Svelto: High-Level Synthesis of Multi-Threaded Accelerators for Graph Analytics

Marco Minutoli, *Member, IEEE,* Vito Giovanni Castellana, Nicola Saporetti, Stefano Devecchi, Marco Lattuada, Pietro Fezzardi, Antonino Tumeo, *Senior Member, IEEE,* Fabrizio Ferrandi, *Member, IEEE*

Abstract—Graph analytics are an emerging class of irregular applications. Operating on very large datasets, they present unique behaviors, such as fine-grained, unpredictable memory accesses, and highly unbalanced task level parallelism, that make existing high-performance general-purpose processors or accelerators (e.g., GPUs) suboptimal. To address these issues, research and industry are developing a variety of custom accelerator designs for this application area, including solutions based on reconfigurable devices (Field Programmable Gate Arrays). These new approaches often employ High-Level Synthesis (HLS) to accelerate the development of the accelerators. In this paper, we propose a novel architecture template for the automatic generation of accelerators for graph analytics and irregular applications. The architecture template includes a dynamic task scheduling mechanism, a parallel array of accelerators that enables supporting task-level parallelism with context switching, and a related multi-channel memory interface that decouples communication from computation and provides support for fine-grained atomic memory operations. We discuss the integration of the architectural template in an HLS flow, presenting the necessary modifications to enable automatic generation of the custom architectures starting from OpenMP annotated code. We evaluate our approach first by synthesizing and exploring triangle counting, a common graph algorithm, and then by synthesizing custom designs for a set of graph database benchmark queries, representing series of graph pattern matching routines. We compare the synthesized accelerators with previous state-of-the-art methodologies for the synthesis of parallel architectures, showing that the proposed approach allows reducing resource usage by optimizing the number of accelerators replicas without any performance penalty.

Index Terms—Parallel architectures, Multi-threading, Dynamic Task Scheduling, Context switching, RDF, SPARQL, High Performance Data Analytics, Big Data.

1 INTRODUCTION

Graph analytics enable understanding relationships among objects and structural characteristic of a graph as a whole. The need to quickly analyze networks (social, communication, transportation networks, financial transactions, biomedical data, the Web) of ever-increasing size to extract actionable insights from the data has led to the development of a variety of methods and solutions to accelerate the execution of graph algorithms.

Graph methods are said to be irregular [1]. They typically employ pointer- or indirection-based data structures that induce unpredictable fine-grained data accesses, with poor spatial and temporal locality. While potentially exposing massive amounts of parallelism, when for example a large graph is traversed for each edge or vertex, the parallel activities may be significantly unbalanced and synchronization intensive. Even when graph algorithms require arithmetic for the computation of weights, ranks, and heuristics, they are largely dominated by memory operations. Graph algorithms are best developed with shared memory abstractions, because partitioning efficiently graphs is a complex and time-expensive problem by itself. Conventional high-performance general-purpose architectures are optimized for regular computation, arithmetic (integer or floating-point) intensive workloads, and locality, through complex cache hierarchies that aim at reducing latency. Even if accelerators such as general-purpose graphic processing units (GPGPUs) employ latency tolerance techniques through massive multithreading, they require threads with identical behaviors to maximize utilization. Their parallel memory subsystems provide high-performance with well aligned, coalesced data accesses, and their non-uniform memory access (NUMA) design achieves the highest performance when data are accurately partitioned across the domains.

For all these reasons, research and industry have recently started to look at custom accelerators for graph analytics. These include a variety of approaches, ranging from custom application-specific integrated circuit (ASIC) designs that decouple computation from memory access and specialize the memory subsystem [2] to processing-in-memory designs with stacked memory [3]. The Tera MTA, and its successor, Tera MTA 2, Cray XMT, and Cray XMT 2 [4], were large-scale multithreaded machines optimized for irregular workloads. The EMU system [5] provides a fine-grained multithreaded design where thread migrates to the data. We are experiencing a significant uptake in the use of accelerators based on reconfigurable devices, and in particular Field Programmable Gate Arrays (FPGAs) for data analytics in the data center. These solutions allow specializing the compute architecture and the memory interfaces for the spe-

• *N. Saporetti, S. Devecchi, M. Lattuada, P. Fezzardi and F. Ferrandi are with the Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy. E-mail: {marco.lattuada, pietro.fezzardi, fabrizio.ferrandi}@polimi.it*
• *M. Minutoli, V.G. Castellana and A. Tumeo are with Pacific Northwest National Laboratory, Richland, WA, USA. E-mail: {marco.minutoli, vitogiovanni.castellana, antonino.tumeo}@pnnl.gov*

cific workload characteristics. For example, the ConveyMX [6] provided a custom multithreaded design with a parallel memory interface, and the related programming model, for irregular applications. The Microsoft Brainwave [7] design exploits on-chip memories to persist model parameters. However, even if they use reconfigurable substrates, these still are mostly optimized, hand-designed architectures in hardware-description languages.

At the other end of the spectrum, High-Level Synthesis (HLS) approaches provide a way to automatically generate hardware descriptions from high-level languages. HLS is typically employed to reduce the development time of custom accelerators, often to generate modules of larger systems, for both ASIC and FPGA designs. However, current HLS tools are effective in generating serial or parallel (e.g., through OpenCL annotations) accelerators for regular, easily partitionable, arithmetic-intensive workloads typical of digital signal processing. They mainly target extraction of instruction level parallelism, and consider simple memory subsystems. Additionally, they typically do not consider the need to operate with large datasets that cannot fit on-chip memories or cannot be localized. Thus, they normally only consider known, fixed memory access latencies and perform optimizations that focus on reducing such latencies. In general, they do not consider all of the previously listed characteristics of irregular applications and graph algorithms, namely the massive, but fine-grained, memory parallelism due to datasets that can barely fit in the external accelerator memory, the data-dependent operations, the highly unbalanced parallel activities, the synchronization through atomic memory operation. Thus, research has started looking to approaches that could generate effective designs for graphs and irregular algorithms [8, 9, 10].

In this paper, we propose *Svelto*, an architectural template for the generation of efficient parallel accelerators for irregular algorithms, and the related HLS methodology. Svelto is the culmination of several key contributions that we have integrated in our opensource HLS framework [11] to enable the efficient synthesis of parallel algorithms from a natural C/C++ syntax. These includes a parallel controller to synthesize efficient accelerators from task-parallel specification [12], and a hierarchical memory interface [13]. Task-level parallelism is the primary form of parallelism exploited in shared-memory implementations of graph algorithms, for example by parallelizing the loop iterations on the vertices or edges. The hierarchical memory controller introduced support for fine-grained parallel data accesses from the array of parallel accelerators to multiple memory banks with configurable address scrambling, and atomic memory operations. In [9] we have demonstrated how combining together the two solutions we could automatically generate parallel accelerators for the queries (graph walks) processed by our high-performance Graph Database [14]. In [15] we further extended the approach to support dynamic task scheduling, to overcome the issue of the severe load unbalance that may happens in graph algorithms, for example when some parallel walks terminates early and other have to proceed until they match a specific vertex. The design of [15] however employed only spatial parallelism to provide the memory parallelism needed to maximize memory bandwidth utilization. Svelto builds on these solutions and significantly extends them by adding the dimension of temporal parallelism through context switching, hence providing latency tolerance like interleaved multithreaded processors and generating much more area efficient designs. This paper focuses on presenting the details of the Svelto architectural template and the related synthesis methodology.

While the use of the template in a synthesis flows allows us to apply it to any type of parallel specification, the solutions implemented in *Svelto* specifically address characteristics and patterns of memory bound workloads with irregular memory access patterns that are typical of graph analytics. In summary, this work makes the following contributions:

- We propose an architectural template that exploits both instruction-level and task-level parallelism. The proposed solution includes a hardware scheduler that dynamically binds tasks to the available hardware resources. To maximize memory parallelism and throughput, Svelto introduces single-cycle hardware context switching for the components implementing the parallel region.
- We integrate the proposed template in an HLS flow by defining a methodology for the automatic generation of custom accelerators, starting from an imperative language with annotation expressing parallel regions (e.g., C + OpenMP).
- We empirically evaluate our approach by synthesizing and exploring the design space for a classic graph algorithms example (triangle counting). We then show the use of Svelto with an actual graph analytics use case, the synthesis of a set of queries in the form of graph pattern matching routines, derived from a typical graph database benchmark [16]. The experimental evaluation shows scalability in area and performance over our previous approaches. Specifically, with respect to the solutions presented in [9], which provided spatial parallelism with a simple fork-join approach, we obtain a performance improvement up to 3:72 . With respect to the solution implementing dynamic scheduling [15], the improvement is up 1:16 , but it is reached with much more area efficient designs afforded by the addition of temporal multithreading.

