# Throughput studies for data analysis applications

## Introduction

High-speed data throughput is a requirement for most high energy physics software. Whether improving old software or developing new software, we must keep an eye on this aspect of its performance.

The metrics described in this section were measured as part of the development process for two new software products: root4j[1], which provides access to ROOT files in Java (and therefore Apache Spark), and Femtocode[2], which is a query system, intended to produce plots from large (petabyte) datasets in real time.

Throughput bottlenecks can appear in three places. The first is loading into memory, which includes reading from physical disks and the network, deserialization, and decompression. The second is loading from memory into the processing unit, which is often much faster than the first and doesn't involve any transformations. The third is the computation itself. All three are relevant because caches can hide the effect of a slow load-into-memory or a slow load-from-memory.

For instance, root4j is called by spark-root[3] to load data from ROOT files into Spark as a `DataFrame`. After the first request (and until subsequent loads cause it to spill to disk), Spark caches the `DataFrame`, distributing its contents across the DRAM of an entire cluster. Similarly, Femtocode is designed with the expectation that physicist users want to re-plot the same data several times in a row, tweaking aspects of the plot, so it also has a distributed, in-memory cache. In the first request, the disk or network bottleneck is relevant; afterward, the memory-to-processor bandwidth is relevant. The calculation itself may or may not overwhelm both of these, depending on what is being calculated. Simple histogram filling (the most common use-case for both Spark and Femtocode) requires much less computation time than load time.

We will therefore focus on the first two bottlenecks only: load-to-memory and memory-to-processor. Furthermore, each of these depend strongly on the choice of hardware, so our metrics are *comparisons* of one software solution to another, both on the same hardware. The results should be understood as relative, except where specific hardware is described.

Code for all of these tests can be found in the `diana-hep/femtocode-metrics` GitHub repository[4].

## ROOT file reading: Java vs optimized C++

The root4j library is a pure-Java implementation of ROOT file reading. This has technical advantages over alternatives that would call the standard C++ implementation of ROOT from Java (JNI, UNIX pipes, or sockets): root4j is easy to install as a JAR from Maven

[1] `https://github.com/diana-hep/root4j`
[2] `https://github.com/diana-hep/femtocode`
[3] `https://github.com/diana-hep/spark-root`
[4] `https://github.com/diana-hep/femtocode-metrics`

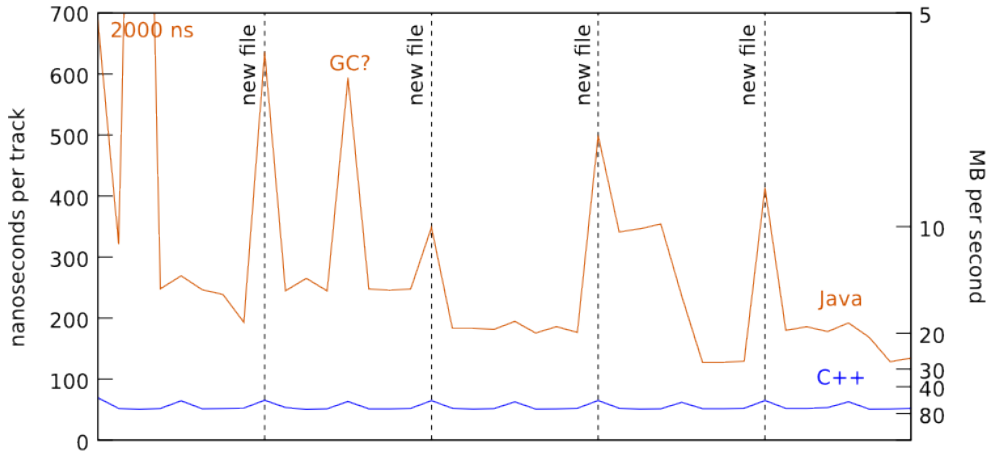


Figure 1: Rate of reading track $\chi^2$ in root4j and C++ ROOT as a function of time. (Samples on the horizontal axis are every 100,000 tracks and "MB" refers to *decompressed* data.)

Central and is free of several classes of bugs, including segmentation faults and failures in interprocess communication. It does, however, have the potential to be slower than the C++ implementation, since Java is a virtual machine with garbage collection.

Early in the development process, the read performance of root4j and C++ ROOT were compared to get a sense of this cost. Track $\chi^2$ values for all tracks (hundreds per event) were read from a CMS public dataset (AOD). This study focuses on a single attribute of a variable-length collection of objects within the event, so it relies heavily on the fact that these objects are laid out in a columnar fashion in the ROOT files.

