantly increasing the amount of parallelism.

Another feature of the computation model is composability, which means the computation can be expressed in a combination of data-parallel, fork-join, or general task-parallel patterns, and the resulting program should still work. This is from the fact that all higher-level patterns are ultimately transformed to the continuation passing primitives.

C. Scheduling the Computation

The model described above enables expressing concurrency in the computation. However, it is up to the *scheduler* to schedule the tasks onto processing elements at run-time for parallel execution. Our framework supports both dynamic work scheduling using work stealing [17], [20] for general dynamic computation, and static task distribution for simple data-parallel computations. Here, we briefly describe the work stealing model.

We model each processing element as a datapath that can process tasks, a local task queue that stores ready tasks, and a pending task storage that holds pending tasks. Each processing element operates on its own task queue in a LIFO (Last-In First-Out) manner, that is, operating on the tail of the queue. When a processing element is idle, it first tries to dequeue and execute a task from the tail of its local task queue. If a task is spawned or a pending task becomes ready, it is appended to the tail of the task queue. If the task queue is empty, the processing element (thief) begins work stealing by randomly selecting another processing element (victim) and trying to steal a task from the head of victim's task queue, that is, the oldest task in the queue. If the continuation of a task refers to a pending task on another processing element, and sending the task's return value caused the pending task to become ready, the newly created task is transferred back to the original processing element to be executed. This is needed to implement *greedy* scheduling, which means the processing element that produces the last missing argument of a pending task should continue execution with the successor task [21]. Greedy scheduling is important for the space bound.

It can be shown that for fully strict computations, where every task sends its result only to its parent's successor task [20], the scheduling policy described above has the same behavior as Cilk's scheduler [20], which is provably efficient. Specifically, it can be shown that the space to store the tasks required for an execution with P processing elements is bound by $S_P \leq S_1 P$, where S_1 is the space required for a serial execution on one processing element [17]. This bound is important to put a limit on the task queue sizes.

D. Function Calls

Traditional HLS tools provide insufficient support of function calls, especially the ones that are deeply nested or recursive. This is not surprising because executing recursive function calls usually requires a stack, which a hardware accelerator does not have. As a result, these tools can only handle simple functions that can be inlined. Our computation model naturally supports recursive function calls by reusing the task spawn and continuation mechanisms. This is similar to how classic continuation passing style programs handle function calls.

III. Accelerator Architecture

The accelerator architecture implements the computation model described in Section II. Specifically, the architecture is designed to fulfill two major goals: (1) implementing task spawning and explicit continuation passing in hardware, and (2) scheduling the computation. Figure 3(a) shows the high-level system architecture, with the accelerator shown in the shaded box. The accelerator consists of multiple tiles, and each tile is composed of a configurable number of processing elements (PEs), each with a unique ID. A tile serves as a basic building block in the architecture, which is a fully-functional task processing engine. The accelerator can consist of any number of tiles without changing its functionality. This tile-based architecture enables an accelerator designer to easily scale the number of tiles and PEs, and also reduces design effort as one component is reused multiple times. The tiles are connected together using an on-chip network. Each tile has an L1 cache, shared by the PEs in that tile.

TABLE I
Comparison between tile architectures.

Pattern	FlexArch	LiteArch
Data-Parallel	Yes	Yes
Fork-Join	Yes	No
General Task-Parallel	Yes	No
Task Scheduling	Work-Stealing	Static Distribution

In this study, we present the architecture in the context of an integrated CPU-FPGA SoC where general-purpose cores and FPGA fabric share a single address space as well as the last-level cache through a cache-coherent interconnect. The integrated SoC platform is becoming increasingly popular and attractive for general-purpose acceleration because it allows fine-grained data sharing between the CPU and FPGA. For applications that do not need fine-grained sharing, the proposed architecture can also be adapted to discrete FPGAs with changes to the memory hierarchy.

We present two variants of our architecture, named FlexArch and LiteArch, which support different trade-off points between flexibility and overhead. FlexArch supports the full continuation passing model, and allows programmers to implement algorithms using many parallel patterns including data-parallel and fork-join patterns as well as nesting in flexible ways. It uses work-stealing for task scheduling. In comparison, LiteArch only supports the data-parallel pattern, and uses static task distribution for task scheduling. LiteArch is intended as a lightweight alternative for applications where a static data-parallel pattern is sufficient. Table I summarizes the features of the two architectures.

A. FlexArch Tile and PE Architecture

Figure 3(b) shows the architecture of a FlexArch tile. A tile contains multiple processing elements, as well as a pending task storage (P-Store), an argument/task router, and network interfaces (Net IF). These components are connected via intra-tile buses. Each PE consists of a worker and a task management unit (TMU).

The worker performs task-specific computations. Because this is the part that an accelerator designer needs to describe for each application, the architecture is designed to keep the worker simple. We factor out common functionalities such as task management into separate modules that can be reused, and provide the worker an interface to communicate with these modules by sending and receiving messages. It has a `task_in` port for receiving a task, a `task_out` port for spawning a task, an `arg_out` port for returning a result value, and a pair of `cont_req` and `cont_resp` ports for creating a successor task

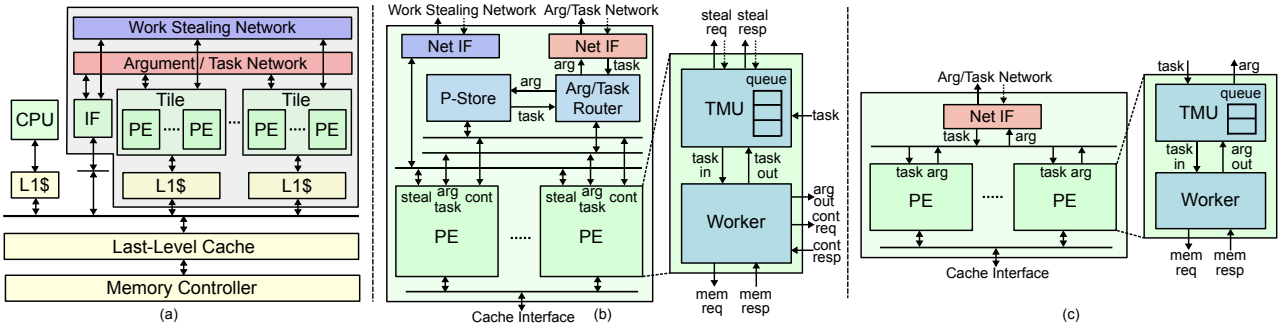

Fig. 3. Accelerator architecture. (a) System architecture. The accelerator is shown in the shaded box. (b) FlexArch tile. (c) LiteArch tile.

