# Example Benchmark

This example benchmark demonstrates the file formats that will be used for the challenge. It shows a small problem to highlight all aspects of the problem definition. The actual challenge will involve larger and more complex problems.

The 'kernel placement' problem is a key component of the flow that transforms a deep learning network description (e.g. TensorFlow) into a 2-dimensional array of homogenous processor tiles. The actual system consist of approximately 400,000 tiles, arranges in a 633 by 633 array.

Each processor tile has 48 Kilobytes of RAM memory to store the coefficients and its program. The computation is very efficient because the coefficient data is kept close to the processor. Each processor tile is also part of fabric that connects it to the 4 neighboring tiles. Each clock cycle, a packet of data can be forwarded to each neighboring tile. The tile can also receive a data packet to perform its processing task

The layers in the deep learning network are mapped to 'kernels' that each perform an individual deep learning computational task. For example, one kernel can compute a '5x5 convolution' while the next may implement a '256->16 fully connected layer'. The deep learning network is a pipeline of such kernels, each performing the computation in parallel. A state-of-the-art deep learning network consists of several dozens of such kernels.
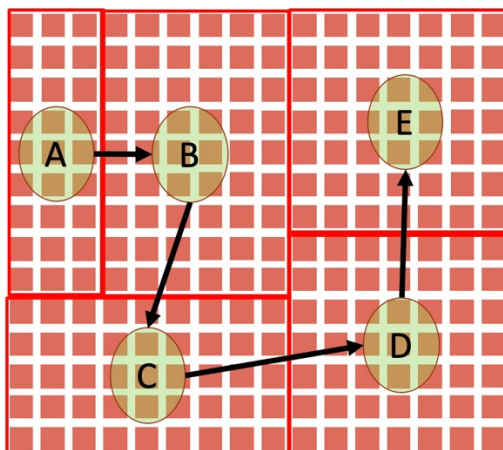
Example of an array of processing tiles with a chain of kernels mapped onto it.
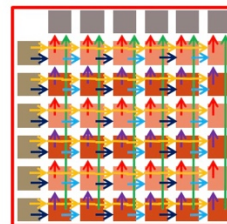
Kernel D instance
Is a 7x7 array
of processing tiles with
internal connections

Physically each kernel is a rectangular array of processor tiles. Its size and shape are malleable. The larger the physical area of kernel is implemented, the higher its throughput because more compute tiles are working on the problem in parallel. The kernel instance with the slowest throughput determines the overall performance of the pipeline. The incentive is to equalize the throughput of each kernel in the network. This means that kernels with high computational load should be larger than kernels with 'easy' jobs. At the same time the kernels have to

physically fit together in a floorplan and utilize as many of the given compute tiles as possible. So, in contrast to the tradition IC floorplan, there is an incentive to maximize the area.

We can now formally split the 'kernel placement problem' into three parts:

1. A *kernel library* specifies the performance functions for each placeable kernel. These are the 'blocks' that need to be placed.
2. A *kernel graph* specifies the particular kernel instances that must be placed. This is the netlist that defines the set of kernel instances and the connections between them.
3. The *problem statement* specifies the optimizer settings such as the solution area and scoring coefficients.

Kernel Library

The example kernel library contains three kernels that can be used to implement the [Residual Convolutional Neural Networks](). We include a brief description of Residual Networks to help make the example library understandable. Understanding the particulars may help build intuition for working with and developing heuristics for this type of problem.

Convolutional networks depend on the [Image Convolutional operator](). The convolution operator accepts an image represented as a 3-Tensor with height H, width W, and feature count C. It applies K different filters at stride T, each with a receptive field size of height R, width S. An image convolution described this way can be implemented as a loop-nest of six loops. The example kernel has execution parameters h' w' c' and k' that split each loop in the nest into some number of tiles (in either the x or y dimension of compute fabric). Ultimately each individual tile owns a region of the iteration space of the loop nest equal to H/h' * W/w' * C/c' * K/k' * R * S. When training a neural network, the same convolution must be run three times for each input. Once for the forward pass of the neural network, once for the backward pass, and once for the weight update pass. Because the loop nests are all the same, one way to implement this is to use three copies of the same kernel placed side-by-side. This is the implementation strategy described by the convperf function below.

The other two kernels in the library implement aggregate residual blocks. Residual blocks in ResNet contain either three or four convolutions. The kernels described below enforce a hierarchical strategy of implementing each residual block side-by-side to its neighbors. They share some loop distribution parameters, which helps the convolutions communicate efficiently with each other.

