High Level Synthesis of RDF Queries for Graph Analytics

Invited Paper

Vito Giovanni Castellana, Marco Minutoli, Alessandro Morari, Antonino Tumeo
Pacific Northwest National Laboratory
Richland, WA, USA
{vitoGiovanni.castellana, marco.minutoli, alessandro.morari, antonino.tumeo}@pnnl.gov

Marco Lattuada, Fabrizio Ferrandi
Politecnico di Milano - DEIB
Milano, Italy
{marco.lattuada, fabrizio.ferrandi}@polimi.it

***Abstract*—In this paper we present a set of techniques that enable the synthesis of efficient custom accelerators for memory intensive, irregular applications. To address the challenges of irregular applications (large memory footprint, unpredictable fine-grained data accesses, and high synchronization intensity), and exploit their opportunities (thread level parallelism, memory level parallelism), we propose a novel accelerator design that employs an adaptive and Distributed Controller (DC) architecture, and a Memory Interface Controller (MIC) that supports concurrent and atomic memory operations on a multi-ported/multi-banked shared memory. Among the multitude of algorithms that may benefit from our solution, we focus on the acceleration of graph analytics applications and, in particular, on the synthesis of SPARQL queries on Resource Description Framework (RDF) databases. We achieve this objective by incorporating the synthesis techniques into Bambu, an Open Source high-level synthesis tools, and interfacing it with GEMS, the Graph database Engine for Multithreaded Systems. The GEMS' front-end generates optimized C implementations of the input queries, modeled as graph pattern matching algorithms, which are then automatically synthesized by Bambu. We validate our approach by synthesizing several SPARQL queries from the Lehigh University Benchmark (LUBM).**

I. Introduction

The exponential growth in the availability of data in many areas, such as finance, commerce, government, healthcare, cybersecurity, communication networks, transportation networks, social networks, and the Web, is driving the needs for effective methods to extract value from the data itself. The challenges reside not only in the unprecedented size of the data to process, but also in the requirement to process them as quickly as possible to provide actionable answers to queries. Data, in fact, dynamically changes and late results could be not useful or even generate major risks. To address these issues, researchers have started to explore the use of High Performance Computing (HPC) approaches and techniques to Data Analytics, hence giving birth to High Performance Data Analytics (HPDA).

A distinct characteristic of these data is that they usually are unstructured or poorly structured. Data structures such as graph appear capable of organizing the collected data in a supportive manner: graphs are space efficient and can effectively represent the dynamically changing relationships among the elements of the datasets by adding or removing edges. Unfortunately, graphs are prototypical irregular data structures. Their exploration is inherently parallel, but algorithms proceed by executing fine-grained, unpredictable data accesses and exhibit high synchronization intensity. In addition, parallelism is dynamic and datasets are difficult to partition without generating load imbalance among the concurrent activities. Modern HPC systems are multi-node clusters that implement multi-core processors with complex cache hierarchies, which reduce memory access latency and improve performance of regular workloads. They also have high floating point performance, which the coupling with accelerators, such as Graphic Processing Units (GPUs), increases even more. Finally, the network interconnect between nodes is optimized for large, batched transfers, and becomes heavily under-utilized with small messages. Because of these characteristics, graph-based algorithms and, in general, data analytics perform poorly on these systems. The Graph database Engine for Multithreaded Systems (GEMS) [1] is one of the first relevant examples of HPDA applications. GEMS implements a Resource Description Framework (RDF) database on a commodity cluster by mainly employing graph methods at all levels of his stack. To address the limitations of HPC systems, GEMS employs a runtime that provides: a global address space across the cluster, so that data do not need to be partitioned, lightweight software multithreading, to tolerate data access latencies, and message aggregation, to improve network utilization with fine-grained transactions. A graph application programming interface (API) and a set of methods to ingest RDF triples and generate the related graph and dictionary are built with the functions provided by the runtime. On top of the whole system, a translator converts query expressed in SPARQL to graph-pattern matching operations.

Accelerators based on reconfigurable devices such as Field Programmable Gate Arrays (FPGAs) are emerging as a promising platform for HPDA applications. The Microsoft Catapult [2] project has integrated FPGAs in Microsoft-designed servers to improve performance, reduce power consumption, and provide new capabilities in the datacenter. The Convey HC and MX hybrid platforms integrate high density FPGAs with general-purpose processors, providing optimized high-bandwidth, host-coherent, memory controllers. The Convey

WX (Wolverine) accelerator is a PCI-Express drop-in solution that provides similar features. The high density of latest generation FPGAs enables exploiting the inherent parallelism of data analytics applications by physically replicating the kernels. The possibility to customize the accelerators' design allows better exploiting the available memory bandwidth, coping with the peculiar applications' features. However, designing accelerators by employing Hardware Description Languages (HDL) is hard and time-consuming. Hand-designed accelerators usually provide very high performance, but can address only a very specific set of algorithms. High Level Synthesis approaches, which generate HDL starting from descriptions in higher-level languages (such as C/C++), try to bridge this productivity gap. Historically, HLS tools have targeted regular, compute intensive applications. The reason is that regularity allows applying a large number of transformations, enabling extraction of Instruction Level Parallelism (ILP). Although modern HLS solutions try to better support task-parallel specifications, they still mainly focus on regular and compute intensive code. This allows primarily dealing with known latencies and exploiting latency reduction techniques. Only a very limited number of approaches started looking at possibilities to address issues of irregular and memory-intensive workloads.