and receiving a continuation that points to it. The worker also has a memory port. The architecture does not stipulate how the worker is implemented as long as it follows the interface protocol. For example, it can be implemented in either HLS or RTL. Our architecture currently use homogeneous workers that can run any task in the computation graph. The type of a task is identified by the `type` field in the task message, which corresponds to the f in the computation model. It is also possible to extend the architecture to use heterogeneous workers where each worker is designed to process a subset of task types. This allows coarse-grained resource sharing at the tile level, that is, the hardware for a worker is shared within a tile, rather than dedicated to a PE. In contrast, when using homogeneous workers, HLS tools perform fine-grained resource sharing between the logic for different tasks at the worker level.

The task management unit (TMU) is responsible for feeding the worker with tasks. Internally, it has a task queue that stores ready tasks. The task queue is implemented as a double-ended queue that supports operations on both ends. The worker enqueues and dequeues tasks at the tail of the queue in a LIFO order, which is important because it results in much better task locality than FIFO order, by traversing the task graph in a depth-first manner. When the task queue becomes empty, the TMU initiates work stealing. It uses a linear feedback shift register (LFSR) to pick a random PE as the victim. Then it sends a steal request to the victim through the network. When the TMU on the victim PE receives the request (shown as dotted arrows in Figure 3(b)), it dequeues a task from the head of its queue and sends it back to the stealing PE. Stealing from the head is important for efficiency because it enables stealing a larger chunk of work with each request (i.e. the task at the head is closer to the root of the task spawn tree).

The P-Store holds pending tasks that are waiting for arguments, and keeps track of whether they are ready for execution. Its function is analogous to the reservation stations in an out-of-order processor. A straightforward design is to implement the P-Store as a centralized structure for the entire accelerator, where all pending tasks are kept. However, this would lead to severe contention when scaling up the number of PEs. We address this challenge by proposing a distributed architecture, where each tile has a local P-Store, but is still able to access P-Stores on other tiles over the network. Because of locality in the processing of the task graph, pending tasks created by a tile are likely to be consumed within that tile, which means most accesses would go to the local P-Store without incurring network traffic.

Each P-Store consists of a control unit, a free list, a join counter array, a metadata array, and argument arrays. The free list keeps track of the available entries in the P-Store. When a PE requests to create a pending task, an entry is allocated, and a continuation ID is returned. The join counter array stores the number of missing arguments for each pending task. When an argument is received, the data is written to the argument array specified by the continuation, and the join counter is decremented. If the counter reaches zero, the task that became ready is sent to the PE that produced the last argument, and the entry is deallocated.

The argument/task router steers argument and task messages between local and remote tiles. This is needed for two reasons. First, when a worker returns an argument, it may refer to a pending task on a remote tile. Second, when the P-Store receives an argument from a remote PE and outputs a task, the task needs to be sent back to the remote PE in order to implement *greedy* scheduling, which is critical for guaranteeing the asymptotic bound on space [20].

B. LiteArch Tile and PE Architecture

Figure 3(c) shows the architecture of a LiteArch tile. Compared to a FlexArch tile, it does not have a P-Store or argument/task router as there is no support for creating pending tasks or routing arguments and tasks between tiles. In this architecture, the accelerator does not have a work stealing network. Within a PE, the TMU is simplified to remove work stealing capabilities, and the worker does not have P-Store ports. This architecture supports the data-parallel pattern with the host CPU splitting the range into smaller subranges, and enqueuing the tasks for execution on the PEs.

C. Networks

The argument and work stealing networks shown in Figure 3(a) are two logical networks. Our architecture does not specify the physical implementation of these two networks,

as long as they are compatible with the network interface protocol. They can be implemented with different topologies, or even be combined into one physical network. In our implementation, each network uses a crossbar.

D. Memory Hierarchy

In our architecture, the accelerators are integrated into the general-purpose memory hierarchy via the last-level cache, and share the same address space as the general-purpose cores. We investigate a single address space, cache-based memory system for two reasons. First, caches reduce programming effort by removing the need to manually orchestrate data transfers, which is a significant portion of the design efforts in today's hardware accelerators. Second, caches and a single address space enable fine-grained data sharing between the CPU and FPGA, e.g. sharing pointer-based data structures, which broadens the range of applications that can be mapped to the architecture. The accelerator has a number of L1 caches (shown in Figure 3(a)), one per each tile. The architecture supports cache coherence; the caches can be kept coherent among themselves and with the last-level cache to support applications that require fine-grained data sharing. The accelerator caches can be implemented using the block RAMs on FPGAs. Future FPGAs can also include hardened L1 cache blocks. Also note that the workers can have local memory structures such as scratchpads that are not a part of the cache-coherent memory system. Some accelerators rely on the massive internal memory bandwidth provided by such local memory to achieve high performance. When a task is stolen, data movement is performed transparently through shared memory with coherent caches. The proposed framework can also be used with non-coherent caches or DMA-based accelerators if fine-grained data sharing is not needed, and designers are willing to explicitly control data transfers. A PE can initiate cache flushing or DMA transfers to read input / write output data for a task.

Integrating accelerators into the general-purpose memory hierarchy represents a challenge for traditional HLS tools. The traditional HLS tools assume a fixed latency for all memory accesses, and generate large monolithic designs which struggle when facing the variable memory latency of a general-purpose memory system [22]; a delay in any memory response would cause the entire design to stall. Our accelerator architecture overcomes this problem by making PEs independent so that one stalled PE would not affect others, and using dynamic work scheduling to balance the load on the PEs should any imbalance arise due to memory latencies.

E. CPU-Accelerator Interface

The accelerator contains an interface (IF) block that serves as the interface between the CPU and the accelerator. The IF block implements a memory-mapped interface. The CPU can send tasks to the accelerator and read results back using memory-mapped accesses. Once the IF receives a task, it needs to pass it to the PEs for processing. For FlexArch, we leverage work stealing for this purpose. A PE can steal a task from IF via the work stealing network. For LiteArch, the IF passes the task via the argument/task network to one of the PEs based on a static assignment, where the task is then executed.