The benchmark challenge will include a kernel library with tens of kernels. The final challenge will re-use the same kernel definitions as the benchmark. Each kernel is parameterized by formal and execution arguments. The formal arguments will be specified in the kernel graph. The execution arguments must be optimized as part of the compilation process.

```
convperf(H, W, R, S, C, K, T; h', w', c', k') = {
      height = h'*w'*(c'+1)
      width  = 3*k'
      time   = ceil(H/h') * ceil(W/w') * ceil(C/c') * ceil(K/k') * R * S / T / T
      memory = C/c' * K/k' * R * S + (W+S-1)/w' * (H+R-1)/h' * K/k'
}

dblockperf(H, W, F; h', w', c1', c2', c3', k1', k2', k3') = {
      conv1 = convperf(H, W, 1, 1, F,   F/4, 1, h', w', c1', k1')
      conv2 = convperf(H, W, 3, 3, F/4, F/4, 1, h', w', c2', k2')
      conv3 = convperf(H, W, 1, 1, F/4, F,   1, h', w', c3', k3')
      height = max(conv1.height, conv2.height, conv3.height)
      width  = conv1.width + conv2.width + conv3.width
      time   = max(conv1.time, conv2.time, conv3.time)
      memory = max(conv1.memory, conv2.memory, conv3.memory)
}

cblockperf(H, W, F; h', w', c1', c2', c2', c4', k1', k2', k3', k4') = {
      conv1 = convperf(H,   W,   1, 1, F/2, F/4, 1, h', w', c1', k1')
      conv2 = convperf(H,   W,   3, 3, F/4, F/4, 2, h', w', c2', k2')
      conv3 = convperf(H/2, W/2, 1, 1, F/4, F,   1, h', w', c3', k3')
      conv4 = convperf(H,   W,   1, 1, F/2, F,   2, h', w', c4', k4')
      height = max(conv1.height, conv2.height, conv3.height, conv4.height)
      width  = conv1.width + conv2.width + conv3.width + conv4.width
      time   = max(conv1.time, conv2.time, conv3.time, conv4.time)
      memory = max(conv1.memory, conv2.memory, conv3.memory, conv4.memory)
}
```

Kernel graph

The following kernel graph uses three kernels to implement a 9-layer neural network. The input and output kernels may be ignored. Each kernel graph is specified with two sections. The first section lists all of the kernels that will be used and provides values for the formal arguments. The second section lists the connectivity of the kernels and the size of the tensor that must be communicated over each connection. The benchmark challenge will include graphs more diverse and non-linear connectivity graphs, and with larger graphs including thousands of kernels.

```
(* Node Definitions *)
output[0] n=[28 28 512] name='y'
 input[1] n=[56 56 256] name='x'
cblock[2] f=512 h=56 w=56
dblock[3] f=512 h=28 w=28
dblock[4] f=256 h=56 w=56
(* Connectivity *)
 input[1]:_ -> dblock[4]:x, shape:[56][56][256]
cblock[2]:y -> dblock[3]:x, shape:[28][28][512]
dblock[3]:y -> output[0]:_, shape:[28][28][512]
dblock[4]:y -> cblock[2]:x, shape:[56][56][256]
```

Problem statement

Each particular problem statement will provide the following specific arguments.

Kgraph: filename of input kernel graph
Output: file name of output placement solution
Wire-penalty: Co-efficient of penalty for L1 distance between connected kernel centers.
Time-limit: Maximum allowed runtime in seconds
Width, Height: All kernels must fit inside a rectangle with the specified width and height.

```
./candidate-solver kgraph=kgraph.txt output=solution.txt wirepenalty=120 timelimit=60
width=633 height=633
```

Example Solution

The solution file format expresses the formal arguments, execution arguments, coordinates and orientations of each kernel. Each kernel is first instantiated with an assignment statement, and then placed at an (x, y, rotation) location on the fabric.

```
k1 = dblock( 56 56 256 4 4 8 8 8 8 64 16 )
k1 : place(0 148 R0)
k2 = cblock( 56 56 512 4 4 8 8 8 8 32 32 8 19 )
k2 : place(150 0 R0)
k3 = dblock( 28 28 512 4 4 8 8 8 8 32 8 )
k3 : place(0 0 R90)
```

The floor plan looks like this: