

Large Fast Fourier Transforms with FFTW 3.3 on Terabyte-RAM NUMA Servers

Andrey Vladimirov
Stanford University
for Colfax International

February 2, 2012

Abstract

This paper presents the results of a Fast Fourier Transform (FFT) benchmark of the FFTW 3.3 library on Colfax's 4-CPU, large memory servers. Unlike other published benchmarks of this library, we study two distinct cases of FFT usage: sequential and concurrent computation of multithreaded transforms. In addition, this paper provides results for very large (up to $N = 2^{31}$) and massively parallel (up to 80 threads) shared memory transforms, which have not yet been reported elsewhere.

The FFT calculation is discussed: parallelization techniques and hardware-specific implementations; motivation for a specific astrophysical research is given. Results presented here include: dependence of performance on the transform size and on the number of threads, memory usage of multithreaded 1D FFTs, estimates of the FFT planning time. The paper shows how to optimize the performance of concurrent independent calculations on these large memory systems by setting an efficient NUMA policy. This policy partitions the machine's resources, reducing the average memory latency. Such optimization is not specific to FFT algorithms, and can be useful for a variety of applications in large memory NUMA systems. Our conclusion is that the FFTW implementation of multithreaded one-dimensional FFTs scales very well with the number of threads for large transforms, but worse for small transforms. Having a large amount of shared memory in the system is beneficial for the performance of large concurrent FFTs, as it allows to reduce instruction-level parallelism in favor of task-level parallelism.

Contents

1	Introduction	2
1.1	Fast Fourier Transform Parallelization	2
1.2	Sequential vs Concurrent Multithreaded Transforms in Shared Memory	3
1.3	Motivation of research: large 1D FFTs and gamma-ray pulsars	4
2	System configuration	4
3	Results	5
3.1	Performance of Sequential and Concurrent Multithreaded FFTs	5
3.2	Memory requirements of multithreaded 1D FFTs	5
3.3	Optimizing NUMA policy to partition resources	9
3.4	FFTW planning time in the MEASURE mode	9
4	Conclusions	11

Colfax International (<http://www.colfax-intl.com/>) is a leading provider of innovative and expertly engineered workstations, servers, clusters, storage, and personal supercomputing solutions. Colfax International is uniquely positioned to offer the broadest spectrum of high performance computing solutions, all of them completely customizable to meet your needs - far beyond anything you can get from any other name brand. Ready-to-go Colfax HPC solutions deliver significant price/performance advantages, and increased IT agility, that accelerates your business and research outcomes. Colfax International's extensive customer base includes Fortune 1000 companies, educational institutions, and government agencies. Founded in 1987, Colfax International is based in Sunnyvale, California and is privately held.

1 Introduction

1.1 Fast Fourier Transform Parallelization

A ubiquitous mathematical operation in data processing and simulations, the Fourier Transform, is usually numerically performed using one of the Fast Fourier Transform (FFT) algorithms, such as the common Cooley-Tukey method. The FFT algorithm needs to be parallelized in applications that require high performance and/or a large amount of operating memory. The parallelization of one-dimensional FFTs is different from multi-dimensional FFTs, and distributed-memory parallelization is harder than shared-memory parallelization. The following paragraph outlines two important cases:

- *Case 1.* Shared memory parallelization is possible if the problem fits in the operating memory of a node. In this case,
 - a) multi-dimensional FFTs are parallelized by dividing them into a number of one-dimensional FFTs. These one-dimensional transforms are distributed across available cores, where they are computed serially;
 - b) one-dimensional FFTs are parallelized by distributing the calls of the recursive Cooley-Tukey algorithm across cores.

Parallelization in shared-memory architectures is a well studied subject, and efficient implementations exist for CPU-based, as well as GPU-based parallel FFT calculations. Popular solutions for serial and shared-memory parallel FFT on CPUs include [the FFTW library](#)¹ and [the Intel MKL](#)². On GPUs, [the CUFFT library](#)³ for CUDA-capable hardware provides good performance. For OpenCL, only sample code exists (e.g., project [OpenCL.FFT](#)⁴ based on research by Volkov and Kazian⁵), along with white papers and presentations of independent groups⁶.

- *Case 2.* Distributed memory parallelization is necessary when the problem does not fit in the operating memory of a node. This case is much harder to parallelize, because FFT algorithms, in general, require all-to-all communication.
 - a) multi-dimensional parallel transforms with distributed memory are well studied, and efficient MPI-based implementations exist for CPU clusters (e.g., in the FFTW library and the Intel MKL) and GPU clusters (see, e.g., project [DiGPUFFT](#)⁷; and research by [Chen et al. 2010](#)⁸).
 - b) one-dimensional transforms in distributed memory systems are severely limited by interprocessor communication speed and are, in general, less efficient than in-core transforms. For CPU cluster transform tests see, e.g., [this paper on the Intel MKL benchmark](#)⁹). At the same time, there seem to be no efficient implementations out there for *one-dimensional* distributed-memory FFTs on GPU clusters.