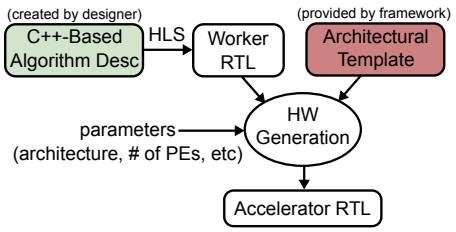

Fig. 4. Accelerator design flow using ParallelXL.

IV. Design Methodology and Framework

In this section, we discuss the methodology and software framework we developed for designing accelerators with low manual effort. Figure 4 shows the overall design flow using the ParallelXL framework. Accelerator designers describe the algorithm using a C++-based worker description format, then the framework synthesizes the worker RTL using HLS. Next, the framework combines worker RTL with an architectural template it provides, and generates the final RTL of the accelerator.

A. Architectural Template

We implemented the proposed accelerator architecture as an architecture template in PyMTL [23], [24], a Python-based hardware generation language. The template is parameterized so that the designer can configure the architecture (FlexArch or LiteArch), the number of tiles and PEs, the number of entries of the task queue and P-Store, as well as the cache size.

B. Algorithm Description Format

While the accelerator architecture does not specify how the task processing logic (worker) should be implemented, in practice, high-level synthesis is usually preferred because of its productivity compared to RTL design. We support the HLS approach by defining a C++-based worker description (CPPWD) format. As an example, Figure 5 shows the CPPWD code for the Fibonacci algorithm described in Section II. The worker is defined as a function, and the arguments of the function are the ports of the worker. The function header is standard for all workers, except for the function name and task type, which are defined by the designer. The body of the function defines the Fibonacci algorithm, which recursively splits the problem into sub-problems by dynamically spawning child tasks until reaching the base case, and then merging the results back to obtain the answer. This algorithm is challenging to express using today's accelerator design methodologies because it involves dynamically bounded parallel recursion, but is trivial to express using our framework.

For the data-parallel pattern, the framework provides a helper function (`parallel_for`) similar to Intel TBB [14], which wraps the details of implementing dynamic spawning/joining of tasks in an easy-to-use interface. CPPWD also

```
void FibWorkerHLS
(
  TaskInPort<FibTaskType>  task_in,
  TaskOutPort<FibTaskType> task_out,
  ContReqPort              cont_req,
  ContRespPort             cont_resp,
  ArgOutPort               arg_out
) {
  const FibTaskType task = task_in.read();

  // continuation
  task_k_t k = task.k;

  if (task.type == FIB) {
    int n = task.x;
    if (n < 2)
      send_arg(Argument(k, n), arg_out);
    else {
      // create successor task
      k = make_successor(SUM, k, 2, cont_req, cont_resp);
      // spawn tasks
      spawn(FibTaskType(FIB, k, 1, n-2, 0, 0), task_out);
      spawn(FibTaskType(FIB, k, 0, n-1, 0, 0), task_out);
    }
  } else if (task.type == SUM) {
    int sum = task.x + task.y;
    send_arg(Argument(k, sum), arg_out);
  }
}
```

Fig. 5. C++-based worker description for Fibonacci.

supports the `blocked_range` concept, which allows splitting a linear range into blocks of configurable size.

C. Accelerator RTL Generation

The framework generates the accelerator RTL by combining the synthesized worker RTL with the architecture template according to the parameters specified by the designer, including the choice of the architecture, the number of PEs, the number of task queue entries, cache size, etc. The framework then elaborates the template and perform hardware generation to output the final RTL of the accelerator. Design space exploration can be done easily by changing the parameters given to the framework, without rewriting any code.

V. Evaluation

In this section, we present the evaluation results for the proposed accelerator framework. We first present a hardware prototype of accelerators on today's FPGA platform. Then, in order to perform a more detailed study of the architecture, and to avoid the limitations of the current FPGA platform, we present a simulation-based study in the context of a future integrated CPU-FPGA SoC.

A. Benchmarks

We use a set of ten benchmark algorithms that cover a variety of application domains, including linear algebra, graph search, sorting, combinatorial optimization, image processing, and bioinformatics. Some of the benchmarks are developed in house, while others are adapted from benchmark suites such as Cilk apps [19], Unbalanced Tree Search [25]. and MachSuite [26]. We coded parallel implementations of these algorithms using the proposed C++-based algorithm description format. Task granularity depends on application characteristics, but is chosen to strike a balance between parallelization overhead and load balancing. Table II summarizes the benchmarks and shows the characteristics of each benchmark. Among them, the ones that are recursive, or have nested or data-dependent parallelism are especially challenging to express in existing accelerator design frameworks, but our framework allows writing these algorithms.

TABLE II
Summary of benchmarks. PA: Parallelization Approach, PF=parallel-for, FJ=fork-join, CP=continuation passing. R/N: Recursive/Nested Parallelism. DP: Data-Dependent Parallelism. MP: Memory Access Pattern. MI: Memory Intensity.

Name	From	PA	R/N	DP	MP	MI
nw	In-house	CP	Yes	Yes	Regular	Medium
quicksort	In-house	FJ	Yes	Yes	Regular	Medium
cilksort	Cilk apps	FJ	Yes	Yes	Regular	Medium
queens	Cilk apps	FJ	Yes	Yes	Regular	Low
knapsack	Cilk apps	FJ	Yes	Yes	Regular	Low
uts	UTS	FJ	Yes	Yes	Regular	Low
bbgemm	MachSuite	PF	Yes	No	Regular	Medium
bfsqueue	MachSuite	PF	No	No	Irregular	High
spmvcrs	MachSuite	PF	No	No	Irregular	High
stencil2d	MachSuite	PF	No	No	Regular	High