Figure 1 shows the reading rate as nanoseconds per track (left axis) and, equivalently, as megabytes per second for the *decompressed* data (right axis), measured every 100,000 tracks. Measuring this rate as a function of time is important because Java does HotSpot optimization at the beginning of the run, sacrificing start-up latency for long-term throughput.

These files were also pre-read before measuring any data, so that they would be in Linux's virtual memory, effectively removing physical disk access from the measurement and ensuring that neither software product is unfairly discounted by this large bottleneck, simply due to being run first.

Both C++ and Java readers are deserializing and decompressing (deflate) the same bytes. The C++ ROOT version is 6.08/04 (pre-built binary for Ubuntu Linux). The root4j version is 0.1-pre2; spark-root and Spark were *not* involved in this test (which would complicate matters, due to caching). Both C++ and Java were executed in a single process, single thread.

In the plot, we see that standard C++ ROOT is consistently faster than root4j. We also see that, unlike C++, root4j starts slow and accelerates. We see the HotSpot optimization phase as a spike (temporary slow-down) at the beginning of reading, a spike when opening each file, and a single spike in the middle of one of the files. Spikes like these appear at

different times in different runs, so they are likely garbage collector pauses.

However, the asymptotic speed of root4j, which is all that matters for large files and large datasets, is about 4 times slower than C++. Depending on the user's intended purpose, this cost may be worthwhile, since it opens direct access to all the "big data" and machine learning tools that have been developed for the Java platform.

More recent performance benchmarks are in development, using Spark's built-in performance counters[5].

## ROOT file reading: CMSSW vs optimized C++

The second test of ROOT file reading was motivated by Femtocode, which will take advantage of ROOT's columnar data representation on disk to perform calculations in the same format in memory, without reconstructing objects. This is in contrast to a traditional framework for analyzing physics data, such as CMSSW.

CMSSW uses ROOT to build instances of C++ classes, and physicist users write C++ code to interact with these objects. Since the data in CMS data files are are mostly arbitrary-length collections of record structures (e.g. `std::vector<Muon>`), these objects usually aren't contiguous in memory (to allow appending). And since the user code could access any attribute from these classes, all columns associated with the whole class must be loaded, even if the user actually accesses only one or two.

Femtocode, on the other hand, is a high level language that gets JIT-compiled to machine code. Part of this compilation process determines which attributes are needed and only loads those. The code is also compiled in such a way that it runs on contiguous arrays of data loaded directly from the ROOT file without constructing objects first.

Femtocode's strategy is similar to `TTree::Draw`, a special-purpose function for aggregating data into histograms (and immediately drawing them). Femtocode and `TTree::Draw` differ in that Femtocode is JIT-compiled and more general, but the similarities are close enough that a comparison to `TTree::Draw` is interesting.

Table 1 compares the file-reading performance of CMSSW, `TTree::Draw`, and a ROOT-reading routine designed for Femtocode. This specialized reader uses the ROOT libraries, but avoids function calls that would reconstruct C++ objects from the data, as an ordinary user would want. (This routine was also used in the previous subsection, as the C++ comparison to Java.) We thank Philippe Canal for his help in writing it.

The CMSSW reader is a custom EDAnalyzer in version `8_0_25` of CMSSW, following current best-practices for file-reading (`EDGetTokenT<>` and `consumes<>` to inform CMSSW of which particles will be loaded). The goal was to extract the $p_T$ (or $E_T$) value from a variety of particles (individually, in separate runs). CMSSW is at a disadvantage for this kind of test because it must load all attributes of the selected particles. Most particle types have 95–231 such attributes (called "branches" in ROOT). For this task, CMSSW is reading about a hundred times more data than is necessary. This roughly correlates with its performance.

[5] `https://github.com/diana-hep/spark-root/blob/master/md/PerformanceBenchmarksPublicDS.md`

Table 1: File-reading rates in events per millisecond per process (kHz per process). In each case, the goal was only to access the $p_T$ (or $E_T$) attribute of each particle, though CMSSW loads all attributes associated with the particle.

| particle | # of particles per event (avg) | # of attributes ("branches") | CMSSW EDAnalyzer | `TTree::Draw()` | Femtocode ROOT reader |
|---|---|---|---|---|---|
| photon | 2.9 | 205 | 1.14 kHz | 435 kHz | 769 kHz |
| electron | 2.5 | 231 | 1.02 | 417 | 833 |
| muon | 2.7 | 192 | 1.02 | 16.5 | 770 |
| tau | 6.3 | 88 | 1.55 | 244 | 417 |
| jet | 16.7 | 95 | 1.15 | 123 | 182 |
| AK8 jet | 1.8 | 95 | 2.10 | 556 | 1000 |

(Collaboration frameworks like CMSSW were also designed for event reconstruction, which requires many more attributes per particle. CMSSW's file reading is closer to optimal for this important task.)

