Memory-Aware Loop Mapping on Coarse-Grained Reconfigurable Architectures

Abstract:

The coarse-grained reconfigurable architectures (CGRAs) are a promising class of architectures with the advantages of high performance and high power efficiency. The compute-intensive parts of an application (e.g., loops) are often mapped onto the CGRA for acceleration. Due to the extra overhead of memory access and the limited communication bandwidth between the processing element (PE) array and local memory, previous works trying to solve the routing problem are mainly confined in the internal resources of PE arrays (e.g., PEs and registers). Inevitably, routing with PEs or registers will consume a lot of computational resources and cause the increase of the initiation interval. To solve this problem, this paper makes two contributions: 1) establishing a precise formulation for the CGRA mapping problem while using shared local data memory as a routing resource and 2) extracting an effective approach for mapping loops to CGRAs. The experimental results on loops of the SPEC2006, Livermore, and MiBench show that our approach (called MEMMap) can improve the performance of the kernels on CGRA up to 1.62×, 1.58×, 1.28×, and 1.23× compared with the edge-centric modulo scheduling, EPIMap, REGIMap, and force-directed map, respectively, with an acceptable increase in compilation time. The proposed architecture of this paper analysis the logic size, area and power consumption using Xilinx 14.2.

Enhancement of the project:

Existing System:

Coarse-grained reconfigurable architectures (CGRAs) are becoming a very attractive platform because of its high performance for computation, high flexibility, and low power dissipation. A CGRA typically consists of an array of processing elements (PEs), a shared local data memory, and a context memory. CGRAs have four major architectural parameters:
1) PE array size;
2) interconnection topology of PEs;
3) local data memory architecture;
4) Configuration method of PEs.

Since it is an indisputable fact that a program spends 90% of its execution time on only 10% of
the code (loops), the most important task for CGRA compiler is efficiently and automatically
mapping loops onto CGRA. Software pipelining is usually adopted to exploit more loop-level
parallelism. Software pipelining improves the parallelism of loops by overlapping loop
iterations. Modulo scheduling is widely used for software pipelining. The goal of modulo
scheduling is to find a valid mapping with a minimum initiation interval (MII), where II
represents the delay between the initiations of iterations of the loop. We know that II is inversely
proportional to the performance and is limited by both the resource constraints and the loop-
carried dependence. In CGRA, the existing mapping techniques usually use PEs and registers to
route data dependence. However, if the PEs and registers are overused to route data dependence,
it will inevitably cause the shortage of PEs for computation, and finally, result in a large II. In
particular, when the loop contains some loop carried dependence, the number of PEs or registers
used to route dependence will increase dramatically. Therefore, in order to improve the
performance of application mapping onto CGRA, the key is to reduce the occupation of PEs and
registers for routing dependence, and eventually, the II can be reduced. As the on-
chip storage
resources, local data memory is an alternative to PE or register for routing data dependence.

Although local data memory is a kind of storage resource just like registers, the hardware
structure and access manner of local data memory are a quite different from those of registers.
First, registers are distributed in PE array, and each register can only be accessed by its
neighboring PEs, while the local data memory is centralized and can be accessed by all PEs.
Second, data transfer by registers between two PEs is in a hop-by-hop manner, which occupies
multiple registers, while data transfer by local memory only involves a pair of load and store
operations. Last but most importantly, the latency of memory access is usually longer than that
of the logic operation, while the latency of register access is usually the same as that of the logic
operation. Due to these three major differences, we cannot simply treat local data memory as an
extension of registers and use an existing mapping approach, e.g., REGIMap [5], to utilize local data memory for routing data dependence.

Disadvantages:

- Power efficiency is low
- Low performance

Proposed System:

Target Architecture

In the past two decades, more than a dozen of CGRAs have been proposed, covering a wide range of application fields, e.g., ADRES, MorphoSys, Silicon Hive, and Raw. The proposed CGRAs differ in four major architectural parameters: 1) PE array size; 2) interconnection topology of PEs; 3) local data memory architecture; and 4) configuration method of PEs. To make our approach have wide applicability, we summarize these four parameters and parameterize the problem formulation with these four parameters.