This paper demonstrates results for Case 1a, shared-memory 1D transforms on CPUs, and covers the relatively unexplored area of large transforms. ‘Large’ in this study means that the transform exceed the size of the L2 CPU cache in the system, but one or several transforms fit in the operating memory of the system (up to $N = 2^{31}$ in this research).

¹‘Fastest Fourier Transform in the West’, a portable open source library: <http://www.fftw.org/>

²Intel Math Kernel Library: <http://software.intel.com/en-us/articles/intel-mkl/>

³<http://developer.nvidia.com/cufft>

⁴http://developer.apple.com/library/mac/#samplecode/OpenCL_FFT/Introduction/Intro.html

⁵http://www.cs.berkeley.edu/~kubitron/courses/cs258-S08/projects/reports/project6_report.pdf

⁶see, for example, <http://developer.amd.com/documentation/articles/pages/OpenCLOptimizationCaseStudy-PartI.aspx> and http://developer.amd.com/afds/assets/presentations/2913_3_final.pdf

⁷<http://code.google.com/p/digufft/>

⁸<http://dx.doi.org/10.1145/1810085.1810128> or <http://sei.pku.edu.cn/~cyf/ics10.pdf>

⁹<http://software.intel.com/en-us/articles/mkl-fft-performance-using-local-and-distributed-implementation/>

1.2 Sequential vs Concurrent Multithreaded Transforms in Shared Memory

In most cases, applications need to compute multiple FFTs in one of the following situations.

Sequential Multithreaded Computations: If data dependency between transforms exists, or if only one transform fits in the operating memory, then only one FFT process will run on the system at any given time. It may be a multithreaded process, but there will be only one. In this case, the optimization strategy is to find a regime (the number of threads, Q), in which one transform performs the fastest. Intuitively, for large transforms, using the maximum number of threads should yield the best results in this case.

Concurrent Multithreaded Computations: When more than one FFT fits in memory, and the transforms are independent from one another, they can be computed concurrently. Intuitively, the optimization strategy in this case should be the opposite to the sequential case. That is, one should use as few threads per process, Q , as possible, but run a large number of concurrent processes, P . Ideally, serial transforms should be used ($Q = 1$), if the machine's RAM can fit as many transforms as there are thread contexts in the system.

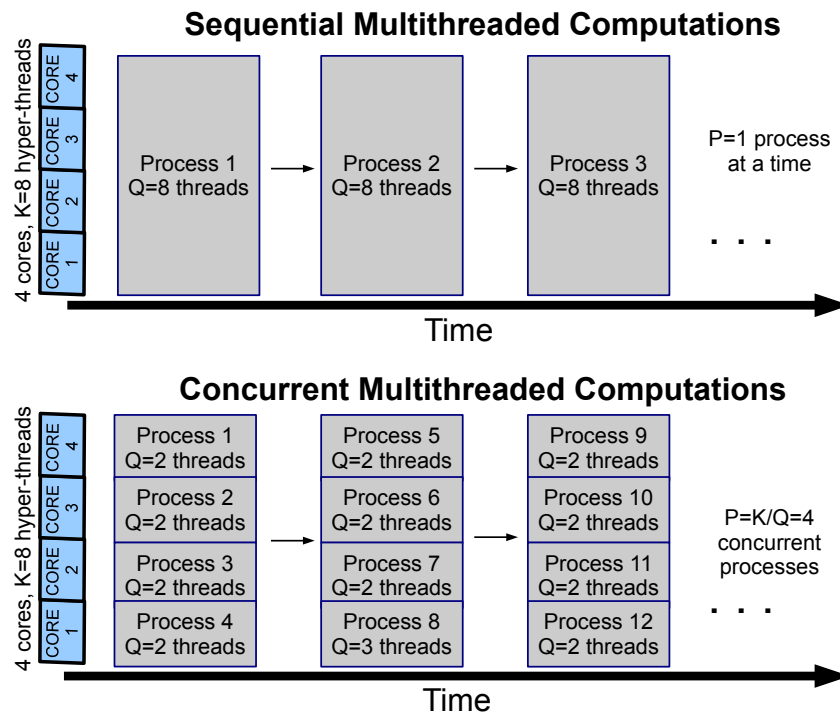


Figure 1: *Sequential calculations* are necessary when inter-process dependency exists, or only $P = 1$ process fits in memory. Large number of threads Q typically benefits sequential calculations, as long as the problem size is large enough to justify the overhead of multithreading.

Concurrent calculations are possible when more than one process fits in memory, and processes are independent of one another. Each of the P concurrent processes may be multi-threaded with Q threads. A system with K hardware threads is fully utilized when $P \times Q = K$. For concurrent calculations, low Q and correspondingly large $P = K/Q$ should benefit performance. Ideally, $Q = 1$ and $P = K$ should be used, if the amount of available RAM permits.

Typically, benchmarks of FFTs focus on sequential transforms situation, with one process running on the system. However, when one's task is to compute many independent transforms, these results are irrelevant, because in concurrent transforms, the system is under full load, and resources are utilized differently. Therefore, a separate benchmark must be made for concurrent transforms.