Here we give a brief description of each benchmark algorithm, as well as the approach we take to parallelize them: ① `nw` implements the Needleman-Wunsch algorithm, which is a dynamic programming algorithm that aligns two DNA sequences. The algorithm fills values of a two-dimensional matrix, where the value of each element depends on its neighbors on the north, west, and northwest. We parallelize `nw` by blocking the matrix, and using continuation passing to construct the task graph, similar to Figure 2(c). ② `quicksort` implements the classic Quicksort algorithm, which is a divide and conquer algorithm that recursively partitions an array into two smaller arrays and sorts them. We use the Hoare partition scheme [27] in our implementation, and use fork-join to parallelize across the divide-and-conquer tree. ③ `cilksort` is a parallel merge sort algorithm first described in [28]. It recursively divides an array into smaller arrays and sorts them, and also performs the merging in parallel. When the sub-array size gets small, it uses quicksort to sort the sub-array, which in turn partitions and sorts the sub-arrays, and uses insertion sort when the sub-array size becomes sufficiently small (tens of elements). We use fork-join to parallelize across the divide-and-conquer tree. ④ `queens` solves the classic N-queens problem. We use fork-join to parallelize searching of the solution space. ⑤ `knapsack` solves the 0-1 knapsack problem. Our implementation uses a branch-and-bound algorithm, and is parallelized using fork-join. ⑥ `uts` is a benchmark that dynamically constructs and searches an unbalanced tree. The unbalanced nature of the tree stresses the load balancing capability of the architecture. We use fork-join to parallelize across the subtrees. ⑦ `bbgemm` is a matrix multiplication

kernel that use blocking to achieve good memory locality [29]. We use a block size of 32 and parallelize the loop nest with two nested parallel-for's. ⑧ `bfsqueue` is a breadth-first search algorithm that uses a queue to store frontier nodes. We parallelize across the frontier with a parallel-for loop. ⑨ `spmvcrs` is a sparse matrix-vector multiplication algorithm using compressed row storage format. We parallelism across the matrix rows using parallel-for. ⑩ `stencil2d` performs stencil computation on a 2D image. We break the image into blocks and use parallel-for to parallelize across the blocks.

For each worker that is generated by HLS, we applied standard HLS optimization techniques such as loop pipelining and unrolling, and use application-specific local memory structures such as scratchpads and buffers to achieve high internal memory bandwidth when possible. In that sense, a single PE in our architecture can be considered to represent optimized accelerators designed using today's HLS tools without additional parallelization support.

For benchmarks that use fork-join or continuation passing, we also tried to implement a version that only uses parallel-for, targeting the LiteArch. The high-level idea is to use multiple rounds, with each round processing one level of the task graph using a parallel-for, and at the same time constructing the next level. This requires the tasks in the same level to be homogeneous. For benchmarks that cannot be parallelized this way, we also tried to rewrite the algorithm using a different approach if that helps mapping it to parallel-for. In the end, we were able to implement parallel-for versions of `nw`, `quicksort`, `queens` and `knapsack`, but not `cilksort`, due to the complexity and irregularity of its dynamic task graph.

We also coded a parallel software implementation for each algorithm using Intel Cilk Plus [30], and compiled with `-O3` optimization and auto-vectorization targeting NEON SIMD extensions.

B. Hardware Prototype on Today's FPGA

To demonstrate the proposed framework, we implemented a prototype system using the Xilinx Zynq-7000 [31] FPGA SoC on Zedboard. The SoC includes two ARM Cortex-A9 cores and an integrated FPGA fabric equivalent to Artix-7. We implemented the FlexArch template for the FPGA and generated accelerators using the flow described in Section IV-C. The Zynq-7000 platform has some limitations compared to future integrated CPU-FPGA platforms that we envision (Figure 3(a)). First, the FPGA fabric does not have a shared-cache interface that can be used to implement coherent caches on the FPGA. As a result, we implemented stream buffers instead of L1 caches to connect PEs to the L2 cache, and a few benchmarks that rely on fine-grained cache accesses were not implemented. Second, the bandwidth from the FPGA to the L2 cache is limited by a single ACP port and is much lower than the CPU-to-L2 bandwidth. The memory bandwidth becomes a bottleneck when scaling up the number of PEs.

We compare the performance of the accelerators to an optimized parallel Cilk Plus implementation of the benchmarks running on the two ARM cores on the SoC. Performance

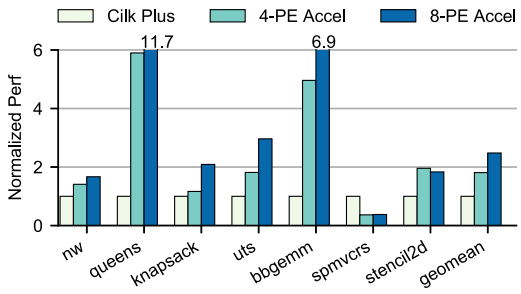

Fig. 6. Accelerators performance compared to parallel software on Zedboard.

TABLE III
PLATFORM CONFIGURATION.

Technology	28nm
CPU	ARM ISA, eight-core, four-issue, out-of-order, 32 entries IQ, 96 entries ROB, 1GHz
CPU L1 Cache	L1I/L1D: 32KB, 2-way, 64B line size, 1-cycle hit latency, next-line prefetcher
Accel logic	In FPGA fabric, 200MHz
Accel L1 Cache	32KB, 2-way, 64B line size, 400MHz, 1-cycle hit latency, next-line prefetcher
L2 Cache	2MB, 8-way, 1GHz, 10-cycle hit latency, inclusive, shared between cores and accelerator
Coherence	MOESI snooping protocol
DRAM	64-bit DDR3-1600, 12.8GB/s peak bandwidth

numbers are obtained by comparing whole program execution time, which include initialization and data transfers. Figure 6 shows the performance of FPGA accelerators with 4 PEs and 8 PEs, normalized to the parallel software implementation. The results show that the 4-PE accelerators achieve up to 5.9x speedup over parallel software (geomean 1.8x), and the 8-PE accelerators achieve up to 11.7x speedup (geomean 2.5x). The results also reveal the limitations of the Zynq-7000 platform. For example, the accelerators show a slowdown for `spmvcrs`, which is a memory-bound benchmark, because the FPGA has lower memory bandwidth to the L2 cache compared to the ARM cores. Similarly, there is little performance improvement for `nw`, `spmvcrs`, and `stencil2d` when increasing the number of PEs, again due to limited memory bandwidth.

C. Simulation Methodology

The limitations of today's FPGA platform makes it difficult to evaluate the proposed architecture in the context of future integrated CPU-FPGA platforms with support for cache coherent accelerators [3] and higher memory bandwidth. For the rest of the section, we present a simulation-based study, which allows us to further explore the design space and perform more detailed evaluation.

