# CPEN 513 Project Report: Optimizing CNN Accelerator Dataflow Using Simulated Annealing

Dingqing Yang 38800141

2020 May

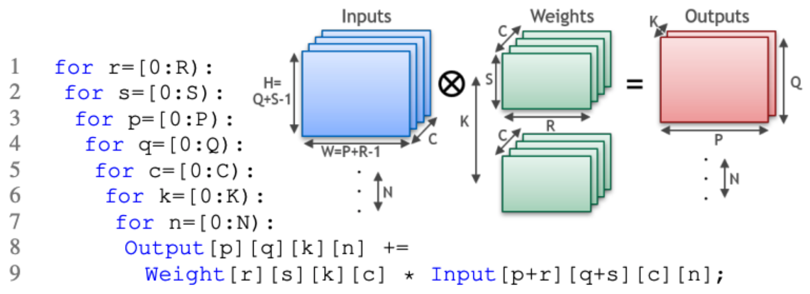
## 1 Project Overview

Deep Learning has become revolutionary in many AI applications at the cost of huge computation and memory requirements. To reduce the burden, many accelerator such as GPUs, FPGAs, and ASICs are proposed to improve the enrgy efficiency. However, the hardware itself cannot maximumly boost the energy efficiency. A smart "compiler" is required to figure out how to optimally map the DNN execution onto the underlying hardware. Many existing work [Ragan-Kelley et al., 2013, Ma et al., 2017, Parashar et al., 2019] has studied the problem under different hardware platform. All studies show a significant impact of different scheduling (i.e. mapping) on even the same hardware. We refer to the processing of finding optimal mapping on a hardware accelerator as *dataflow optimization*.

However, dataflow optimization is difficult due to many reasons. First, the search space is extremely huge such that it is challenging to exploit the entire search space efficiently given reasonable amount of time. Second, the search space is discrete and its geometry is complex and non-convex. This leads to gradient based method inappropriate. The current dataflow optimization tool such as Timeloop Parashar et al. [2019] is aware of such difficulty and requires the user to provide a mapspace constraint to reduce the search space size. A simple random sampling based search algorithm is used to discover the optimal mapping. This algorithm is inefficient since no knowledge is transferred from evaluating one mapping to another.

Given the above description on the search space, I find that simulated annealing could be a proper optimization algorithm for it. Therefore, in this project, I extend Timeloop to use simulated annealing for finding the optimal mapping. This is not straight-forward since Timeloop's internal mapspace representation [1] is unsuitable for simulated annealing. The rest of the report is organized as follow: section 2 introduces appropriate background on dataflow optimization techniques, and further elaborates on mapping and mapspace; section 3 proposes a new mapspace representation that is suitable for simulated annealing;

[1]Details are described in subsection 2.3



```
for r=[0:R):
 for s=[0:S):
  for p=[0:P):
   for q=[0:Q):
    for c=[0:C):
     for k=[0:K):
      for n=[0:N):
       Output[p][q][k][n] +=
         Weight[r][s][k][c] * Input[p+r][q+s][c][n];
```

Figure 1: Conv Layer Execution as 7D For Loop. Figure Credit from: [Parashar et al., 2019]

section 4 discusses some implementation details; section 5 presents results and discussion; section 6 concludes the report.

## 2 Background

### 2.1 Dataflow Optimization Techniques

Execution of DNN layers can be fomulated as a deep loop nest as shown in Figure 1. However, there are many valid ways to map this problem on a hardware accelerator:

1. we can tile the loop to adjust the working set size to the size of different levels of the memory to exploit better reuse.

2. we can permute loop order which result in different execution order, and thus different reuse pattern.

3. we can spatially unroll some loop to exploit available hardware parallelism.

The above 3 ways are the most common dataflow optimization techniques, namely loop tiling, loop reordering, and spatially partitioning. A specific tiling, ordering, partitioning choice specifies exactly a mapping (i.e. scheduling decision).

### 2.2 Mapping Representation

We need an efficient "language" (i.e. representation) to represent each mapping. Loop representation is commonly used and very expressive. An example mapping for a toy 1D convolution is shown in Figure 2. In this example, tiling is performed across multiple tiling levels; ordering is self-explanatory; spatial unrolling in performed in the spatial tiling level where each for loop is annotated as parallel_for.

```
// === 1D Convolution Workload ===
for r=[0:R):
 for p=[0:P):
  Output[p] += Weight[r] * Input[r+p];

// === Mapping ===
// DRAM
for r3=[0:R3):
 for p3=[0:P3):
  // GBuf
  for r2=[0:R2):
   for p2=[0:P2):
    // Spatial: GBuf->RFile
    parallel_for r1=[0:R1):
     parallel_for p1=[0:P1):
      // RFile
      for r0=[0:R0):
       for p0=[0:P0):
        p = p3*P2*P1*P0 + p2*P1*P0 + p1*P0 + p0;
        r = r3*R2*R1*R0 + r2*R1*R0 + r1*R0 + r0;
        Output[p] += Weight[r] * Input[r+p];
```

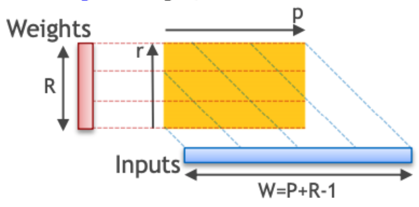


Figure 2: Example Mapping using Loop Representation. Figure Credit from: [Parashar et al., 2019]

## 2.3 Mapspace

It is clear that one can evaluate different performance metrics give a mapping and it's underlying hardware. But in order to search across different mappings, we need to define the problem's entire search space, namely the *mapspace*. Timeloop Parashar et al. [2019] naturally define the mapspace by 3 subspaces, namely `IndexFactorizationSpace`, `PermutationSpace`, and `SpatialSplitSpace`, where each of them specifies different tiling, ordering, and partitioning choices respectively. The user can specify constraints on different subspace to reduce the search space size to speedup the search process. However, this decoupled mapspace is hard to formulate a move in simulated annealing. We therefore construct a new unified mapspace that is suitable for simulated annealing.