In this paper we discuss how we extended Bambu, an open-source HLS synthesis tool, with techniques to better support HPDA applications. Differently from other HLS approaches, which focus on extracting Instruction Level Parallelism (ILP) and generate a statically scheduled Finite State Machine with Datapath (FSMD), our tool is able to exploit Task Level Parallelism (TLP) by employing adaptive Distributed Controllers (DCs). The DCs, structured as a set of interacting control elements, enable dynamic scheduling. The resulting set of kernels can concurrently access multi-banked/multi-ported high bandwidth shared-memories through a Memory Interface Controller (MIC). The MIC provides concurrency control, dynamic routing of memory accesses and conflicts management, and supports atomic memory operations. We show a real use case for this HLS flow, starting from GEMS and queries from the Lehigh University Benchmark (LUBM). Whereas GEMS executes the queries converted to C/C++ through its multithreaded runtime, we synthesize the resulting C/C++ graph pattern matching algorithms onto an FPGA through Bambu. The generated accelerator implements a full query. We evaluate the flow with datasets of varying size, while changing the number of parallel kernels that implement the queries and the number of memory ports.

The paper proceeds as follows. Section II briefly surveys the related work. Section III presents the proposed architecture template, detailing the DC, the MIC, and discussing how they are implemented in the HLS flow. Section IV describes the integration of GEMS and Bambu. Section V is about the experimental evaluation. Finally, Section VI concludes the paper.

II. Related Work

The approach discussed in this paper touches several areas of research in reconfigurable architectures and HLS. These include the synthesis of distributed controllers, the exploitation of task-level parallelism and the synthesis of parallel specifications, the design of accelerators for irregular and graph kernels (such as graph breadth first exploration) on reconfigurable devices, and accelerators for databases.

Synthesis of distributed controllers. Because conventional HLS tools generate accelerators that exploit centralized controllers, the majority of the approaches look at decomposing the FSM to reduce its complexity. Among the variety of works, we highlight approaches that restructure the controller in a hierarchical way [3], [4], even using State Charts descriptions [5]. Some solutions, like [6], employ a pseudo-distributed approach that enables supporting Speculative Functional Units. The final architecture still relies on a static schedule, but a local controller dynamically checks results of SFUs without stalling the whole datapath. Our approach, instead, is built from the beginning with distributed controllers, and does not consider any fixed schedule, avoiding runtime conflicts on shared resources through arbiters.

Task-Level Parallelism Exploitation. The use of a centralized FSM to exploit parallelism across the boundary of basic blocks may lead to an exponential increase in complexity. Several approaches solve the problem by synthesizing *tasks* independently and then managing their execution through custom schedulers or dedicated processors [7], [8]. Our approach, instead, does not require any additional control unit.

HLS of Parallel Specifications. Various commercial and research HLS flows started considering parallel specifications as input descriptions. These include specifications annotated in CUDA, OpenMP, OpenCL, and pthreads [9], [10]. LegUP [11] also supports OpenMP specifications, but requires the instantiation of an additional general purpose processor for scheduling. Our approach does not require an additional processor, and can support any level of nested parallelism. The generated hardware supports nested parallelism, but limited to only two levels of the call structure. OpenCL [12] is finding some success, also in commercial tools [13]. However, having been designed mainly for vector-based processors, it does not adapt well to irregular applications.

Accelerators for irregular applications. Prominent examples of designs to accelerate graph traversal and, in general, irregular kernels, are the BFS personalities for the Convey HC systems [14], and the Convey MX system, which couples a multithreaded custom processor on the reconfigurable logic with an OpenMP programming environment (CHOMP - Convey Hybrid OpenMP) [15]. Betkaoui et *al.* [16] discuss reconfigurable hardware methodologies for efficient parallel processing of large-scale graph exploration. These, however, either are custom accelerators for a specific kernel, or employ general-purpose designs on the FPGA. Our approach exploits a HLS approach. In [17], Halstead et *al.* discusses how to extend the ROCCC framework to support irregular applications, introducing multithreading to tolerate long memory access latencies. However, they do not address atomic memory operations and focus on the simple case study of pointer chasing.

Accelerators for databases. In the last few years, research in reconfigurable computing has focused on finding solutions to accelerate database operations and queries. IBM has proposed a FPGA-based system to accelerate expensive operations in relational databases queries, including data decompression and predicate evaluation [18]. [19] discusses FPGA acceleration of hash-joins on a ConveyMX, exploiting multithreading and the support for atomic memory operations provided by the system. IBM has also explored FPGA support for DB2 with BLU acceleration: compression techniques, paired with the Column-Store approach, enable performing most SQL operations on the compressed value, so that they can be processed in a Single Instruction Multiple Data (SIMD) fashion. Compilation of queries to FPGA for streaming databases has also been explored [20]. [21] presents a compiler based approach that translates SQL-based queries for software based Complex Event Processing systems in hardware. Casper and Olukotun in [22] show the potential of hardware acceleration for in-memory databases with select, sort, and join operations. Dennl et *al.* [23] discuss acceleration of the SQL restrict and aggregate operators, employing partial dynamic reconfiguration to compose query-specific datapaths. The poster [24] hints at the potential of the use of HLS to fully implement queries for in-memory databases by employing Vivado HLS. The integration of custom units in general purpose processors to accelerate analytics workloads and database operations has also been a recent topic of interest (e.g., [25]). Our approach is different from all these works. GEMS is a RDF database, that mostly uses graph methods at all levels of its stack and the SPARQL query language. Acceleration of conventional relational and table-based operations can only improve management of the result tables. We aim at fully synthesizing SPARQL queries, but GEMS translates them in graph pattern matching operations that require different architectural designs. We provide support for task parallel workloads, irregular memory accesses, and atomic memory operations.

III. Proposed Architecture

The majority of HLS techniques adopts the FSMD model for the target architecture. While very effective in exploiting ILP, this execution paradigm is inherently serial and does not efficiently exploit coarser granularities of parallelism, such as Task Level Parallelism (TLP). This is a significant limitation in several application domains. HPDA applications and, in general, irregular applications, although providing some ILP, typically are task parallel. To overcome the limitations of the FSMD model, we devised an alternative architecture design, able to handle concurrent execution flows through a DC.

A. Distributed Controller Architecture

The proposed design supports parallel execution and dynamic scheduling through the introduction of an adaptive DC [26]. The DC consists of a set of communicating modules, each one associated with an operation. The approach does not require the definition of any execution order (scheduling) at design time, and allows run-time exploitation of parallelism. The controller modules, called Execution Managers (EMs), start execution of the associated operations as soon as all their dependencies are satisfied and resource conflicts are resolved. The minimum set of dependencies each operation is subject to, called Activating Conditions (ACs), is computed by analyzing the Extended Program Dependencies Graph (EPDG) of the algorithm, which extends a typical Program Dependence Graph (PDG) with control-flow information, such as loops' back edges. ACs are expressed as logic functions, and specifically synthesized for each EM. Instead, dedicated arbiters, called

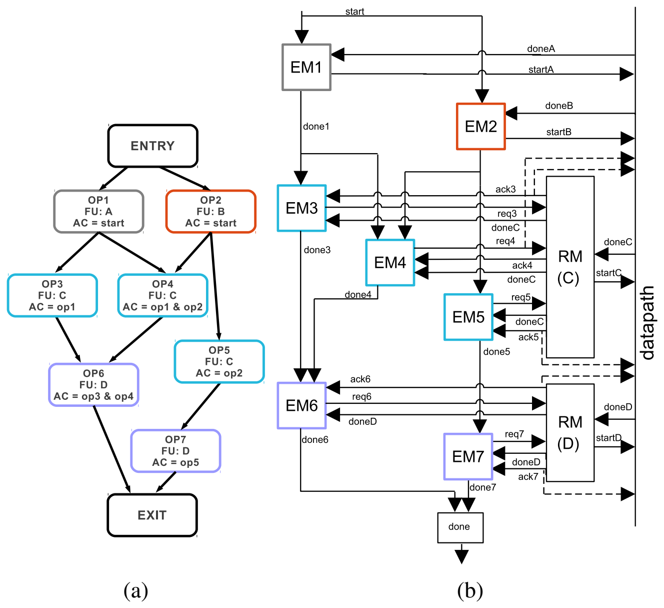


Fig. 1: Example Extended Program Dependencies Graph (a) and corresponding Distributed Controller architecture (b).

Resource Managers (RMs), associated to shared resources manage resource conflicts: if multiple operations compete for a resource, the arbiter establishes which one executes first, according to a priority ordering. EMs communicate through a lightweight token-based schema: each EM receives a token signal whenever a dependency gets satisfied. When the controller has collected all the AC tokens (i.e., all dependencies are satisfied), it checks for resource availability. If the resource associated with the operation is free, execution starts. The approach does not introduce any communication overhead, because it does not use any sophisticated protocol. Since every operation and function is managed independently, the DC can efficiently control several concurrent execution flows. Obtaining the same behavior with centralized FSMs is possible, but not cheap: in fact, the complexity of a FSM controller, in terms of number of states and transitions, is exponential with respect to number of flows. This complexity would lead to unfeasible designs even for relatively small degrees of TLP. The complexity of the DC instead, grows linearly with the number of operations, regardless of the latency of the operations and of number of concurrent flows. Figure 1 proposes an example of EPDG, annotated with ACs and binding information, and the associated parallel controller architecture. Operations 3,4,5 are bound to the same resource C, while operations 6,7 are bound to D: the corresponding EMs interface with RMs to avoid structural conflicts. In this example all the operations have unknown latency (e.g. external memory accesses, function calls, speculative operations) and the completion of their execution is notified through explicit done signals from the datapath to the EMs. If the execution latency is known at design time, this signaling is not required, and the EMs directly manage the timing.

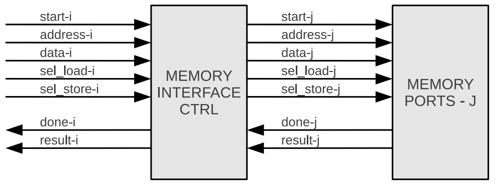


Fig. 2: Top Level Memory Interface Controller Structure.