Benchmarks for both sequential and concurrent multithreaded calculations are presented in this paper; however, the concurrent calculation result is applicable to the astrophysical calculation that motivates this research (see Section 1.3).

1.3 Motivation of research: large 1D FFTs and gamma-ray pulsars

This study was motivated by an astrophysical project, blind search for short-period, radio-quiet gamma-ray pulsars in the data of the [Fermi Gamma-Ray Space Telescope](#). The blind search method, outlined in a paper by [Atwood et al., 2006](#)¹⁰, relies on FFT to detect periodicity in the differences between photon arrival times. Details of the project will be discussed elsewhere; however, an important aspect of the problem is that for each candidate source, it requires a calculation of a large number (10^5 - 10^6) of one-dimensional Fourier transforms ranging from $N = 10^{29}$ to 10^{33} in size.

In single precision, $N = 10^{33}$ translates into 32 GB of data in each transform, which exceeds the onboard memory capacity of today's frontline GPUs. Due to lack of efficient *distributed-memory* implementations of 1D FFTs on GPU clusters (see Section 1.1), the calculation must be run on CPU-based systems.

Even for CPUs, such large arrays pose a significant problem, as blades with more than 32 GB of RAM are still hard to come by in many of today's HPC clusters available to the academia. This is why we turned to Colfax's large memory servers featuring up to 1 TB of RAM and 4 Intel Xeon CPUs in a shared-memory system (see Section 2). These machines allow us to do the transforms in shared memory, and even run several concurrent FFT calculations.

One should note that the lack of efficient distributed-memory implementations of 1D FFTs is at least partially due to the fact that this is not a common operation. Indeed, if the transformed data are a signal evolving in time, then the accuracy of timing should be less than one part in N . For example, if $N = 2^{33}$, the timing accuracy should be better than $2^{-33} \approx 10^{-10}$. While such accuracy is not beyond reach, it remains the domain of high precision scientific experiments rather than commonplace industrial operations. Another interesting indication of the unusually large size of these transforms is that the FFTW library has to be used with special 64-bit interface functions in order to specify FFTs of size $N = 2^{31}$ and greater. The documentation of the FFTW library candidly notes¹¹:

We expect that few users will require transforms larger than this, but, for those who do, we provide a 64-bit version of the guru interface...

That said, this study enters the uncharted territory of very large 1D FFT transforms in shared memory.

2 System configuration

The hardware system used in this benchmark is one of Colfax's [large memory servers](#)¹² with 1 TB of RAM shared between four [Intel Xeon E7-4870 Westmere CPU](#)¹³. Each CPU has 10 cores with two-way hyper-threading (a total of 80 hyper-threads in the system). [An earlier Colfax Research publication](#)¹⁴ provides more information about these machines.

Access to the RAM in these machines is non-uniform in the sense that the memory *local* to the CPU accessing it has lower latency than *remote* memory, which needs to be fetched via a bus. This configuration is known as the [NUMA \(Non-Uniform Memory Access\) architecture](#)¹⁵. The complex memory hierarchy is hidden from the user. That is, from the application's perspective, the machine appears to have a 1 TB of operating memory, all of which can be allocated using standard programming tools, and no message passing between CPUs is required.

The system was running 64-bit [CentOS Linux 6.0](#)¹⁶ with kernel version 2.6.32-71.el6.x86_64. The FFTW 3.3 library¹ was compiled with flags

```
--enable-openmp --enable-sse2 --enable-float CFLAGS='-m64 -O3'
```

using the [Intel Parallel Studio XE 12.0.4 compilers](#).

¹⁰<http://dx.doi.org/10.1086/510018>

¹¹http://www.fftw.org/doc/64_002dbit-Guru-Interface.html

¹²http://www.colfax-intl.com/jlrid/SpotLight_more_Acc.asp?L=122&S=45&B=2329

¹³[http://ark.intel.com/products/53579/Intel-Xeon-Processor-E7-4870-\(30M-Cache-2.40-GHz-6.40-GTs-Intel-QPI\)](http://ark.intel.com/products/53579/Intel-Xeon-Processor-E7-4870-(30M-Cache-2.40-GHz-6.40-GTs-Intel-QPI))

¹⁴<http://research.colfaxinternational.com/post/2012/01/04/Terabyte-RAM-Servers-Memory-Bandwidth-Benchmark.aspx>

¹⁵<http://software.intel.com/en-us/articles/optimizing-software-applications-for-numa/>

¹⁶<http://www.centos.org/>

3 Results

This section reports the results of several measurements: performance (Section 3.1), memory usage (Section 3.2) and FFTW planning time in the MEASURE mode (Section 3.4) for single precision, real-to-real transforms of kind FFTW_R2HC. All results were obtained optimized NUMA policy (see Section 3.3). In all cases where concurrent transforms are discussed, the number of concurrent transforms, P , is equal to K/Q , where $K = 80$ is the number of hyper-threads in the system, and Q is the number of threads per process.