We model a future integrated CPU-FPGA SoC where the CPU and FPGA share a cache-coherent memory system. The parameters of the platform are shown in Table III. We use gem5 [32] to model the integrated CPU-FPGA SoC. To simulate the accelerators, we modified gem5 by integrating

an RTL simulator (Verilator) into gem5 as a `ClockedObject` that is ticked every cycle, similar to gem5's CPU models. We wrote adapters to perform synchronization between gem5's event-based components (memory-system) and the cycle-based accelerator RTL simulator. In this way, we can perform detailed RTL simulations of the accelerators, while retaining the flexibility in configuring the system components such as cores, caches, interconnect, and DRAM.

We estimate FPGA resource utilization by synthesizing the RTL using Vivado targeting Xilinx's 7-series FPGA to obtain LUT/FF count, the number of DSP slices, and the number of block RAMs. We estimate the resource utilization of the accelerator caches using numbers from Xilinx's cache IP [33].

To estimate the energy of the accelerators, we run Vivado's power estimation tool on the synthesized netlist using signal activity factors from RTL simulation. We model the energy of the cores using McPAT [34], using event statistics from gem5 simulations.

D. Performance Results

1) Scalability: Here we present the scalability of the proposed accelerator architecture using parallel speedup, which is the speedup of a n-PE implementation over a single PE implementation. In our experiments, we configure each tile to have 4 PEs and simulate up to 8 tiles (32 PEs) for both FlexArch and LiteArch. For comparison, we also show the scalability of the CilkPlus baseline on 1 to 8 cores. Because a PE is much smaller and lower-power than an out-of-order core, we can fit more PEs than cores in the same area and power budget. On the memory system side, the 8-tile and 8-core configurations have the same number of L1 caches. Table IV shows the scalability results. Comparing the software and accelerators results, the accelerators achieve similar speedups (from 1 to 8 cores/PEs) compared to CilkPlus, which is a state-of-the-art task-based parallel programming framework and runtime. In addition, the accelerators continue to get more speedups with more PEs for most benchmarks. This shows that the proposed accelerator architecture is effective in harnessing the parallelism in applications.

Comparing the two accelerator architectures, LiteArch accelerators match the scalability of the FlexArch accelerators when algorithms map naturally to the data-parallel pattern (`bbgemm`, `bfsqueue`, `spmvcrs` and `stencil2d`) However, for benchmarks that have dynamic data-dependent parallelism or are irregular (parallelized with fork-join or explicit continuation passing), FlexArch accelerators generally achieves better scalability, except for `knapsack`. The `knapsack` implementation on LiteArch uses a different algorithm that sacrifices algorithmic efficiency in order to map to parallel-for. Though it has good scalability, we will see later that the absolute performance is actually much lower. These results indicate that LiteArch is adequate to support regular data-parallel algorithms. FlexArch, on the other hand, is a better fit for most other parallel algorithms. This is because although some of these algorithms can be rewritten to map to LiteArch, their dynamic and irregular nature makes the implementation less efficient, due to less effective load balancing and/or reduced algorithmic efficiency.

The results also show that some benchmarks have better scalability than others. For example, the two sorting algorithms, `cilksort` and `quicksort`, show similar speedups when there are only a small number of cores/PEs. However, when the number of cores/PEs increases, `cilksort` can continue to scale its performance, achieving 26.20x speedup with 32 PEs using FlexArch, while the performance of `quicksort` quickly tapers off. The reason is that these two algorithms have different amount of dynamic parallelism. `quicksort` has a significant non-parallelizable portion. Specifically, the partitioning step is performed serially, thus the achievable speedup is limited by Amdahl's law. In contrast, `cilksort` (a.k.a. parallel merge sort) generates a large number of parallel tasks during execution, hence it achieves better scalability.

The results also indicate that the FlexArch architecture achieves good load balancing using its hardware-based work stealing mechanism. For example, `uts` (Unbalanced Tree Search) is particularly difficult to load balance and requires frequent work stealing operations. CilkPlus only achieves 3.91x speedup with 8 cores. In comparison, the FlexArch accelerator achieves 6.50x speedup with 8 PEs, and is able to continue to scale the performance with more PEs. The hardware implementation of work stealing is more efficient than software because it incurs less overhead. A work stealing operation may require hundreds of instructions in software, but only needs several cycles on the accelerator.

2) Normalized Performance: Figure 7 shows the performance of FPGA accelerators normalized to a single out-of-order core. The horizontal line represents the performance of parallel software using CilkPlus running on eight cores. The numbers are obtained by comparing whole program execution time. The results show that the accelerators outperform the 8-core software implementation for most benchmarks. When using 32 PEs, the FlexArch accelerators are up to 9.1x (geomean 4.0x) faster than eight cores, and up to 69.5x (geomean 24.1x) faster than a single core. For many applications, the accelerators can outperform an 8-core processor that has a much higher frequency because each PE can perform more operations in a cycle than a processor core. We use standard HLS optimizations (loop pipelining and unrolling) to increase internal parallelism with a PE. In addition, the PEs also exploit application-specific parallelism. For example, in `queens`, each PE is designed to check multiple candidate locations on a chessboard in parallel. These types of optimizations require hardware customization, and are difficult to implement in processors. On the other hand, the accelerators cannot significantly outperform an 8-core processor for `quicksort` and `spmvcrs`. As discussed earlier, `quicksort` has a significant serial portion, so the processor with a high frequency runs faster. `spmvcrs` is limited by memory bandwidth, as a result all implementations eventually reach similar performance.

The LiteArch accelerators achieve similar performance as the FlexArch accelerators for data-parallel benchmarks. For most other benchmarks, FlexArch significantly outperforms

TABLE IV
BENCHMARK SCALABILITY. THE NUMBERS ARE THE SPEEDUP OF A N-CORE/PE IMPLEMENTATION OVER A SINGLE CORE/PE IMPLEMENTATION.