B. Memory Interface

The mainstream approach for TLP in hardware synthesis is based on the replication of computing resources. Custom hardware components implements different tasks/threads, and the final design allocates multiple instances of such modules. In our approach we adopt the same strategy, binding concurrent function calls to distinct hardware components, thus allowing parallel execution. However, not all the resources can be straightforwardly replicated: this is the case of memory resources. Parallel applications usually present tasks that share data. Consequently, memory can be a shared resource, and allowing parallel execution of tasks requires managing concurrent memory operations. The memory bottleneck can considerably degrade performance, especially in memory bound applications. In fact, those applications may not have sufficient computation intensity to hide the memory latency. Solutions based on caching require the adoption of coherency protocols, and provide limited benefits, if any, in the absence of locality. More suitable architectural approaches are mostly based on memory distribution and/or partitioning. These techniques allow concurrent access to the memory by multiple operations, but introduce additional challenges:

- memory addresses usually are not statically known, thus destination locations must be identified at runtime;
- tasks may access the memory in parallel, thus they need synchronization;
- structural conflicts on shared memory resources have to be avoided.

In our approach we address these issues by incorporating in the synthesized architectures an adaptive Memory Interface Controller (MIC). The MIC completely manages concurrency and synchronization of the memory resources [27]. It dynamically maps memory operations across multiple, distributed and/or multi-ported memories, such as those available in hybrid systems. Figure 2 shows a high-level schematic representation of the MIC. The MIC takes in input memory access requests from N ports, which have an address, a data and an operation type (load/store) line. The MIC routes requests towards one of the M output ports by evaluating their addresses. It serves a request as soon as the corresponding port is available. In a similar way, it routes back M done signals (which notify termination of an operation) and the results (in case of loads) to the requesting operation. The memory is composed of M different and independent banks, and each output port accesses one bank. Each memory bank has non-overlapping addresses. Accesses are routed towards a specific memory port at runtime, providing efficient support of the unpredictable memory access patterns typical in irregular applications. Customizable control logic, synthesized according to the particular scrambling function that distributes the data on the memory system, performs the routing. A lightweight arbitration scheme, which avoids any structural conflict on shared resources and does not introduce any further delay, provides concurrency management. For arbitration, we employ RMs also in the MIC, similarly to the DC. Access routing and resource availability checks both occur at runtime, enabling the MIC to issue concurrent memory operations, provided that they do not address the same memory locations. This improves system memory bandwidth utilization. Support of atomic memory operations, such as fetch-and-add and compare-and-swap enables synchronization. The RMs reject further memory requests on a memory location accessed by an atomic memory operation, guaranteeing atomicity.

C. Synthesis Flow

We have implemented a complete HLS flow that automatically generates the proposed architecture by extending Bambu, a state-of-the-art HLS tool available under GPL. Bambu takes in input a C-code specification and synthesis objectives (e.g. target frequency and area), and outputs a Verilog implementation, directly synthesizable on a variety of devices from several vendors (Altera, Xilinx, Lattice). Bambu's conventional target architecture is a FSMD. Bambu's flow has three main components: front-end, synthesis and back-end. The front-end phase processes the input specification, employing the GNU Compiler Collection (GCC). The front-end analyzes the input specifications and applies code transformations and optimizations (loop unrolling, function inlining, constant propagation, etc). The process generates several graph-based Internal Representations (IR), such as Control Flow Graphs, Data Flow Graphs, Program Dependence Graphs and Call Graphs. The synthesis phase takes those IRs in input and synthesizes the application one function at a time, following the structure of the call graph. This results in a modular, hierarchical design. The main activities that the flow performs, as in most HLS approaches, are: operation scheduling; allocation and binding of functional units, registers and interconnections. Finally, the back-end generates the final circuit description in Verilog, together with the simulation and synthesis scripts that enables Bambu to directly interface with 3rd-party tools.

To generate the DC architecture, we either designed novel synthesis techniques or heavily customized previous approaches. In fact, the majority of HLS algorithms requires the definition of an execution schedule. The proposed approach, instead, does not consider any pre-determined execution ordering, because the DCs dynamically execute operations. When compared to the FSMD flow, the proposed approach mandates additional front-end analysis steps to build the EPDG and compute the ACs. From the point of view of HLS techniques, instead, we adopt custom algorithms for register [28] and module binding [29]. On the other hand, the introduction of the MIC does not require significant changes to the synthesis algorithms.

IV. Query Processing

Our framework generates the hardware implementation of SPARQL queries translated into C++ graph pattern matching routines. To obtain efficient C++ implementations, we exploit some functionality offered by the GEMS software stack. GEMS, the Graph database Engine for Multithreaded Systems [1], is a multilayer software infrastructures for graph analytics, composed of three main components:

1) GMT, Global Memory and Threading: GMT is a custom runtime library that provides features to improve efficiency of irregular applications on commodity clusters. GMT implements a global address space across all the (distributed) memories of the nodes in a cluster, removing the requirements to partition dataset. It tolerates latencies to access data in remote nodes through lightweight software multithreading. Finally, it increases network bandwidth utilization through message aggregation.

2) SGLib, Semantic Graph Library: SGLib implements the methods to load and access the database, to perform graph traversal, and to manipulate the data structures. The methods exploit GMT primitives, hiding the runtime primitives to higher levels of the stacks or to users that want to implement their own queries in C++.

