A Synthesis Approach for Mapping Irregular Applications on Reconfigurable Architectures

Vito Giovanni Castellana*†, Antonino Tumeo*, Fabrizio Ferrandi†
*Pacific Northwest National Laboratory - Richland, WA, USA
†Politecnico di Milano, DEIB - Milano, Italy

1. INTRODUCTION AND MOTIVATION

Data mining, bioinformatics, pattern recognition and, in general, knowledge discovery are new, emerging irregular applications. They features irregular data structures such as graph, unbalanced trees or unstructured grids, which employ pointers or linked lists. These data structures provide large amounts of inherent dynamic parallelism, because the application can potentially spawn new tasks for each explored element. However, they also present very poor spatial and temporal locality, leading to substantially unpredictable, fine-grained, memory accesses. Irregular applications are mostly memory bandwidth bounded, and the key in maximizing their performance is maximizing bandwidth utilization in presence of fine-grained memory accesses. Several systems targeting irregular applications that employ hybrid architectures have appeared. Hybrid architectures integrate both general purpose processors and reconfigurable logic such as FPGAs to accelerate some specific workloads. Solutions such as the Convey HC-1 or the HC-2 include custom personalities (hand-designed accelerators) for some irregular algorithms, such as Breadth First Search (BFS). These platforms integrate complex, custom memory controllers with multiple distributed or banked memories, which supports many concurrent fine-grained parallel memory requests, high bandwidths and large memory sizes. However, they still are custom designs, with hand-developed accelerators or even processors, loaded on the reconfigurable logic.

On the other end of the spectrum, High Level Synthesis (HLS)tools allow automatic generation of hardware accelerators starting from high level languages such as C. They appear very promising for hybrid architectures: the application developers can decide to offload some of the kernels (usually, the ones that mostly constrain the application performance) to the reconfigurable logic, and let the tool generate all the register transfer level code to synthesize. In this paper we introduce a HLS approach for irregular applications, starting from a parallel C specification. Our approach generates dynamically scheduled components. It enables extraction of parallelism from independent coarse-grained code segments, generating different "hardware tasks", with limited complexity. This also makes easy replicating kernels (i.e., generating multiple hardware tasks from independent loop iterations) to increase the number of parallel operations. A key element of our approach is the definition of a memory interface controller which dynamically maps memory operations across multiple, distributed and/or multi-ported memories, such as those available in hybrid systems. This improves memory bandwidth utilization. The interface also enables atomic memory operations, such as fetch-and-add and compare-and-swap. We evaluate the proposed approach by performing design space exploration in terms of parallelism (number of concurrent kernels) and memories for a typical irregular kernel, the BFS algorithm.

2. ARCHITECTURE OVERVIEW

Our approach is based on the definition of a dynamic scheduling accelerator design, able to manage multiple execution flows concurrently. The generated accelerators feature an adaptive controller [2] which supports dynamic scheduling. The controller is composed of a set of interacting modules, which communicate with a token-based paradigm, without adding any communication overhead. Each operation is subject to a set of activating conditions, which are identified by analyzing the program dependencies. A dedicated module verifies when, at runtime, the conditions are satisfied, and allows starting the execution of the related operation as soon as possible. If multiple operations compete for a hardware resource, an arbiter establishes which one executes first, according to a priority ordering. This avoids structural conflicts. Because all operations are independently managed, this approach enables concurrent execution of variable latency operations. Support for concurrent variable latency operations, together with the ability to resolve structural conflicts at runtime, makes the design suitable for memory bound applications: memories in fact are usually modeled as variable latency functional units. Furthermore, these features also allow to effectively exploit task level parallelism since function calls may be modeled as common variable latency operations. The synthesis process binds each function to a dedicated module, synthesized as a stand alone accelerator and then integrated in the caller design. If a kernel function performs memory operations on shared memories, it forwards execution requests to the caller. In case of nested calls, forwarding is recursive until the top level is reached. At the top level, the Memory Interface Controller manages the memory accesses.