To the best of our knowledge, this is the first integrated methodology that automatically synthesizes Verilog designs of architectures implementing hardware context-switch for task parallelism coupled with a parallel memory subsystem starting from annotated C code, without user intervention.

2 Related Work

The proposed methodology generates efficient accelerators for graph kernels. The methodology enables the programmer to naturally write shared memory graph algorithms, annotated with OpenMP-like pragmas and using atomic memory operations, while generating related architecture templates that maximize external memory utilization through latency tolerance. To achieve this objective, this work touches several areas of designs for reconfigurable architectures and HLS approaches.

Accelerators for graph algorithms. The breadth-first search (BFS) *personalities* of the Convey HC systems were one of the

most prominent commercial examples of designs targeted at accelerating graph traversal. These have been followed up by the Convey MX [6] system, which couples a large number of small general purpose cores implemented on the reconfigurable logic and a multi-port memory controller template, with an OpenMP programming environment (CHOMP - Convey Hybrid OpenMP). Subsequently, various research groups have started investigating parallel FPGA designs for large-scale graph exploration [8, 17], even looking at the implementation of more generic accelerators for vertex-centric [18] or edge-centric [19] graph frameworks. However, many of these either are hand-designed accelerators for specific kernels, or general-purpose design implemented on the FPGA, thus they do not tackle the challenge of automatically generating the hardware description of the accelerators. While recent FPGA-based graph frameworks can generate hardware descriptions, they do so by either employing libraries or pre-defined accelerators [20], or gather-apply-scatter approaches, which may require to rewrite graph algorithms in non-intuitive ways, to partition the data, and not always fit well general cases.

HLS from Parallel Specifications. Because FPGAs need to be programmed spatially, synthesis tools need to extract parallelism as much as possible. A variety of research and commercial HLS tools have thus started considering parallel specifications as input descriptions. These include C or C++ specifications annotated in CUDA, OpenCL, OpenMP, or pthreads. OpenCL efforts are mainly led by FPGA vendors. However, OpenCL has been mainly designed to support data parallel workloads that exhibit similar behaviors across work units, and requires significant efforts in code rewriting for graph algorithms. Additionally, the current synthesis tools are not able to support more than one nested loop and atomic operations. Similar limitations are present in CUDA-to-HDL synthesizers [21]. LegUP [22] supports OpenMP specifications, but the generated memory interface only supports parallel operations through double buffering. The generated hardware supports nested parallelism, but limited to only two levels of the call structure. All these approaches replicate a datapath that implements a loop iteration, but can only execute a task per datapath, thus potentially wasting resources for memory-bound workloads.

Synthesis of graph algorithms. Tan et *al.* [10] describe a pipelined architecture to enable the synthesis of irregular loop nests and validate the approach with typical graph kernels. The approach generates a set of Loop Processing Units (LPUs). A *Distributor* then dynamically assigns each loop iteration to one of the available LPUs. All the LPUs are connected to a *Collector* that passes results to the next stage of the pipelined loop. The architecture implements a *reorder buffer (ROB)* to ensure that results are committed in the same order as in the original loop. However, the approach only uses scratchpad memories, and does not support atomic memory operations, relying on the ROB to guarantee consistency. In [9], we *al.* presented an HLS methodology for the synthesis of graph databases queries. These are expressed as graph pattern matching routines in C, where nested loops are annotated with OpenMP pragmas. The approach leverages an adaptive Distributed Controller (DC) to implement (task) parallel accelerators. The DC employs dataflow-like mechanisms to map concurrent execution flows (tasks) to a parallel array of accelerators, each one implementing a loop iteration. However, the solution adopts a fork/join strategy that spawns tasks in group. If tasks within the same group are unbalanced (i.e., have different duration) resources remain underutilized. In a subsequent work, we extended the design to support dynamic task scheduling, improving utilization. The approach, however, still relies only on spatial replication of the datapaths to provide enough operations to saturate the memory subsystem.

Synthesis of multithreaded accelerators. Halstead and Najjar extend the ROCCC HLS compiler with the CHAT methodology to generate temporally multithreaded accelerators starting from loops constructs [23]. However, they do not address atomic memory operations and focus on the simple case study of pointer chasing. Similarly to CHAT, Huttmann et *al.* [24] extend the Nymble compiler to support multiple execution contexts on the same datapath. The approach duplicates registers in stages that perform operations of unknown latency (e.g., inner loops or external memory access), but only focuses on a single datapath. Sommer et *al.* [25] further extend Nymble to synthesize spatially and temporally multithreaded accelerators from loops annotated with OpenMP worksharing constructs. However, the implementation only statically assigns threads to datapaths, and does not provide dynamic load balancing, which is critical in managing irregular workloads. While justified by the objective of increasing utilization of the datapaths with external memory accesses of unknown latency, neither of these works really consider complex, realistic shared memory subsystems and realistic parallel programming paradigms that use synchronization to coordinate tasks. The shared memory abstraction is, in fact, discussed by assuming that each thread just operates on its *thread-private* memory between synchronization points. Synchronization through atomic memory operations is not discussed at all and, as a consequence, the solution is just evaluated with compute intensive benchmarks with partitionable datasets, and not irregular workloads accessing shared data structures. Irregular applications, and graph algorithms in particular, instead, need frequent synchronization operations to coordinate accesses to the same elements (e.g., vertices, edges) when executed in parallel.

3 Proposed Architecture

We model the parallel specifications to be accelerated through the application template proposed in Listing 1. In this paper, we employ OpenMP pragmas to express parallelism information. However, our accelerator design is not tied to any specification language. The application template includes a parallel loop, whose iterations may concurrently access the memory. We expect that access to shared resources (and memory locations, in particular) is coordinated through atomic operations (lines 6-7). With atomic operations we do not imply only atomic memory operations, but more in general critical sections of parallel code that need to operate in mutual exclusion. Atomic memory operations provide a necessary element to guarantee the consistency model. We do not impose any constraint on the loop body, which can include additional loops and atomic/critical sections: without this limitation, any function containing

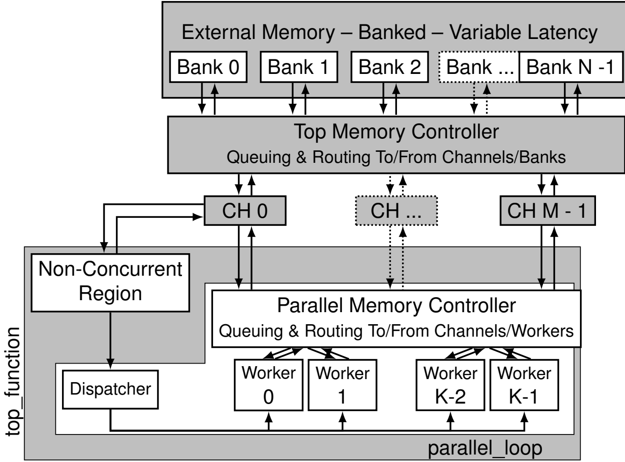


Fig. 1: Overview of the proposed architecture, with N memory banks, M channels, and K workers. Dotted components can be arbitrarily replicated.

```
1 void top_function(...) {
2   {...} // code block A
3   #pragma omp parallel for
4   for (size_t i = 0; i < N; ++i) {
5     {...} // loop body X
6     #pragma omp atomic
7     update_results(...);
8     {...} // loop body Y
9   }
10  {...} // code block B
11 }
```

Listing 1: Example of OpenMP application to be synthesized.

```
1 void atomic_update(...) {
2   update_results(...);
3 }
4 void loop_iteration(size_t i, ...) {
5   {...} // loop body X
6   atomic_update(...);
7   {...} // loop body Y
8 }
9 void parallel_loop(...) {
10  for (size_t i = 0; i < N; ++i)
11    loop_iteration(i, ...);
12 }
13 void top_function(...) {
14   {...} // code block A
15   parallel_loop();
16   {...} // code block B
17 }
```

Listing 2: Example of OpenMP application to be synthesized after High Level Synthesis transformations.

parallel loop(s) can be mapped on the presented application template. Listing 2 provides an equivalent formulation of the application template, which can be automatically generated through code transformations during the synthesis process. This formulation facilitates the mapping of the original template to our proposed architecture via HLS. Similarly to what would normally happen during software compilation, the transformation wraps the parallel loop and the loop body into functions. In addition, we also wrap critical sections composed of multiple instructions.

