POLITECNICO
MILANO 1863

A Dynamically Scheduled Architecture for the Synthesis of Graph Database Queries

M. Minutoli, V. G. Castellana, A. Tumeo, F. Ferrandi, M. Lattuada

M. Minutoli, V. G. Castellana, A. Tumeo, F. Ferrandi, and M. Lattuada. A dynamically scheduled architecture for the synthesis of graph database queries. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 136–136. IEEE, May 2016

The final pubblication is available via http://dx.doi.org/10.1109/FCCM.2016.41

A Dynamically Scheduled Architecture for the Synthesis of Graph Methods

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

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

Data analytics applications, such as graph databases, exibit irregular behaviors that make their acceleration non-trivial. These applications expose a significant amount of Task Level Parallelism (TLP), but they present fine grained memory accesses. Thus, accelerators for this class of applications should exploit coarse-grained parallelism and support parallel memory architectures. These two features alone do not guarantee optimal performance. The parallelism in these applications is highly dynamic, and the concurrent tasks often are unbalanced: they can have different duration, touch different amount of data, or require mutual exclusive access on shared data. Statically estimating the latency of tasks is often impossible due to their strictly data dependent behavior. Typical High Level Synthesis (HLS) flows employ execution paradigms based on static scheduling that do not provide effective load balancing in these applications.

Previous work investigated many techniques to exploit TLP in hardware accelerators. [1] presents a HLS flow translating a SPARQL query into a hardware accelerator using a Parallel Controller (PC) and Memory Controller Interface (MCI) to support the parallel execution of tasks. However, the PC that they propose employ a fork and join model that shows sub-optimal performance when tasks in the same group are highly unbalanced. Our solution overcome this limitation through dynamic task scheduling on the available resources. [2] describes a pipelined architecture targeted at the HLS of irregular loop nests. They synthesize pipelined loops as a set of Loop Processing Units (LPUs). Iterations of the loop are dynamically assigned to one of the available LPUs through a Distributor. LUPs are connected to a *Collector* that passes results to the next stage of pipeline. To preserve the order of memory operation, the architecture employs a *reorder buffer (ROB)*. Our solution also employs dynamic task scheduling with the difference that applies to more general cases than loop pipelining.

In our study, we structure graph methods and graph pattern matching algorithms as loop nests. We parallelize them by transforming each iteration of the outer loop in a task. Synchronization between concurrent tasks is achieved through the atomic memory operations implemented in the Hierarchycal Memory Interface (HMI). The proposed architecture includes three basic components: a *Kernel Pool*, the *Dynamic Task Scheduler (DTS)* and, the *Termination Logic*. The Kernel Pool exploits spatial parallelism including replicas of custom Processing Units (PUs) that execute a single task. The DTS manages the parallel execution with the objective of maximizing resource utilization. It includes three components: a *Task Queue* that stores tasks ready for execution; a *Status Register* that holds run-time information about PUs utilization (available/computing); a *Task Dispatcher* that dynamically assigns ready tasks to available PUs. When the Task Queue has some elements and a PU is available, the Task Dispatcher pops a task from the queue and starts its execution, updating the Status Register accordingly. Similarly, when a PU completes a task, it updates the Status Register to signal its availability to the Task Dispatcher. The *Termination Logic* counts all the tasks that have been pushed in the queue by the parallel loop and the completed tasks by the PUs. When the number of spawned task is equal to the number of completed tasks, the *Termination Logic* asserts a done signal, denoting the conclusion of the parallel phase.

We evaluated our approach by generating the accelerators for a set of queries from the Lehigh University Benchmark (LUBM). We profiled queries execution, showing that the execution time among tasks can differ even of order of magnitudes. We synthesized accelerators for the queries and compared their performance against a serial implementation. Experimental results show that our solution provides a speedup up to *3.76* over the serial implementation. We explored the design space to achieve maximum memory channels utilization. The best design used at least 3 of the 4 memory channels for more than *80%* of the execution time.

In conclusion, our approach overcomes, with the introduction of the DTS, some of the limitations of the current state of the art solutions when the task workload is highly unbalanced.

References

[1] 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*, 2015, pp. 323–330.

[2] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang, "Elasticflow: A complexity-effective approach for pipelining irregular loop nests", in *ICCAD'15*, 2015, pp. 78–85.