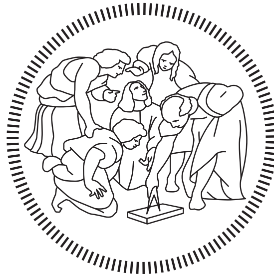


Efficient Synthesis of Graph Methods: A Dynamically Scheduled Architecture

Marco Minutoli, Vito Giovanni Castellana, Antonino Tumeo, Marco Lattuada, Fabrizio Ferrandi

Marco Minutoli, Vito Giovanni Castellana, Antonino Tumeo, Marco Lattuada, and Fabrizio Ferrandi. Efficient synthesis of graph methods: A dynamically scheduled architecture. In *Proceedings of the 35th International Conference on Computer-Aided Design*, ICCAD '16, pages 128:1–128:8, New York, NY, USA, 2016. ACM

The final pubblication is available via http://dx.doi.org/10.1145/2966986.2967030

©ACM, 2016. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Proceedings of the 35th International Conference on Computer-Aided Design http://doi.acm.org/10.1145/2966986.2967030

Efficient Synthesis of Graph Methods: a Dynamically Scheduled Architecture

Marco Minutoli, Vito Giovanni Castellana, Antonino Tumeo
Pacific Northwest National Laboratory
907 Battelle Blvd, Richland, 99354 WA, USA
marco.minutoli@pnnl.gov
vitogiovanni.castellana@pnnl.gov
antonino.tumeo@pnnl.gov

Marco Lattuada, Fabrizio Ferrandi
Politecnico di Milano — DEIB
Piazza Leonardo Da Vinci 32, 20133, Milan, Italy
marco.lattuada@polimi.it
fabrizio.ferrandi@polimi.it

ABSTRACT

RDF databases naturally map to a graph representation and employ languages, such as SPARQL, that implements queries as graph pattern matching routines. Graph methods exhibit an irregular behavior: they present unpredictable, fine-grained data accesses, and are synchronization intensive. Graph data structures expose large amounts of dynamic parallelism, but are difficult to partition without generating load unbalance. In this paper, we present a novel architecture to improve the synthesis of graph methods. Our design addresses the issues of these algorithms with two components: a Dynamic Task Scheduler (DTS), which reduces load unbalance and maximize resource utilization, and a Hierarchical Memory Interface controller (HMI), which provides support for concurrent memory operations on multi-ported/multi-banked shared memories. We evaluate our approach by generating the accelerators for a set of SPARQL queries from the Lehigh University Benchmark (LUBM). We first analyze the load unbalance of these queries, showing that execution time among tasks can differ even of order of magnitudes. We then synthesize the queries and compare the performance of the resulting accelerators against the current state of the art. Experimental results show that our solution provides a speedup over the serial implementation close to the theoretical maximum and a speedup up to *3.45* over a baseline parallel implementation. We conclude our study by exploring the design space to achieve maximum memory channels utilization. The best design used at least three of the four memory channels for more than *90%* of the execution time.

CCS Concepts

•Hardware → Operations scheduling; •Computer systems organization → *Reconfigurable computing;*

ACM acknowledges that this contribution was authored or co-authored by an employee, or contractor of the national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. Permission to make digital or hard copies for personal or classroom use is granted. Copies must bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. To copy otherwise, distribute, republish, or post, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICCAD '16, November 07-10, 2016, Austin, TX, USA

© 2016 ACM. ISBN 978-1-4503-4466-1/16/11...$15.00

DOI: http://dx.doi.org/10.1145/2966986.2967030

Keywords

High-Level Synthesis; Dynamic Task Scheduling; SPARQL; Big Data

1. INTRODUCTION

Graph databases are one of the most prominent examples of large-scale Data Analytics applications. The Resource Description Framework (RDF) data model[14], recommended by the W3C, has experienced a significant uptake for organizing data coming from data providers of a variety of areas, including finance, government, health care, transportation, communication and social networks, cybersecurity, and, of course, the World Wide Web (in form of the Semantic Web). RDF organizes data as statements about resources in the form of subject-predicate-object expressions, known as triples. RDF triples naturally maps to labeled, directed graphs. SPARQL is the predominant query language for RDF datasets. It basically expresses queries as a combination of graph methods (graph walks and graph pattern matching operations) and analytic functions.

Graph methods are said to be *irregular*[18], because they usually employ pointer-based data structures that determine fine-grained and unpredictable data accesses with poor spatial and temporal locality. They are inherently parallel, because the algorithms can spawn concurrent activities for many data elements (e.g., for each vertex, or for each edge), and they are synchronization intensive, because concurrent activities can access the same elements at the same time and, thus, require coordination. The parallelism is also dynamic, because activities can start or finish at any point of the computation as they explore new data elements. The computation is mostly composed of data accesses, so increasing memory parallelism for fine-grained memory operations is key in providing higher performance. However, for applications like graph databases, the data sets are extremely large, and the irregularity makes the data structures difficult to partition across distributed memories without generating load unbalance.