3) SPARQL-to-C++ Translator: the translation process is composed of three main stages. The translator initially parses the query, and then constructs an algebraic representation (High Level Intermediate Representation - HLIR) of the query, expressed as a Directed Acyclic Graph (DAG). Then, the translator processes the HLIR to identify an optimal query plan, and serializes it, producing a Low Level Intermediate Representation (LLIR). Finally, the code emitter generates the output C++ code, built upon SGLIB primitives.

GEMS allows implementing RDF databases on top of Commodity Clusters. RDF is a data model proposed by the W3C that organizes data in forms of subject-predicate-object triples that naturally maps to directed, labeled graphs. It is typically used to collect and organize data coming from finance, government, healthcare, cybersecurity, transportation networks, communication networks, social networks and for the semantic Web. SPARQL is a common query language for RDF databases that expresses a query as the search of a graph pattern on the dataset. Differently from other RDF *triplestores*, which are typically built on top of conventional relational databases and have to resort to relational select and joins at some point during query processing, GEMS employs mainly graph methods in all layers of its stack. For this reasons, GEMS heavily adopts High Performance Computing techniques to speed up the graph algorithms.

Coupling GEMS and Bambu

Figure 3 illustrates how GEMS is integrated with Bambu. We have customized some of GEMS' layers to make them generate C implementations of the queries that Bambu can synthesize. Specifically, we extended SGLib, developing an alternate version of the graph API written in C-language that does not use the GMT runtime. We modified the code emitter accordingly. Figure 4 shows a sample SPARQL query, together with its graph pattern representation.

When processing this query, the custom SPARQL-2-C translator generates the C-code implementation as listed in Figure 5a.

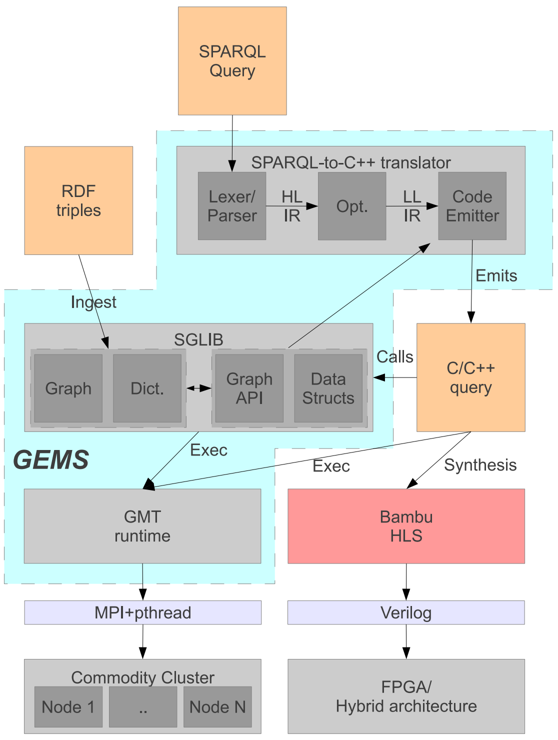
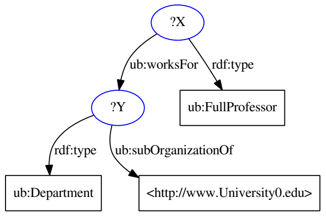


Fig. 3: Structure of the GEMS stack and interacion with Bambu HLS.

```
SELECT ?x ?y
WHERE {
 ?y ub:subOrganizationOf
   <http://www.University0.edu> .
 ?y rdf:type ub:Department .
 ?x ub:worksFor ?y .
 ?x rdf:type ub:FullProfessor
}
```

(a)



(b)

Fig. 4: Example Query Q6: *full professors working at a department of University0.*