The `TTree::Draw` function was designed to plot small numbers of branches, so it does not load unnecessary attributes. Similarly, Femtocode is primarily intended for plotting, so its specialized reader calls a similar set of functions as `TTree::Draw`. In most cases, their results are within a factor of 2 of each other, though the muon comparison is a factor of 50. (This has been reported to the ROOT team; they are investigating.) The factor of 2 can be expected, since `TTree::Draw` is additionally invoking an interpreter to generate a plot, unlike the CMSSW and Femtocode readers.

All tests were single-process, single-thread, reading the same input file that was pre-loaded into Linux virtual memory by previous reads. The `TTree::Draw` tests were rotated in order and the first call was discarded, since `TTree::Draw` does some initialization the first time it is called.

In addition to comparing these three ways of calling ROOT routines to read ROOT files, we tested the time required to read a flat array of the same size from a Numpy file. Numpy files have a very simple format: the contiguous array is compressed (deflate, same as ROOT) and written to disk as-is (with a small header). The Femtocode ROOT reader was only 5% slower than reading the equivalent data from Numpy files, which suggests that the procedure is near optimal.

## Memory bandwidth: conventional vs GPU vs KNL

Because of the high cost of loading data into memory, Spark and Femtocode cache data for subsequent computations. This fits the work-pattern of most data analyses, in which a computation is repeated with subtle variations as part of exploratory data analysis, systematics studies, or machine learning iterations. Assuming that the whole dataset can be loaded

into memory, all but the first pass benefits from the cache, often by orders of magnitude (100–1000×).

At these high rates, the bottleneck shifts from disk or network access to DRAM memory access. To similate a low-arithmetic intensity plotting problem, we performed a simple addition by a constant on 10 billion double-precision floating point numbers. The motivation for this order of magnitude comes from an assumption of 100 million events (e.g. all $t\bar{t}$ data collected by CMS since its inception) accessing about 100 values per event (e.g. hundreds of tracks or combinations of a few different attributes). In total, this amounts to 80 GB of in-memory data to be evaluated while the user waits.

The throughput from DRAM to CPU back to DRAM is highly affected by compiler optimizations, which target this specific problem. In a single process, single thread, the following times are required to either add values to the array in-place or add them and store them elsewhere (simulating an immutable operation):

| | with `-O0` | with `-O1` | with `-O2` | with `-O3` |
|---|---|---|---|---|
| in-place | 87.8 sec | 27.2 sec | 27.1 sec | 25.6 sec |
| immutable | 91.2 sec | 38.7 sec | 39.2 sec | 39.2 sec |

Without compiler optimizations, the in-place and immutable times are nearly the same, even though the first accesses one memory location and the second accesses two. With optimizations, both operations are faster, but the one with fewer memory accesses is more so. These optimizations involve such things as loop unrolling, which hide the latency of memory access and encourage CPU prefetching. Automatic vectorization, which is a potential optimization unrelated to memory access, is only applied in `-O3`, and we don't see that effect here.

Plotting problems are embarrassingly parallel and can be horizontally scaled to a nearly arbitrary degree, dividing this 25 or 40 seconds by a large factor of $N$. However, we also want to see if specialized hardware could be beneficial. We performed the same simple operation on several different GPUs and a Knight's Landing (KNL) chip.

The GPU results are summarized in Table 2. For comparison, the single-threaded CPU rates are 0.39 GHz (in-place) and 0.26 GHz (immutable) while fully vectorized GPU rates range from 4.0 GHz to 56.9 GHz for laptop/gaming GPUs (GeForce GTX) to high-end supercomputing GPUs (Tesla P100).

Another thing to notice is that memory-to-CPU and memory-to-GPU bandwidths are similar to each other for most architectures: the biggest deviation is 60% slower for copying to the GPU on one machine. In designing software, we can consider the GPU to be just as "distant" from memory as the CPU.