3.1 Overview of the Architecture

Figure 1 shows a high-level schematic of the architecture template designed for the Svelto HLS flow.

parallel_loop - a hierarchy of Finite State Machine (FSM) controllers with Datapaths. To exploit spatial parallelism, parallel constructs (i.e., *parallel_loop* in Figure 1) include several instances of modules (i.e., *Worker*) implementing the loop body (e.g. `loop_iteration`), each capable of executing several iterations (temporal parallelism) through context switching. This module orchestrates the execution of the iterations through a *Dispatcher* (one per parallel section).

Top Memory Controller - our design takes advantage of banked, multi-ported external memories to increase memory-level parallelism. The external memory is accessed through a hierarchy of memory controller interfaces: the accelerator is connected to a *Top Memory Controller* (directly interfacing with the external memory) through a set of narrow (1 word sized) full-duplex channels. Each channel accepts a new memory request every cycle, while concurrently routing back to the *top_function* responses coming from the *Top Memory Controller*. Read/Writes from the parallel workers proceed through a preassigned channel, and a *Parallel Memory Controller Interface* manages their routing and concurrency. Non-concurrent memory operations (e.g. coming from sequential code in the top module) use one of these channels since their execution is not overlapped with parallel code and thus do not contend any resources. Notice that in our design the number of memory banks, channels and parallel worker instances are decoupled; in the experimental evaluation we explore different design configurations varying such parameters.

The support for context switching, and hence temporal multithreading, is the key difference of the Svelto approach with respect to our previous solutions [15]. Introducing context switching requires a significant redesign of the architecture template and of the whole synthesis methodology, including Dispatcher, generation of the datapaths and FSMs, and memory controller.

3.2 Dispatcher and Workers

The proposed design implements a fork/join execution paradigm. The fork event corresponds to the call of a function embedding a parallel loop (section) from the top, followed by an implicit join when the function returns. As anticipated in the previous section, the parallel loops are implemented through a set of modules executing the loop iterations, and a *Dispatcher*. In the following, we denote a single iteration of the loop as *Task* and a single instance of the loop body modules as *Worker*. Moreover, K identifies the number of *Workers* of the architecture, CS the number of tasks concurrently assigned to the same *Worker*, M the number of channels between *Parallel Memory Controller* and *Top Memory Controller*, and N the number of external memory *banks*. For the sake of simplicity, in the rest of the paper we assume that CS is a power of 2 and that K is a multiple of M. The *Dispatcher* is responsible for distributing the tasks to the available workers at runtime, with a Round-Robin issuing mechanism: at each clock cycle it schedules a single *task* to a different *Worker*. Each *Worker* can queue more than one *task*, and the *Dispatcher* keeps sending *tasks* until all the *Workers* signal that their queues are full. Whenever a *Worker* completes *task*, the *Dispatcher* sends it another *task* following the Round-Robin policy. Theoretically, this may introduce some resource under-utilization (i.e., workers with empty queues waiting for tasks to be scheduled) but, in practice, it

has a negligible impact on the performance due to the actual duration of the tasks.

Figure 2 shows the design of Worker modules, depicting the components required to support context switching. All workers can execute in parallel, and the memory controllers manage contention on the memory resources. The idea motivating context switching is simple, yet effective: whenever a worker executes a high latency memory operation (e.g., an access to external memory), other tasks execute on the datapath, hiding the long latency. There is obviously a tradeoff on the number of tasks required to fully hide latency of long (memory) operations, the number of pending memory operations, and the cost of control logic and register files to store task contexts. In our architecture, a context includes the state of the accelerator controller and of the datapath registers at the time of the switch.

Among the components of the *top_function*, the *Workers* arguably play the most important role. These modules, implementing the function `loop_iteration` of the example in Listing 2, are all identical replicas. Each instance of a *Worker* can run in parallel with the others. Each one implements the multi-threading logic with context-switching.

The architecture of a *Worker* consists of the module *loop_iteration*, generated by the HLS engine, and of the module *Scheduler*. The *Scheduler* has two main responsibilities: to control which *task* a *Worker* executes at any given time, and to act as an intermediary between the *tasks* in execution on the *Worker* and the *Parallel Memory Controller*. Each *Worker* provides CS logical *slots*. Each *slot* can hold data and information about a *task* at any given time and is characterized by a unique identifier *Slot ID*, internal to each *Worker*. *Slot IDs* go from 0 to $CS \quad 1$.

The *Scheduler* receives the *tasks* submitted for the execution from the *Dispatcher*. It uses three First-In-First-Out (FIFO) queues for internal bookkeeping. Each of these queues is statically dimensioned to hold up to CS *task* identifiers. The first holds the list of *free slot IDs*, i.e. the identifiers of the *slots* that are not currently storing any *task*. The second queue holds the list of *waiting slot IDs*, i.e. the identifiers of the *slots* that are currently assigned to suspended *tasks* waiting for the completion of a memory operation. The third queue holds the list of *ready slot IDs*, i.e. the identifiers of the *slots* that are currently assigned to *task*s not waiting for any pending memory operation and ready to run. The *task* assigned to the first element of the list of *ready slot IDs* is the one in execution. If the *ready list* is empty, the *Worker* does not execute any *task* and waits either for a new *task* to come from the *Dispatcher* or for a *task* in the *waiting queue* to become *ready*. The execution of a *task* continues until one of the following conditions happens: it completes the execution; it reaches a state where it must issue a memory request with variable latency. In the first case, the *task* terminates and notifies the *Scheduler*, which in turn moves the associated *slot ID* to the list of *free slot IDs* and starts executing the next *ready task*, if any. In the second case, the *task* forwards the memory request to the *Scheduler*, which in turn forwards it to the *Parallel Memory Controller* attaching some metadata. This forwarding happens in a single clock cycle, during which the *Scheduler* also moves the *slot ID* associated with the requesting *task* from the list of *ready slot IDs* to the list of *waiting slot IDs*. In the next clock cycle, the first available *ready task* continues its execution. A selector signal, driven by the first *slot ID* in the list of *ready slot IDs*, controls the register files. In this way, whenever a *task* is suspended and a new *task* is woken up, the `selector` directly changes the active register in all the register files, effectively saving the context of the previous *task* while enabling the new one. As previously mentioned, the *Scheduler* handles memory requests that tasks issue to the *Parallel Memory Controller*.

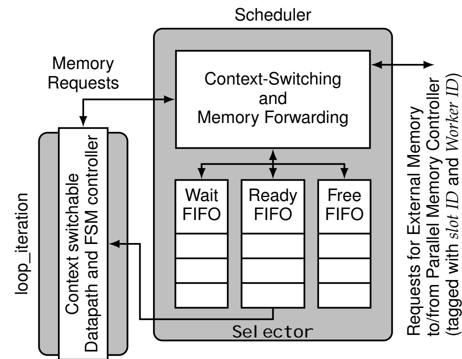


Fig. 2: Architecture of a single multi-threaded *Worker* with the *Scheduler* and the top context switchable component.

As we traverse the memory hierarchy, there will be multiple memory requests in flight at the same time. Thus, all the components involved require a mechanism to properly route the replies back to the *task* that originated the request. To achieve this goal, a unique identifier, called *worker ID* is statically assigned to each *Worker* instance. Each worker tags each memory request before sending it to the *Parallel Memory Controller*, attaching its own *worker ID* and the *slot ID* of the *slot* whose *task* issued the memory request. Moreover, an extra flag bit is added to indicate if the memory request is atomic, leading to an overall size of the tag of $\log_2(M \quad K) + \log_2(CS) + 1$. These tags are moved up and down the memory hierarchy with the requests and the associated replies, as described later in Sections 3.3 and 3.4. Their presence introduces an area overhead, because all the components on the path from *Workers* to *External Memory* need to forward them, but they make routing replies much faster, which is paramount to reduce latencies and congestion of pending requests. It is worth noting that, because of the massive physical and logical parallelism in the execution of the *Workers*, they cannot be attached directly to the *Channels*, but they need the *Parallel Memory Controller* as an intermediate layer.

3.3 Parallel Memory Controller

The *Parallel Memory Controller* (PMC) is a component designed to handle the arbitration of the requests from the *Workers* to the *Top Memory Controller*. The PMC contains two different mechanisms for routing and arbitrating communications: one for requests going from the *Workers* to the *Top Memory Controller* (outgoing) and one for replies coming from *Top Memory Controller* to the *Workers* (incoming).

The PMC manages *outgoing* requests with a Round-Robin policy. The PMC contains M Round-Robin arbiters, one for each *Channel*. Each arbiter is attached to the K/M *Workers*. In turn, arbiters are connected with an associated *Channel*. Each outgoing request from a *Worker* is stalled in

the PMC until the Round-Robin arbiter selects the requesting *Worker*. In the worst case (i.e., multiple simultaneous requests), the latency increases linearly with K/M. However, even for large values of K, this added latency is negligible when the system is fully running, because of the larger memory latency which entirely masks the delay effects that are visible at this level.

