

# Understanding Performance Cold Spots using pfmon

Emery Davis, 16 June 2007

## Abstract

This paper briefly describes using the pfmon utility on the SiCortex platform. This machine is based on a modified MIPS r5kf core running at 500 Mhz. The core supports double precision FMA instructions at full throughput, giving a theoretical maximum performance of 1 Ghz per core in double precision. Algorithms designed for this platform should scale successfully to many thousands of cores.

We used pfmon to time a 3D convolution routine and observed several performance cold spots. These points of decreased performance arrived at 256, 512, 768 etc columns respectively. (Data in row major order). We used pfmon to analyze a 2D convolution, because it is a proper subset of the 3D case – with a similar memory profile – and executes faster.

pfmon allows us increased understanding of our algorithms, enabling us to find a solution to the points of decreased performance.

## The 2D problem

We began our analysis by using pfmon to count cycles in the 2D convolution. All data is for a 15x15 tap filter, with 380 rows. We vary the number of columns and observe the execution time in cycles, as given in Figure 1. These measurements are done using

```
pfmon -e CPU_CYCLES --trigger-code-start-address=conv2d --trigger-code-stop-address=conv2d.
```

All subsequent measurements were performed similarly, varying the counted event.

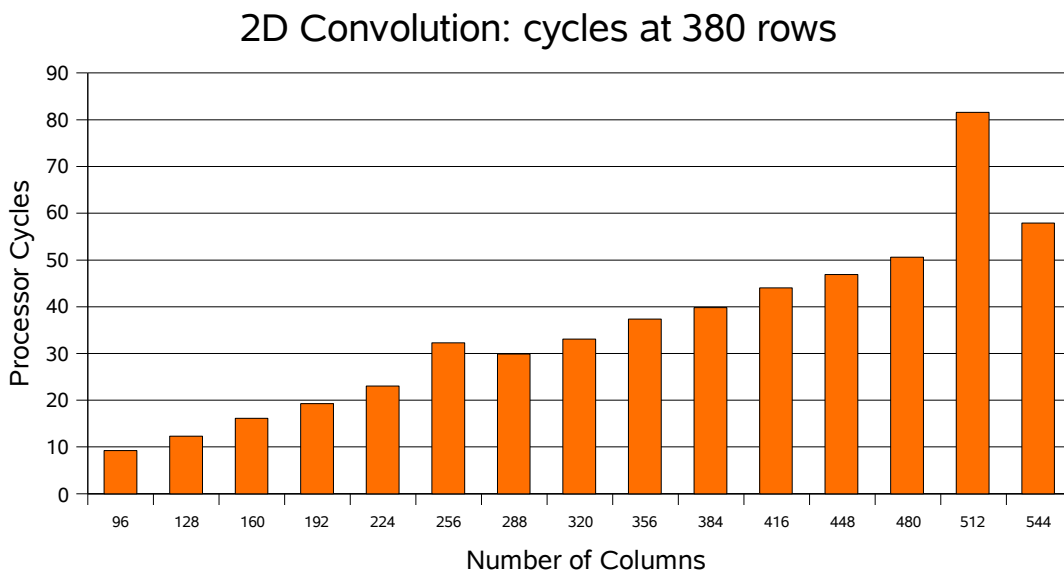


Figure 1: Processor Cycles (millions) for 15x15 tap 2D Convolution, 380 rows by X columns

Although Figure 1 shows the cold spots, these are easier to see when we look at the Mflops at each size. Because the algorithm is tiled, we expect the performance to asymptotically approach a constant figure as the size increases. The Mflops can be derived, for our convolution, from processor cycles by applying the formula:

$$Mflops = 2*tr*tc*(rows-tr-1)*(cols-tc-1)/(cyc/Mhz)$$

with tr, tc each being the number of taps in each direction (15), and rows fixed at 380. Figure 2 clearly shows the cold spots.