3.1 Performance of Sequential and Concurrent Multithreaded FFTs

Figure 2 shows the measured performance of the system for multithreaded FFTs with transform sizes from $N = 2^9$ to $N = 2^{31}$ and the number of threads per process, Q , from $Q = 1$ to $Q = 80$. The performance metric, T_{1000} , used here is the time to calculate 1000 FFTs. For sequential transforms, T_{1000} equals the mean time of 1 transform times 1000. With concurrent transforms, T_{1000} is the mean time of 1 transform *measured with the machine under full load with P concurrent Q -threaded transforms*, multiplied by 1000 and divided by P . Note that this measurement includes only the transform time; initialization, planning and output are excluded from T_{1000} .

1. In the case of *sequential calculations benchmark* (top panel of Figure 2), the optimal number of threads, Q , depends on the size of the transform. For small transforms ($N < 2^{13}$), the overhead of multithreading does not pay off, and $Q = 1$ provides the best performance. For larger transforms, the parallel scalability improves, and for $N \gtrsim 2^{17}$, choosing $Q = 40$ provides better results than $Q = 1$. For the largest transforms we studied, $N \geq 2^{27}$, $Q = 80$ is the optimal number of threads.
2. For *concurrent multithreaded calculations* (bottom panel of Figure 2), single-threaded transforms ($Q = 1$ and $P = 80$) get the job done in the shortest time. However, choosing the optimal Q for large transforms involves another consideration: does the system have enough RAM to perform $P = 80$ transforms at once? If not, then for larger transforms one must resort to using more and more threads per process, Q , until P concurrent processes do fit in memory. Luckily, for large transforms ($N \geq 2^{21}$), the curves in the bottom panel of Figure 2 are nearly flat up to $Q = 20$. This means that the penalty for high parallelism in concurrent transforms is small, and, in fact, the FFTW has good parallel scaling in the situation when the system is under full load.

Figure 3 shows the same performance data as Figure 2 but plotted as a function of the array size, N , and expressed in GFLOP/s. This figure is provided to simplify comparison with other benchmarks. The FLOP count was estimated as $2.5N \log_2 N$. This is not the actual operations count, but a [convenient estimate](#)¹⁷.

Note: There is no data point for $Q = 1$ and $N = 2^{31}$ in the concurrent calculation benchmark, because there is not enough memory in our system to fit $P = 80$ processes of this size in memory. Memory requirements of sequential and concurrent multithreaded FFTs are presented in Section 3.2.

3.2 Memory requirements of multithreaded 1D FFTs

The measured memory requirement of FFTs sized from $N = 2^{21}$ (8 MB of data in single precision) to $N = 2^{31}$ (8 GB of data) is shown in Figure 4.

For both sequential and concurrent calculation setups, the best performing parallelization scheme requires the most memory. That is, for sequential calculations, $Q = 80$ is the most memory-hungry regime, and for concurrent calculations, the optimal $Q = 1$ requires the most memory for $N \geq 2^{24}$.

Peak memory usage for a single process was obtained by monitoring the quantity `VmPeak` in the resource `/proc/$FFTW_PID/status` with a sampling rate of 1000 Hz. The environment variable `$FFTW_PID` contains the FFT process ID. For concurrent runs, the memory requirement was estimated as that for a single process times the number of concurrent calculations P . Note that P is proportional to K , and for systems with fewer CPUs, the memory requirement for concurrent runs (and, correspondingly, the performance) is lower. For example, a system with 2 instead of 4 CPUs needs half the amount of RAM shown in Figure 4 for a run with a given number of threads per process Q .

