MemPool Benchmarks

Here are the results of the initial benchmarks on MemPool. I ran the benchmarks on four different interconnect configurations. All configurations contain 256 cores distributed over 64 tiles. The numbers were extracted using RTL simulation and using the traces generated by the snitch cores.

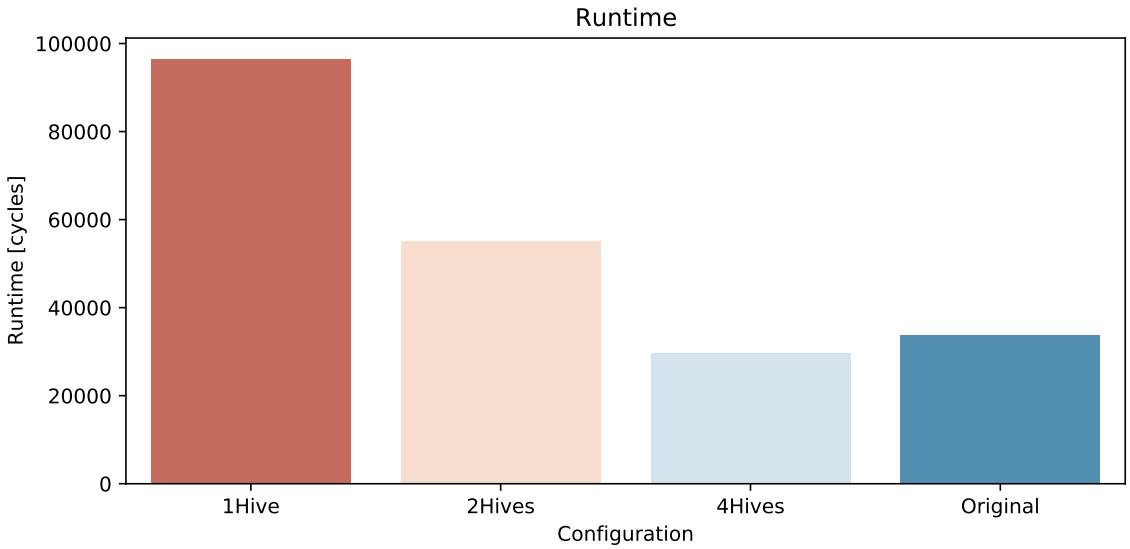
- **1Hive**
  This configuration is close to the one we used originally with one 64Mx64S butterfly interconnect. This means, each tile has only one master port shared by four cores. This implementation is physically feasible and runs at 500MHz post placement and routing.
- **2Hives**
  Here we use the new hierarchical approach that Matheus has described. In this case, instead of connecting all 64 tiles with one interconnect, the tiles are split in two 'hives' of 32 cores each. There are four 32Mx32S butterfly networks here. One per hive to connect the cores inside each hive, and one in each hive connecting to the other one. We did not yet run the back-end for this system, as we hope to implement the next one.
- **4Hives**
  This is exactly the new interconnect that Matheus has mentioned. The cores are split into 4 hives of 16 tiles. Each hive has a local 16Mx16S interconnect, and three connections to the other three hives (north, northeast, east). This structure seems physically feasible, but the back-end is becoming very difficult.
- **Original**
  This is the 'original' design we worked with up to a couple of week ago. It contains 4 parallel 64Mx64S interconnects to give each core in a tile its private master port. This design is physically infeasible, but we benchmark it to compare the new approaches.