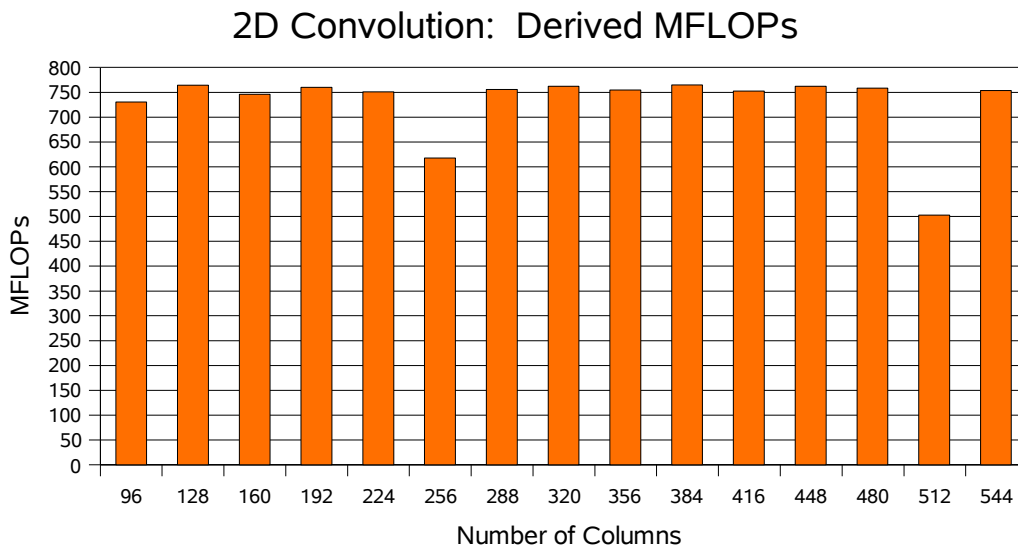


Figure 2. Mflops per size, derived from CPU\_CYCLES

As these cold spots show up just where we are likely to expect the algorithm to be broken up across multiple cores, we need to understand the problem and find a solution.

### Verifying the Algorithm

The first step we take is to verify that the algorithm is indeed working correctly at the problem sizes. Aside from the correctness of results, we can do this by seeing if the number of floating point instructions and cpu load/store instructions looks right. This will teach us whether the tiling algorithm is causing outputs to be calculated multiple times. These events can be measured using CPU\_FLOAT, and the sum of CPU\_LOAD and CPU\_STORE. These measurements are shown in Figure 3. A more evocative view is the ratio between floating point instructions and loads/stores. If this ratio remains constant at all sizes, and the Figure 3 also shows a linear relationship with size, then the algorithm is working correctly. This ratio is shown in Figure 4.

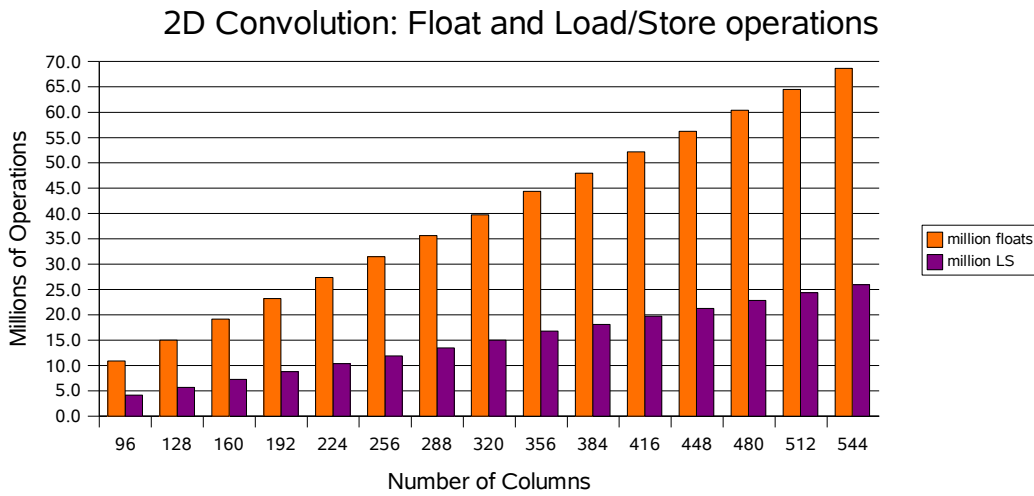


Figure 3. Number of floating point operations, number of load and store operations, in millions