The pattern matching function consists of a nest of parallel loops: each loop corresponds to matching a particular edge of the graph pattern that composes the query. In SPARQL queries, both vertices and edges may be either constant (represented through their value in the input data) or variable. The labels of constant elements, used the perform value checking during the query execution, acts as input parameters for the search function. This allows supporting with just one procedure different queries that differ only for those labels. Bambu processes the query implementation generated by GEMS. It applies code transformations to better expose TLP for the proposed architecture. In particular, Figure 5b shows how parallel loops are partially unrolled, with an unrolling factor equal to the number of kernel instances allocated in the synthesized architecture. Currently, the user must provide this number. The flow bounds all the kernel instances in the unrolled loop to different hardware modules during the synthesis, enabling concurrent execution. The DC manages the concurrent execution.

```
1  void search(Graph * graph, NodeId var_2, Label p_var_3, LabelId
        p_var_4, LabelId p_var_5, LabelId p_var_7, LabelId p_var_8,
        LabelId p_var_9) {
2    size_t in_degree_var_2 = getInDegree(graph, var_2);
3    Edge * var_2_1_inEdges = getInEdges(graph, var_2);
4    for(size_t i_var_3 = 0; i_var_3 < in_degree_var_2; i_var_3++) {
5      LabelId var_3; //el. with label "ub:subOrganizationOf"
6      var_3 = var_2_1_inEdges[i_var_3].property;
7      NodeId var_1; //el. with label "?Y"
8      var_1 = var_2_1_inEdges[i_var_3].node;
9      if(var_3 == p_var_3) {
10       size_t in_degree_var_1 = getInDegree(graph, var_1);
11       Edge * var_1_3_inEdges = getInEdges(graph, var_1);
12       for(size_t i_var_7 = 0; i_var_7 < in_degree_var_1; i_var_7++)
             {
13         LabelId var_7; //el. with label "ub:worksFor"
14         var_7 = var_1_3_inEdges[i_var_7].property;
15         NodeId var_6; //el. with label "?X"
16         var_6 = var_1_3_inEdges[i_var_7].node;
17         if(var_7 == p_var_7) {
18           size_t out_degree_var_6 = getOutDegree(graph, var_6);
19           Edge * var_6_5_outEdges = getOutEdges(graph, var_6);
20           for(size_t i_var_9 = 0; i_var_9 < out_degree_var_6;
                 i_var_9++) {
21             LabelId var_9; //el. with label "rdf::type"
22             var_9 = var_6_5_outEdges[i_var_9].property;
23             NodeId  var_8; //el. with label "ub:FullProfessor"
24             var_8 = var_6_5_outEdges[i_var_9].node;
25             if((var_9 == p_var_9) && (var_8 == p_var_8)) {
26               size_t out_degree_var_1 = getOutDegree(graph, var_1);
27               Edge * var_1_7_outEdges = getOutEdges(graph, var_1);
28               for(size_t i_var_5=0; i_var_5<out_degree_var_1;
                     i_var_5++) {
29                 LabelId var_5; //el. with label "rdf::type"
30                 var_5 = var_1_7_outEdges[i_var_5].property;
31                 NodeId  var_4; //el. with label "ub:Department"
32                 var_4 = var_1_7_outEdges[i_var_5].node;
33                 if((var_5 == p_var_5) && (var_4 == p_var_4))
34                   insertResults(var_6);
35               }
36             }
37           }
38         }
39       }
40     }
41   }
42 }
```

(a)

```
void kernel(size_t i_var3, Edge * var_2_1_inEdges, Graph
      * graph, NodeId var_2, Label p_var_3, LabelId
     p_var_4, LabelId p_var_5, LabelId p_var_7, LabelId
     p_var_8, LabelId p_var_9) {
  LabelId var_3; //el. with label "ub:subOrganizationOf"
  var_3 = var_2_1_inEdges[i_var_3].property;
  NodeId var_1; //el. with label "?Y"
  var_1 = var_2_1_inEdges[i_var_3].node;
  if(var_3 == p_var_3) {
    size_t in_degree_var_1 = getInDegree(graph, var_1);
    Edge * var_1_3_inEdges = getInEdges(graph, var_1);
    for(size_t i_var_7 = 0; i_var_7 < in_degree_var_1;
        i_var_7++) {
      //  Same as Fig. 5a lines [13--38]
      ...
    }
  }
}


void search(Graph * graph, NodeId var_2, Label p_var_3,
     LabelId p_var_4, LabelId p_var_5, LabelId p_var_7,
     LabelId p_var_8, LabelId p_var_9) {
  size_t in_degree_var_2 = getInDegree(graph, var_2);
  Edge * var_2_1_inEdges = getInEdges(graph, var_2);
  size_t i_var_3;

  for(i_var_3=0; i_var_3 < in_degree_var_2%4; i_var_3++)
      {
    kernel(i_var3, var_2_1_inEdges, graph, p_var_3,
         p_var_4, p_var_5, p_var_7, p_var_8, p_var_9);
  }

  for(; i_var_3 < in_degree_var_2%4; i_var_3+=4) {
    kernel(i_var3, var_2_1_inEdges, graph, p_var_3,
         p_var_4, p_var_5, p_var_7, p_var_8, p_var_9);
    kernel(i_var3+1, var_2_1_inEdges, graph, p_var_3,
         p_var_4,  p_var_5, p_var_7, p_var_8, p_var_9);
    kernel(i_var3+2, var_2_1_inEdges, graph, p_var_3,
         p_var_4, p_var_5, p_var_7, p_var_8, p_var_9);
    kernel(i_var3+3, var_2_1_inEdges, graph, p_var_3,
         p_var_4, p_var_5, p_var_7, p_var_8, p_var_9);
  }
}
```

(b)

Fig. 5: Pseudo code for the pattern matching routines of example query Q6

V. Experimental Results

We validated our approach by synthesizing a set of queries, such as the one in Figure 5, from the LUBM benchmark. We consider seven queries from the set used in [30]. Objective of LUBM is to evaluate performance of Semantic Web repositories in a standard and systematic way. It evaluates performance considering queries over a datasets originated from a single realistic ontology. LUBM consists of a university domain ontology, customizable and repeatable synthetic data, a set of test queries, and several performance metrics. We generated two different datasets: *LUBM-1*, consisting of 100,573 triples, and *LUBM-40*, consisting of 5,309,056 triples. A RDF triple corresponds to a subject-predicate-object clause. A set of triples naturally maps to a directed labeled graph.

To evaluate the effectiveness of the proposed approach, we synthesized the queries with two different configurations. In the first (*Serial*), the generated accelerator is serial (i.e., a single task). In the second, the generated accelerator is parallel, implementing 4 hardware kernels ($T = 4$) and a MIC with 4 memory channels ($M = 4$). We synthesized all the designs with Vivado 2015.1, targeting a Xilinx Virtex-7 xc7vx690t (the same device used in a Convey Wolverine WX690). We set a target frequency of 100 MHz for the synthesis. Table I reports the performance of the design in terms of execution latency (clock cycles) and maximum clock frequency. We can see that the synthesized architecture ($T = 4$, $M = 4$) is able, in general, to provide speed ups with respect to the serial one. With the small dataset, the average speed up is around 2.1, raging from 1.03 (Q1) to 3.13 (Q3), depending on the query. With the large datasets, the average speedup is similar (2.05), with a minimum of 1.08 (Q1 and Q6) and a maximum of 3.13 (again Q3) . For the large datasets there are two queries with minimum speedup. For the queries that show only a modest improvement, the reasons reside in the structure of the query and the high dependency of graph-like methods from the datasets. In particular, the outer loops of Q1 and Q6 execute only a few iterations. Hence, these queries have only a few tasks. Our current design implements a fork-join scheme that spawns a group of T tasks (identified from

TABLE I

	Serial			T=4, M=4			Speed up	Speed up
	Latency (#Cycles)		Max Freq. (MHz)	Latency (#Cycles)		Max Freq. (MHz)		
	LUBM-1	**LUBM-40**		**LUBM-1**	**LUMB-40**		**LUBM-1**	**LUMB-40**
Q1	5,339,286	1,082,526,974	130.34	5,176,116	1001581548	113.37	1.03	1.08
Q2	141,022	7,359,732	143.66	54,281	2801694	130.11	2.60	2.63
Q3	5,824,354	308,586,247	121.27	1,862,683	98163298	114.53	3.13	3.14
Q4	63,825	63,825	143.20	42,851	42,279	122.97	1.49	1.51
Q5	33,322	33,322	133.92	13,442	13,400	138.31	2.48	2.49
Q6	674,951	682,949	136.76	340,634	629,671	113.26	1.98	1.08
Q7	1,700,170	85,341,784	131.98	694,225	35,511,299	106.71	2.45	2.40

TABLE II

	Serial		T=4, M=4		Area Ov.	
	LUTs	**Slices**	**LUTs**	**Slices**	**LUTs**	**Slices**
Q1	5,600	1,802	13,469	4,317	2.40	2.39
Q2	2,690	8,24	5,280	1,607	1.96	1.95
Q3	5,525	1,775	13,449	4,308	2.43	2.43
Q4	3,477	1,073	7,806	2,399	2.24	2.24
Q5	2,785	848	5,750	1,738	2.06	2.05
Q6	4,364	1,369	10,600	3,426	2.43	2.50
Q7	6,194	1,943	15,002	4,953	2.42	2.55

loop iterations) and assigns them to the T hardware kernels. The group of tasks runs to completion before a new group can start execution. Whenever one of the parallel hardware kernels tries to access a memory location concurrently accessed by another kernel, the MIC, by design, denies the request, and the kernel stalls. Consequently, tasks of he same group could have different execution times, but the group terminates only when all the tasks have completed, leading to underutilization of hardware resources and memory bandwidth. The dataset dependency is highlighted by Q6, which takes almost the same time with the sequential architecture with both the dataset sizes. However, while with the smaller datasets there is a speed up of 2, for the larger dataset the different layout of data in memory does not allow to maximize concurrency. Regarding frequency, the parallel implementations always meet the 100 MHz constraints, but they are in average 10% slower than the serial implementation, with a maximum of around 20% for the biggest designs (Q7). With the smallest designs the difference is obviously lower, and in one case (Q5) the parallel accelerator also reaches a slightly higher maximum frequency.

Table II reports the area of the synthesized accelerators in terms of number of Look Up Tables (LUTs) and Slices. The results are post place and route. Occupation of the parallel implementations goes from around 2 times to 2.5 times the occupation of the serial implementations. As kernels are replicated 4 times, the synthesis tool provides some optimization that makes occupation not linear. Because we see an average speed up of 2, the average increase in area is somewhat balanced by the higher performance. Obviously, for queries with speed ups over 2.5, the parallel implementation is highly profitable, while for the others, although there still is an advantage in using the parallel controller, it does not outweigh the increased occupation.

VI. Conclusions

In this paper we have presented a template architecture for the HLS of irregular, memory intensive algorithms, such as graph traversal. These algorithms are inherently task parallel, they normally are developed with shared memory abstractions to simplify their implementation, as they generate fine-grained unpredictable data accesses on difficult to partition datasets, and usually are synchronization intensive. We described two components: an adaptive Distributed Controller (DC), which provides an easier way to exploit task level parallelism than the conventional FSMD model, and a Memory Interface Controller (MIC), which transparently manages concurrent accesses to a multiported/multibanked shared memory by the generated shared kernels, while also providing support for atomic memory operations. Irregular algorithms are common in the new class of HPDA applications, which have to deal with large amounts of unstructured or poorly structured data. We show a case study where our synthesis flow is interfaced to GEMS, an infrastructure that allows implementing RDF databases on commodity clusters by mainly exploiting graph methods, to synthesize SPARQL queries on a common benchmark for Semantic Web repositories (LUBM). We show that our HLS flow can synthesize serial and parallel versions of the queries (converted to graph pattern matching operations in C by GEMS), and that the parallel implementations generally provide good speedups with respect to the serial implementations.