Matrix Multiplication

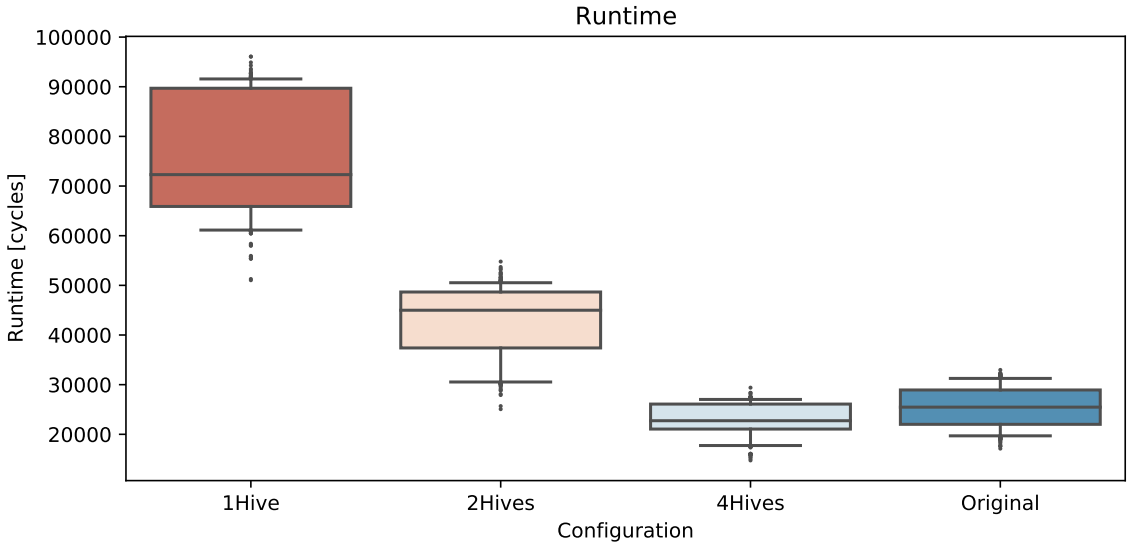
The matrix multiplication kernel consists of calculating C=AB with A being a 256x8 matrix, B a 8x256 matrix and C a 256x256 matrix. Each core calculates one row of C. The implementation uses data reuse and calculates four elements of C simultaneously to reduce the number of loads by a factor of 1.6 compared to a naive implementation.

Runtime

The following figure shows the runtime of the kernel on different HW configurations, so lower is better. The runtime is calculated as the interval from the timestamp at which the earliest core starts until the last cores finishes. Compared to the original version, the 1Hive configuration is 2.86 times slower, and the 2Hive version 1.63 times. The 4Hive version is the fastest with a speedup of 1.14 compared to the original configuration.



The following figure shows the distribution of the runtime of each core. It can be seen that the variability between individual cores is much larger in the 1Hive and 2Hive configuration.