Commodity general purpose processors employ complex cache hierarchies to extract as much performance as possible by exploiting locality, while emerging accelerators such as Graphic Processing Units focus at providing high arithmetic performance and high memory bandwidths for aligned data accesses. For these reasons, graph methods do not exploit these architectures at their best, requiring a careful

and often long optimization process. Reconfigurable devices such as Field Programmable Gate Arrays (FPGAs) thus provide an opportunity to design more efficient custom accelerators for these workloads[9, 10]. However, this may simply redirect the development efforts from optimizing the code in high-level language to describing complex architectures in hardware description languages. High-Level Synthesis (HLS) approaches provide a way to quickly generate hardware descriptions from high-level languages, but current HLS tools are effective in generating serial or parallel accelerators for regular, easily partitionable, arithmetic-intensive workloads typical of digital signal processing. They do not generate efficient accelerators for irregular workloads. At the same time, research started looking for solutions to accelerate big data applications, data analytics and databases on reconfigurable devices. Some approaches even consider not only the acceleration of database operators, but also the synthesis of queries, in particular for streaming datasets, with the objective to quickly find the same patterns in dynamically changing data. Additionally, recent works have looked into architectures for graph and irregular algorithms[1, 4, 17].

In this paper, we propose a novel architecture design for the synthesis of custom accelerators for irregular workloads. We consider a set of SPARQL queries from the Lehigh University Benchmark (LUBM), a realistic benchmark that aims at evaluating the performance of Semantic Web repositories in a standard and systematic way. We analyze their behavior, identifying issues that make these applications difficult to synthesize with conventional approaches for parallel accelerators. We demonstrate, in particular, the challenges connected with the large datasets and the memory parallelism, and the load unbalance among concurrent activities (i.e., concurrent tasks executing on the parallel accelerator arrays). We then propose two new components that address these issues: a Hierarchical Memory Interface (HMI) controller, which enables using multi-banked and multi-ported shared memories, and a Dynamic Task Scheduler (DTS), which enables out-of-order task completion and maximizes resource utilization in presence of unbalanced tasks. We synthesize parallel accelerators for the LUBM queries exploiting these two components. We demonstrate speed ups with respect to conventional approaches for serial accelerators, and with respect to synthesis techniques for parallel accelerators that do not employ them. We then finally show how the generated accelerators mainly are bandwidth bound, performing a design space exploration and identifying the configurations (in terms of number of parallel hardware kernels and memory ports) that maximize bandwidth utilization. In summary, the contributions of this paper are:

- the study of load unbalance and its effect on performance for custom accelerators of prototypical irregular algorithms such as graph database queries;
- the design of the HMI;
- the design of the DTS;
- their validation and evaluation in the context of the synthesis of custom parallel accelerators.

The paper proceeds as follows. Section 2 motivates the papers. Section 3 presents the design of the HMI and of the DTS. Section 4 proposes the experimental evaluation, while Section 5 compares our approach to other solutions. Finally, Section 6 concludes the paper.

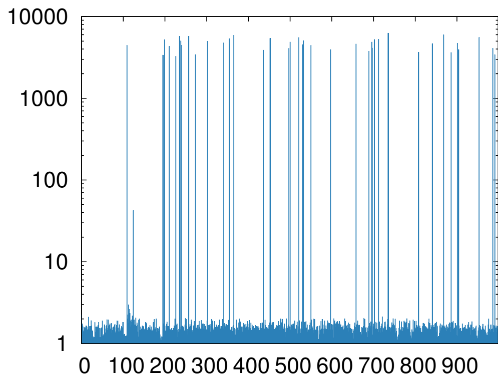
2. MOTIVATION

Graph methods, as employed in graph databases, and, in general, irregular applications present unique behaviors that complicate the design of workload-optimized accelerators. These applications typically expose significant Task Level Parallelism (TLP), but only limited Instruction Level Parallelism (ILP) . They are mainly bound by memory operations. Thus, accelerators for this class of applications should mainly exploit coarse-grained parallelism and support parallel memory architectures. However, these two features alone do not guarantee optimal performance. In fact, the parallelism in these applications is highly dynamic and the (potentially concurrent) tasks often are unbalanced, because they can have different durations, touch different quantities of data, or require mutual exclusion on the same data elements. Additionally, it is often impossible to statically estimate task latencies, because their behavior strictly depends on the actual data they process. Thus, the execution paradigms based on static scheduling typically employed in HLS do not provide effective load balancing.

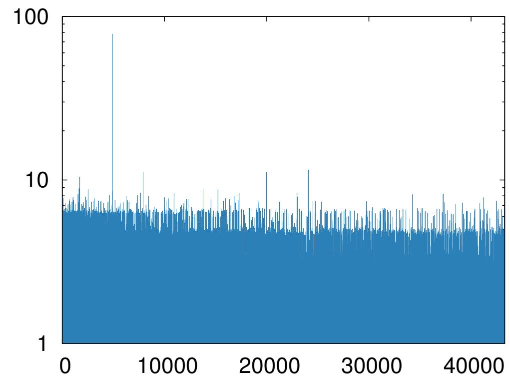
We highlight the impact of load unbalance by profiling a representative case study. For this purpose, we have chosen SPARQL queries Q1-Q7 [16] from the Lehigh University Benchmark (LUBM) suite. LUBM consists of a university domain ontology, customizable and reproducible synthetic data, and a set of test queries [11]. We considered parallel C-language implementations of these queries that model them as graph pattern matching procedures executed on a graph representation of the data, similarly to [4]. There exist RDF databases infrastructures [5] that converts SPARQL queries to graph methods in C/C++. For this type of databases, accelerating whole queries rather than only some primitives is not a limitation: usually, analysts have a pre-defined set of queries that do not change much over time. Datasets, instead, are periodically updated with varying frequencies depending on the availability of new data and regulations on data retention.