Once the data are on the GPU (as "global" memory, not "unified"), calculations are much faster. In particular, copying data within the GPU has approximately the same cost as a calculation. The following hierarchy emerges:

$$\text{DRAM to CPU} \sim \text{to GPU} \ll \text{GPU global memory to calculation} \sim \text{to copy.}$$

Table 2: Data throughput rates in GHz (billions of 32-bit floating point values per second) for various GPUs and architectures. The first line is a data copy that does not involve the GPU, for comparison with the ones that do. "Unified" memory (also known as "pinned") are pointers that can be accessed by both host (CPU) and device (GPU), but with a price in performance, as we see below. "Global" memory is bound to the GPU and requires an explicit copy-to-device.

| | GeForce GTX 660M | GRID K520 (AWS) | Tesla K20m (Princeton) | Tesla K40c (CERN) | Tesla K20Xm (CERN) | Tesla P100-SXM2 (CERN) |
|---|---|---|---|---|---|---|
| Data transfers: | | | | | | |
| DRAM to *CPU* | 1.8 | 1.5 | 0.82 | 0.77 | 0.77 | 3.5 |
| DRAM to GPU | 1.7 | 1.0 | 0.87 | 0.82 | 0.51 | 3.4 |
| GPU to DRAM | 1.7 | 1.0 | 0.51 | 0.49 | 0.76 | 1.8 |
| CPU write unified | 2.3 | 1.5 | 0.87 | 0.82 | 0.50 | 3.6 |
| CPU read unified | 2.3 | 1.5 | 0.50 | 0.48 | 0.46 | 2.1 |
| within GPU | 7.0 | 15.2 | 18.9 | 23.6 | 23.6 | 56.9 |
| Calculations: | | | | | | |
| in-place global | 4.0 | 13.7 | 16.9 | 21.4 | 21.1 | 56.9 |
| immutable global | 4.1 | 14.3 | 16.9 | 21.5 | 21.2 | 56.9 |
| in-place unified | 2.6 | 2.2 | 1.4 | 2.6 | 1.6 | 7.1 |
| immutable unified | 2.9 | 2.4 | 1.5 | 2.9 | 2.9 | 8.3 |

For designing software, this (a) that there's no memory bandwidth advantage to performing operations in-place, rather than immutably, and (b) that the GPU is advantageous only if several operations can be performed before copying the data back to normal memory. We may even consider keeping an input data cache in GPU global memory as well as DRAM. That cache could be implemented as a circular buffer that copies the most recently used data to the head of the buffer with every iteration, since data copies within the GPU are essentially free.

Finally, it's worth pointing out that the "unified" memory model provided by modern CUDA disguises but does not eliminate the cost of the transfer. In the unified model, host (CPU) code and device (GPU) code use the same pointers without an explicit copy, but GPU calculations on unified memory run at a much lower rate than explicit global memory. Presumably, it is copying on demand.

(GPU "shared" memory is even faster than global, but much too small for our datasets.)

Finally, this comparision of single-threaded CPU rates (0.26 GHz) to GPU rates (4–57 GHz) is unfair because modern architectures support multiple CPU cores, often many more (8–64) than the number of GPUs (1–4). We should run CPU processes in parallel, too.

Given the embarrassingly parallel nature of the problem, we would naïvely expect perfect scaling with the number of worker threads up to the number of cores. On conventional and even NUMA architectures, however, we hit a limit of about 2 GHz. Since these processes do not interact, we conjectured that the limiting factor was contention on the memory bus, since all workers are doing little more than requesting data. Even in NUMA architectures, half of the CPUs share a single memory bus.

To prove this conjecture, we obtained access to a Knight's Landing (KNL) architecture at Princeton. KNL has one feature in common with GPUs: small groups of CPU cores have a local pool of memory called MCDRAM, which is a rough equivalent of a GPU's global memory (in size, granularity, and distance from the processing unit). If the 2 GHz scaling limit is due to memory bus contention, moving data to a KNL's MCDRAM would reduce that contention and allow higher scaling.

Figure 2 shows that the conjecture was right: switching from a conventional architecture to KNL does not, by itself, break the 2 GHz limit, but allocating memory on the MCDRAM does. In fact, scaling on a single KNL machine reaches 8 GHz with 128 processes, twice the number of physical cores (though it drops precipitously beyond that).

All of these studies point to the same general conclusion, that novel hardware like GPUs and KNL are advantageous for our project not because of superior computational ability but because they have local stores of memory, close to the computing elements. Our gains in throughput depend exclusively on how well we exploit the device memory.

While the computational problem Femtocode poses is embarrassingly parallel, not all parts of it are vectorizable. To perform the equivalent of object oriented programming on data represented as flat arrays, some operations need to correlate values in arrays of different lengths. These operations, known in Femtocode as "explode" and "combine", are not a good match to the GPU's vectorized programming model. Explode operations must interpret an array sequentially from start to end, and combine operations must either be sequential or require $\log_2 N$ vectorized passes over a dataset of $N$ elements. These operations are best suited for a sequential processor like KNL or a conventional CPU, with GPUs used only for the "flat" operations that correlate equal-length arrays, element by element.