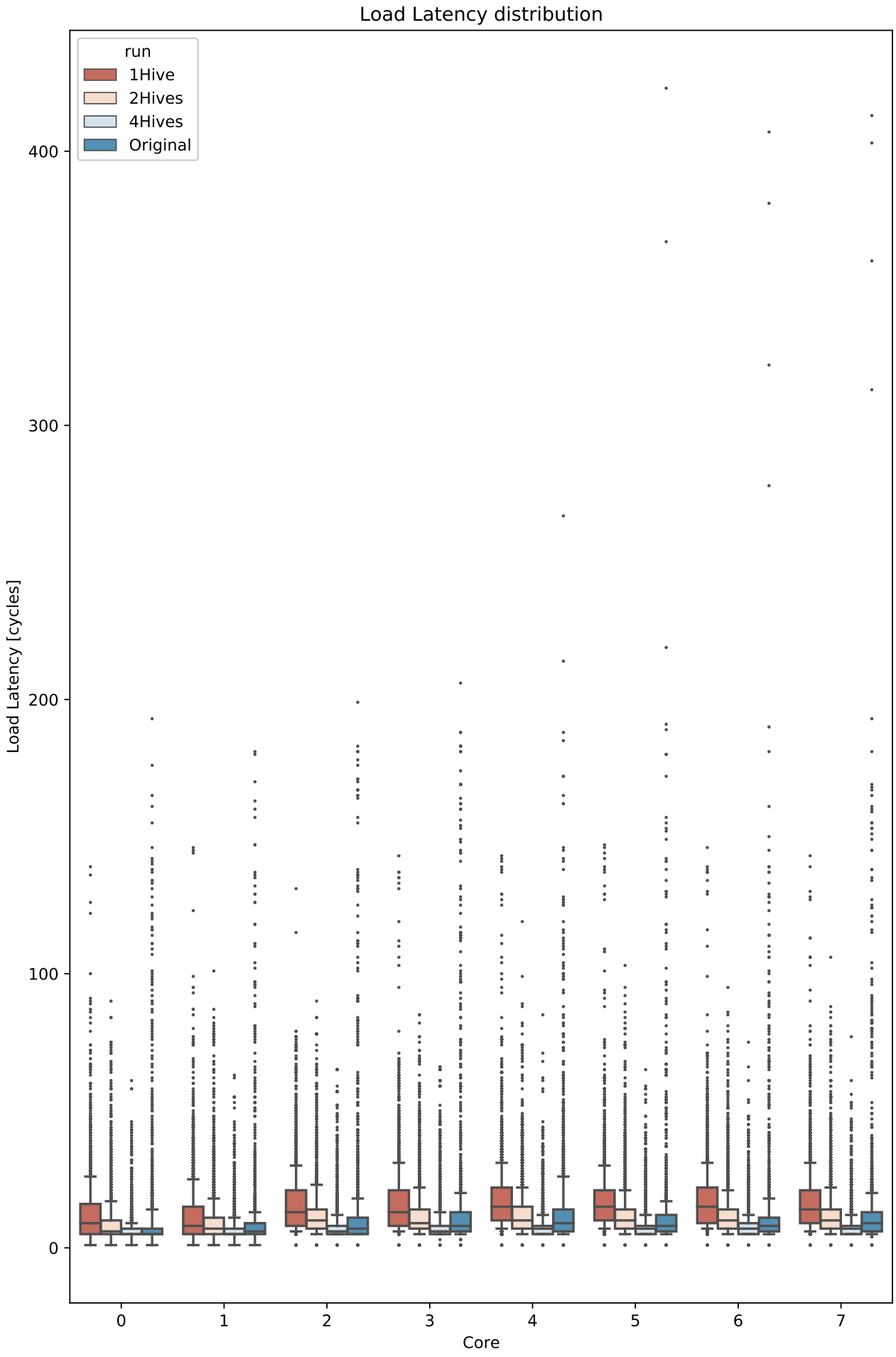
Load Latency

The following table gives some statistics over the load latency of all cores. Similarly the figure on the next page visualizes the data. It plots the load latency for the first 8 cores in the system.

Here, we can see why the 1 and 2Hive configuration perform so much worse than the 4Hives or Original systems. The average load latency of the 1Hive is almost twice as large as the 4Hive's metric. The Original implementation has almost the same average load latency as the 2Hive, but this is due to very extreme outliers. 90 percent of the Original's loads have a latency smaller than 27 cycles, while the 2Hive configuration uses 33 cycles for this number. The difference at the 50% mark is even larger.

	1Hive	2Hives	4Hives	Original
samples	656641	656641	656641	656641
mean	20.38	16.86	11.12	15.96
std	16.82	13.34	8.72	25.82
min	1	1	1	1
10.00%	7	6	5	5
25.00%	10	8	6	6
50.00%	16	13	8	9
75.00%	25	21	13	15
90.00%	38	33	21	27
max	184	156	148	503

The extreme distribution of the outliers can also be seen in the figure and can be explained by the way the matrix multiplication is parallelized. Each core calculates one row, starting with the first column of C. Therefore, all cores load the same values of B in the very beginning. After the initial conflicts, the loads become more aligned, but assuming no conflicts and stalls, each core will load the exact same values of B at the exact same time throughout the kernel. This is something that has to be improved, and I expect the load latency to improve with such changes.


