ACA Project Report: Architectural Synthesis for Machine Learning Models

Quintino Francesco Lotito (215032)
MSc in Computer Science, 2nd year
University of Trento
Trento, Italy
quintino.lotito@studenti.unitn.it

1 INTRODUCTION

Architectural synthesis is the process of automatic mapping from a high-level description of the circuit behavior (e.g., the computational graph of the operations to carry out) to a low-level circuit description of the architecture, by considering also constraints given by the available resources, the area available for the circuit and the latency time.

Given a computational graph, a set of available resources and a set of constraints, fundamental problems of architectural synthesis are to perform the scheduling of the operations (i.e., define the execution starting time of the operations, considering the dependencies given by the computational graph) and the binding of the operations (i.e., specify what resources perform a given operation).

Of course, multiple solutions exist given a high-level description, and different algorithms have been proposed to explore the design space and optimize different designer-imposed criteria. Typical measures to evaluate the quality of a circuit description (design evaluation) are the area, the clock period and the latency.

The goal of this project is to develop a software that, given as input the description of a neural network, it constructs the computational graph induced by its inference step and performs the scheduling and the binding of the operations described by the graph.

2 BACKGROUND AND PROBLEM DEFINITION

In this section, we give some basic definitions related to the problem of our interest and formalize some concepts mentioned in the introduction.

Definition 2.1 (Computational graph). A computational graph is a graph $G = (V, E)$ in which V is the set of vertices and E is the set of edges. Moreover, the vertices represent *operations*, and the edges are *directed*, meaning that they have a direction that embed the notion of *dependency* among the operations. The graph G is also *acyclic*, meaning that it contains no cycles (i.e., there cannot be an operation a that depends from an operation b that depends from the operation a again). An example of a computational graph is reported in fig. 1.

Definition 2.2 (Convolutional Neural Network). In short, Convolutional Neural Networks (CNNs) are a class of neural networks that are mostly used to deal with images. The name is given by the *convolution* operation, which appears in at least one layer of the network. A convolution is the application of a filter (the *kernel*) that maps the input to a feature space, which highlights

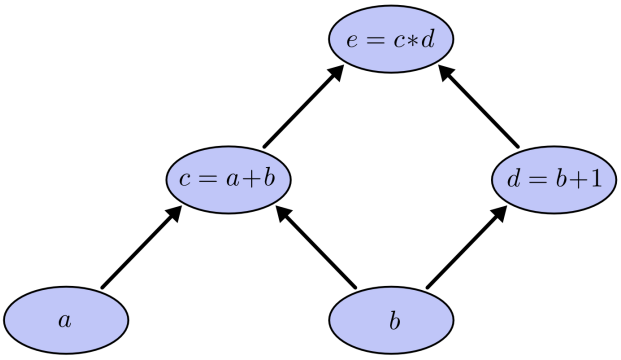

Figure 1: Example of a computational graph.

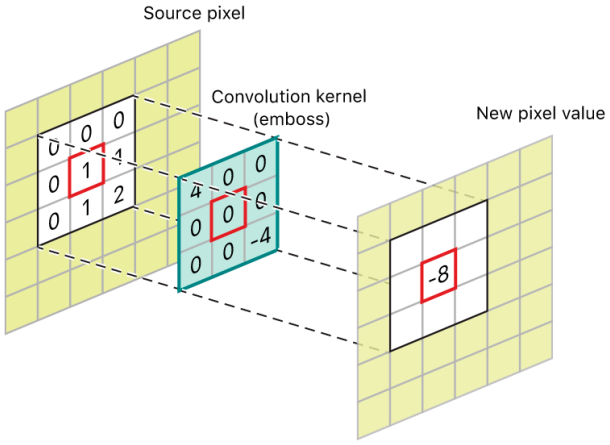

Figure 2: Mathematics of a convolution layer.

the locations and strength of a detected feature in an input. The breakthrough introduced by neural networks is that these filters are learned at training time and are optimized to solve a certain task, they are not handcrafted. The mathematical operations performed by a convolution layer are pretty simple: the kernel is applied multiple times over all the possible windows of pixels of the image, a point-wise multiplication is performed between the pixels of the input and the respective points of the kernel and the everything is summed up. A summary of these operations is reported in fig. 2.