For *incoming* replies, communication is even simpler. The *Top Memory Controller* partially handles the arbitration and the routing of the replies, so that only a single incoming reply arrives from a given *Channel* at every clock cycle, directed to one of the connected K/M *Workers*. As a consequence, the PMC does not need any arbitration nor complex routing mechanisms for handling incoming data. Incoming data are broadcasted without filters directly from the *Channel* to all the attached *Workers*. Remember that all the transferred data have a *tag* that includes the identifier of the *Worker* that has originated it. Hence, all the *Schedulers* of all the *Workers* attached to a *Channel* can constantly wait for incoming data, and use the *tag* to automatically recognize two types of information: 1) if the incoming reply is for a task running on the associated *Worker*; 2) the slot ID of the *task* that requested the data. Using this information, the *Scheduler* can select the *task* to move from the list of *waiting slot IDs* to the list of *ready slot IDs*, allowing the *task* target of the reply to be rescheduled whenever a context-switch is necessary.

3.4 Top Memory Controller

From Figure 3 we can see that the *Top Memory Controller* integrates an arbiter associated with each bank and routing logic. There are two types of routing logic: the *direct routing logic*, represented by black arrows, and the *indirect routing logic*, represented by red dashed arrows.

This dual routing mechanism copes with the fact that the number of *Channels* M may be different from the number of *banks* N. Indeed, the request coming from a given *Channel* can target any of the *banks*, but each *Channel* only has direct access to N/M of them. The indirect routing mechanism handles accesses to the remaining banks. The *banks* are organized so that the memory location with word address a resides on *bank* a mod(N), providing an efficient way to compute the *bank* where an address is mapped (N must be a power of two). In this way, whenever a request arrives through a channel, the destination *bank* can be computed by analyzing the $\log_2(N)$ least significant bits of the address.

Moreover, the *direct routing logic* extracts the least significant $\log_2(N/M)$ bits from the target word address and uses them to decide the destination *arbiter* where to forward the request. Consider for example the *Top Memory Controller* shown in Figure 3 and two requests coming from *Channel* CH 0 with target address 1 and target address 6. Since $N=M = 4$, the arbiter is selected on the basis of the bits [1:0]: A1 for the first request, A2 for the second request. When an arbiter processes a request, it checks the lowest $\log_2(N)$ bits of the address: if the target word address is mapped on its associated *bank* it directly manages the request, it forwards the request to the correct arbiter through the indirect routing mechanism otherwise. Note that the arbiter only needs to check the bits in the range from $\log_2(N \quad 1)$ to $\log_2(N/M)$. As shown in the figure, the *indirect routing logic* connects all the arbiters whose number is equal modulo N/M. For example, when A1 processes request with target address 1, it recognizes that the request targets its associated bank. On the contrary, when A2 processes request with target address 6, it forwards the request to A6. Note that that in Figure 3 N/M = 4. This means that A0 directly connects to A4 and vice-versa, and so on for all the other pairs of associated *arbiters*.

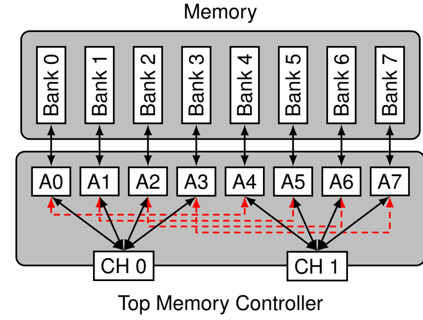


Fig. 3: Architecture of the *Top Memory Controller*. In this example, the number of *Channels* (M) is 2, whereas the number of memory *banks* (N) is 8. Each *bank* has a dedicated *arbiter* that manages the requests to the associated bank. *Red* dashed arrows represent the indirect routing mechanism.

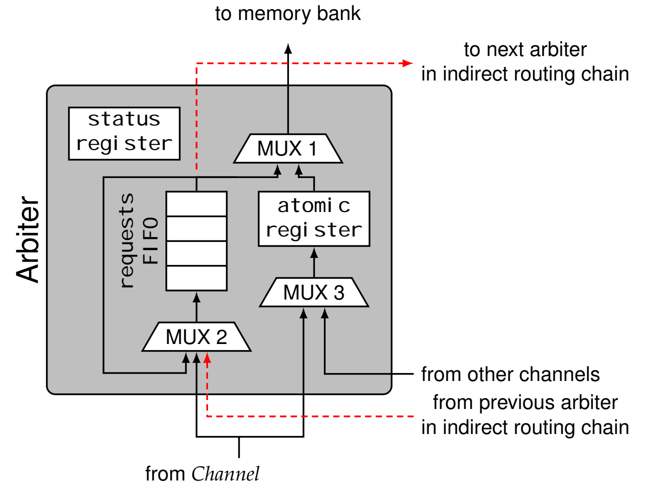


Fig. 4: Detail of an *arbiter* in the *Top Memory Controller*.

When M > 2, there are two possible ways of implementing such connections: circular chain or crossbar. The latter requires a larger area, but has lower latency. The former presents the opposite trade-off. In this work, we adopt the the first solution, because the extra delay is negligible with respect to the long latency of the memory accesses.

Finally, Figure 4 details the *Arbiter*. Beside steering the direct and the indirect connections, it manages *Atomic* memory operations. These operations allow implementing the typical synchronization mechanism in parallel programming. In the presented architecture, we implement the uninterruptible read-modify-write sequence of atomic memory operations as follows. We assume that the atomic memory operations consist of a *Load* operation, a sequence of operations that (possibly, but not necessarily) modify the loaded value, and a related *Store* operation. During an atomic memory update only one *Load* and *Store* pair for a memory *bank* is allowed to proceed at any given time. Other concurrent non-atomic *Store*s are blocked. A *status*

register stores the current status of the *bank* associated with the *Arbiter* as follows:

locked busy, if an atomic memory transaction started, and there is another ongoing memory operation.
locked free, if an atomic memory transaction started, but there are not other ongoing memory operations.
unlocked busy, if there are not ongoing atomic memory transactions, but there is another ongoing memory operation.
unlocked free, if there are not ongoing atomic memory transactions, nor other ongoing memory operations.

Atomic *Store*s must have higher priority than the other requests to unlock the *bank* as early as possible. For this reason, they are not stored in the *requests FIFO* like all the other requests, but in a register. In fact, there can only be at most one atomic memory transaction targeting the associated *bank*. At the same time, at every clock cycle, the top of the *request FIFO* can have two possible destinations: the current *bank* or another *bank*. In the former case, there are three possible scenarios:

status is *locked busy* and the first request is a *Load* not part of an atomic memory transaction: the request is popped and queued at the end of the *FIFO*.
atomic register is empty, *status* is *locked free* and the first request is a *Load* not part of an atomic memory transaction or the *status* is *unlocked free*: the request is popped and forwarded to the memory bank.
all the other cases: the request remains in the *request FIFO*.

In the latter case, the request is popped from the queue and forwarded to next arbiter of the chain through the *indirect routing chain*.

There are three possible sources for the *request FIFO* (i.e., the three inputs of *MUX 2*): the *Channel* associated to the *Arbiter*, the previous *Arbiter* in the *indirect routing chain*, and the head of the FIFO itself. Note that the requests coming from the *channel* are not only the requests targeting the associated *bank*, but they also include the requests targeting the *Arbiter*s connected through the *indirect routing chain*.

The management of *Incoming requests* is much simpler, and in Figure 4 we do not portray their handling logic. There still is a FIFO for each arbiter (in the opposite direction), and a similar *indirect routing chain* infrastructure. However, no special handling of the atomic operations is required, because the replies are just routed from memory to *Channel*s with no consequences for memory locking.

4 Automatic Generation

We have implemented the proposed architectural template and the related synthesis methodology in Bambu [26], a state-of-the-art HLS tool inside Politecnico di Milano's PandA framework [11].