Benchmark	OOO CPU				Flex Accelerator						Lite Accelerator					
	1-C	2-C	4-C	8-C	1-PE	2-PE	4-PE	8-PE	16-PE	32-PE	1-PE	2-PE	4-PE	8-PE	16-PE	32-PE
nw	1.00	1.74	3.21	5.54	1.00	1.98	3.69	7.11	13.23	21.19	1.00	1.81	3.09	5.10	7.54	9.90
quicksort	1.00	1.91	3.42	5.40	1.00	1.89	3.24	5.15	6.52	6.81	1.00	1.61	2.54	3.46	4.55	5.17
cilksort	1.00	1.98	3.78	7.05	1.00	1.99	3.50	6.94	13.66	26.20	N/A	N/A	N/A	N/A	N/A	N/A
queens	1.00	1.99	3.92	7.65	1.00	1.89	3.10	6.20	12.12	24.20	1.00	2.00	3.96	7.45	12.08	13.21
knapsack	1.00	2.05	3.92	8.20	1.00	1.97	3.22	6.13	12.55	23.94	1.00	1.93	3.80	7.64	15.15	29.99
uts	1.00	1.75	2.81	3.91	1.00	1.95	3.66	6.50	11.32	15.64	1.00	1.92	3.52	5.76	7.51	7.44
bbgemm	1.00	1.99	3.85	7.04	1.00	1.99	3.88	7.50	13.38	17.48	1.00	1.95	3.42	6.39	11.29	18.27
bfsqueue	1.00	1.77	3.11	4.64	1.00	1.78	3.36	6.13	9.93	12.40	1.00	1.56	4.23	6.95	9.99	12.55
spmvcrs	1.00	1.95	3.50	5.45	1.00	1.99	3.59	6.86	13.16	16.51	1.00	1.93	2.91	5.52	10.16	17.42
stencil2d	1.00	1.99	3.85	7.04	1.00	1.99	3.17	6.22	12.12	20.13	1.00	1.98	2.73	5.36	10.32	17.35
geomean	1.00	1.91	3.52	6.04	1.00	1.94	3.43	6.44	11.57	17.35	1.00	1.85	3.31	5.82	9.37	12.98

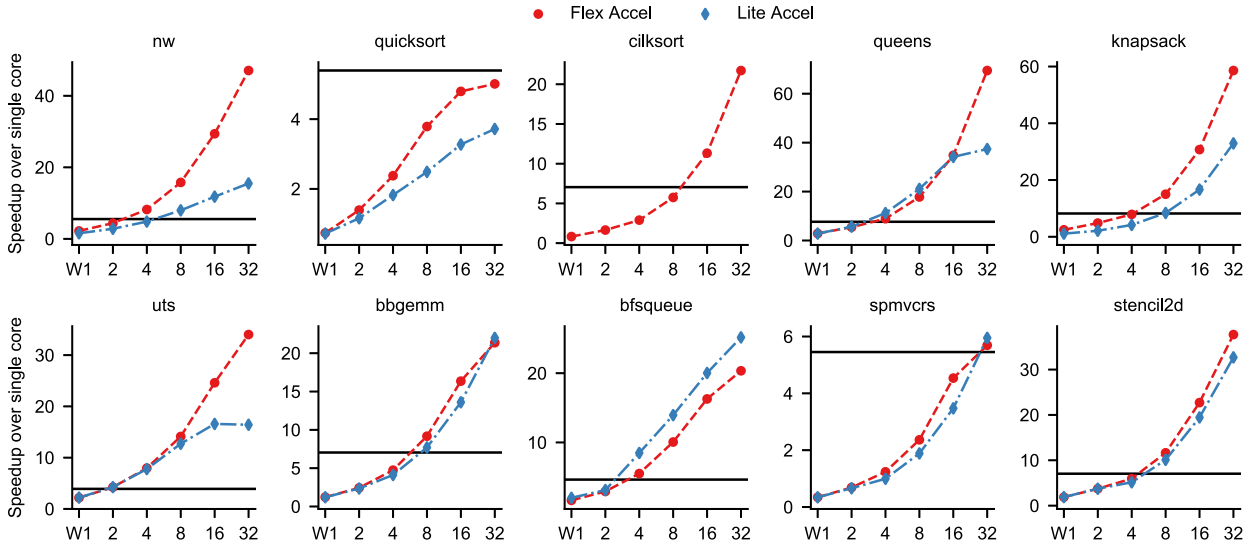

Fig. 7. Normalized accelerator performance. The x-axis is the number of workers (PEs). The y-axis is performance normalized to a single OOO core. The horizontal bar indicates the performance of an eight-core CilkPlus implementation.

LiteArch, especially with a large number of PEs. Also note that the performance difference of `knapsack` comes from the algorithmic inefficiency as discussed earlier.

These results also demonstrate that FPGA accelerators need to exploit parallelism in order to provide performance benefits over parallel software. Traditional HLS tools that use sequential C/C++ code as input can only generate accelerators that roughly match the performance of a single PE, which is often slower than parallel software. Our framework enables easily mapping diverse parallel algorithms to FPGA and achieve compelling performance (and shown later, energy) advantages over parallel software.

E. Resource Utilization

Table V shows the per-PE and per-tile resource utilization of the accelerators. Each tile consists of four PEs and a cache. The DSP blocks are mainly used to implement multipliers, and the BRAMs are used as local scratchpads and buffers, task storage, and caches. The results show that the LiteArch accelerators generally use less resources than the FlexArch accelerators. The reduction is most apparent for regular data-parallel benchmarks (`bbgemm`, `bfsqueue`, `spmvcrs`, and `stencil2d`) whose tasks graphs can be determined statically. On the other hand, the resource reduction by using LiteArch for other benchmarks is less significant, as the task graphs need to be constructed dynamically in both architectures.

To put the resource utilization numbers into context, we studied how many PEs can be mapped to typical FPGA devices. We experimented with two FPGA devices: a low-cost FPGA (Artix XC7A75T) similar to the one on Zedboard, and a mainstream FPGA (Kintex XC7K160T). The low-cost FPGA can fit on average 4 tiles (16 PEs) for FlexArch, and 5 tiles (20 PEs) for LiteArch. The mainstream FPGA can fit 8 tiles (32 PEs) for most benchmarks (except for `cilksort`) for both FlexArch and LiteArch.

TABLE V
ACCELERATORS RESOURCE UTILIZATION. EACH TILE CONSISTS OF FOUR PES AND A CACHE. DSPS ARE SHOWN IN THE NUMBER OF DSP48 SLICES. BRAMS ARE SHOWN IN THE NUMBER OF RAM18'S (EACH RAM36 COUNTS AS TWO RAM18'S).