Load Latency distribution
run
1Hive
2Hives
4Hives
Original
Load Latency [cycles]
0
100
200
300
400
0
1
2
3
4
5
6
7
Core

Convolution

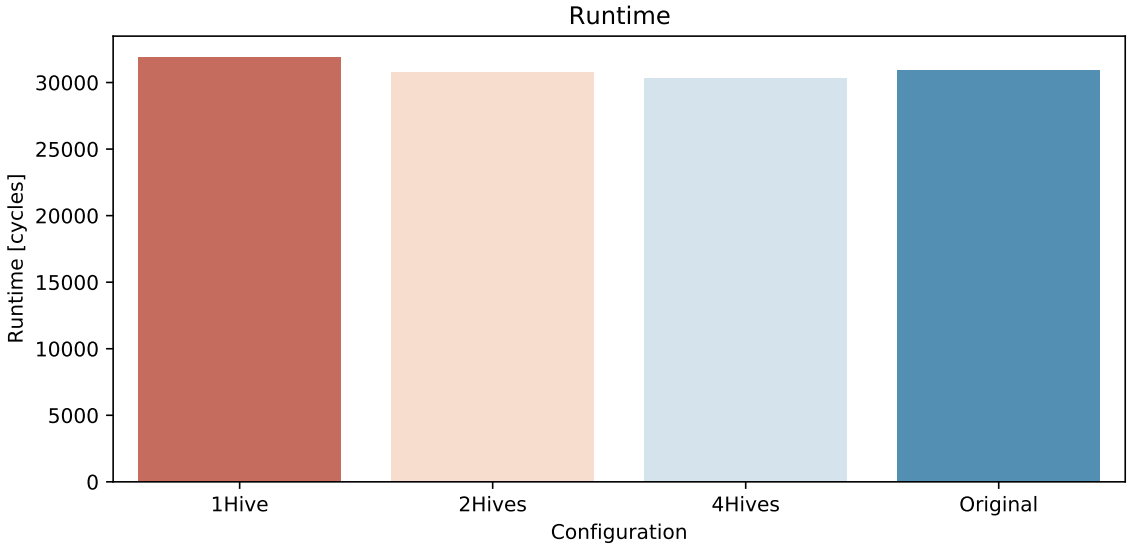
The convolution kernel in this benchmark calculates a Gaussian blur with a 3x3 window. Even though the kernel is fix, it is programmed to be runtime configurable, i.e. each core will have to load the kernel at the beginning and the compiler cannot optimize kernel operations for this scenario.

The kernel was optimized for MemPools memory layout. Specifically, the parallelization is done in such a way that each core will only calculate pixels that are stored in the same tile as the core is. This means that all store operations are 'local' and newer go through the global butterfly networks. The same is true for the loads, with the exception of pixels at the boundary of the tiles. For these pixels, the cores will have to also load one column of values (for the 3x3 kernel) from the neighboring tile.

The benchmark shows the results from processing an image of 1024x80 pixels.

Runtime

All four systems show almost the same performance. This is expected as the kernel hardly uses the global interconnect, which is the only difference in the four evaluated systems. Again the 4Hive configuration is the fastest, being 2% faster than the Original one. The 2Hive is half a percent faster than the Original one and the 1Hive 3% slower.