Finally, memory bus contention could also be defeated by purchasing a large number of few-core machines. This is another way to make memory more local: by physically separating them on the rack. We have not investigated how the price of four 16-core servers compares to the price of one 64-core KNL, but this question is not a blocker for writing software.

## Conclusions

In this section, we described several quantitative analyses performed as part of designing software for data analysis applications. We confirmed the disk access $\ll$ memory access $\ll$ processing hierarchy, with some useful details.

C++ ROOT remains the fastest way to read ROOT files, though it depends on how this library is used. In our application of extracting only one or a few attributes of each particle for plotting, we can pull data from the file at a rate of about a MHz per thread. Java is 4

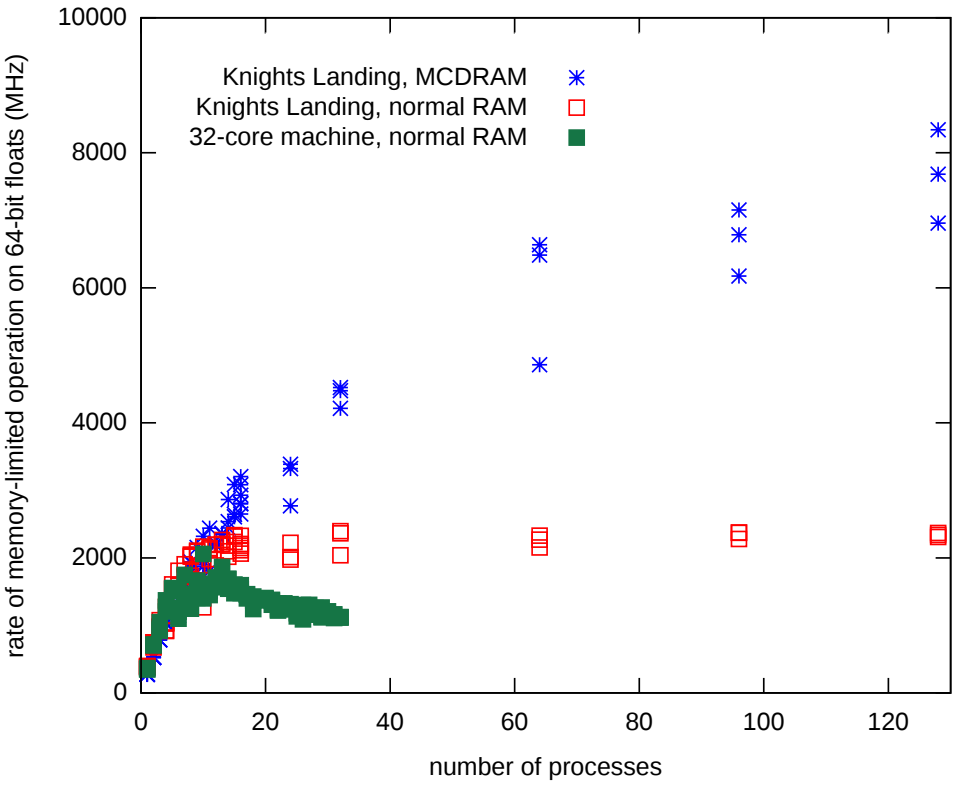


Figure 2: Memory bandwidth-limited scaling on a conventional 32-core machine, a Knight's Landing machine with input data in globally shared DRAM and local MCDRAM, respectively.

times slower, but it brings the benefit of providing access to "big data" and machine learning tools like Apache Spark.

Using (one might say "misusing") a collaboration's framework like CMSSW for the purpose of extracting only one attribute per particle is three orders of magnitude slower: about a kHz. This is primarily because CMSSW loads all attributes per particle (which are needed for the usual application of event reconstruction) but partly because CMSSW creates and destroys C++ objects in non-contiguous memory for the convenience of the user.

The MHz rate of `TTree::Draw` and the Femtocode ROOT reader is nearly identical to the rate of reading numbers from a Numpy file, which has minimal overhead. Therefore, this MHz rate is probably close to the maximum possible.

If the input data are in a memory cache, however, rates of a quarter GHz per thread are possible (almost three orders of magnitude faster). Adding non-interacting threads allows us to scale up to 2 GHz before running into memory bandwidth limits of a conventional memory bus. However, using local memory attached to a KNL or GPU device breaks through this limit, allowing up to 8 GHz on a KNL and 57 GHz on a high-end GPU (though not all parts of the problem can be ported to the GPU).

Amusingly, these event-processing rates even exceed the collision rate in the LHC, though this numerology is coincidental. Collision events are highly filtered, and we need a system to process years of collected data in seconds if it is to serve for real-time data analysis.