Problem 2.1 (Scheduling). Given a computational graph G, the problem of the scheduling consists in labeling each vertex of G with the starting time of its execution, respecting the dependencies of G. More formally, the scheduling is a function:

$$\phi : V \mapsto \mathbb{Z}^+ \tag{1}$$

where $\phi(v_i)$ denotes the execution time of v_i, and ϕ respects the dependencies of G.

Problem 2.2 (Binding). Given a computational graph G, the problem of the binding of the operations consists in choosing what resource performs each operation. More formally, the binding is a function:

$$\beta : V \mapsto R \times \mathbb{Z}^+ \tag{2}$$

where $\beta(v_i) = (t, r)$ indicates that the node v_i is executed by the r-th instance of the resource of type t.

Definition 2.3 (Constrained / Unconstrained scheduling and binding). The two problems mentioned above can be considered in the *constrained* setting, meaning that we have a limited number of resource to allocate at a given time, and in the *unconstrained* setting, meaning that we have no resource limitations. The binding in the unconstrained setting is not particularly relevant.

Problem 2.3 (Architectural Synthesis for Machine Learning Models). The goal of this project is to:

- Develop a software able to generate the computational graph given by the inference performed by a single convolutional layer on a matrix input.
- Perform the scheduling and the binding of the generated computational graph, by implementing different algorithms from literature and considering the case of resource-constrained and resource-unconstrained setting.

3 IMPLEMENTATION

In this section we survey our implementation choices. The programming language of our choice is Python 3.x.

3.1 Building the computational graph

The first step for this part of the project is to implement a graph data structure. This data structure should implement all the basic attributes and methods of a graph, such as containing the label of the vertices and storing the edges. This implementation is contained in the file *graph.py*.

The non-trivial part is to generate the DAG from the description of a neural network. In this project, we decided to first focus only on classical mathematical operation in infix notation, and then proceed to the subset of the convolutional neural networks with only one step of convolution. The first choice is for debug purposes, the second is to simplify the problem with respect of the time allocated to develop this project.

Generating the expression tree from a mathematical operation is a well-known problem. It is performed into two steps:

- Converting from infix (e.g., $A + B$) to postfix notation (e.g., $AB+$), since it is easier to be handled by a machine.
- Creating the tree by relying on a stack. This is done in the following way. We iterate across the postfix formula, if the character is an identifier we create a node in the graph and add the identifier to the stack, otherwise it is an operator and we add a node to the graph and extract from the stack the previous two values, which will be linked to the operator node.

To generate the computational graph of the convolutional layer, we proceed in the following way:

- We iterate over all the possible windows of pixels induced by kernel applications.
- At iteration i, after having selected the window of pixel M_i, we create a node for every element of M_i and for every element of the kernel K.
- We need to multiply point-wise M_{ij} with K_{ij}, therefore for every multiplication we create a node for the multiplication and link it to the respective M_{ij} and K_{ij}. For each multiplication, we store the id of the operator node.
- We iterate over all the stored ids and construct the nodes for the summations. We limit each operator to consider only two inputs, therefore multiple summations in cascade are created.

A limitation of this approach is that we create multiple nodes in the graph related to the same value, and this is a waste of memory access time, since we could store the values in registers and use them again when need. However, we opted for this simplification.

We can summarize the content of a graph by saying that each node which contains a variable represents accessing the memory and storing the value in a register, and each node which contains an operator represents a mathematical operation (computed by an instance of the available resources) between two predecessor nodes.

The implementations of these parsers are contained in the files *expression_parser.py* and *conv_parser.py*.

