Lego_v0: A domain-specific compiler for testing of sparse tensor algebra applications on coarse-grained reconfigurable arrays

Sai Gautham Ravipati[1]
[1]Department of Electrical Engineering, Stanford University
sgautham@stanford.edu

16th June, 2024

Introduction: The Hardware Stack

Most domain-specific accelerators today involve a dedicated piece of hardware for accelerating the computation tied to small memories with faster accesses, generally packed with data meant to be reused. One such domain is sparse tensor algebra which is a fundamental block in many machine learning applications like graph machine learning algorithms. One of the domain-specific accelerators of interest are the coarse-grained reconfigurable arrays (CGRAs) which offer a lower level flexibility in terms of reconfiguration over FPGAs, but are more flexible than ASICs. The architecture of a typical CGRA SoC has been illustrated in Figure-1. The CGRA has memory tiles and processing elements (PEs). The on-chip control processor (CP) orchestrates the data between the global buffer and the CGRA memory tiles. The data initially resides on the off-chip main memory and the off-chip application processor (AP) orchestrates this data between the main memory and the chip via the TLX interface.

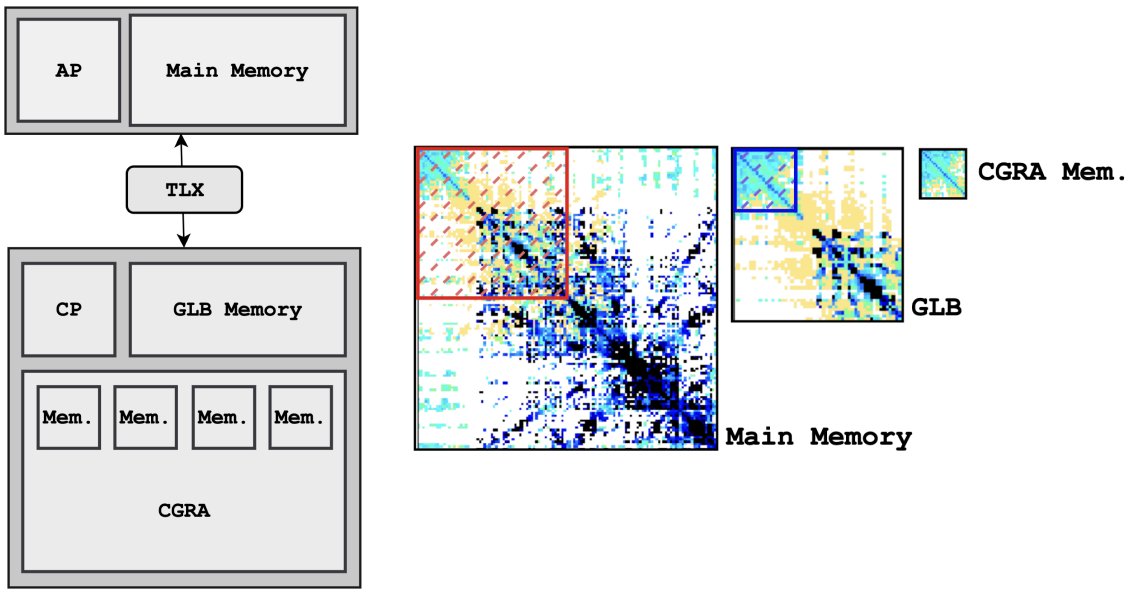

Figure-1: (left) Abstraction of the hardware architecture in Agile Hardware project. (right) Hierarchical tiling of the input tensor into tiles and subtiles.

One of the important constraints that applications should satisfy is the fact that the memory capacities of the global memory and CGRA memory are limited. Hence, the large input tensor that resided on the DRAM is tiled hierarchically into tiles and subtiles. The CGRA accelerator operates on the subtiles. The size of a given subtile is bounded by the size of the CGRA memory tile. And the number of subtiles that can be packed onto the global memory and in turn the size of a tile is bounded by the size of global memory (4 MB). For each application the user needs to write code that runs on the application processor and the control processor to break and orchestrate the tiles and subtiles across the memory hierarchies.

Why do we need to generate code?

The fact that users need to write different code for every application doesn't scale, particularly in the case of sparse tensor algebra. In the case of dense tensor algebra, the process of breaking the tensors into tiles and tiles into subtiles would be a set of nested for-loops. For the case of sparse tensor algebra, we need to handle the additional processing of pairing the tiles and subtiles. To perform the same, we preprocess the input tensor to add ranks and the additional ranks encode the information about the tile and subtile of a given element. Figure-2 illustrates the pre-processing of input tensor data. Now, pairing of the tiles and subtiles would be the generation of co-iteration loop nests similar to TACO (Figure-3), to intersect and drop the tiles and subtiles if any of the input operands doesn't have a corresponding pair. Since this co-iteration loop nest varies from application to application and also the compression of a given tensor dimension, handwriting code to pair tiles/subtiles is not scalable for sparse tensor algebra. In this work, we use TACO style code-generation for pairing the titles and subtiles.

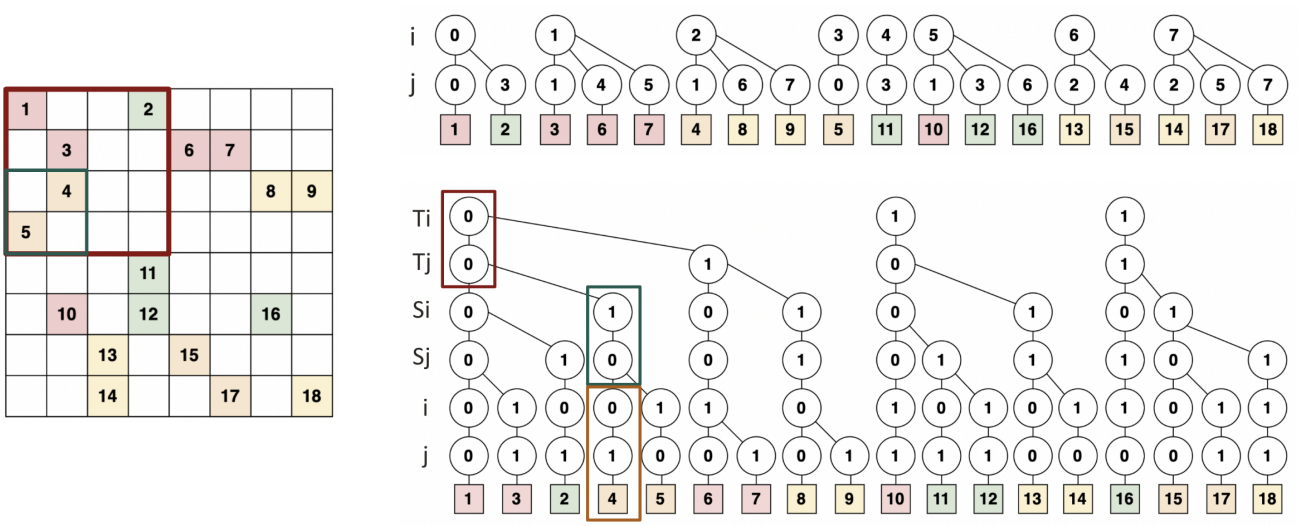

Figure-2: (left) (8, 8) tensor with (4, 4) tiles and (2, 2) subtiles. (right) Fibre-tree before and after pre-processing. For example, element 4, belongs to (0, 0) tile and (1, 0) subtile within a tile and location (0, 1) within the subtile.

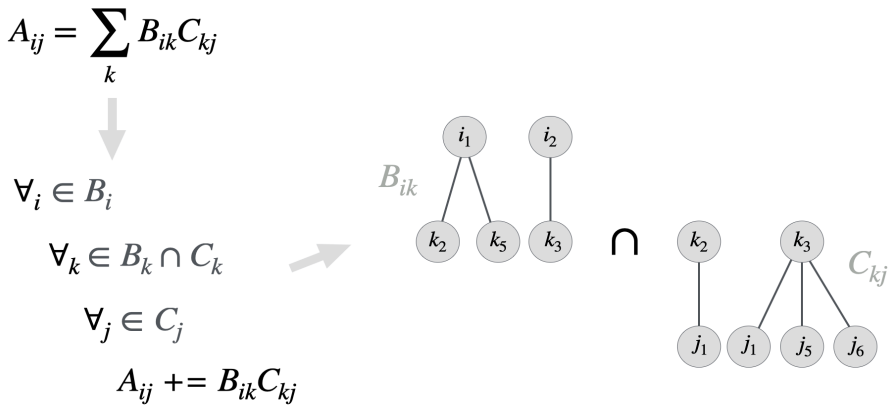

Figure-3: TACO style co-iteration for loop nest generation of matrix multiplication. The dimension common to both the tensors is intersected to eliminate redundant computation.