¹⁷<http://www.fftw.org/speed/method.html>

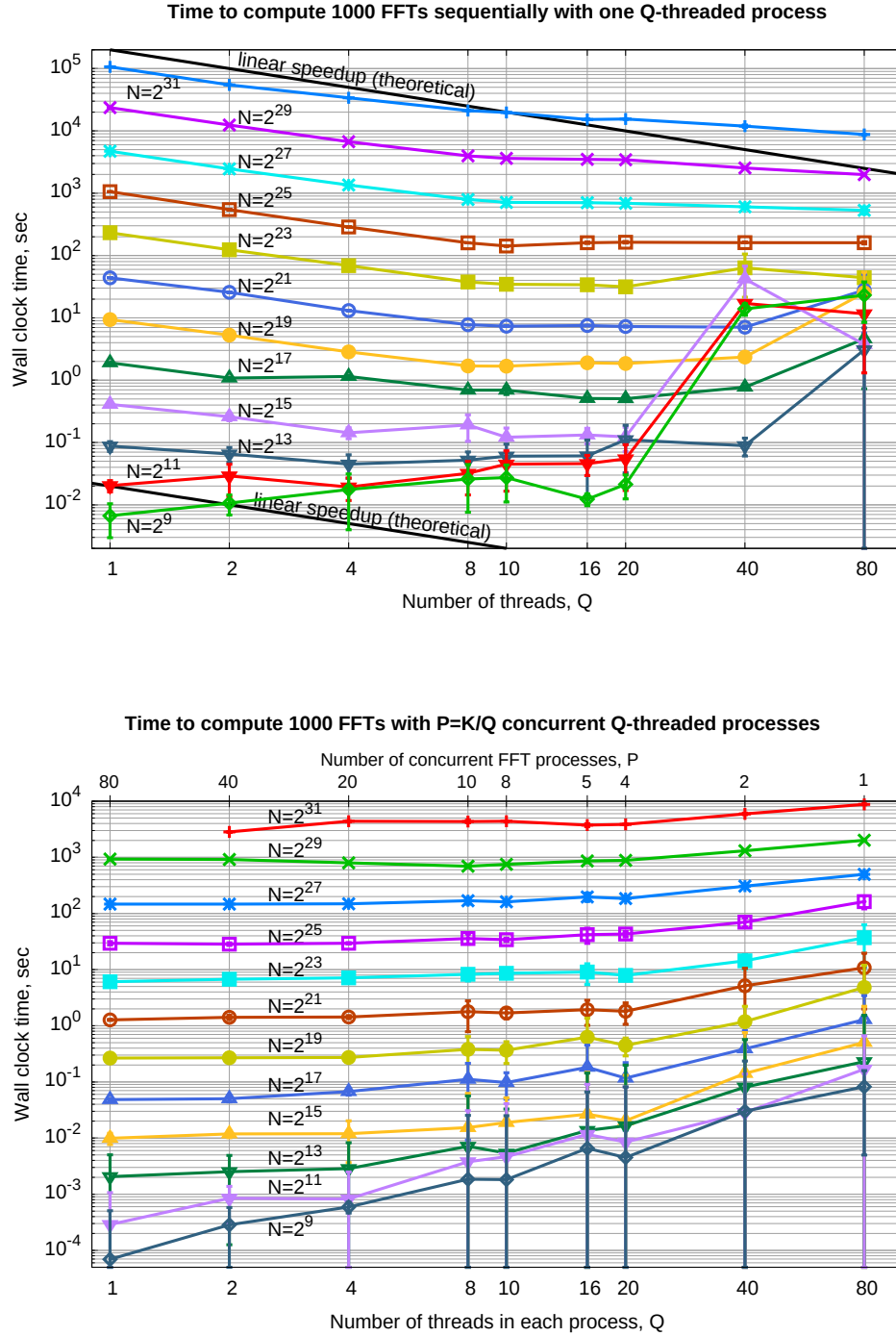


Figure 2: Performance of one Q -threaded FFT (*top panel*) or P concurrent FFT instances with Q threads each (*bottom panel*), as a function of the number of threads Q . Transforms are in-place, 1-dimensional, in single precision, of real-to-real kind FFTW_R2HC. Performance values in the bottom panel (concurrent transforms) are *not* inferred from the measurements in the top panel (sequential transforms); they are measured in a separate test with P concurrent processes fully utilizing the machine ($P \times Q = K$). For performance measured in GFLOP/s, see Figure 3. See Section 3.1 for discussion.