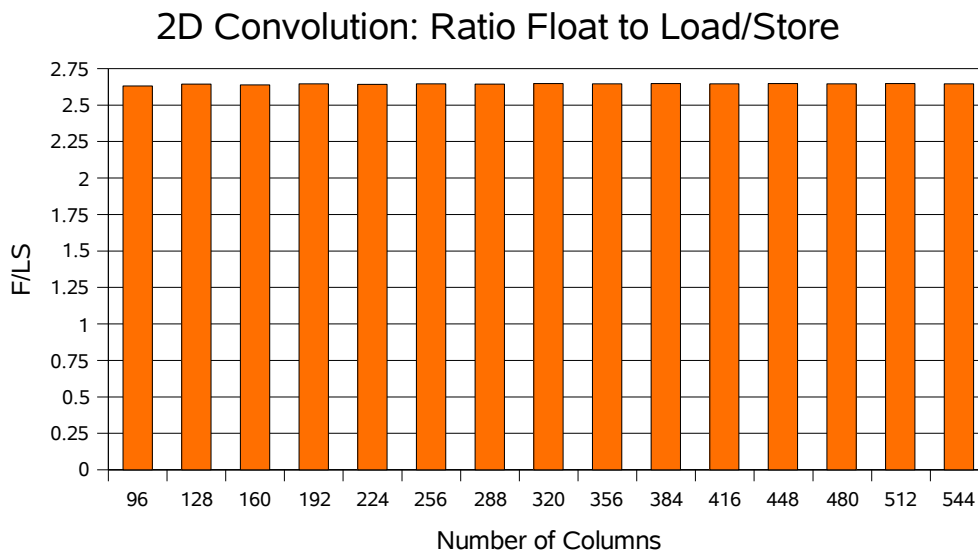


Figure 4. Measured Ratio of Float to Load/Store Instructions

Figures 3 and 4 combined show that the algorithm is working correctly. Figure 4 also bodes well for scalability since it shows that there is a favorable relationship between calculation and the memory footprint.

### Memory Interactions

Next we look at whether there are memory stalls. We can start by looking at how much the processor stalls while waiting for data, with CPU\_MSTALL. This number should be more or less constant per problem size, but as Figure 5 shows, there are a large number of stalls at the trouble spots.

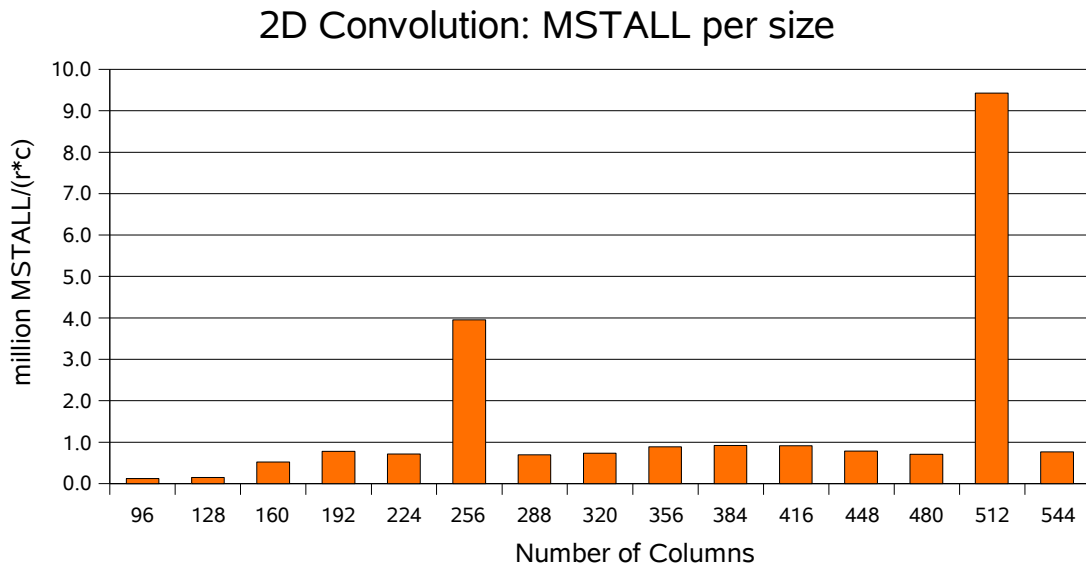


Figure 5. Stalls in the M-stage are not constant per size

This leads to an immediate suspicion of the TLBs. But by using pfmon to count TLB misses and TLB traps, we saw that a TLB thrash could not be the culprit. Indeed the TLB management appears to be reasonably efficient.

However using pfmon to count Dcache misses and evictions (of dirty lines) we can clearly show that there is a thrash situation (sometimes whimsically referred to as a Murphy's Scenario) at the problem spots.

Figure 6 uses knowledge of the 2D Convolution internal tiling mechanism to calculate the rough theoretical Dcache misses, and compares this to the measured value. (The calculation is rough because tile edge effects are not accounted for, and these can contribute significantly). The graph stays fairly flat except at the cold spots.