Benchmark	Flex PE				Flex Tile (incl. Cache)				Lite PE				Lite Tile (incl. Cache)			
	LUT	FF	DSP	RAM	LUT	FF	DSP	RAM	LUT	FF	DSP	RAM	LUT	FF	DSP	RAM
nw	1487	1547	3	7	8914	8668	12	51	1273	1346	1	4	6431	6838	4	36
quicksort	1828	1484	0	6	10618	8484	0	47	1857	1490	0	2	8665	7387	0	28
cilksort	5961	3785	0	8	27233	17622	0	58	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
queens	549	535	0	4	5744	4684	0	40	704	606	0	0	4164	3851	0	20
knapsack	737	770	5	5	6083	5674	20	45	575	466	0	0	3591	3295	0	20
uts	2227	2216	0	5	11510	11438	0	44	2541	2158	0	0	10997	10063	0	20
bbgemm	1551	1789	15	19	9671	9620	60	100	1019	1361	15	14	5401	6736	60	76
bfsqueue	1481	1190	0	6	9353	7348	0	48	887	822	0	1	4901	4791	0	24
spmvcrs	1441	1273	3	13	9303	7660	12	76	875	905	3	8	4777	5119	12	52
stencil2d	1741	2334	12	10	10316	11905	48	64	1200	1964	12	5	6175	9359	48	40

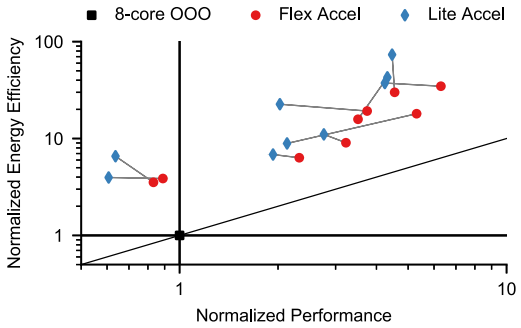

Fig. 8. Normalized performance and energy efficiency. Energy efficiency is the inverse of energy consumption. Both performance and energy efficiency are normalized to the CilkPlus implementation on 8 OOO cores. Points to the right of the vertical line have better performance. Points above the horizontal line have better energy efficiency. The diagonal line represents the iso-power line. Points above the diagonal line have lower power. Points for the same benchmark are linked. Note that both axes are in log scale.

F. Power and Energy Efficiency

Figure 8 shows the performance and energy efficiency of the accelerators (16-PE configuration) normalized to a CilkPlus implementation on eight out-of-order cores. The results show that the proposed accelerators are lower power and more energy efficient for all benchmarks, with most benchmarks showing more than 10x gain in energy efficiency. Comparing the two accelerator architectures, there exists a clear trend in the performance/energy efficiency profile: FlexArch usually achieves better performance, while LiteArch often have better energy efficiency. On average, FlexArch achieves a normalized energy efficiency of 11.8x compared to the out-of-order cores, while LiteArch achieves 15.3x.

G. Cache Size Customization

The accelerator L1 cache (tile cache) are built using the BRAMs in the FPGA fabric. The size of the cache can be customized according to application characteristics. For benchmarks that are not memory intensive, or have good locality, the cache sizes can be made smaller to reduce BRAM usage without significantly degrading performance. Figure 9 shows the performance of the FlexArch accelerators (16-PE configuration) when varying the L1 cache size from 4kB to 32kB. The benchmarks that have an irregular memory access pattern (`bfsqueue` and `spmvcrs`) show the largest performance loss. `nw` and `bbgemm` also showed some performance loss because the reduced temporal reuse with smaller cache sizes. The other benchmarks perform relatively well even with a small cache size. Among them, `cilksort`, `quicksort`, and `stencil2d` have good locality, and the other three have relatively low memory intensity.

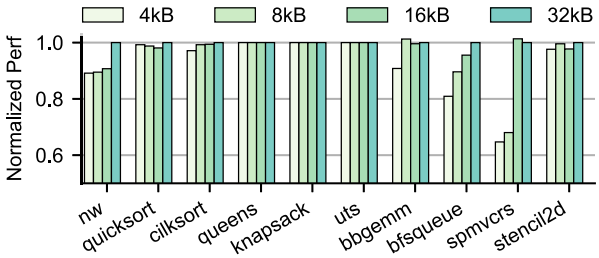

Fig. 9. Performance when varying accelerator L1 cache size.

VI. RELATED WORK

Task-Based Parallel Programming. Task-based parallel programming was first proposed in [35], and recently gained popularity with the introduction of languages and frameworks such as Cilk [12] and Intel TBB [14]. Task-based programming has been shown to allow programmers to think at a higher level while providing good performance and load balancing. Carbon [36] implements hardware task queues in a processor that can be accessed using special instructions. Our work is inspired by task-based programming, but uses the task-based framework for hardware acceleration.

Work stealing was developed along with task-based programming and has been extensively studied [14], [17], [20]. It has been shown to have provable bounds in terms of the space and time needed for a parallel execution compared to serial execution [17], and also works well in practice. We implement work stealing in hardware and show that it can efficiently distribute and balance load in parallel accelerators.

Design Methodologies for Parallel Accelerators. Generating parallel accelerators from a high-level description was explored and implemented in several languages and frameworks [7]–[10], [37]. For example, OpenCL [8] has been adopted for generating accelerators based on data parallelism. Delite [10] is a domain-specific language for generating accelerators based on a collection of parallel patterns. Liquid Metal [9] extends Java to support accelerators with pipeline parallelism. Legup [7] supports a subset of POSIX threads. Kiwi [37] extends C# to generate accelerators with threads and channels. The existing frameworks require parallelism to be specified at compile time and statically scheduled. As a result, it is difficult to map dynamic or irregular algorithms to these frameworks. The proposed framework supports dynamic parallelism with dynamic work generation and dynamic scheduling.

A few prior studies explored dynamic parallelism in hardware. Li et al. [11] propose to extract parallelism from irregular applications by considering dynamic data dependencies [11]. It focuses on pipeline parallelism within a single thread of execution. Our work targets many types of parallelism including data-parallel, fork-join and general task parallelism. We focus on efficient scheduling of dynamically generated tasks onto multiple processing elements for parallel (multithreaded) execution, and propose a hardware-based work-stealing mechanism to achieve good load balancing. Ramanathan et al. [38] explored implementing software-based work stealing runtimes on FPGAs using OpenCL atomic operations, which incurs high performance and resource overhead. In contrast, we propose a hardware architecture that implements native support for work stealing, which is more efficient, more scalable, and uses less resources.

VII. Conclusion

In this paper, we introduce ParallelXL, an architectural framework to design high-performance parallel accelerators with low manual effort using techniques inspired by task-based parallel programming. We propose an accelerator architecture that implements a task-based computation model with explicit continuation passing, and handles task distribution and load balancing efficiently using work stealing. The architecture supports dynamic and nested parallelism in addition to static parallelism, and can execute both irregular and regular applications efficiently. We also introduce a design methodology that allows creating task-parallel accelerators from high-level descriptions. Evaluation results show that our approach can generate high-performance and energy-efficient accelerators targeting FPGAs with low manual effort. While we focus on FPGAs in this paper, we believe that the methodology has potential to be adopted to generate ASIC accelerators as well.

Acknolwedgements

This work was supported in part by the Office of Naval Research (ONR) grant #N0014-15-1-2175, NSF XPS Award #1337240, NSF CRI Award #1512937, AFOSR YIP Award #FA9550-15-1-0194, and donations from Intel and Xilinx.

References

[1] "Amazon EC2 F1 instances," Online Webpage, 2017 (accessed Apr 17, 2018), https://aws.amazon.com/ec2/instance-types/f1/.

[2] P. K. Gupta, "Xeon-FPGA platform for the data center," *Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL)*, Jun 2015.

[3] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel, "CAPI: A coherent accelerator processor interface," *IBM Journal of Research and Development*, vol. 59, Jan/Feb 2015.

[4] S. Ahmad, V. Boppana, I. Ganusov, V. Kathail, V. Rajagopalan, and R. Wittig, "A 16-nm multiprocessing system-on-chip field-programmable gate array platform," *IEEE Micro*, vol. 36, Mar/Apr 2016.

[5] M. Hutton, "Stratix 10: 14nm FPGA delivering 1GHz," in *Hot Chips 27 Symposium (HCS)*, 2016.

[6] *Vivado Design Suite User Guide: High-Level Synthesis*, Xilinx, Inc.

[7] J. Choi, S. D. Brown, and J. H. Anderson, "From pthreads to multicore hardware systems in LegUp high-level synthesis for FPGAs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, 2017.

[8] *Intel FPGA SDK for OpenCL Programming Guide*, Intel Corporation.

[9] J. S. Auerbach, D. F. Bacon, I. Burcea, P. Cheng, S. J. Fink, R. M. Rabbah, and S. Shukla, "A compiler and runtime for heterogeneous computing," in *Proceedings of the 49th Annual Design Automation Conference 2012 (DAC)*, 2012.

[10] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun, "Automatic generation of efficient accelerators for reconfigurable hardware," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016.

[11] Z. Li, L. Liu, Y. Deng, S. Yin, Y. Wang, and S. Wei, "Aggressive pipelining of irregular applications on reconfigurable hardware," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[12] C. E. Leiserson, "The Cilk++ concurrency platform," in *Proceedings of the 46th Design Automation Conference*, 2009.

[13] "Intel Cilk Plus language extension specification, version 1.2," Intel Reference Manual, Sep 2013, https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm.

[14] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ For Multi-Core Processor Parallelism*. O'Reilly, 2007.

[15] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of OpenMP tasks," *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, vol. 20, Mar 2009.

[16] "OpenMP application program interface, version 4.0," OpenMP Architecture Review Board, Jul 2013, http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[17] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, 1999.

[18] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, Aug 1996.

[19] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1998.

[20] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, 1995.

[21] A. Robison, "A primer on scheduling fork-join parallelism with work stealing," The C++ Standards Committee, Tech. Rep., 01 2014.

[22] T. Chen and G. E. Suh, "Efficient data supply for hardware accelerators with prefetching and access/execute decoupling," in *Proceedings of the 49th International Symposium on Microarchitecture (MICRO)*, 2016.

[23] D. Lockhart, G. Zibrat, and C. Batten, "PyMTL: A unified framework for vertically integrated computer architecture research," in *Proceedings of the 47th International Symposium on Microarchitecture (MICRO)*, 2014.

[24] S. Jiang, B. Ilbeyi, and C. Batten, "Mamba: closing the performance gap in productive hardware development frameworks," in *Proceedings of the 55th Annual Design Automation Conference (DAC)*, 2018.

[25] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C. Tseng, "UTS: an unbalanced tree search benchmark," in *19th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2006.
[26] B. Reagen, R. Adolf, Y. S. Shao, G. Wei, and D. M. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *Proceedings of the 2014 IEEE International Symposium on Workload Characterization (IISWC)*, 2014.
[27] C. A. R. Hoare, "Algorithm 64: Quicksort," *Communications of the ACM*, vol. 4, 1961.
[28] S. G. Akl and N. Santoro, "Optimal parallel merging and sorting without memory conflicts," *IEEE Transactions on Computers*, vol. 36, 1987.
[29] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proceedings of the Forth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, 1991.
[30] "Intel Cilk Plus," Online Webpage, 2015 (accessed Aug 2015), https://software.intel.com/en-us/intel-cilk-plus.
[31] "Zynq-7000 all programmable SoC," Online Webpage, 2017 (accessed Apr 17, 2018), https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.
[32] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, 2011.
[33] "LogiCORE IP system cache v3.0," Xilinx Product Guide, https://www.xilinx.com/support/documentation/ip_documentation/system_cache/v3_0/pg118_system_cache.pdf.
[34] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd International Symposium on Microarchitecture (MICRO-42)*, 2009.
[35] F. W. Burton and M. R. Sleep, "Executing functional programs on a virtual tree of processors," in *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture (FPCA)*, 1981.
[36] S. Kumar, C. J. Hughes, and A. D. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," in *34th International Symposium on Computer Architecture*, 2007.
[37] D. J. Greaves and S. Singh, "Kiwi: Synthesis of FPGA circuits from parallel programs," in *Proceedings of the 16th IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2008.
[38] N. Ramanathan, J. Wickerson, F. Winterstein, and G. A. Constantinides, "A case for work-stealing on FPGAs with OpenCL atomics," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016.