All the implementations basically consist of loop nests. Each loop iterates over the neighbor list of a specific vertex of the graph. In turn, each iteration of a loop matches an edge of the query pattern on the graph data. Each iteration potentially executes in parallel, and thus can be an independent task. However, the latency of each task can widely vary. For example, tasks can have different durations because of different sizes of the neighbors lists to explore. In another example, some loop iterations may find several complete or partial matches and thus explore a wide portion of the solution tree, while others may find no matches and terminate early. The behavior only depends on the input graph, and thus there is no way to determine or infer it statically. Our analysis profiles the execution latencies of the LUBM queries on a graph consisting of 5,309,056 edges (LUBM-40).

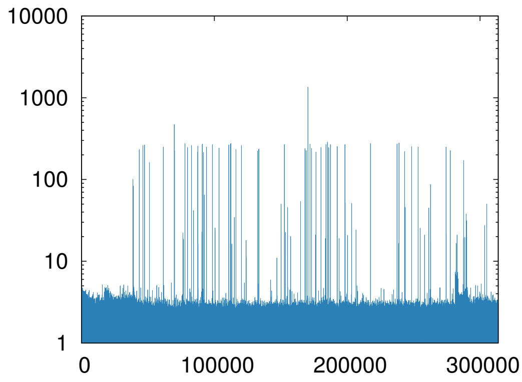
Figure 1 reports the execution latency of each iteration of the outer loop of each query, normalized to the shortest one. Our experiments show that most queries are highly unbalanced and that the execution time between subsequent iterations can differ of several orders of magnitude. In particular, we see that Q1, Q3 and Q6 are the most unbalanced queries, with iterations that last hundred times more than others. Moreover, the execution time of each iteration is



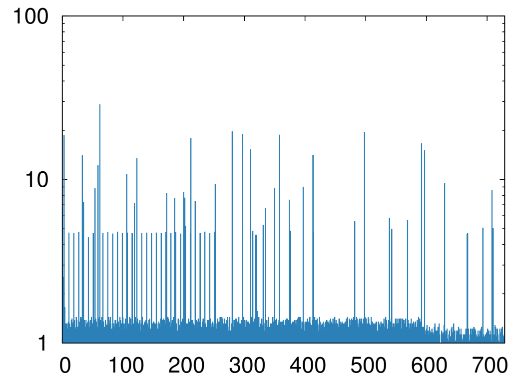
(a) Q1



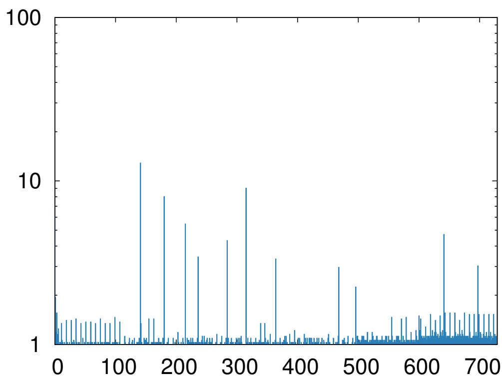
(b) Q2



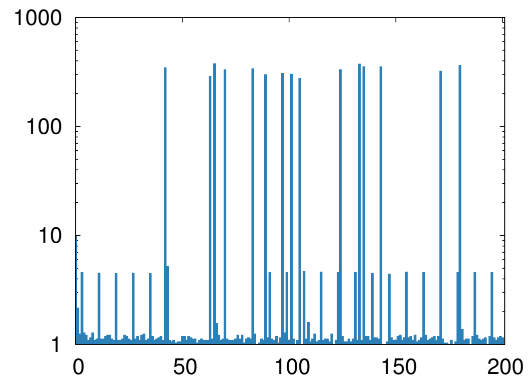
(c) Q3



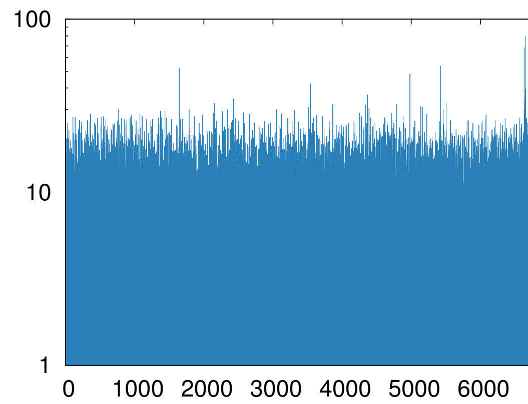
(d) Q4



(e) Q5



(f) Q6



(g) Q7

Figure 1: Analysis of the outer loop execution time of queries on LUBM-40. Data is normalized against the minimum.

```
#pragma parallel
for (int i = init();
      i != termination();
      i = nextValue()) {
  kernel(i, args);
}
```

Listing 1: Pseudo code of the target application template.

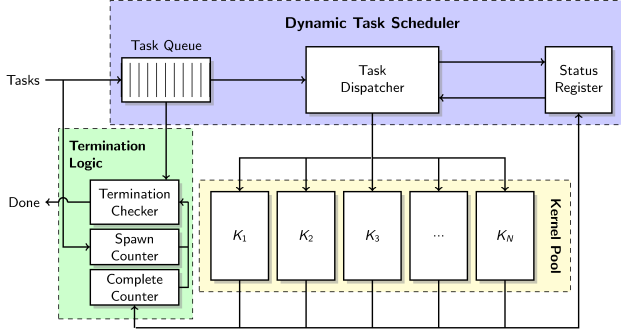


Figure 2: Schematic representation of the architecture template using the dynamic task scheduler

data dependent and, consequently, unpredictable at design time. For these reasons, executing multiple iterations in parallel with common fork/join paradigms results in sub-optimal performance: many available resources, associated with the shorter iterations, will indeed stay unused until the longer iterations complete, even though other new iterations could start execution. Thus, the overall execution latency is dominated by high-latency iterations/tasks, and concurrent execution provides very limited benefits. We address these issues by proposing a new accelerator design that allows issuing and executing tasks (in our case, different loop iterations) as soon as computing resources become available, i.e. as soon as other tasks complete.