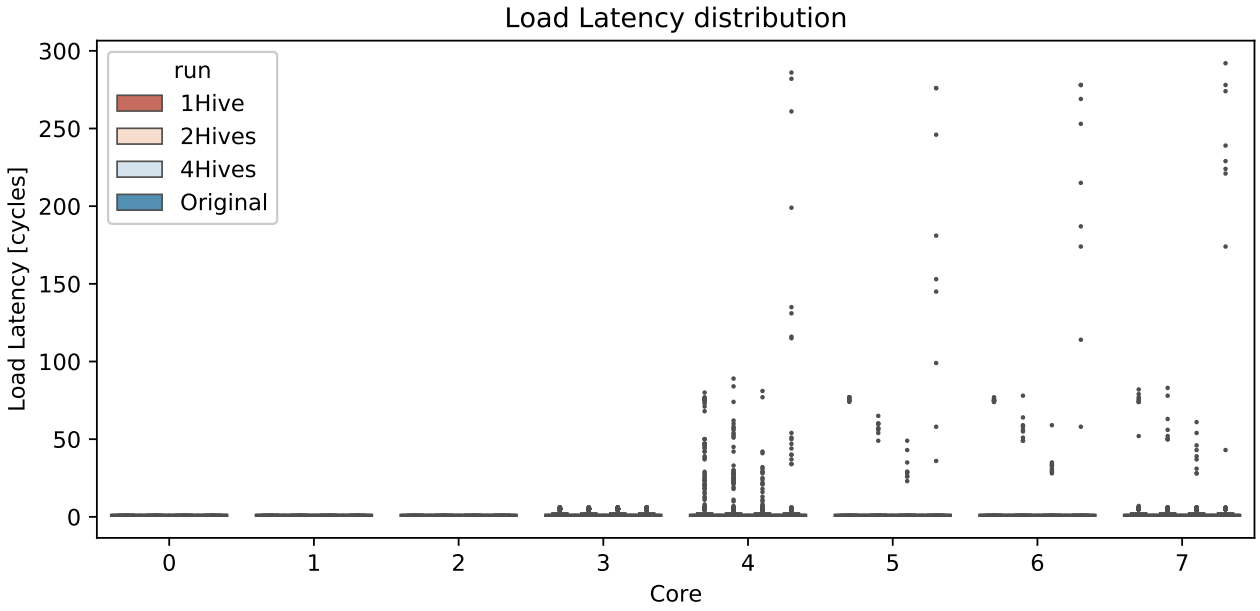
Load Latency

The load latency of the convolution is shown in the table below. Despite the low cycle count for the 90 percentile mark of only 1 cycle, all configurations have an average load latency significantly above 1 and a large standard deviation.

	1Hive	2Hives	4Hives	Original
samples	723843	723843	723843	723843
mean	1.63	1.5	1.39	2.02
std	7.78	5.67	3.89	16.75
min	1	1	1	1
10.00%	1	1	1	1
25.00%	1	1	1	1
50.00%	1	1	1	1
75.00%	1	1	1	1
90.00%	1	1	1	1
max	150	156	152	525

This is easily explained by looking at the figure. While the average bar is at one cycle, there are again outliers that reach almost 300 cycles load latency in the first 8 cores of the system. This is from loading the kernel once at the very beginning. Therefore, there are no more than 9 outliers and since the kernel is stored in tile 0, only the cores outside this tile show this behavior.

Another interesting effect can be seen when looking at core 3, 4, and 7 in this figure. Those are the cores at the boundary of the tile that have to load one twelfth of the values (they calculate four pixels of which one is at the boundary) from the neighboring tile. These loads are the collection of outlier dots, just above the average box, at 5 cycles latency.