3.2 Unconstrained scheduling

After having built the computational graphs, we can focus on the problem of scheduling the operations defined by the DAG G. Since from now on the input is a graph, we can ignore if the graph comes from a convolutional operation or from a mathematical formula.

The first problem we consider is that of resource unconstrained scheduling, meaning we have infinite availability of every resource. We implemented two algorithms to solve this problem: *asap* (i.e., as soon as possible), and *alap* (i.e., as late as possible).

Asap. In asap each operation is scheduled as soon as it is allowed by its dependancies. It can be solved by simply operating a topological sort of G. Every time all the predecessors of a node v are scheduled, we can schedule also v at a time computed considering the execution time of all the predecessors.

Alap. In alap, the scheduling must satisfy an upper bound on the latency time, and each operation is scheduled at the maximum allowed time (whereas in asap we scheduled at the minimum allowed time). The implementation of alap is similar to asap, but in reverse. In alap, a solution may not exist, since not all schedules may finish until the upper bound.

3.3 Scheduling and binding under resource constraints

In this case, beside the input graph G, we also have a list of resources with the respective number of instances allowed. Meaning that, if the availability of resource R at time t is R_t, at time t there cannot be assigned more than R_t instances of R.

The problem of scheduling under resource constraints and minimum latency is know to be *intractable*. Therefore, a number of proposals have been made to approximately solve the problem.

In this project, we implement the algorithm *LIST_L*, which belongs to a family of heuristics known as listing scheduling algorithms.

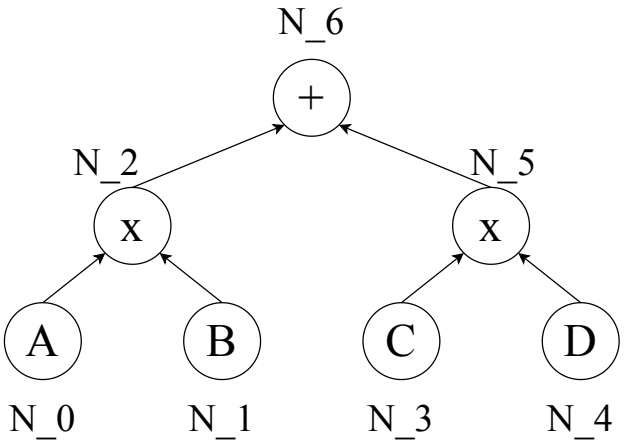

Figure 3: Computational graph of equation 3.

The idea is simple. Until all the nodes are scheduled, at each timestep t we inspect the availability of each resource R_i at time t and schedule (until we have available resources) all the operations that are ready to be scheduled and need R_i. We free the resource when the operation is scheduled to finish. On a more implementation side, the free of the resources is implemented using a priority queue which orders the resources in use by time of finish schedule. Since every time we schedule one operation we need to reserve one resource, also a form of binding is performed.

All the algorithms presented in this section are implemented in the Graph data structure in the file *graph.py*.

4 RESULTS

In this section we present some examples of the execution of the software. We consider the availability of the following resources: registers with a cost of 1, adders with a cost of 2 and multipliers with a cost of 4. Access to memory costs 1. With one access to memory we can read only one value.

For illustrative purposes, let us consider this simple formula:

$$(A \times B) + (C \times D) \tag{3}$$

The resulting computational graph is reported in fig. 3.

The output of the software for all the different algorithms are reported in fig. 4.

In the list reported after the execution of each algorithm we report the scheduling time of every node, i.e., T_i is the scheduling time of node i. Alap is executed with a latency bound of 20. LIST_L is executed considering the availability of 3 multipliers, 5 adders and 3 registers.

In fig. 5 we repeat again the experiment with the same graph but setting the availability of registers to 4 instead of 3. We can notice an improvement in the overall latency, since A, B, C, D can all be read from memory, stored in registers and the required multipliers can operate almost in parallel. Another thing one can notice is that if the availability is set to 2, the scheduling become impossible, and the software prints an error.