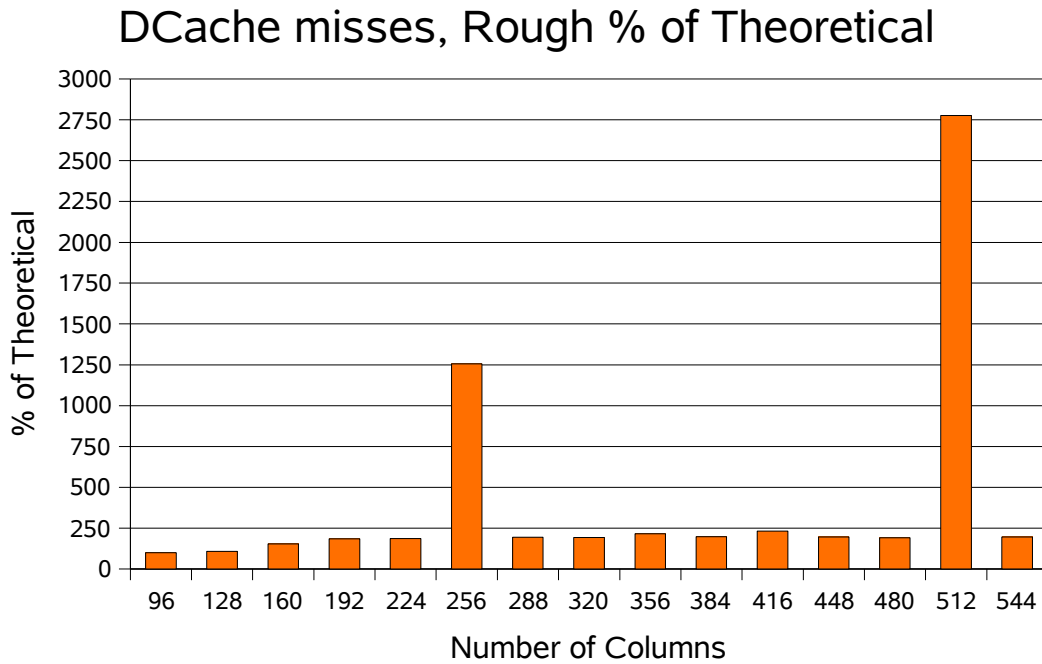


Figure 6. L1 Dcache misses shown as percentage of the rough theoretical minimum

Figure 7 shows the number of measured Dcache evictions per destination element. Since the cache line size is 32 bytes, there are 4 double precision elements per cache line, and a measure of 0.25 represents perfect efficiency. In fact the algorithm is very efficient indeed, approaching that number everywhere but at the cold spots. This is also a good sign for scalability. Still pfmon shows that at 512 columns, this implementation writes each dirty cache line more than 25 times.

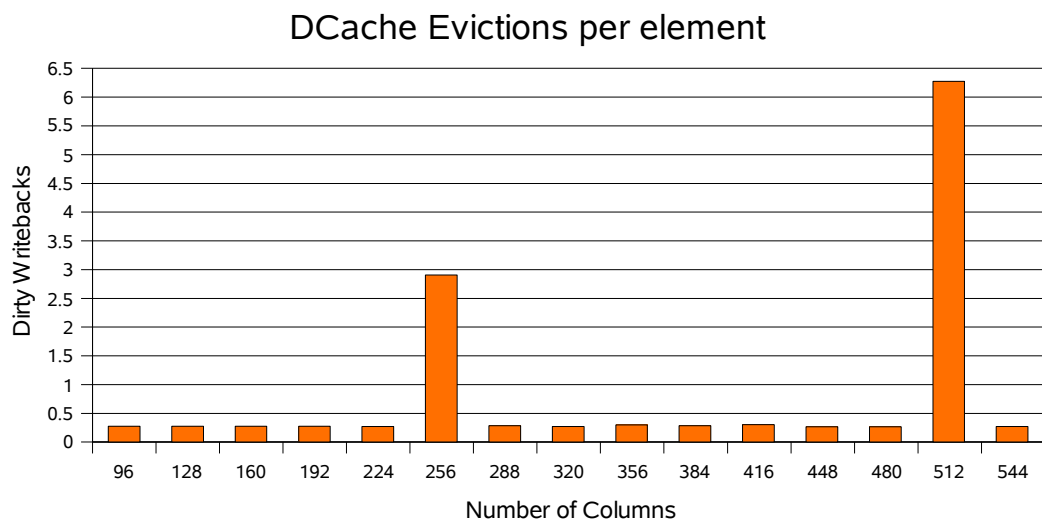


Figure 7. L1 Dcache dirty writebacks per double precision destination element. 0.25 is perfect.

## **Conclusion**

We used pfmon to analyze the performance of our 3D and 2D convolutions. We discovered similar performance cold spots in both algorithms.

pfmon helped us identify a classic L1 cache thrash scenario in these unwieldy codes. By analyzing the 2D convolution, which has an identical memory footprint, we were able to understand the 3D case. pfmon also helped us understand our algorithm and its efficiency and scalability in various areas.

Armed with the knowledge that pfmon teaches us about our code, and remembering that our L1 is 4-way associative, we can easily see how to avoid the Murphy's Scenario in the L1 which occurs when the number of columns is a multiple of 256. We need to simply change the dimensions of the input and output matrices so that the number of columns can be allocated away from the problematic spots. We convolve sub-matrices at the desired sizes. Since the difficulty occurs with a thrash of dirty lines, this is sufficient to smooth out the performance. Then we simply add another parameter to the 2D and 3D convolution APIs. Inside our tiling algorithms we can adjust for the additional pointer offset without any loss of performance.