3. METHODOLOGY

In our study, we structure graph methods and graph pattern matching algorithms as loop nests. We can parallelize them following the general template in Listing 1. Such a template represents the loop bodies as function calls (*kernel*). As usual in parallel programming, synchronization among multiple concurrent kernels is achieved through atomic memory operations. As discussed in Section 2, to effectively accelerate these algorithms we need to jointly: 1. exploit TLP; 2. mitigate the effects of load unbalance; 3. allow concurrent access to the memory. We meet these requirements by introducing a novel accelerator design template that adopts a dynamic execution paradigm.

3.1 Dynamic Task Scheduling

Like other solutions for TLP, our proposed architecture exploits spatial parallelism, i.e., it replicates custom Processing Units (PUs) that execute a single instance of a task.

However, in opposition to fork-join execution paradigms, our approach neither issues tasks in blocks (with size equal to the number of replicas), nor it statically binds a particular task to a particular PU. Instead, our design allows task to execute as soon as any PU becomes available, thus reducing the effects of load unbalance for irregular workloads.

Figure 2 shows a high-level schematic representation of the proposed template. It includes three basic components: the *Kernel Pool*, the *Dynamic Task Scheduler* (DTS) and, the *Termination Logic*.

The *Kernel Pool* is the set of PUs implementing the replicated hardware kernels. The DTS manages the parallel execution with the objective of maximizing resource utilization. It integrates three components:

- a *Task Queue* that keeps track of tasks ready for execution by storing their input parameters, such as the value of the induction variable;
- a *Status Register* that holds runtime information about resource utilization (available/computing) of the PUs in the Kernel Pool;
- a *Task Dispatcher* that dynamically assigns ready tasks to available PUs.

The component implementing the function that contains the parallel loop inserts a new task in the Task Queue every clock cycle, until the queue is full or all the tasks have been inserted. Then, it stalls waiting for additional space in the queue or for the completion of the parallel phase. The size of the Task Queue is configurable; in the current implementation, its size is not critical, because tasks are pushed and popped at the same frequency. However, the Task Queue allows decoupling the parallel architecture template from the rest of the circuit, enabling its sharing among multiple components. Whenever there are elements in the Task Queue, the Task Dispatcher checks the Status Register to find available PUs. If there is a PU available, the Task Dispatcher pops a task from the queue and starts its execution, updating the Status Register accordingly. Similarly, when a PU completes a task, the PU itself updates the Status Register to signal its availability to the Task Dispatcher. The Task Dispatcher schedules one task per clock cycle on an available PU. When multiple PUs are free at the same time, the Task Dispatcher picks one according to a static priority. In our experiments, we found that the best solution to implement the Task Dispatcher is to use a static table. Each entry in the table is a one-hot value that corresponds to a PU. The table rows are in accordance to the static priority of the PUs in the Kernel Pool. The Task Dispatcher addresses the table using the value stored in the Status Register, and obtains from it the next PU that is ready to start a new computation. In our implementation, the one hot will be at the position of the first zero of the Status Register. The table is automatically generated according to all the possible values that can be contained in the Status Registers. We found that this design provides higher frequency than a combinational circuit implementing the same functionality, as in [3]. The output of the decision table is then used to trigger the start signal of the selected PU and to update the Status Register accordingly. Starting a single computation every clock cycle is not a limitation, because tasks usually have a long execution time. On the other hand, this assumption significantly reduces the complexity of the circuit. We designed the PUs to register their inputs at front edge of their start signal so that the Task Dispatcher can broadcast task parameters from the Task Queue to the all the PUs in the Kernel Pool and trigger the start signal only of the selected PU. This design decision increases the fan-out at the output of the Task Dispatcher but avoids a potentially big mux tree. The

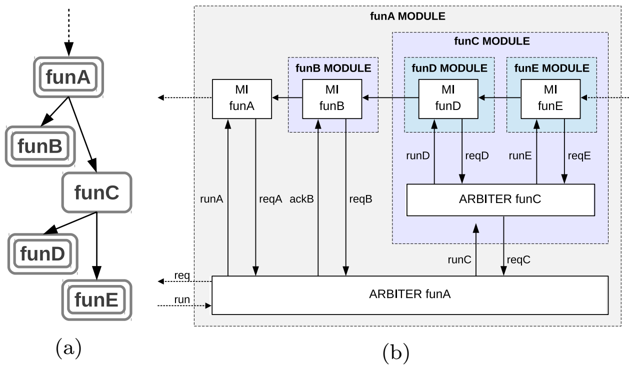


Figure 3: Example Call Graph (a) and associated memory interface structure (b). The framed nodes in the CG are associated with functions that directly perform memory accesses.

Termination Logic checks the termination condition of the parallel loop. It counts all the tasks spawned by the parallel loop and the completed tasks. In our design, we consider a task as spawned as soon as it has been pushed to the task queue, even tough its execution has not started yet. When the number of spawned task is equal to the number of completed tasks, the *Termination Logic* asserts a done signal, denoting the conclusion of the parallel phase. A particular feature of the proposed design is its modularity: every task can, in turn, spawn other tasks with the same architectural solution. While this dynamic execution paradigm considers different tasks as independent, tasks still share memory resources. Thus, this paradigm requires arbitration mechanisms to manage concurrency and synchronization among tasks.