For the application template in Listing 1, the HLS engine needs to discriminate functions that include or are included in parallel code from functions that do not require any customized HLS step. So, after the code factoring and wrapping done during the transformation from Listing 1 to Listing 2, the functions are either tagged as standard or as non-standard HLS functions. In the first set, there are all the functions that could be synthesized using the current HLS approaches and that do not have any coarse grain parallelism or context switch to manage. For example, in Listing 2 the function *top_function* does not present any specific pattern that requires a custom HLS step: it is composed only of simple instructions as well as calls to sub-functions. On the other hand, functions such as *parallel_loop* are built starting from the *OpenMP parallel for*, so they need to be synthesized accordingly to the architecture proposed in Section 3. Moreover, functions such as *parallel_iteration*, implementing the body of the parallel for, have to be synthesized in a way that allows stopping and switching the execution to a different iteration as specified by the *Scheduler* shown in Figure 2. Finally, the body of the *parallel for* may include atomic operations that have to be context-switchable as well (e.g., *atomic_update* in Listing 2). Within this classification these functions really capture the meaning of a critical section (i.e., a sequence of operations that needs to be done in mutual exclusion from other critical sections). However we need to map these to the hardware mechanisms that maintain the memory (and program) consistency within the hardware template, hence the synthesis process needs to deal with the atomic memory operations. In summary, we have four function classes: *standard*, *parallel OMP for*, *context switchable*, and *atomic*. Note that an *atomic* function can also be *context switchable* (if invoked inside a context switchable function), but not necessarily.

4.1 Synthesis of *parallel OMP for* functions

The HLS flow for this type of functions is quite different from the synthesis approach for all the other functions of the architecture. It requires instantiating parametric components taken from a library and building the *loop_iteration* as described in the following paragraphs. The main customizations applied are the number of hardware accelerators and the number and types of the function parameters.

The Datapath of a *parallel OMP for* function contains the replicas of the worker component described in Figure 2. Worker components are defined by wrapping the top context-switchable function (e.g., *loop_iteration*) and the *Scheduler* component, which controls the context switch between tasks and steers the memory operations requests. The component *Dispatcher* works as a controller of the *parallel OMP for* function. Its role is to submit executable tasks to the workers with free slots for task execution. Since each task is identified with the value of the induction variable of the parallel loop, the flow must instantiate a counter storing its value for the next task to be submitted.

4.2 Synthesis of *context switchable* functions

The custom HLS flow for this class of functions is specialized with respect to the FSM and Datapath generation.

First, the Datapath of these context-switch-enabled components must be connected with the control signals coming from the *Scheduler* instantiated in the associated worker, as shown in Figure 2. As noted, the *Scheduler* will also mediate the memory requests to the *Parallel Memory Controller*. Second, all the registers in the Datapath, used to store live variables in the states where the *task* can be suspended,

need to be replaced with register files, controlled by the `selector` of the *Scheduler*.

The register files have a number of *slots* equal to CS (from 0 to CS - 1), but only the one associated with the running *task* is selected for reading and writing. For each parameter of the *context-switchable* functions, a register file is allocated to store the parameter values of each task currently in execution. In case the *context switchable* function is also classified as an *atomic* function, all its memory operations are classified as *atomic* memory operations.

The FSM controller of these context-switchable functions needs customization as well. In particular, the FSM controller state is stored in a register file that the *Scheduler* controls. In this register file, it is possible to save and restore the current state of each *task* during the context-switch. The management of memory accesses with variable latency is obtained by associating a stall state at any state performing a memory operation. In this way, it is easy to define the state in which the memory operation starts and the state in which the memory operation continues the execution after a context switch.

4.3 Breadth First Search example

To briefly illustrate how the synthesis approach complies with typical ways shared memory parallel graph algorithms are written, we discuss the textbook example of the queue based breadth first search ((BFS) [27] using the Compressed Sparse Row (CSR) format (the array of vertices contains the offsets where neighbors lists are located in the edge array). As in the typical high-performance implementations, we declare the queues (Q and Qnext) of the current frontier and the next frontier as arrays, and define (shared) counters to identify slots currently occupied (Q_N, Qnext_N). We never reallocate these queues for each new frontier, but rather just swap the pointers at the end of each iteration. Additionally, we use a map to indicate vertices that have been already visited. Listing 3 shows relevant part of the code after the HLS Transformation of Listing 1. It is possible to see, in particular, the *parallel_loop*, working in parallel on each vertex in the current frontier, and the *loop_iteration* that explores the neighbor list of the vertex, verifies if a neighbor has been already visited through a compare and swap (*cas*), and if it has not, adds it to the next frontier through an integer fetch and add (*ifa*), booking the slot of the queue and writing the id of the neighbor in it. We also show how the *cas* is implemented. Since we are synthesizing C code to Verilog, and not compiling directly for an architecture that has, for example, a *cas* instructions (exposed as an intrinsic), we need to identify the (common) sequence of instructions that implement *cas* as an uninterruptible read-test-write sequence. This is achieved by encapsulating the sequence of instructions in the pragma OMP atomic. We omit the *ifa* code, but its implementation is similar. We also highlight that cas and ifa, as well as other of these *atomic* operations, can obviously be provided as a library to the user of our synthesis flow.

```
1  void bfs(vertex, offset, map, root) {
2    unsigned Q[NV], Qnext[NV];
3    unsigned Q_N = 0, Qnext_N=0, level=0;
4    unsigned *p_Q = ( unsigned*) Q;
5    unsigned *p_Qnext = (unsigned *) Qnext;
6    Q_N = 1; Q[0]=root; Qnext_N=0;
7    while ( Q_N != 0) {
8      parallel_loop(0, Q_N, p_Q, p_Qnext, &Qnext_N);
9      unsigned * p_tmp;
10     p_tmp = p_Q; p_Q = p_Qnext; p_Qnext = p_tmp;
11     Q_N = Qnext_N; Qnext_N = 0;
12     level++;
13   }
14   return level;
15 }
16 void parallel_loop(...) {
17   unsigned i;
18   #pragma omp parallel for
19   for (i = start; i < end; ++i)
20     loop_iteration(i, p_Q, p_Qnext, Qnext_N);
21 }
22 void loop_iteration(..){
23   unsigned i;
24   for (i = offset[vertex]; i < offset[vertex+1]; i++) {
25     if (cas(&map[edges[i]],0,1) == 0) {
26       p_Qnext[ifa(Qnext_N,1)] = edges[i];
27     }
28   }
29 }
30 unsigned cas(* addr,old, new) {
31   unsigned ret = 1;
32   #pragma omp atomic {
33     if(*addr==old) {
34       *addr=new;
35       ret = 0;
36     }
37   }
38   return ret;
39 }
```

Listing 3: Breadth First Search code example

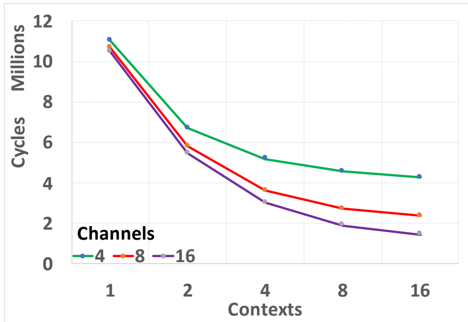
5 Experimental Evaluation

In this section, we validate and evaluate the Svelto approach. We first synthesize a prototypical graph kernel, *triangle counting* (TC), to validate how the proposed methodology adapts to conventional graph algorithms. We then proceed with the evaluation by synthesizing a more peculiar category of graph kernels - pattern matching routines corresponding to queries on a graph database. While for the most common graph algorithms there is a large literature of approaches that look at optimizing the algorithms at all the levels (from their formulation to their specific hardware implementation), queries present a significant variety of computational patterns that hand-designed architecture may not completely capture. The variety of behaviors that the queries presents can justify the use of a synthesis approach to generate the hardware descriptions of the accelerators (of at least some of their key elements), rather than the complex and time-consuming hardware/software codesign process needed to distill behaviors and design the specific components that would be needed in a specialized architecture.

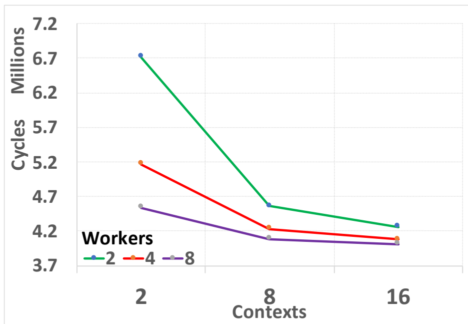
We used a Xilinx Virtex-7 (xc7vx690t) as target device for all the experiments. The only constraint for the HLS tool is the frequency (100 MHz). We measure performance, reported as number of clock cycles needed to execute a query, by simulating the RTL designs with Modelsim 10.5. We perform logic synthesis of the generated Verilog designs with Vivado v2017.2, and report resource utilization after the place and route phase. All the generated accelerators after place and route meet the time requirement.

