# Designing Accelerators for Data Analytics: A Dynamically Scheduled Architecture

Marco Minutoli, Vito Giovanni Castellana, Antonino Tumeo
Pacific Northwest National Laboratory
902 Battelle Blvd, Richland, 99352 WA, USA
{marco.minutoli, vitoGiovanni.castellana, antonino.tumeo}@pnnl.gov

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

***Abstract*—Conventional High Level Synthesis (HLS) tools mainly target compute intensive kernels typical of digital signal processing applications. We are developing techniques and architectural templates to enable HLS of data analytics applications. These applications are memory intensive, present fine-grained, unpredictable data accesses, and irregular, dynamic task parallelism. We introduce a dynamic task scheduling approach to efficiently execute heavily unbalanced workloads, at the opposite of conventional HLS flows that employ execution paradigms based on static scheduling. Our approach is validated by analyzing and synthesizing queries from the Lehigh University Benchmark (LUBM), a well know SPARQL benchmark.**

Data Analytics applications, such as graph databases, often employ pointer or linked list-based data structures that, although convenient to represents dynamically changing relationships among the data elements, induce an irregular behavior. These data structures, in fact, allow spawning many concurrent activities, but present many unpredictable, fine-grained, data accesses, and require many synchronization operations. Partitioning the datasets without generating load imbalance is also very difficult. General-purpose processors typically exploit locality and try to reduce access latencies through caches. Thus, they perform poorly with these workloads, making application-specific accelerators (implemented, for example, on Field Programmable Gate Arrays - FPGAs) an appealing solution. However, traditional High Level Synthesis (HLS) flow generally target compute intensive workloads (i.e., digital signal processing) that mainly expose instruction level parallelism, and can easily be distributed across replicated functional units. They also usually assume a simple memory system, where each unit has private data with exclusive access.

Resource Description Framework (RDF) databases have become one of the most prominent example of data analytics application. RDF is the metadata data model typically used to describe the Semantic Web. RDF databases naturally maps to graphs, and query languages for these databases such as SPARQL basically express queries as a combination of graph methods (graph walks and graph pattern matching operations) and analytic functions. Among these, we consider GEMS, the Graph database Engine for Multithreaded Systems (GEMS) [2]. GEMS implements a RDF database on a commodity cluster that mainly employs graph methods at all levels of his stack. To address the limitations of HPC systems, GEMS is based around a runtime (Global Memory and threading - GMT) 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 (collectively named SGLib) 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 in C/C++. We have investigated acceleration of queries, as generated for the GEMS software stack, on FPGAs. We have developed a set of architectural templates and HLS methodologies to automatically generate specifications in hardware description language (Verilog) of graph methods starting from C descriptions and have integrated them in a modified version of an openly available High Level Synthesis tool, Bambu [1]. We have then interfaced GEMS with the modified Bambu.

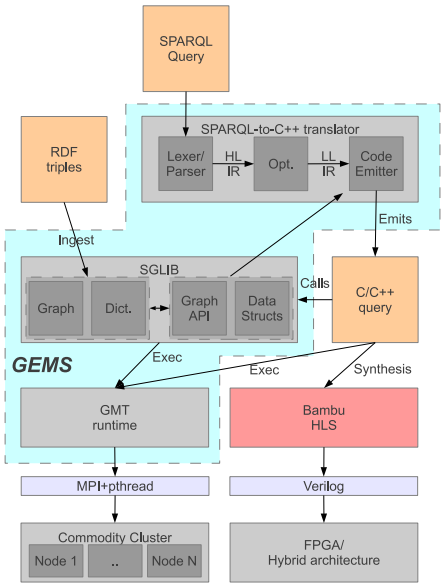


Fig. 1. Structure of the GEMS stack and interaction with Bambu HLS

Figure 1 shows such integration. We modified all GEMS layers so to not be anymore dependent on the GMT runtime. We developed a pure C version of the graph API that does not exploit any of the functionalities of GMT. We modified

the code emitter accordingly, so that the C code generated from the SPARQL queries only invokes the new C graph API to perform graph walks and matching and pure C functions to execute any analytic operation. Such a C code in practice corresponds to a set of nested loops, where each loop matches a particular edge of the graph pattern that composes the query, and becomes the input of Bambu for the synthesis. Note that synthesizing full queries is not a limitation: in the many analytics applications, once the query is defined, it remains stable in time, while the dataset dynamically changes. So, if we can significantly accelerate query execution, we can afford the time to synthesize the query on FPGA. We modified Bambu by integrating three architectural templates, and the corresponding methodologies to generate their instances. The three components aim at providing better support for certain of the typical structures and behaviors that characterize parallel graph methods, ad employed in GEMS for the query processing. The components include a Parallel distributed Controller (PC) [3], a Hierarchical, multi-ported, Memory Interface (HMI) [4], and a novel Dynamic Task Scheduler (DTS).