3.2 Hierarchical Memory Interface Controller

The design of the Hierarchical Memory Interface controller (HMI) is based upon the custom Memory Interface Controller (MIC) described in [6]. The MIC:

- dynamically maps N unpredictable memory requests onto M memory ports, computing the destination addresses at runtime. Memory requests correspond to all the memory operations performed by the synthesized hardware kernels (i.e., the synthesis tool generates the appropriate signals and buses that connect the kernels to the memory interface whenever data are required). Kernels stall until the data operation terminates. A hardware addressing unit dynamically resolves the addresses, as data are distributed on different banks accessed from the different memory ports. The hardware unit distributes ("scrambles") data across the banks, following different policies with the objective of reducing contention with irregular data structures;

- manages concurrency: it collects memory requests, and if their target addresses collide, it serialize them. A dedicated arbiter manages each memory port. The arbiters only employ combinational logic that does not introduce any delay penalty;

- implements atomic operations (e.g. atomic increment, compare and swap) through dedicated hardware units. From a behavioral perspective, while running, atomic operations lock the associated memory port.

Table 1: LUBM-1: performance comparison of the implementation using the DTS+HMI against the serial and the parallel controller implementations

	Single Acc. # Cycles	Parallel Controller # Cycles	Dynamic Scheduler # Cycles	Speedup Single Acc.	Speedup Parallel Controller
Q1	5,339,286	5,176,116	5,129,902	1.04	1.01
Q2	141,022	54,281	50,997	2.77	1.06
Q3	5,824,354	1,862,683	1,805,731	3.23	1.03
Q4	63,825	42,851	19,928	3.20	2.15
Q5	33,322	13,442	9,016	3.70	1.49
Q6	674,951	340,634	197,894	3.41	1.72
Q7	1,700,170	694,225	492,280	3.45	1.41

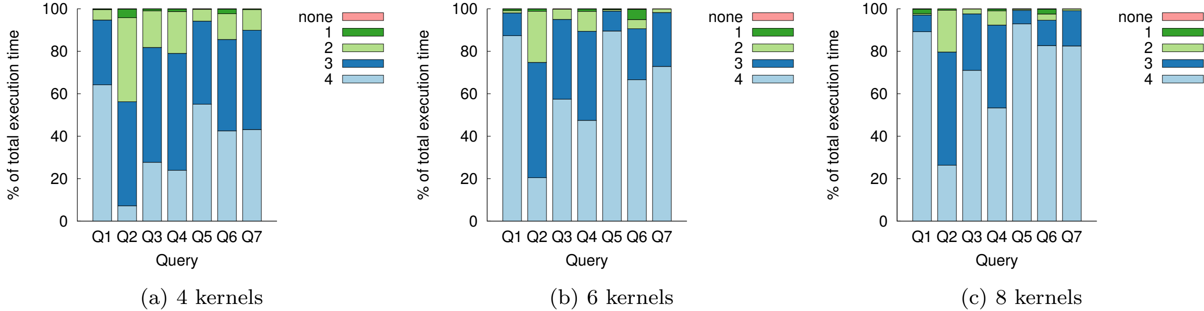
Table 2: LUBM-40: performance comparison of the implementation using the DTS+HMI against the serial and the parallel controller implementations

	Single Acc. # Cycles	Parallel Controller # Cycles	Dynamic Scheduler # Cycles	Speedup Single Acc.	Speedup Parallel Controller
Q1	1,082,526,974	1,001,581,548	287,527,463	3.76	3.48
Q2	7,359,732	2,801,694	2,672,295	2.75	1.05
Q3	308,586,247	98,163,298	95,154,310	3.24	1.03
Q4	63,825	42,279	19,890	3.21	2.13
Q5	33,322	13,400	8,992	3.71	1.49
Q6	682,949	629,671	199,749	3.42	3.15
Q7	85,341,784	35,511,299	24,430,557	3.49	1.45

We have refined the design of the MIC, proposing a hierarchical implementation of the component. While the centralized MIC is a customizable component allocated at the top level of the design hierarchy (i.e., the root of the call graph of the application), the proposed HMI distributes the arbitration logic across the design hierarchy to preserve the design modularity. By exploiting the HMI, every level of the design, from root to leaves, presents the same structure. This basically improves modules re-usability, since every sub-module can be used as top-level module as well. Figure 3 provides a schematic view of the HMI. Every module in the design hierarchy presents the same Memory Interface (MI), and MIs are *chained* from leaves to top. The first element in the chain (*MI funE* in Figure 3a) receives inputs from the memory (e.g. data loaded), while the last element provides inputs to the memory (e.g. memory addresses, data to be written). For each level in the hierarchy there is an arbiter per memory partition that avoids simultaneous execution of operations targeting the same memory partition, providing concurrency management. Each arbiter sends an execution request token to the upper level arbiter (until the top is reached) and forwards ack signals coming from the upper level to the lower levels arbiters. Figure 3a shows an example of Call Graph (CG): framed nodes denote functions (*funA*, *funB*, *funD*, *funE*), which directly access the shared memory. Figure 3b shows the associated memory interfaces. Function *funC* does not perform any memory access, but the called functions (*funD*, *funE*) do. For this reason, funC is involved in the management of memory concurrency at the caller level (funA). Another key difference of the HMI design is that the MIC also embeds the hardware components that implements atomic memory operations. In our design instead, atomic operations are supported through dedicated signaling: when a kernel executes an atomic memory operations, it employs dedicated signals that request to lock the associated memory partition.