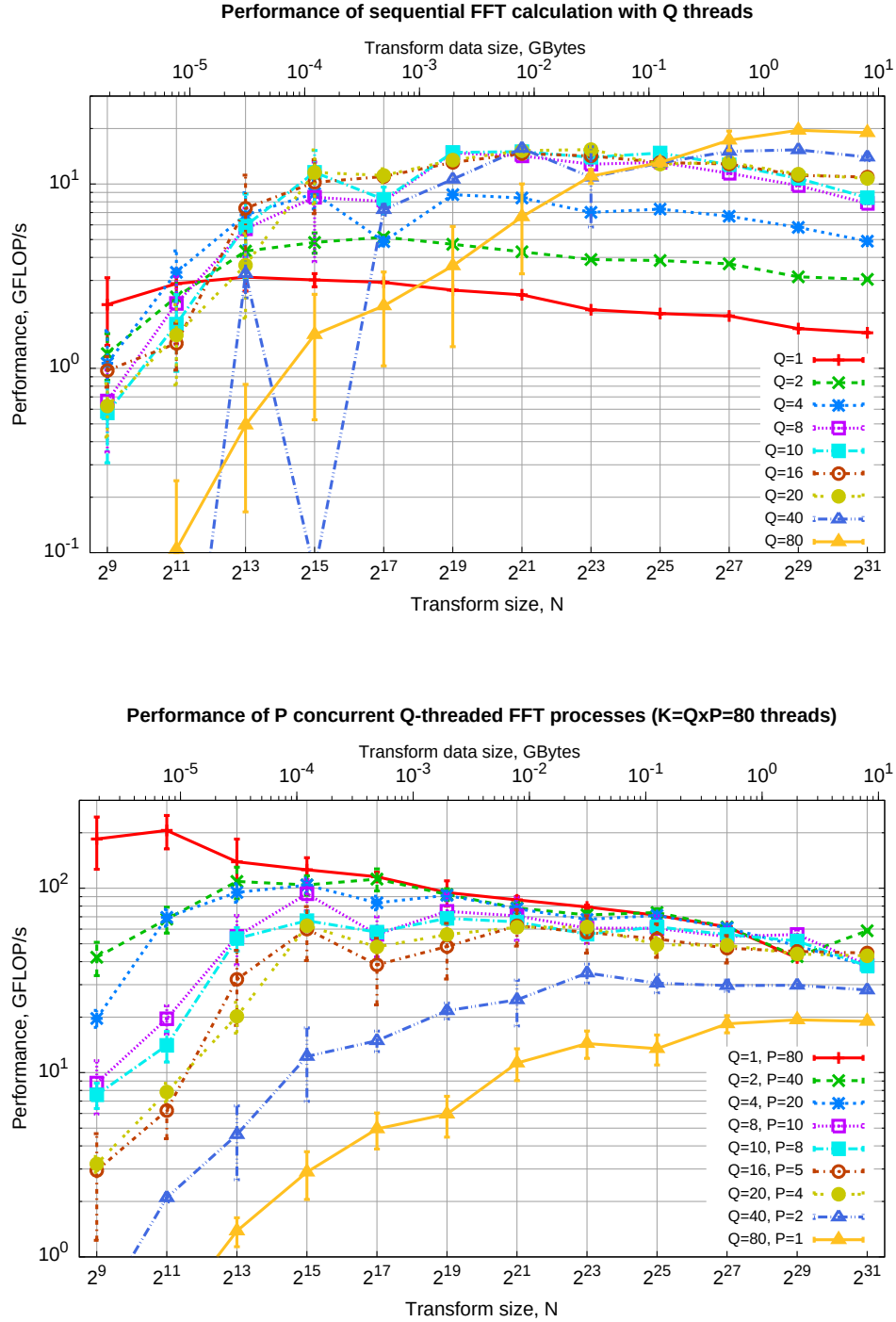


Figure 3: Performance of one Q -threaded FFT (*top panel*) or P concurrent FFT instances with Q threads each (*bottom panel*), as a function of the transform size N . Transforms are in-place, 1-dimensional, in single precision, of real-to-real kind FFTW_R2HC. Performance values in the bottom panel (concurrent transforms) are *not* inferred from the measurements in the top panel (sequential transforms); they are measured in a separate test with P concurrent processes fully utilizing the machine ($P \times Q = K$). This is an alternative graphical representation of the data shown in Figure 2. FLOP count is estimated as $2.5N \log_2 N$. See Section 3.1 for discussion.

