The Future of Dyalog APL Performance

Elsinore 2017

Marshall Lochbaum

#dyalog17

Improvements in version 16

- dyalog.com/dyalog/dyalog-versions/160/performance.htm
- Many structural functions are much faster
- Enlist and set functions on nested arrays are about twice as fast
- Base encoding/decoding and equals/not-equals scan sped up

“Latency” and Throughput

- Most functions are faster, relative to array size, on large arguments
- A typical APL primitive takes constant time for each element, but also has a small delay (20-100ns) for each overall invocation
- Most of the speedups in 16.0 target throughput, although reshape and transpose have a reduced constant cost. Monad -×÷|○~ and dyad ⌈⌊⍱⍲ are faster on simple scalar arguments.

Optimising for small operations

- Try to run operations on large arguments to reduce function overhead whenever possible
- Most primitives cost the same on small arguments, but operators take around 10x as long
- Simple statements and dfns with few operators will run the fastest when function overhead is a factor
- Make sure your problem is really function overhead before trying to optimise this way!

Optimising for large operations

- Know how fast each function is, and profile to find expensive functions in your application
- Try to fit intermediate results in the smallest datatype possible (e.g. multiply prices by 100 and store as integers instead of using floats)
- Apply functions to smaller arrays when possible (when multiplying by both a matrix and a scalar, multiply the two together first)
- Never optimise without testing!

Instruction set extensions

- Extensions to x86 and other architectures allow the CPU to do some operations much more quickly
- Historically Dyalog has used these in only a few cases, but they are used for more, and more important, operations in versions 16.0 and 17.0

x86 instruction sets used by Dyalog APL

- SSE2 (2001): operations on 128 bits at a time
- SSE4.1 (2007): adds max and min to SSE2
- POPCNT (2008): count the number of bits in a word
- BMI2 (2013): compress and expand for booleans
- AVX2 (2013): operations on 256 bits at a time
- AVX-512 (2018): basically all of APL in hardware

x86 instruction sets used by Dyalog APL

- Version 15.0:
  - **SSE2 for vector-vector +-×÷**
  - POPCNT for +/ and +.f with boolean functions f
  - CRC32 (SSE4.2) for hashing in some cases
  - SSE4.1 for ⌈/ and ⌊/
  - BMI2 for faster 8x8 boolean transpose
- Version 16.0:
  - BMI2 for boolean ,⍉↑/\
- Version 17.0 (so far):
  - **AVX2 for scalar dyadics**
  - BMI2 for scalar replicate boolean (used in index/outer product)

Checking for instruction set support

- The processor provides a list of which instruction sets it supports (CPUID flags)
- Dyalog checks these when it runs to determine which code to use
- On Windows, use CPU-Z to check flags
- On Linux, use grep flags /proc/cpuinfo
- On macOS, use sysctl -n machdep.cpu

Which processors to use?

- For 15.0, instruction support doesn’t matter much—processors too old for SSE4.1 will be very slow anyway
- For 16.0 and 17.0, a Haswell or newer processor is recommended for BMI2 (16.0) and AVX2 (17.0)
- Version 17.0 will make use of AVX-512, which will be supported in all Cannonlake processors if Intel ever decides to release those (early 2018, we hope)

Version 17.0 improvements

- SSE2 or AVX2 will be used for +-×÷⌈⌊<≤=≥>≠ when available
- Integer comparisons are 6-8x faster with SSE2 and 10-16x faster with AVX2 (about 2x and 4x for floats)
- Tolerant comparison with a scalar is converted to intolerant comparison and is 5x or 10x faster
- Altivec (for POWER architecture) and NEON (for ARM) will most likely be supported, as well as AVX-512

Ways to move forward

- Improve the performance of primitives on large arguments (in progress)
- Reduce function overhead
  - Optimised parsing; pre-parsing
  - Better allocation and garbage collection
- JIT compilation (generate machine code at runtime)
  - Perform multiple operations in one pass
  - Compile simple functions if they are used a lot