4. EXPERIMENTAL EVALUATION

To evaluate the effectiveness of the proposed approach we



(a) 4 kernels (b) 6 kernels (c) 8 kernels

Figure 4: Profiling of memory operations during query execution on LUBM-40.

have synthesized the queries with three different configurations. The first configuration implements 4 hardware kernels ($T = 4$) and the HMI with 4 memory channels ($M = 4$), to be directly comparable to [4], which proposes an architecture for query processing implementing a fork-join paradigm. The two other configurations differ from the first one only in the number of hardware kernels implemented (respectively, $T = 6$ and $T = 8$), while the number of memory channels of the HMI stays unchanged at 4. We have synthesized all the designs with Vivado 2015.4, targeting a Xilinx Virtex-7 xc7vx690t. We set the target frequency for the synthesis to 100 MHz and, for the performance analysis, simulate the designs with Modelsim 10.4c. We report all the synthesis results post-place and route.

Table 1 and Table 2 shows the performance comparison among the serial implementations, the parallel implementations ($T = 4, M = 4$) with the Distributed Controller (DC) architecture introduced in [4], and the proposed parallel implementation with the HMI and the DTS of the queries, on the LUBM-1 (100,573 triples) and on the LUBM-40 (5,309,056 triples) datasets, respectively. The two tables report performance in terms of execution latency. Comparing the performance against the serial implementations, the architectures employing the DTS and the HMI generally show a speed up close to the theoretical maximum (up to 3.76). Moreover, the simulations highlight that the DTS provides significant speed ups also against the DC parallel accelerators (between 1.41 and 2.25 on LUBM-1, between 1.45 and 3.48 on LUBM-40), especially on queries that present high load unbalance among tasks (queries Q1, Q4, and Q6 as shown in Figure 4). This result confirms that the adopted dynamic execution paradigm, in opposition to fork-join approaches, effectively mitigates the effects of load unbalance.

We have measured the utilization of the memory channels in our accelerators integrating both the HMI and the DTS. We recorded how many memory channels are in use in each clock cycle during the parallel phases of the queries. Figure 4 reports the data collected as a percentage of the execution time of the parallel sections. The plots show that the configuration with $T = 4, M = 4$ is using at least 3 memory channels for more than 75% of the execution time for all the queries except Q2.

The performance trends show two outliers: Q1 with the LUBM-1 dataset and Q2. The profiling of Q1 on the LUBM-1 datasets shows that a single task lasts for 87% of the whole execution time, thus explaining the limited speed up (very close to 1) of the parallel implementations with respect to the serial one. At the opposite, Figure 1b shows that Q2 is not heavily unbalanced. However, in Figure 4a we see that Q2 has the lowest utilization of the memory channels. This may be caused by structural conflicts on the memory resources: Figure 4 shows indeed that increasing the number of parallel kernels in Q2 does not increase the utilization of the memory channels.

Table 3 compares the area of the accelerators implemented with the HMI and the DTS against the serial accelerators and the accelerators implemented with the parallel controller as in [4]. The table reports the number of Look Up Table (LUTs) and Slices post place and route. For each design point, we also report the maximum reachable frequency. The implementations with the HMI and the DTS range from 1.72 up to 1.94 times the occupation of the serial implementations. However, they are 20% smaller on average than the corresponding implementations with the parallel controller. Considering that the speedup over the serial implementations is always greater than 2.5, the adoption of the DTS results highly profitable in terms of the area/performance ratio.

Table 4 shows area (LUTs and Slices), performance (execution latency in clock cycles), and maximum frequency of accelerators implementing the HMI an the DTS respectively with $T = 6$ and $T = 8$ kernels. We can see that, in accelerators employing the DTS, the area scales linearly with the number of kernels. The maximum frequency slightly drops, but it meets the imposed timing constraint (100 MHz) for all the designs. These two configurations provides higher utilization of the memory channels, as shown in Figure 4b and Figure 4c. Both the configurations use at least 75% of the memory channels for 90% of the execution time. The configuration with 8 kernels uses all the memory channels for more than 80% of the execution time in four of the seven queries. We also see that increasing the number of parallel kernels from $T = 6$ to to $T = 8$ with a fixed number of memory channels (4) provides diminishing returns in terms of performance. In fact, the performance only increases from 1% and 6% for all the queries except Q1. Q1 even shows a small performance drop, presumably because of memory contention. We can infer that increasing the number of kernels beyond 8 would not be worth the cost in area without also increasing the number of memory channels.

5. RELATED WORK

Our proposed approach aims at accelerating irregular applications by jointly exploiting TLP and multi-ported/distributed shared memories. By implementing dynamic scheduling and out-of-order execution, our approach also addresses

Table 3: Comparison of Synthesis results of the DTS+HMI implementation against the serial and parallel controller implementation

	Serial			Parallel Controller			Dynamic Scheduler			Area Overhead Serial		Area Overhead P. Controller	
	LUTs	Slices	Max. Freq.	LUTs	Slices	Max. Freq.	LUTs	Slices	Max. Freq.	LUTs	Slices	LUTs	Slices
Q1	5,600	1,802	130.34MHz	13,469	4,317	113.37MHz	10,844	3,503	113.60MHz	1.94	1.94	0.81	0.81
Q2	2,690	824	143.66MHz	5,280	1,607	130.11MHz	4,636	1,335	132.87MHz	1.72	1.62	0.88	0.83
Q3	5,525	1,775	121.27MHz	13,449	4,308	114.53MHz	10,664	3,467	116.92MHz	1.93	1.95	0.79	0.80
Q4	3,477	1,073	143.20MHz	7,806	2,399	122.97MHz	6,175	1,918	118.68MHz	1.78	1.79	0.79	0.80
Q5	2,785	848	133.92MHz	5,750	1,738	138.31MHz	5,330	1,578	114.51MHz	1.91	1.86	0.93	0.91
Q6	4,364	1,369	136.76MHz	10,600	3,426	113.26MHz	8,125	2,633	118.68MHz	1.86	1.92	0.77	0.77
Q7	6,194	1,943	131.98MHz	15,002	4,953	106.71MHz	11,344	3,747	113.23MHz	1.83	1.93	0.76	0.76

Table 4: Performance and Synthesis results of the DTS+HMI architecture with 6 and 8 kernels

	T=6, CH=4				T=8, CH=4			
	LUTs	Slices	Latency	Max. Freq	LUTs	Slices	Latency	Max. Freq
Q1	15,305	4,822	268,093,088	111.58MHz	20,286	6,469	268,491,462	104.08MHz
Q2	6,507	1,942	2,355,699	113.45MHz	8,429	2,381	2,268,763	112.47MHz
Q3	15,259	4,943	83,327,993	106.19MHz	20,078	6,486	79,649,000	102.67MHz
Q4	9,015	2,807	17,894	113.63MHz	11,830	3,580	17,428	112.23MHz
Q5	7,370	2,190	8,104	113.55MHz	9,241	2,788	8,022	110.02MHz
Q6	11,725	3,806	184,879	107.37MHz	15,467	5,074	173,616	103.58MHz
Q7	16,408	5,347	21,902,616	112.01MHz	21,770	7,079	21,300,052	106.00MHz

load unbalance with tasks of different latencies. Several solutions that exploit TLP involve the automated synthesis of custom parallel accelerators, but require the introduction of custom schedulers or processors to coordinate the execution of tasks. The approach presented in [12] maps tasks, identified by partitioning the input behavior, onto custom Processing Units (PU) and manages their execution through a top-level controller. [2] similarly manages task execution through a *Control Processor* (CP), which represents the top layer of a Multi-Level Computing Architecture (MLCA) [13]. The lower level of the design is a set of PUs: they may be custom accelerators or soft-cores. The CP has the main role of scheduling and mapping tasks onto PUs, considering a top-level control program that consists of *task instructions*. In [12] the authors also identify concurrency on memory resources as a bottleneck for performance, and propose the adoption of distributed memories. The approach relies on the coordinated scheduling of the individual partitions to avoid conflicts for parallel memory accesses. However, it focuses on applications that present nested loops with affine array indices, making the methodology not applicable in the presence of irregular access patterns. [12] further investigates the adoption of an arbiter for managing memory concurrency as an alternative to coordinated scheduling. However, such arbiter implements a handshaking protocol among the different partitions, and introduces several clock cycles of delay. The LegUp framework provides both a conventional HLS flow for the synthesis of full-hardware accelerators, and a MLCA flow [7]. The latter allows the automatic generation of designs that couple a MIPS processor with custom PUs. This approach enables the concurrent execution of parallel kernels, identified from OpenMP and pthreads specifications. The kernels may forward access requests to the shared memory one at a time, managed by a round robin arbiter. The support for spatial parallelism (i.e., kernel modules duplication) makes the approach particularly profitable when targeting computationally intensive specifications, but less convenient when synthesizing memory-intensive applications, especially when increasing the number of kernels. The reason is the contention on the shared memory. Our approach mitigates the impact of the memory bottleneck by allowing concurrent access to distributed/multi-ported memories, supported through the HMI controllers. In addition, compared with these hybrid solutions, our approach generates custom accelerators able to exploit TLP without the intervention of external control units or soft-cores.

[15] discusses a HLS flow that translates OpenMP programs in synthesizable Handel-C or VHDL. However, such a flow imposes severe restrictions on the input specifications: for example, it does not support global variables. Moreover, it does not consider ILP: each kernel executes serially, and only performs one operation at a time. [8] discusses the use of OpenMP for specifications in system-level design. The approach targets hardware/software system architectures, and maps OpenMP threads either on software threads or on custom hardware components, synthesized with a conventional HLS flow. In general, it assigns (mostly) data-intensive threads to dedicated hardware modules, and control intensive threads to software. The authors identify several limitations that make the adoption of full-hardware approaches less attractive:

- complexity and costs of pure hardware implementations require dropping several features such as dynamic scheduling and nested parallelism;
- interactions among threads are managed through centralized mechanisms, leading to scalability issues when increasing the number of threads;
- the support for external memory systems is limited.

Our proposed techniques overcome most of these limitations. Our flow is able to exploit parallelism at every level of the call graph. The design of the accelerators and of the HMI allows concurrent management of independent kernels through lightweight communicating components, greatly improving scalability. Finally, the HMI allows accessing external memories and exploiting the increased bandwidth provided by distributed and multi-ported memories.

In [17] Tan et *al.* describe a pipelined architecture targeted at the HLS of irregular loop nests. Their approach synthesizes a pipelined loop as a set of Loop Processing Units (LPUs). Each iteration of the loop is dynamically assigned

to one of the available LPUs through a Distributor. All the LUPs are then connected to a *Collector* that passes results to the next stage of the pipelined loop. The authors introduce a *reorder buffer (ROB)* to ensure that results are committed in the same order as in the original loop. Our architecture also implements a dynamic scheduling approach, but it mainly targets TLP exploitation rather than loop pipelining. Our design does not require a *ROB* to guarantee correctness, and it achieve consistency of memory operations through the synchronization primitives implemented by the HMI.