References

[1] V. Castellana, A. Morari, J. Weaver, A. Tumeo, D. Haglin, O. Villa, and J. Feo, "In-memory graph databases for web-scale data," *Computer*, vol. 48, no. 3, pp. 24–35, Mar 2015.

[2] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, June 2014, pp. 13–24.

[3] A. Girault, B. Lee, and E. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 6, pp. 742–760, Jun 1999.

[4] C. Papachristou and Y. Alzazeri, "A Method of Distributed Controller Design for RTL Circuits," *DATE '99: Design, Automation and Test in Europe*, pp. 774 – 775, 1999.

[5] A. Seawright and W. Meyer, "Partitioning and Optimizing Controllers Synthesized from Hierarchical High-Level Descriptions," in *DAC '98: 35th Annual Design Automation Conference*, 1998, pp. 770–775.

[6] A. Del Barrio, S. Memik, M. Molina, J. Mendias, and R. Hermida, "A distributed controller for managing speculative functional units in high level synthesis," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 3, pp. 350 –363, march 2011.

[7] 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*, vol. 15, no. 11, pp. 1191–1204, 2007.

[8] F. Karim, A. Mellan, A. Nguyen, U. Aydonat, and T. Abdelrahman, “A Multilevel Computing Architecture for Embedded Multimedia Applications,” *IEEE Micro*, vol. 24, no. 3, pp. 56–66, 2004.

[9] D. Bacon, R. Rabbah, and S. Shukla, “FPGA Programming for the Masses,” *Queue*, vol. 11, no. 2, pp. 40:40–40:52, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2436696.2443836

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

[11] J. Choi, S. Brown, and J. Anderson, “From software threads to parallel hardware in high-level synthesis for fpgas,” in *Field-Programmable Technology (FPT), 2013 International Conference on*, Dec 2013, pp. 270–277.

[12] T. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. Singh, “From OpenCL to High-Performance Hardware on FPGAs,” in *FPL ’12: 22nd International Conference on Field Programmable Logic and Applications*, 2012, pp. 531–534.

[13] “Altera sdk for opencl,” https://www.altera.com/products/design-software/embedded-software-developers/opencl.

[14] C. Computer, “Convey computer doubles graph500 performance, develops new graph personality. available at http://www.conveycomputer.com/files/2413/5095/9078/sc11_graph500_release.final.pdf.”

[15] ——, “Convey MX Series. Architectural Overview. available at http://www.conveycomputer.com.”

[16] B. Betkaoui, Y. Wang, D. Thomas, and W. Luk, “A reconfigurable computing approach for efficient and scalable parallel graph exploration,” in *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, 2012, pp. 8–15.

[17] R. J. Halstead, J. Villarreal, and W. Najjar, “Exploring irregular memory accesses on fpgas,” in *Proceedings of the first workshop on Irregular applications: architectures and algorithms*, ser. IAAA ’11, 2011, pp. 31–34.

[18] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad, “Database analytics acceleration using fpgas,” in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’12. New York, NY, USA: ACM, 2012, pp. 411–420. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370874

[19] R. Halstead, B. Sukhwani, H. Min, M. Thoennes, P. Dube, S. Asaad, and B. Iyer, “Accelerating join operation for relational databases with fpgas,” in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, April 2013, pp. 17–20.

[20] R. Mueller, J. Teubner, and G. Alonso, “Glacier: A query-to-hardware compiler,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’10. New York, NY, USA: ACM, 2010, pp. 1159–1162. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807307

[21] T. Takenaka, M. Takagi, and H. Inoue, “A scalable complex event processing framework for combination of sql-based continuous queries and c/c++ functions,” in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, Aug 2012, pp. 237–242.

[22] J. Casper and K. Olukotun, “Hardware acceleration of database operations,” in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA ’14. New York, NY, USA: ACM, 2014, pp. 151–160. [Online]. Available: http://doi.acm.org/10.1145/2554688.2554787

[23] C. Dennl, D. Ziener, and J. Teich, “Acceleration of sql restrictions and aggregations through fpga-based dynamic partial reconfiguration,” in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, April 2013, pp. 25–28.

[24] G. A. Malazgirt, N. Sonmez, A. Yurdakul, O. Unsal, and A. Cristal, “Accelerating complete decision support queries through high-level synthesis technology (abstract only),” in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’15. New York, NY, USA: ACM, 2015, pp. 277–277. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689151

[25] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, “Q100: The architecture and design of a database processing unit,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’14. New York, NY, USA: ACM, 2014, pp. 255–268. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541961

[26] C. Pilato, V. Castellana, S. Lovergine, and F. Ferrandi, “A Runtime Adaptive Controller for Supporting Hardware Components with Variable Latency,” in *AHS 2011: NASA/ESA Conference on Adaptive Hardware and Systems*, 2011, pp. 153 – 160.

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

[28] V. G. Castellana and F. Ferrandi, “Scheduling Independent Liveness Analysis for Register Binding in High-Level Synthesis,” in *DATE 2013: Design, Automation and Test in Europe*, 2013, pp. 1571–1574.

[29] ——, “An automated flow for the high level synthesis of coarse grained parallel applications,” in *Field-Programmable Technology (FPT), 2013 International Conference on*, Dec 2013, pp. 294–301.

[30] B. Shao, H. Wang, and Y. Li, “Trinity: A distributed graph engine on a memory cloud,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 505–516.