## 3 New Mapspace Formulation: Primitive List Representation

We perform a prime factorization on each bounds of the problem dimension. Each pair of dimension name and prime factor is referred as a *primitive* element to construct a mapping. A special primitive is introduced to separate different primitives across different tiling boundaries. All primitives are concatenated to be a list which specify tiled loop orders in a mapping. We therefore name this mapspace formulation as *primitive list*. An example of this formulation is provided in the next paragraph.

Assume the problem of the interest is a 1D convolution workload similar to Figure 2. Assume loop bounds $R$ and $P$ equal to 12 and 10 respectively. Then the problem will be decomposed to get the following primitives: $[(R,2),(R,2),(R,3),(P,2),(P,5),S,S,S]$ where $S$ denote our spatial primitive: tiling separator. A random shuffling is performed on the list, and the result is used as a initial mapspace representation for mapping generation.

The nice thing about this new mapspace formulation is that swap (or move) across different primitives in the primitive list can be regarded as changes in tiling, ordering, and spatial unrolling. Tiling changes is achieved by moving a primitive from one tiling level to another and merge with all primitives of the same type in the destination tiling level[2]. Reordering happens when primitives of the different dimension exchange its position in the primitive list. Spatial unrolling is expressed by marking all primitives in the spatial tiling level as spatial primitives. All this together specifies a new mappings. Therefore, it is suitable with simulated annealing.

## 4 Implementation

The majority addition to the Timeloop infrastructure is a new mapspace (located at `src/mapspaces/didi.hpp`) and the simulated annealing search algorithm (located at `src/search/simulated-annealing.hpp`). There is one catch with the new mapspace formulation, which is how to handle multiple primitives of the same dimension. This is not allowed by definition since each dimension should only appear once in each tiling level, but this will appear unfortunately because of the pigeonhole principle: we can easily have more number of prime factors than the number of tiling levels which correlates to number of levels in memory hierarchy. So to construct a mapping out of a primitive list representation, we merge the primitives of the same dimension in the same tiling level, and the loop order for each dimension is determined by average position across all primitives. This workaround is implemented in `PrimitiveToSubNest` method in `src/mapspaces/didi.hpp`

## 5 Result and Discussion

We implement simulated annealing and compare with the baseline random sampling based search on a benchmark layer (VGG_conv_5_2). The optimization metric is energy-delay-product. We experiment different hyperparameters such as initial temperature, cooling period, and temperature scaling factor for our simulated annealing approach. No mapspace constraints are added and all invalid mapping are discarded in both case. The results are compared between 1000 iteration of simulated annealing and random sampling terminated at 1000 suboptimal mapping found. The best simulated annealing run have 4% energy

[2] A catch is to determine the loop order for the merged primitives. We describe implementation details on handling it in section 4

improvement compared to basedline but 25% slower than the mapping produced by baseline. In short, we are not able to beat baseline random sampling based search algorithm. The potential reasons are discussed below:

1. *Random initializatin of primitive list* We find that initialization is quite important even under the same hyperparameter setting. Bad initialization can stuck at a invalid point in the mapspace and never get out of it such that the search algorithm is terminated due to no valid mapping found.

2. *Simulated Annealing Runtime* We found that unfortunately simulated annealing based approach is around 5x slower compared to baseline random sampling based search under similar termination condition. The possible reason includes the following: delta cost is hard to evaluate efficiently like placement in assignment 2 since Timeloop's counting based performance evaluation model need to evaluate a mapping from scratch give any change in a mapping. The runtime overhead limits the amount of tuning we can perform, and thus limits the quanlity of the found mapping as well.

3. *Imperfect Primitive List Representation* As identified in section 4, primitive list representation can easily result in multiple primitives of the same dimension allocated in the same tiling dimension, which can be merged but the loop order for the merged primitive is hard to determine. In the meanwhile, it might be desired for a move that happens on a merged primitive instead of single primitive. This is in spirit very similar to partitioning with multi-sink nodes such as assignment3, where distributing a single pin to the other side maybe result in no effect in the overall metric, while it might be desired if it tends out that we move more pins in that multi-sink net to the other side. Here, moving a single primitive can result in no change in the derived mapping, and therefore is less likely to take the swap, while it is possible that the move is actually desired if we further move more primitives of the same dimension, which will actually change the mapping and improve the cost.

## 6 Conclution

In this project, We extend a dataflow optimization tool called Timeloop to search mappings using simulated annealing. The result is not completely satisfying since it does not outperform original simple random sampling based search algorithm, and there are some potential future work to improve upon it. First, initialization of the primitive list representation is so critical such that a solution better than randomly permuting all primitives is desired. Better tuning could be helpful to escape a bad initialization to deliver better search results if we can improve search runtime. To improve swap effectiveness, we can probabilistically swap a merged primitive instead of always swapping single primitives. This can address the bullet point 3 described in the previous section.

## References

Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 45–54, 2017.

Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315. IEEE, 2019.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.