In [4], Castellana *et. al* present a complete HLS methodology for the synthesis of RDF queries. The approach proposed in the paper employs an adaptive Distributed Controller (DC) to implement (task) parallel accelerators. The DC exploits token passing mechanisms to track dependencies among operations and manage the concurrent execution flows. However, the DC implements a fork/join strategy that spawns tasks in groups. If tasks of the same group are unbalanced (i.e., have different execution times), resources become significantly underutilized. Our design overcomes this limitation by employing the DTC to assign tasks to resources as soon as they are available.

6. CONCLUSION

We presented an architectural template that improves the HLS of parallel accelerators for irregular applications. Irregular applications are mostly memory bound, present unpredictable, fine-grained data accesses, are synchronization intensive, and usually employ very large datasets that are difficult to partition without generating load unbalance. In particular, we targeted RDF databases, which map to graphs, and SPARQL queries, which perform graph walks and graph pattern matching operations. We employed the Lehigh University Benchmark (LUBM) a typical benchmark for this class of applications as case study. We have initially shown that the queries present tasks with significantly different execution times, potentially leading to resource underutilization and load unbalancing. We then described two components, the Dynamic Task Scheduler and the Hierarchical Memory Interface controller (HMI). These components address some of the issues of irregular applications. The DTS provides dynamic task scheduling, enabling better load balancing of tasks across parallel hardware kernels. The HMI provides memory parallelism by interfacing the kernels to multi-ported shared memories. It also provides support for atomic memory operations. Parallel accelerators for SPARQL queries generated with this architectural template provide speed ups close to the theoretical maximum (i.e., the number of parallel kernels implemented in the architecture) with respect to serial accelerators, and speed ups from 1.41 to 3.48 with respect to parallel accelerators without load balancing. Accelerators employing the proposed architectural template also have a smaller area footprint with respect to the baseline parallel accelerators. We finally explore how the proposed architectural template exploits the available memory channels, generally reaching high utilization.

7. REFERENCES

[1] B. Betkaoui, Y. Wang, D. Thomas, and W. Luk. A Reconfigurable Computing Approach for Efficient and Scalable Parallel Graph Exploration. In *ASAP'12: IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, pages 8–15, 2012.

[2] D. Capalija and T. Abdelrahman. An Architecture for Exploiting Coarse-grain Parallelism on FPGAs. In *FPT'09: International Conference on Field-Programmable Technology*, pages 285–291, 2009.

[3] V. G. Castellana and F. Ferrandi. An automated flow for the high level synthesis of coarse grained parallel applications. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 294–301. IEEE, 2013.

[4] V. G. Castellana, M. Minutoli, A. Morari, A. Tumeo, M. Lattuada, and F. Ferrandi. High Level Synthesis of RDF Queries for Graph Analytics. In *ICCAD'15: IEEE/ACM International Conference on Computer-Aided Design*, pages 323–330, 2015.

[5] V. G. Castellana, A. Morari, J. Weaver, A. Tumeo, D. Haglin, O. Villa, and J. Feo. In-Memory Graph Databases for Web-Scale Data. *Computer*, 48(3):24–35, Mar 2015.

[6] V. G. Castellana, A. Tumeo, and F. Ferrandi. An adaptive memory interface controller for improving bandwidth utilization of hybrid and reconfigurable systems. In *DATE'14: Design, Automation and Test in Europe*, pages 1–4, 2014.

[7] J. Choi, S. Brown, and J. Anderson. From software threads to parallel hardware in high-level synthesis for FPGAs. In *FPT'13: International Conference on Field-Programmable Technology*, pages 270–277, 2013.

[8] A. Cilardo, L. Gallo, A. Mazzeo, and N. Mazzocca. Efficient and scalable OpenMP-based system-level design. In *DATE'13: Design, Automation, and Test in Europe Conference*, pages 988–991, 2013.

[9] C. Computer. Convey Computer Doubles Graph500 Performance, Develops New Graph Personality. http://www.conveycomputer.com/files/2413/5095/9078/sc11_graph500_release.final.pdf.

[10] C. Computer. Convey MX Series. Architectural Overview. At http://www.conveycomputer.com.

[11] Y. Guo, Z. Pan, and J. Heflin. Lubm: A Benchmark for OWL Knowledge Base Systems. *Web Semant.*, 3(2-3):158–182, Oct. 2005.

[12] C. Huang, S. Ravi, A. Raghunathan, and N. Jha. Generation of Heterogeneous Distributed Architectures for Memory-Intensive Applications Through High-Level Synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(11):1191–1204, 2007.

[13] F. Karim, A. Mellan, A. Nguyen, U. Aydonat, and T. Abdelrahman. A Multilevel Computing Architecture for Embedded Multimedia Applications. *IEEE Micro*, 24(3):56–66, 2004.

[14] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and abstract syntax. W3C Recommendation 10 February 2004. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210.

[15] Y. Y. Leow, C. Y. Ng, and W. Wong. Generating Hardware from OpenMP Programs. In *FPT'06: IEEE International Conference on Field Programmable Technology*, pages 73–80, 2006.

[16] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *MOD'13: ACM SIGMOD International Conference on Management of Data*, pages 505–516, 2013.

[17] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang. Elasticflow: A complexity-effective approach for pipelining irregular loop nests. In *ICCAD'15: IEEE/ACM International Conference on Computer-Aided Design*, pages 78–85, 2015.

[18] A. Tumeo and J. Feo. Irregular Applications: From Architectures to Algorithms. *Computer*, (8):14–16, 2015.