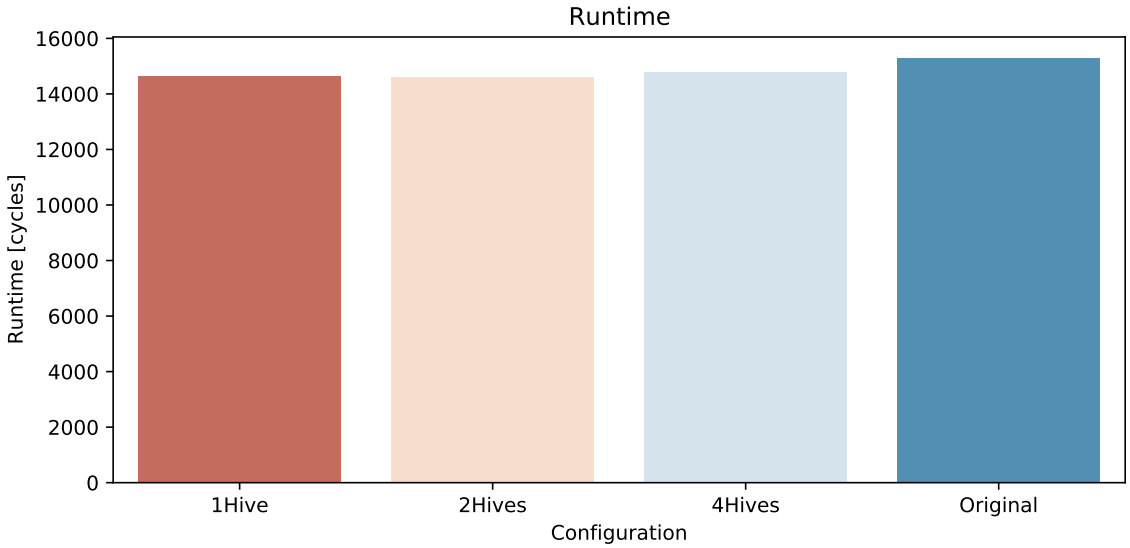
Discrete Cosine Transformation

Finally, the DCT is an application used in the JPEG encoding, where 8x8 pixel blocks are transformed with the 2D DCT to then be quantized. This kernel was implemented to represent this algorithm, and it implements the fast DCT algorithm by Arai, Agui, and Nakajima, often called DCT type-II.

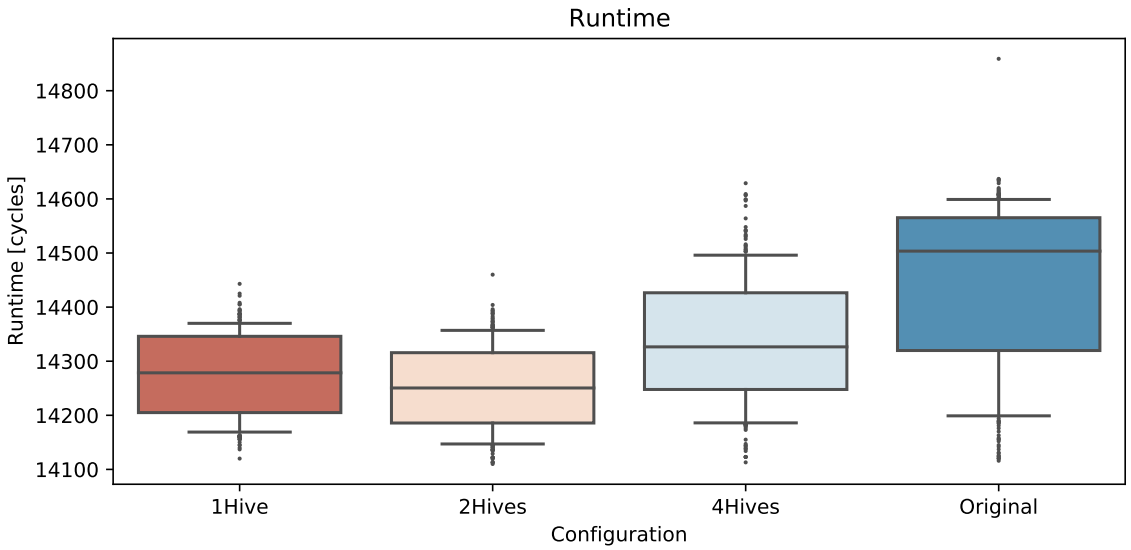
As with the convolution, the algorithm was implemented with MemPools memory map in mind to keep all accesses local. Specifically, each core transforms a 8x8 block of the image in-place, using the stack as temporary buffer. Due to the scrambling logic, the stack is also local to the tile. The performance numbers are from transforming all 8x8 blocks in a 1024x128 pixel image.

Runtime

With all accesses local, the runtime of all four systems is almost the same. In this case, the 1 and 2Hive configurations are the fastest with a 4.5% speedup compared to the Original implementation. The 4Hive version has almost the same runtime and is 3.5% faster than the Original.



The distribution of the runtimes, shown in the figure below, shows the fine difference between the four systems.



Load Latency

The load latency plot shows a very boring average load latency of 1 cycle for the first 8 cores, with no outliers, due to no load crossing a tile boundary.