Fig. 1. CGRA-based computing platform, including a $4 \times 4$ 2-D mesh PE array-based CGRA, a host controller, and a main memory.
If we want to use local memory as the routing resource, we need to solve two main problems. The first one is to select, which dependence should be replaced by memory operations and which CSs in kernel should be allocated to inserted memory operations. The second one is to find a search strategy with low complexity to look for a valid mapping. After some analysis, we find that the first problem is equivalent to finding an efficient method to replace the dependence with load and store operation pairs and schedule them at the suitable CSs in the kernel. Therefore, we switch our perspective to the rows of the kernel (all the operations at the same CS in the kernel construct a row of the kernel).

For example, in Fig. 2(c) and (d), the Nm is equal to 1, because only a single row (third row and fourth row, respectively) contains the memory operations. After applying this formulation to the kernel in Fig. 2(e), we have achieved the IIs of it, as shown in Fig. 2(f).

For the case σ = 1 (including the case that the CGRA supports the partial reconfiguration), the memory operations should be placed in a distributed way. For example, the load operation aL in Fig. 2(c) should be placed at the same row as the operation d, because it will reduce the number
of PEs used for routing the loaded data to operation e and a. In this case, the load operations should be placed at the locations that are close to their target operations, while the store operations should be close to their source operations.

Dependence Replacing

When a pair (IICS, Nm) is given, we need to select the dependence that should be routed by memory and rows, where the memory operations should be inserted. Considering the fact that Nm is not bigger than IICS, there may be many combinations of rows, where the memory operations can be allocated in the kernel. For example, there are six combinations when (IICS, Nm) = (4, 2), namely, (0, 1), (0, 2), (0, 3), (1, 2), (1, 3), and (2, 3). A combination (x, y) means that only xth row and yth row of kernel are allowed to place the memory operations. For example, (2, 3) means that only second row and third row are allowed to place the memory operations, as shown in Fig. 2(e).

Search Strategy

So far, we have solved the problem that dependence should be replaced when a pair (IICS, Nm) is given. To make the search in the space of (IICS, Nm) efficient, we design a search strategy. In order to explain all the rules in our search strategy clearly, we give an example. We map the DDG in Fig. 4(c) onto the 2 × 2 CGRA in Fig. 3(c) whose local memory has 2-read and 1-write ports, and the latency of memory operation is double to that of a logic operation (σ = 2).
Next, we define a dynamic collection IIset to record all the search nodes in the current search space. In order to match them with the combinations (IICS, Nm), we design the IIset as a trituple (line 9). Then, our approach attempts to find a valid mapping in the while loop. Every time, it chooses the combination with the MII from the present search space (line 11) and processes the input DDG with the chosen IICS and Nm. Then, our algorithm attempts to do P&R with the replaced DDG. If it fails, the algorithm will perform the following steps.

1) If the condition $Nm = MNm$ is satisfied, we need to update the dynamic search space with two new search nodes: 1) $(IICS+1, MNm)$ and 2) $(IICS, MNm+1)$, where IICS is the number of CSs in the present iteration.

2) If the condition $Nm = IICS$ is satisfied, the algorithm only needs to delete the current search node from the dynamic search space because of the limitation of $Nm \leq IICS$.

3) If those two conditions are not satisfied, our algorithm will update the current search node with $(IICS, Nm +1)$. Otherwise, we exit because a valid mapping has been achieved.

**DATA MEMORY MANAGEMENT**

In a CGRA-based computing platform, data are transferred between PE array and main memory through local data memory. Thus, the local memory is not only required to serve the PE array, but also to exchange data with the main memory. To improve memory access efficiency, we design a memory management mechanism based on double buffering. By this mechanism, the data exchange between PE array and local memory is parallel with the data exchange between local memory and main memory so that communication time can be eliminated.

In this mechanism, the local data memory is divided into four parts, marked as $P^{1}$, $P^{2}$, $P_{II}^{1}$, and $P_{II}^{2}$, as shown in Fig. 4(c). Each part alternately serves the PE array and exchanges the data with the main memory.
Fig. 4. Example of memory management: (a) Pseudocode of loop, (b) Execution flow of the loop in (a), (c) The process of memory access for the groups from k to k+3.

The data exchange between local memory and main memory is carried out by a multichannel DMA. Each memory part corresponds to an independent channel so that each memory part can have an individual memory access pattern. The DMA is controlled and triggered by the host controller, such as the configuration, initialization, read, or write.

Advantages:
The Master of IEEE Projects

- High Power efficiency
- High performance

Software implementation:

- Modelsim
- Xilinx ISE