3. MEMORY INTERFACE CONTROLLER

The Memory Interface Controller (MIC) has been designed with the objectives of dynamically addressing irregular memory accesses to the corresponding memory port, while managing their concurrency. Basically, the MIC takes in input memory access requests from N ports, which have an address, a data and an operation type (load/store) line. It routes requests towards one of the M output ports by evaluating their addresses. It serves a request as soon as the

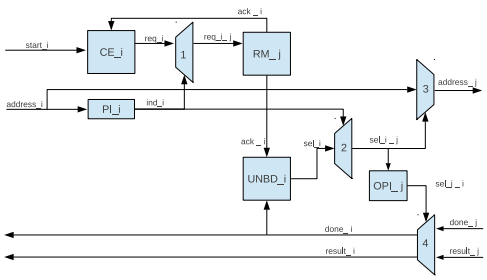

Figure 1: ***Memory Interface Controller schematic representation.***

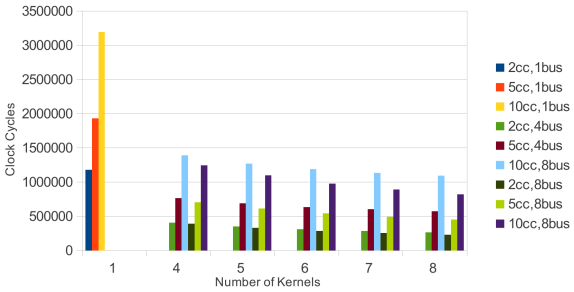

Figure 2: Simulation results; input graph: 5000 nodes, 72887 edges, average out degree: 30

corresponding port is available. In a similar way, it routes back M done signals (which signal 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 access one bank. Each memory bank has non-overlapping addresses. This is equivalent to having M different distributed memories. Figure 1 provides a schematic view of the controller structure. The MIC associates each input operation i to a Control Element (CE) and a module (PI), which analyzes the input address to establish the destination port. For each output j, it is allocated a Resource Manager (RM), which has the role of managing concurrency. Each CE intercepts execution requests and forwards them to the proper RMs, until they are accepted. A Port Index signal produced by the PI, working as selector of the steering logic (connection 1), allows performing the the routing of the requests. Once a RM accepts a request, it sends back an ack signal to the corresponding CE, disabling it, and to the UNBD module, which is responsible of setting the selections while the operation is running. These signals, according to the output of PIs (connection 2), drive the steering logic that feeds the memory ports (connection 3). Figure 1 only show the connections and logic for the address line of an input port. All the other input lines follow a similar approach, duplicating the steering logic and interconnections. Similarly, the MIC routes done signals and results coming from memory (read accesses) according to the requesting input port. In this case, OPeration Index (OPI) modules provide the selectors for the interconnections. OPIs identify the input port requesting the memory access. The MIC is also responsible for managing atomic operations. When it accepts the execution request of an atomic operation, it exclusively binds the associated memory port to the operation, until its completion. The MIC features atomic execution through dedicated hardware. For example, fetch-and-add operations read the value at the specified address, add provided operand to the previously read value, and then store the result in the same memory location. They return the old value they read. The MIC implements this operation as follows. First, it performs the load operation. When the MIC receives the done signal coming from the memory, it intercepts the signal buffers the loaded value, and calculates the sum. Then, it stores the result of the sum into the memory. The subsequent done signal corresponds to the completion of the whole atomic operation, thus it returns the buffered value. The MIC implements other atomic operations following the same approach.

4. EXPERIMENTS

We prototyped the proposed flow extending the *Bambu* HLS framework [1], and evaluated the approach by exploring execution latencies (number of clock cycles) when synthesizing the BFS algorithms.We generated and compared different implementations, varying the number of allocated kernels and the number of available memory ports. Each kernel performs six memory accesses, and two atomic operations, i.e., one fetch-and-add and one compare-and-swap. The kernels require access to the whole memory: the MIC manages all the memory accesses, at the top level. We evaluated the performance also varying the latency model of memory operations (2, 5 and 10 clock cycles per operation). As shown in figure 2, we verified speed-ups when increasing the number of kernels: this highlights the ability of the accelerator to effectively extract thread level parallelism. To demonstrate the effectiveness of the proposed approach in parallelizing irregular memory accesses, we must also consider the relative speed-ups when varying the memory latency. Regardless of the number of kernels, we reported a significant speedup when increasing the number of memory banks. The gains are higher with high latency memories, because in such a case the execution latency is dominated by the memory accesses. This is a valuable result, especially when considering that the speed up increases with the number of kernels, which provides more concurrency on the memory resources.

5. CONCLUSIONS

In this work we propose a HLS flow generating adaptive accelerators which exploit coarse grained parallelism. We designed a memory interface controller able to route multiple unpredictable memory requests on multiple shared memories, while managing concurrency. This makes the approach particularly suitable for the HLS of irregular applications. The experiments validate the approach, showing speed-ups that increase with both the number of memories and concurrent kernels.

6. REFERENCES

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

[2] Pilato, C., Castellana, V.G., Lovergine, S., and Ferrandi, F. "A Runtime Adaptive Controller for Supporting Hardware Components with Variable Latency". *NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2011)*, June 2011.