5.1 Triangle Counting

We synthesized TC from the Graph Algorithm Platform (GAP) [28]. GAP is a benchmark suite that provides parallel shared memory reference implementations (C++ with



(a) Scaling of TC with 2 workers while varying number of contexts and number of memory channels



(b) Scaling of TC with 4 memory channels while varying the number of workers and contexts

Fig. 5: Performance evaluation for triangle counting (TC)

OpenMP) of the six most commonly used graph kernels (BFS, TC, connected components, Page rank, single source shortest path, betweenness centrality). TC is a relatively simple community detection algorithm that identifies all cliques of size 3 in a graph. We actually synthesized the GAP reference implementation starting from C++ (although this work is focused on C, Bambu can also synthesize a large number of C++ constructs). However, we disabled the relabeling heuristic, which provides load balancing when the average degree is high and the degree distribution is heavily skewed. One of the benefits of our multithreaded design is to actually deal with load unbalancing. TC is also included in the Graph Challenge [29], with at least one submission employing FPGA based designs [30]. Such a solution only uses HLS for a small part of the design, employs hand designed components, and relies on a microcontroller for part of the algorithm. Our synthesized design implements the full algorithm in hardware.

We synthesized designs varying the number of workers, contexts per workers, and external memory channels. Figure 5 summarizes our performance scaling evaluation. Figure 5a shows how the execution latency (clock cycles) varies on a graph of scale 13 and edge degree 6 (uniform distribution) generated with GAP itself as we increase the number of contexts per worker in an architecture with 2 workers, and 4, 8 or 16 memory channels, respectively. In all cases, the trend shows that as the number of contexts increases, the performance keeps improving, although it progressively tapers. With 4 memory channels, the speedup with respect to 2 workers with a single context is 2.59 times. With 8 memory channels, 4.50 times. With 16 memory channels, 7.21 times. Thus, in general, the additional active execution contexts seem to always provide a better exploitation of the memory as the number of channels increases. At some point, however, the synchronization costs (we still need to count atomically the found triangles) start impacting on the accelerator. Note that while the graph is uniform in edge degree, the algorithm still remains very unbalanced, as it starts from a sorted neighbor list and, depending on the search, without reordering, can break earlier or later. Figure 5b shows the performance scaling with 4 memory channels, as the number of contexts increases for a fixed number of workers. The main takeaway is that a design with 2 workers but increased number of contexts is competitive with larger designs that hosts more workers. At 16 contexts, the 2.59 speed up of the 2 workers design compares with a speed up of 2.70 and 2.74 for the 4 workers and 8 workers designs respectively. The slight increase in performance is afforded by the additional spatial parallelism of the larger designs, as the context switching, while quick, still costs several cycles, and complicates the management of critical sections. In general, however, context switching generates enough memory parallelism to keep the 4 memory channels busy.

Table 1 shows the resources utilization (post place & route) and speeds ups as the number of contexts increases for architectures with 2 workers and 4 memory channels. The designs with multiple contexts actually show a reduction in the number of registers, up to 16 contexts. However, this is balanced by an increase in the use of look-up tables (LUTs) and, consequently, of FPGA slices. From the analysis of the generated Verilog, we verified that Vivado correctly infers the additional registers for the contexts, but also infers far fewer registers for the interconnection logic (which leads to a significant increase in LUTs and slices).

5.2 Graph Database Queries

We have evaluated the proposed approach through synthesis and simulation of seven queries from the Lehigh University Benchmark (LUBM) [31, 32], a well-known benchmark for Semantic Web repositories. LUBM defines performance metrics and queries over synthetic datasets generated from a realistic ontology from the university domain in the form of Resource Description Framework (RDF) triples. Each RDF triple, consisting of hsubject; predicate; objecti, directly maps to a labeled edge between the subject and the object. Therefore, a set of RDF triples naturally represents a directed graph with labels on its vertices and edges. LUBM queries can thus be expressed as a series of graph pattern matching operations (constrained subgraph isomorphism) on such a graph. To allow a comparison with the current state-of-the-art and our previous approaches, our experiments consider the same set of queries and datasets employed in [9, 15]: *LUBM-40*, consisting of 5,309,056 RDF triples. The queries are generated by converting their SPARQL [33] description into annotated parallel C code using the frontend of the Graph Engine for Multithreaded Systems (GEMS) [14], as in the previous works. The GEMS frontend originally targets a custom runtime for distributed high-performance clusters that provides a shared memory address space, software multithreading, and network data aggregation. The generated C code consists of graph walks traversing edges to match specific graph patterns, lookups

TABLE 1: Resource utilization for triangle counting with 2 workers and 4 memory channels, varying the number of contexts

Cx	LUTs	% Diff.	Slices	% Diff.	Regs	% Diff.	Cycles	Speedup
1	1766	0.00	820	0.00	2157	100.00	11,017,552	1.00
2	3891	120.33	1373	67.44	705	-67.32	6,716,294	1.64
4	4387	148.41	1530	86.59	1027	-52.39	5,179,384	2.13
8	5295	199.83	1727	110.61	1576	-26.94	4,559,944	2.42
16	6984	295.47	2202	168.54	2691	24.76	4,259,145	2.59

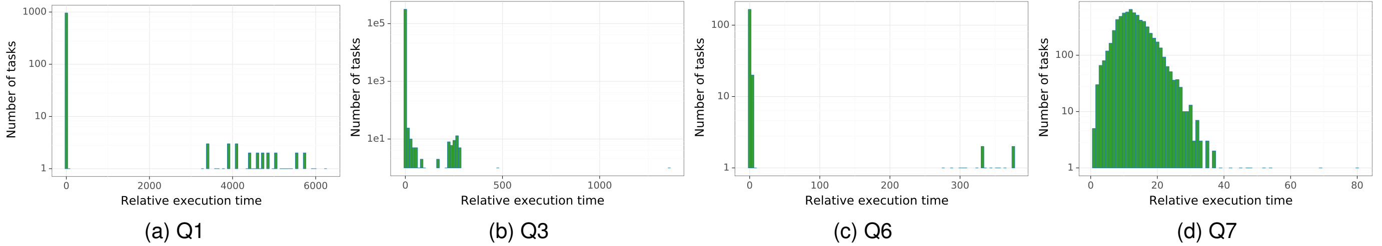


Fig. 6: Histogram of tasks' execution time for the queries on LUBM-40 relative to the fastest task. The execution times are grouped in 100 bins (x-axis).

to match labels of vertices and edges, filter and count operations, and combining (joins) of multiple graph patterns across a variety of policies.

5.2.1 Workload Characteristics

In [15] we have analyzed the runtime characteristics of the seven queries by measuring the execution time of each of their tasks. The analysis shows that the majority of the queries have highly unbalanced tasks: their running time differs of several orders of magnitude. The execution time of each task is data dependent, with no direct (or easily exploitable) relation with the running time of other tasks.

We have further analyzed the workload, looking at the distribution of the execution time of the queries' tasks. Figure 6 shows the distribution of the execution time of the tasks in TQ1, TQ6, and TQ7 with respect to their shortest task. In general, we observe two opposite behaviors. In the first case, queries are composed of a majority of very short tasks, with a few extremely long lasting. Figures 6a to 6c show that the longest task can be up to $6000x$ longer than the shortest task. In the second case, the execution time of the tasks has a relatively smooth distribution. In Figure 6d the longest task takes up to $80x$ more time than the shortest. For the graph pattern matching operations, these differences in execution times depend on the complexity of the searched pattern and the selectivity that each step has on the input data. Thus, a compelling hardware design for these types of workloads needs to provide dynamic load balancing.

In [15] we also measured the utilization of the memory banks using an architecture that only employs spatial parallelism with a dynamic task scheduler and the precursor of the Svelto memory controller. We recorded how many memory banks are in use at each clock cycle during the parallel phases of the queries. Figure 7 reports the data collected as a percentage of the execution time of the parallel sections. The plots show that the configuration with 4 workers and 4 memory banks is using at least 3 memory banks for more than 75% of the execution time for all the queries except Q2, demonstrating the availability of memory parallelism in the queries as well as the effectiveness of the memory controller.

5.2.2 Strong Scaling Analysis

We now discuss strong scaling results for the seven LUBM queries used in [9, 15]. As detailed in Section 3, the *Svelto* architecture has three degrees of freedom: the number of memory channels, the number of workers, and the number of contexts available to each worker. We measure performance scalability using workers with a single context but varying the number of workers (Figure 8) and using two workers with varying number of contexts (Figure 9). In both cases, we change the number of memory channels, and stop the analysis once the performance increase tapers.

Figures 8 and 9 demonstrate the memory-bound nature of the (sub)graph pattern matching performed by the seven LUBM queries. In fact, at saturation (i.e., when performance does not increase anymore with additional workers or contexts), the execution time of the queries reduces by almost a factor of two when doubling the number of memory channels. Before saturation, instead, differences in execution times are less significant (or not observable at all).

In Figure 8 we can see that the *Svelto* architecture provides scaling with all the queries. It achieves a speedup up to 8.67 , with 1 context. Note that, with a single context, the maximum number of outstanding memory operations at each clock cycle coincides with the number of workers.