The execution of the example of a convolutional network is not reported here since it is much more complicated to visualize, however the concept is the same.

5 CONCLUSION

The aim of the project was to perform architectural synthesis applied to a machine learning task. We selected convolutional neural networks as a target, since they are simply formed by summations and multiplications. We broke the problem in two different steps: construction of the computational graph and development of scheduling and binding algorithms (in a resource unconstrained and constrained scenario). We implemented two unconstrained scheduling algorithms: asap and alap, and one heuristic constrained resource scheduling algorithm: LIST_L.

To verify that the algorithms were correct, we also implemented the construction of computational graphs from simple mathematical formulas, which are more understandable, and proposed examples of the execution of the algorithms.

Of course, a lot of improvements could be done. For instance, in the case of convolutional networks we reported the simplifying choice of creating a new node even when a value is sure to be reused. This causes the associated register to be set free and then to require another access to memory. Another things that could be done given more time are more on the “engineering” part, a more user-friendly way to set the parameters instead of setting up the parameters in the code could be nice.

```
----- ASAP Algorithm -----
Operation N_0 = A (read from memory to register) scheduled at time t = 1
Operation N_1 = B (read from memory to register) scheduled at time t = 2
Operation N_2 = N_1 * N_0 (mathematical operator) scheduled at time t = 3
Operation N_3 = C (read from memory to register) scheduled at time t = 3
Operation N_4 = D (read from memory to register) scheduled at time t = 4
Operation N_5 = N_4 * N_3 (mathematical operator) scheduled at time t = 5
Operation N_6 = N_5 + N_2 (mathematical operator) scheduled at time t = 9

T = [1, 2, 3, 3, 4, 5, 9]

----- ALAP Algorithm: latency bound = 20 -----
Operation N_0 = A (read from memory to register) scheduled at time t = 12
Operation N_1 = B (read from memory to register) scheduled at time t = 13
Operation N_3 = C (read from memory to register) scheduled at time t = 14
Operation N_4 = D (read from memory to register) scheduled at time t = 15
Operation N_2 = N_1 * N_0 (mathematical operator) scheduled at time t = 16
Operation N_5 = N_4 * N_3 (mathematical operator) scheduled at time t = 16
Operation N_6 = N_5 + N_2 (mathematical operator) scheduled at time t = 20

T = [12, 13, 16, 14, 15, 16, 20]

---- LIST_L -----
Operation N_0 = A (read from memory to register) scheduled at time t = 1
Operation N_1 = B (read from memory to register) scheduled at time t = 2
Operation N_2 = N_1 * N_0 (mathematical operator) scheduled at time t = 3
Operation N_3 = C (read from memory to register) scheduled at time t = 3
Operation N_4 = D (read from memory to register) scheduled at time t = 7
Operation N_5 = N_4 * N_3 (mathematical operator) scheduled at time t = 8
Operation N_6 = N_5 + N_2 (mathematical operator) scheduled at time t = 12

T = [1, 2, 3, 3, 7, 8, 12]
```

Figure 4: Execution of the different scheduling algorithms on a simple mathematical formula.

```
---- LIST_L -----
Operation N_0 = A (read from memory to register) scheduled at time t = 1
Operation N_1 = B (read from memory to register) scheduled at time t = 2
Operation N_2 = N_1 * N_0 (mathematical operator) scheduled at time t = 3
Operation N_3 = C (read from memory to register) scheduled at time t = 3
Operation N_4 = D (read from memory to register) scheduled at time t = 4
Operation N_5 = N_4 * N_3 (mathematical operator) scheduled at time t = 5
Operation N_6 = N_5 + N_2 (mathematical operator) scheduled at time t = 9

T = [1, 2, 3, 3, 4, 5, 9]
```

Figure 5: Execution of the same computational graphs but with 4 available registers.