Current code-generation scheme and input language

```
stmt: A[i,j] = B[i,k] * C[k,j]
tile_schedule:    [i, k, j]
subtile_schedule: [i, k, j]
cgra_schedule:    [i, j, k]
i:split:3000:240:30
j:split:3000:240:30
k:split:3000:240:30

B:<tensor_type>:<tensor_path>:s:0:60:int16
C:<tensor_type>:<tensor_path>:s:0:60:int16
```

Figure-4: Input language of Lego_v0. The stmt itself is an einsum expression. The tile and subtile schedule govern the data flow order in which the tiles and subtiles are paired, while the CGRA schedule should conform to the data format expected by the chip. The splits of each dimension are static sizes of the dimension, tile dimension and the subtile dimension. We also need to specify the source of the operands, which has additional flags like (sparse/dense), (transpose), (percentage_sparsity), (data-type).

The input language for code-generation has been described in Figure-4. The user needs to specify the einsum expression for the tensor algebra application. In the current implementation, all the dimensions are assumed to be sparse for code-generation. The schedule at each level specifies the co-iteration loop nest of the tile and subtile coordinates. This governs the dataflow order in which the tiles and subtiles are paired. Additionally, for each tensor we have some additional properties in place, for say, the percentage of sparsity for random number generation, the data type of the values, etc. With this language in place, we generate the code that runs on the application and the control processor to pair and stream the tiles and subtiles.

The generated data/code differs based on the mode of testing. Furthermore, we also generate the TACO/numpy code that operates on the raw tensors before pre-processing and generates the output gold tensor.

Implementation Notes: Subtile Pairing on AP and extent duplication

One of the important factors for performance is the fact that we need to send the data over the TLX interface. If we pair the tiles on the application processor and stream the paired tiles to the global buffer, this leads to streaming and storing of redundant subtiles on the global buffer as every subtile doesn't have a corresponding pair in all of the other operands. Hence, we also pair the subtiles on the AP. For the RTL mode, on execution of the AP code, generates a set of subtile pairs that could be fed as input to the Sparse Abstract Machine (SAM) dataflow graph which produces a simulated output. We also generate the gold output for the subtile pairs to be matched with the simulation result. For the case of on-chip test collateral, we generate the code that runs on the control processor to stream the paired subtiles and verify the output from the chip, the paired subtiles in the form of global memory header files and the gold data. Further, in the case of applications involving broadcast that re-use the subtiles, instead of duplicating the subtile, we store pointers to the start and end locations of the subtile (called the extents), and duplicate the extents for reuse of subtiles. Figure-5 and 6 illustrate subtile pairing on AP and extent duplication.