Figure 9 shows how the execution times with two workers improve while increasing the number of workers from 2 to 32. The plots show that the *Svelto* architecture scales linearly when increasing the number of contexts. Actually, we can see that results between Figure 8 and Figure 9 are comparable when considering the same overall number of *Tasks*. E.g., the execution times of 2 workers with 2 contexts each (4 concurrent tasks in total) in Figure 9 is comparable to the execution time with 4 workers (1 context each) in Figure 8. In practice, this demonstrates that with this type of memory bound applications the proposed architecture increases throughput by providing latency tolerance through context switching, and that the context switching mechanisms, as implemented, do not introduce performance overheads.

Figure 10 shows that the *Svelto* architecture with context switching is, as expected, also area efficient. In fact, when considering the same overall number of *Tasks*, designs with fewer contexts and more workers are larger than designs with more contexts and fewer workers. Fixing the number

Fig. 7: Profiling of memory operations during query execution on LUBM-40.

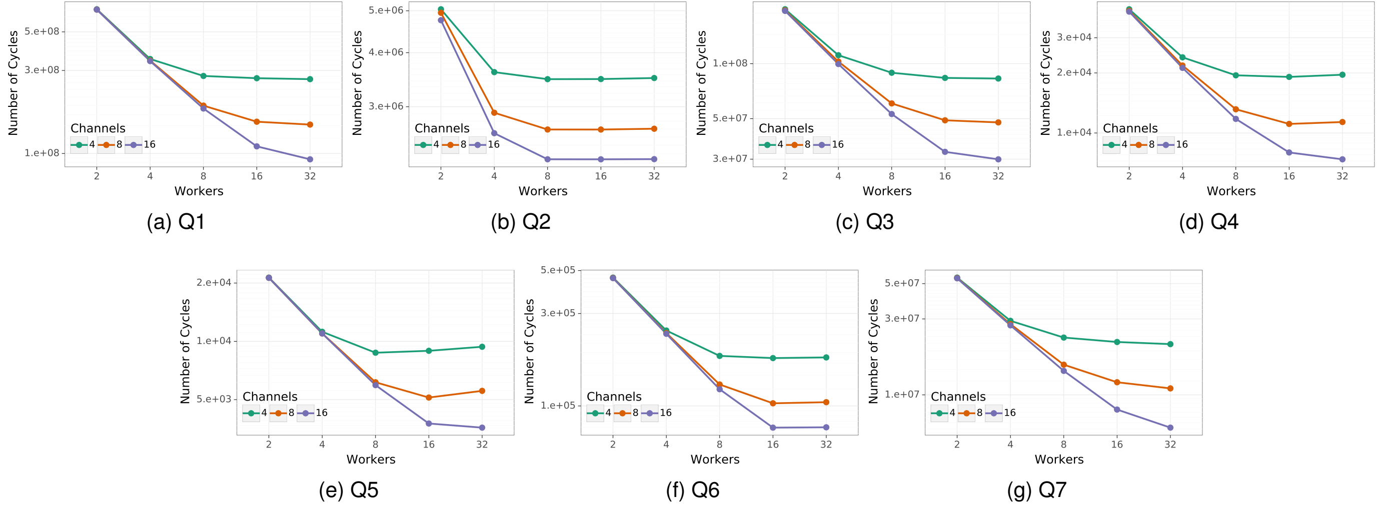


Fig. 8: Strong scaling of the seven LUBM queries with 1 context and varying the number of channels from 4 to 16 and the number of workers from 2 to 32.

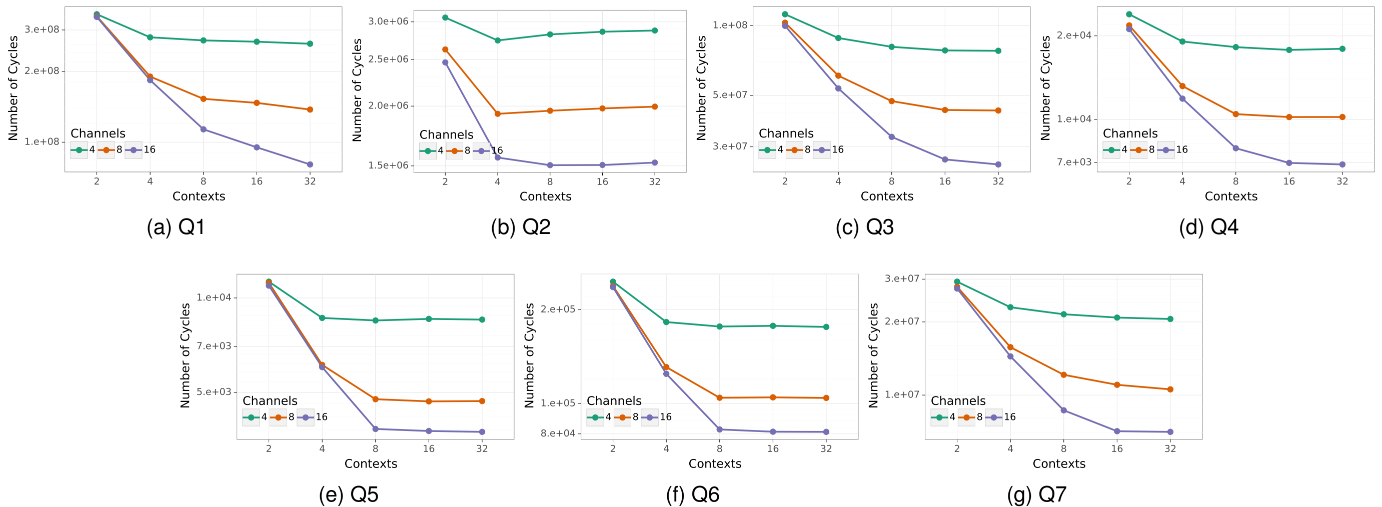


Fig. 9: Strong scaling of the seven LUBM queries with 2 workers and varying the number of channels from 4 to 16 and the number of contexts from 2 to 32.

of contexts, the size in terms of registers and LUT/Slices grows linearly with the number of workers. However, when fixing the number of workers, the size in terms of registers and LUT/Slices apparently grows super-linearly with the number of contexts. We believe that the reason of this trend is a higher complexity in placing and routing. Each additional context leads to the replication of all registers in the datapaths, which in turn require the allocation of a full Slice (LUT and flip-flop pair), thus filling up the device and complicating the interconnection.

5.2.3 Performance Comparison

In this section, we compare the *Svelto* approach with our two previously published approaches: the Parallel Controller [9] (*PC*) and the Dynamic Task Scheduler [15] (DS).

For this comparison, we configured the *Svelto*-generated architectures with 4 memory channels, matching the previ-

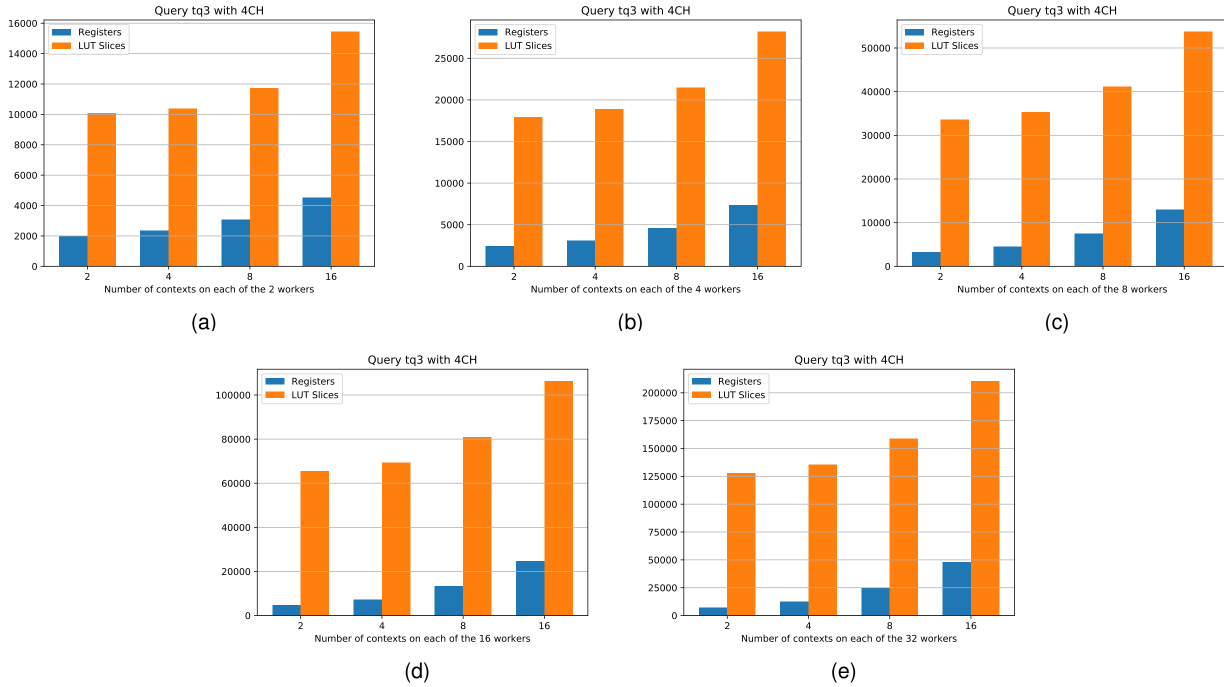


Fig. 10: Resource utilization analysis: Figures 10a to 10e show how Registers and LUTs allocation changes (post place and route) in the synthesis of TQ3 varying the number of contexts (from 2 to 16) and the number of Workers (from 2 to 32). The number of memory channel is fixed to 4.