The PC allows generating more efficient designs that exploit coarse grained (task level) parallelism than the typical centralized controllers of conventional HLS flows based around the Finite State Machine with Datapath (FSMD) model, in terms of performance and area utilization. This is a key element in accelerating graph algorithms that basically are composed of nested loops iterating on vertices or edges, where each iteration could identify a different task. By adopting the PC, it is easy to coordinate parallel execution of tasks (one, or more iterations each) on an array of replicated accelerators. The HMI provides a way to dynamically disambiguate fine grained memory accesses to locations of a large, multi-banked memory, while maintaining an abstract view of a shared memory towards the set of accelerators and, in general, the HLS flow. In cooperation with the PC, the HMI also enables supporting atomic memory operations. Graph algorithms typically access unpredictable memory locations with fine-grained transactions (i.e., they follow pointers), and their implementation is much easier when considering a shared memory abstraction. Also, they often are synchronization intensive when parallelized, because tasks may access the same elements concurrently.

Our latest contribution, the DTS, provides a way to execute new tasks as soon as one of the parallel accelerators is free: instead of simply using a fork/join model, where all currently executing tasks on the set of accelerators must terminate before a new group could be executed, the DTS allow scheduling new tasks as soon as one of the accelerators is free. This adapt to a variety of graph algorithms where certain tasks (iterations) may execute for a long time, while other could terminate early, such as when a graph walk is pruned early because it reached an uninteresting part of the graph. Figure 2 shows an architecture template integrating the DTS. The DTS interfaces to the set of parallel accelerators (Kernel Pool) and to the Termination Logic. The DTS itself contains a Task Queue, the Task Dispatcher, and a Status Register that holds information on the accelerators in the kernel pool. As soon as a task is terminates, the status register is updated and the DTS can schedule a new task. The Termination Logic allows understanding when all the tasks have completed, i.e., when for example all iterations of a parallel loop have completed.

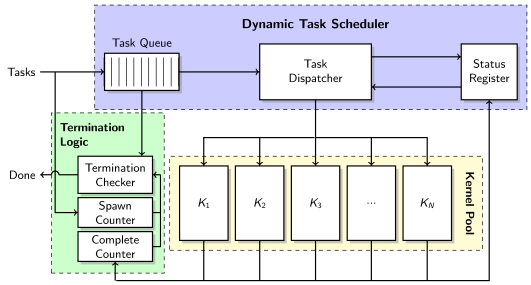


Fig. 2. High level overview of an architecture template using the DTS

TABLE I
PERFORMANCE COMPARISON

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

To validate our architectural templates and our approach, we have synthesized 7 queries from LUBM [5], and we have tested the performance using a dataset of 5,309,056 RDF "triples". In Table I, we compare, in terms of execution latency, a serial implementation of the architecture (Single Acc.), one that employs PC and the HMI [3] (Parallel Controller), and one that also includes the DTS (Dynamic Scheduler). The parallel architectures include 4 accelerators and HMIs with 4 ports. With respect to the serial implementation, the architectures employing the DTS generally show a speed up close to the theoretical maximum. In many cases, the DTS also provides significant speed ups against the PC designs. This happens, in particular, with queries that have some iterations (tasks) that execute order of magnitudes longer than others. Another important effect of the DTS is that it maximizes utilization of the available memory channels as provided by the HMI. In fact, all the architectures with the DTS utilize at least 3 out of 4 of the memory ports for more than 75% of the time.

## REFERENCES


[1] Bambu: A Free Framework for the High-Level Synthesis of Complex Applications. http://panda.dei.polimi.it, 2014.

[2] V. Castellana, A. Morari, J. Weaver, A. Tumeo, D. Haglin, O. Villa, and J. Feo. In-memory graph databases for web-scale data. *Computer*, 48(3):24–35, Mar 2015.

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

[4] 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*, pages 1–4, 2014.

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