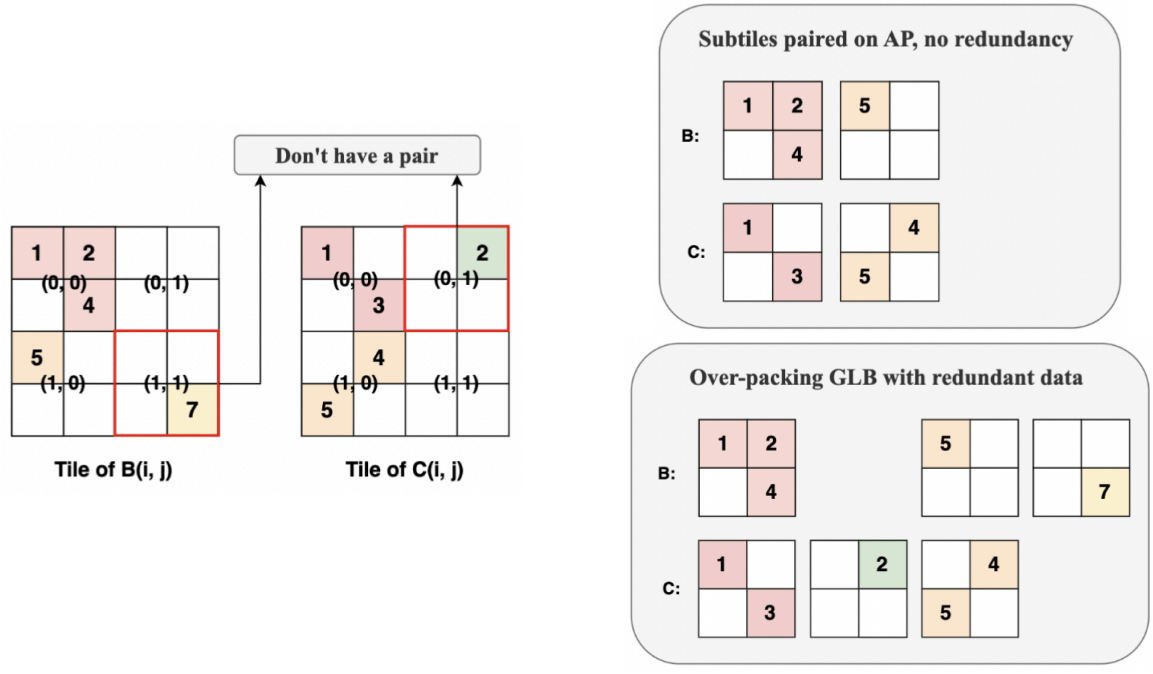

Figure-5: (left) Illustration of subtile pairing for B(i, j) * C(i, j), (right) Streaming subtiles over the TLX interface and storing on the global buffer leads to over-utilisation. Hence, we pair the subtiles on the application processor.

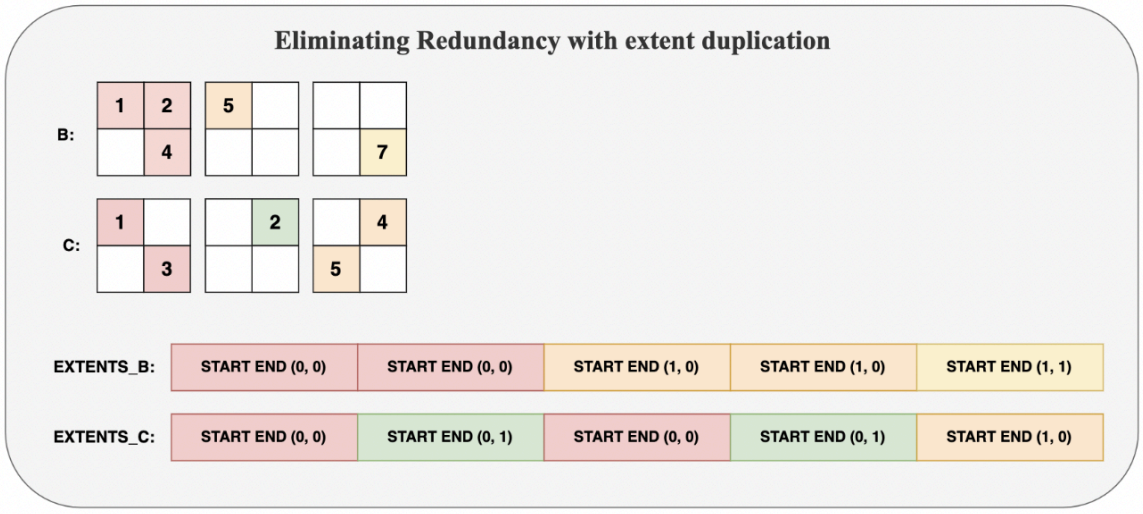

Figure-6: For the tiles shown in Figure-5, and if the operation is B(i, k) * C(k, j), we need to reuse some of the subtiles for multiple computations. Hence, we store them only once on the global buffer and duplicate the pointers to start and end locations on the global buffer (extents).

Conclusion

Lego_v0 acts as a verification collateral for chips in the Agile Hardware Project. It provides the ease of testing different sparse tensor algebra applications in different modes - RTL and on-chip testing. Moving further, we would like to enhance this to deploy sparse applications end-to-end on a heterogeneous hardware stack with multiple processors that tile and orchestrate data across multiple memory hierarchies and the multiple accelerators. There are couple of aspects to consider for building such a compiler and making it more general - dynamically buffering the tiles and subtiles over pre-processing, multiple schedules for binding the operations like matrix multiplication which scatter the output tiles/subtiles which are to be merged to obtain the result, incorporating better tiling methods like position space tiling etc.