TABLE 2: Performance comparison

	Parallel Controller # Cycles	Dynamic Scheduler # Cycles	Svelto # Cycles	Speedup PC	Speedup DS
Q1	1,001,581,548	287,527,463	269,158,569	**3.72**	1.07
Q2	2,801,694	2,672,295	2,422,525	1.16	1.10
Q3	98,163,298	95,154,310	81,911,448	1.20	**1.16**
Q4	42,279	19,890	18,128	2.33	1.10
Q5	13,400	8,992	8,555	1.57	1.05
Q6	629,671	199,749	171,689	3.67	**1.16**
Q7	35,511,299	24,430,557	21,509,718	1.65	1.14

ous works. To fairly compare *Svelto* with *DS* we set up a number of workers and contexts almost matching the *DS* performance from [15]. This implies to have 4 workers for *DS* and 1 worker with 8 contexts for *Svelto*. For consistency, we also report the results obtained by *PC* with 4 workers.

In Table 2 we report the performance obtained by the three approaches, while in Table 3 we show the impacts on the resource consumption and on the maximum frequency. Since the memory accesses done by *DS* and *Svelto* architectures are both almost saturating the 4 memory channels, we were expecting only a limited improvement with *Svelto*, even if it provides a higher degree of parallelism. Anyway, we are still seeing a speedup up to 1:16 over *DS*. On the other hand, the performance difference is higher when we compare *Svelto* with *PC*. In fact, we see that *Svelto* is up to 3:72 faster than *PC*.

While providing higher parallelism, the *Svelto* architectures utilize fewer resources than both the previous approaches. In fact, Table 3 shows that, in all the cases except Q2, *Svelto* has a better resource utilization than *PC* and *DS*. Compared to *PC* architectures, *Svelto* provides reductions up to 50% in LUTs and 49:41% in Slices. With respect to *DS* architectures, *Svelto* provides a reduction in resource utilization up to 33:10% in LUTs and 34:11% in Slices. In terms of timing performance, all the three approaches generate designs meeting the 100 Mhz constraint with maximum frequency differences in the range of +/-10%.

6 Conclusion

This paper presents *Svelto*, an HLS methodology for the generation of custom accelerators optimized for irregular graph kernels. The methodology exploits an architectural template that supports single-cycle context switching, and hides external memory access latency, maximizes memory utilization, and provides dynamic load balancing. We define the required analysis and synthesis steps to generate the accelerators starting from a high-level specification in C with OpenMP annotations. Svelto provides speedups up to 3:72 and reduces resource utilization up to 50% over the current state-of-the-art solutions.

References

[1] A. Tumeo and J. Feo, "Irregular applications: From architectures to algorithms [guest editors' introduction]," *IEEE Computer*, vol. 48, no. 8, pp. 14–16, 2015.

[2] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–13.

[3] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 105–117.

TABLE 3: Resource usage and maximum frequency of the proposed approach against the Parallel Controller and the Dynamic Scheduler approaches

	Parallel Controller			Dynamic Scheduler			Svelto			Svelto vs PC			Svelto vs DS		
	Freq. (Mhz)	LUTs (#)	Slices (#)	Freq. (Mhz)	LUTs (#)	Slices (#)	Freq. (Mhz)	LUTs (#)	Slices (#)	Freq. (%)	LUTs (%)	Slices (%)	Freq. (%)	LUTs (%)	Slices (%)
Q1	113.37	13,469	4,317	113.60	10,844	3,503	123.35	7,434	2,314	8.80	44.81	46.40	8.58	31.45	33.95
Q2	130.11	5,280	1,607	132.87	4,636	1,335	121.18	4,612	1,487	-6.86	12.65	7.47	-8.80	0.52	-11.39
Q3	114.53	13,449	4,308	116.92	10,664	3,467	110.56	7,378	2,390	-3.47	45.14	44.52	-5.44	30.81	31.06
Q4	122.97	7,806	2,399	118.68	6,175	1,918	133.14	5,712	1,765	8.27	26.83	26.43	12.18	7.50	7.98
Q5	138.31	5,750	1,738	114.51	5,330	1,578	153.28	4,776	1,524	10.82	16.94	12.31	33.86	10.39	3.42
Q6	113.26	10,600	3,426	118.68	8,125	2,633	117.90	6,112	1,983	4.10	42.34	42.12	-0.66	24.78	24.69
Q7	106.71	15,002	4,953	113.23	11,344	3,747	115.83	7,589	2,469	8.55	**49.41**	**50.15**	2.30	**33.10**	**34.11**

[4] J. Feo, D. Harper, S. Kahan, and P. Konecny, "Eldorado," in *Conference on Computing Frontiers*, ser. CF '05, 2005, pp. 28–34.

[5] P. M. Kogge and S. K. Kuntz, "A case for migrating execution for irregular applications," in *Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3'17, 2017, pp. 6:1–6:8.

[6] C. Computer, "Convey MX Series. Architectural Overview. available at http://www.conveycomputer.com."

[7] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2016, pp. 1–13.

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

[9] V. G. Castellana, M. Minutoli, A. Morari, A. Tumeo, M. Lattuada, and F. Ferrandi, "High Level Synthesis of RDF Queries for Graph Analytics," in *IEEE/ACM International Conference on Computer-Aided Design*, 2015, pp. 323–330.

[10] 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*, 2015, pp. 78–85.

[11] PandA Project Homepage. [Online]. Available: http://panda.dei.polimi.it

[12] V. G. Castellana and F. Ferrandi, "An automated flow for the high level synthesis of coarse grained parallel applications," in *FPT 2013: International Conference on Field-Programmable Technology*, 2013, pp. 294–301.

[13] 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.

[14] 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.

[15] M. Minutoli, V. G. Castellana, A. Tumeo, M. Lattuada, and F. Ferrandi, "Efficient synthesis of graph methods: a dynamically scheduled architecture," in *IEEE/ACM International Conference on Computer-Aided Design*. ACM, 2016, p. 128.

[16] Y. Guo, Z. Pan, and J. Heflin, "Lubm: A Benchmark for OWL Knowledge Base Systems," *Web Semant.*, vol. 3, no. 2-3, pp. 158–182, Oct. 2005.

[17] S. Windh, P. Budhkar, and W. A. Najjar, "Cams as synchronizing caches for multithreaded irregular applications on fpgas," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 331–336.

[18] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, "Graphgen: An fpga framework for vertex-centric graph computation," in *IEEE International Symposium on Field-Programmable Custom Computing Machines*, May 2014, pp. 25–28.

[19] S. Zhou, R. Kannan, H. Zeng, and V. K. Prasanna, "An fpga framework for edge-centric graph processing," in *ACM International Conference on Computing Frontiers*, ser. CF '18, 2018, pp. 69–77.

[20] T. Oguntebi and K. Olukotun, "Graphops: A dataflow library for graph analytics acceleration," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 111–117.

[21] T. Nguyen, Y. Cheny, K. Rupnow, S. Gurumani, and D. Chen, "Soc, noc and hierarchical bus implementations of applications on fpgas using the fcuda flow," in *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, July 2016, pp. 661–666.

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

[23] R. J. Halstead and W. Najjar, "Compiled multithreaded data paths on fpgas for dynamic workloads," in *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '13, 2013, pp. 3:1–3:10.

[24] J. Huthmann, J. Oppermann, and A. Koch, "Automatic high-level synthesis of multi-threaded hardware accelerators," in *International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2014, pp. 1–4.

[25] L. Sommer, J. Oppermann, J. Hofmann, and A. Koch, "Synthesis of interleaved multithreaded accelerators from openmp loops," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2017, pp. 1–7.