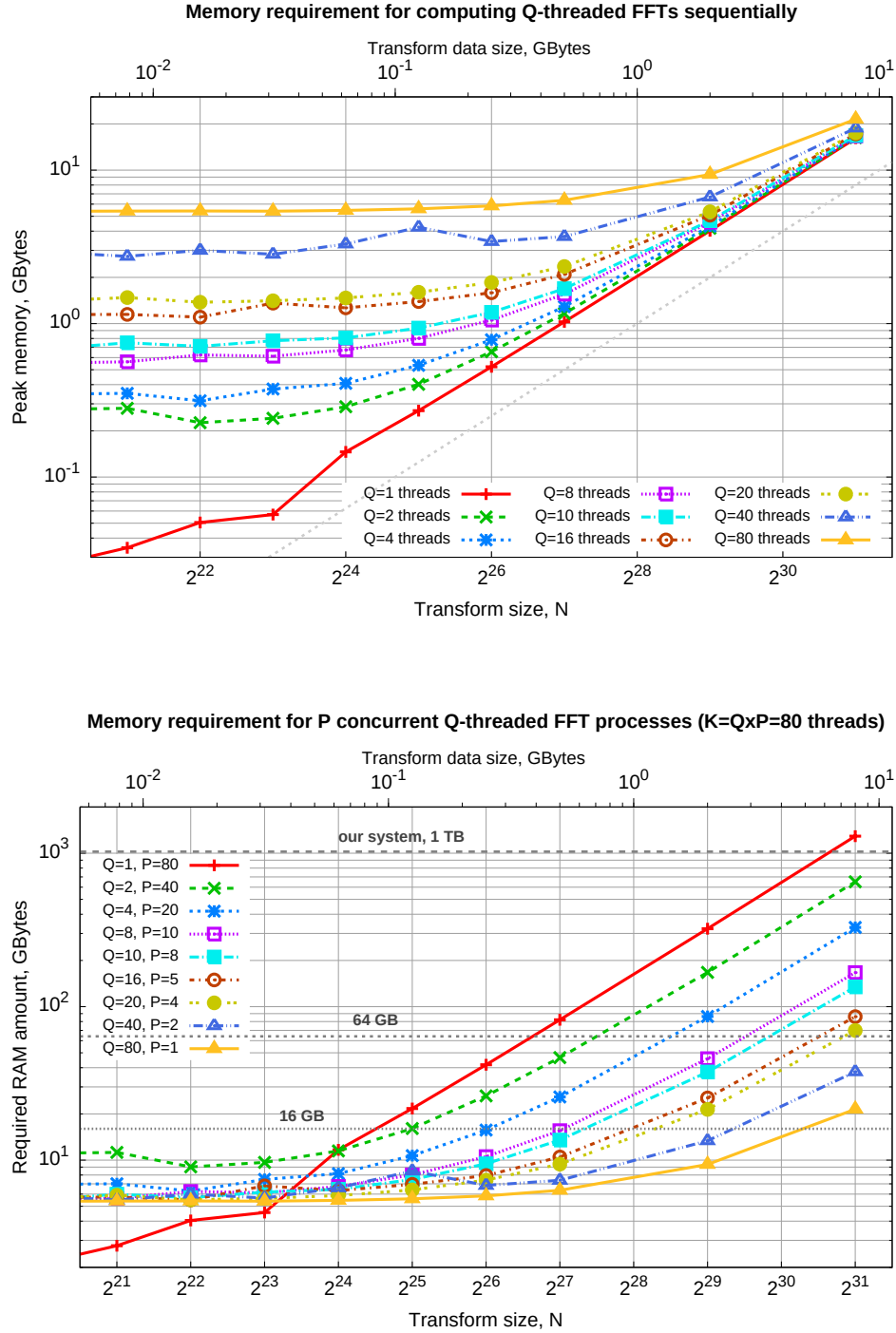


Figure 4: *Top:* multi-threaded sequential transforms incur additional memory overhead.

Bottom: P concurrent processes with Q threads each require P times more RAM than one Q -threaded process. Low Q yields more efficiency (Sec. 3.1), but requires more RAM.

All transforms are performed in-place. Values in the bottom panel (concurrent transforms) are obtained from the measurements in the top panel (sequential transforms) by multiplying them by the number of concurrent processes P . See Section 3.2 for discussion.

3.3 Optimizing NUMA policy to partition resources

NUMA policy aware Linux kernel allows the user to control which CPUs a processes will run on, and where the data will be placed in physical memory.

The default NUMA policy is to allow the operating system to choose at run time where to place the threads and memory of the process. The operating system may also migrate software threads and data to other hardware threads and memory banks when the process is running.

This policy can be modified to partition the machine between processes, so that each process uses only its local, low-latency memory. The FFT requires frequent memory accesses with a complex access pattern, and low latency memory access is beneficial for the FFT performance. In order to optimize the NUMA policy in this way, this benchmark used the `numactl`¹⁸ library. Each shell command launching an FFT instance was prepended with the following call to the `numactl` tool:

```
numactl --membind=$CPUSET --cpunodebind=$CPUSET
```

Here the environment variable `$CPUSET` is the number of the CPU that the process is to be run on. The value of this variable is calculated in the loop that starts the concurrent processes. For example, for $Q = 10$ and $P = 8$, processes 1 through 8 had the following values of `$CPUSET`, respectively: 0, 0, 1, 1, 2, 2, 3, 3. This puts processes 1 and 2 on CPU 0, processes 2 and 3 on CPU 1, etc.

The impact of this optimization is illustrated in Figure 5, where performance before and after the optimization is plotted for $N = 2^{27}$. Note for $Q = 4$, $Q = 10$ and $Q = 20$ the optimization works particularly well, because the number of hyper-threads per CPU, 20, divides evenly into 4, 10 and 20.

A similar result could be achieved by setting the appropriate thread affinity. Intel's OpenMP library supports thread affinity control via the environment variable `KMP_AFFINITY`¹⁹. In our case, we could run the following commands before launching each FFT instance:

```
let KOFFSET=(PNUM-1)*OMP_NUM_THREADS
export KMP_AFFINITY=compact,0,$KOFFSET
```

Here `$KOFFSET` is the offset of the current process from thread 0, variable `$PNUM` is the number of the process (1 through P), and `$OMP_NUM_THREADS` is the number of threads per process, Q . This is a more fine-grained control of thread affinity, as it binds the process to specific hardware threads rather than to specific CPUs.

In tests performed for the present research, the NUMA policy optimization yielded marginally better results than the thread affinity optimization.

Note that this variation of the NUMA policy (or thread affinity) optimizes memory traffic for latency, which is the opposite to what was demonstrated in the [white paper on memory bandwidth benchmark](#)¹⁴, where the affinity of type `scatter` was used to optimize for bandwidth.

3.4 FFTW planning time in the MEASURE mode

The performance of the FFTW library can be significantly improved by planning the transform in the `MEASURE` or `PATIENT` mode. In the course of planning, the library tunes the parameters of the algorithm to adapt it to the machine's architecture (memory bandwidth and latency, cache size, etc.). In our case, planning in the `MEASURE` more improved performance by more than a factor of 10 compared to the faster `ESTIMATE` mode (not shown here). However, the time it takes to plan a transform can be quite long.

Figure 6 shows the time it took to complete planning in the `MEASURE` mode. Planning was done just once, and the FFTW wisdom was saved in files that were re-used in subsequent runs. All benchmarks obtained here used the 'wisdom' (i.e., results of planning) generated in the `MEASURE` mode.

¹⁸<http://linux.die.net/man/8/numactl> and <http://oss.sgi.com/projects/libnuma/>

¹⁹http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/2011Update/cpp/lin/optaps/common/optaps_openmp_thread_affinity.htm

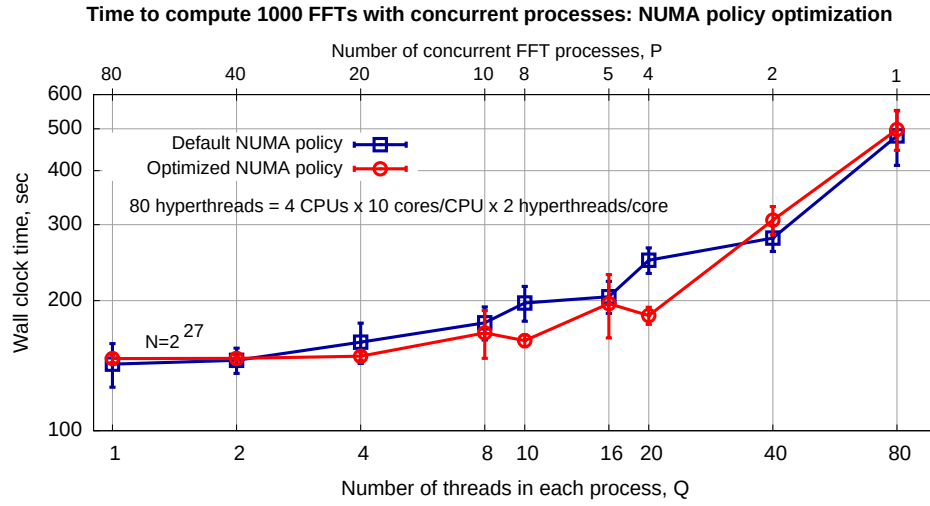


Figure 5: Performance of concurrent FFTs before and after the NUMA policy optimization for $N = 2^{27}$ (see Section 3.3).

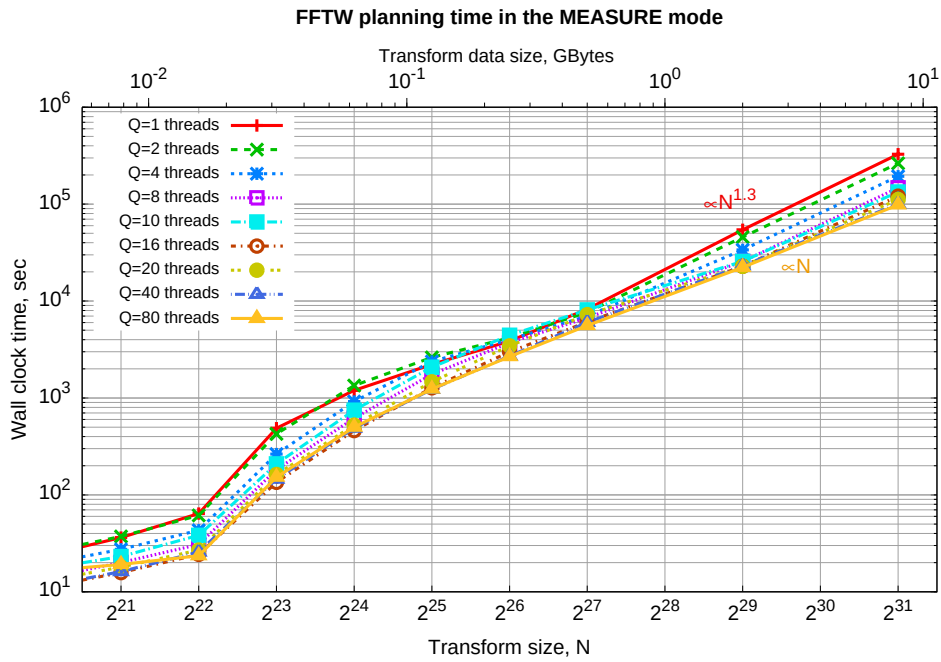


Figure 6: Planning the transform in the MEASURE mode takes a long time for large transforms, but dramatically improves performance. (see Section 3.4).

4 Conclusions

At this time, CPU-based systems are the only efficient solution for computing large 1-D FFTs that exceed the memory of existing GPU devices. And systems with a large amount of memory per core have a significant advantage over lower-RAM alternatives. Consider the following example: computing 1000 FFTs of size $N = 2^{31}$ with concurrent processing. Refer to the bottom panels of Figures 2 and 4 to verify the following numbers:

1. In our system with 1 TB of RAM, we can run with $P = 40$ concurrent processes $Q = 2$ threads per process. This setup uses 650 GB of RAM, and 1000 FFTs are computed in $T_{1000} = 3 \times 10^3$ s.
2. If our system had 64 GB of RAM installed, then we could afford to run just two processes ($P = 2$) with $Q = 40$ threads per core, which would require $T_{1000} = 6 \times 10^3$ s.
3. With 32 GB of RAM, the only possibility is $P = 1$, $Q = 80$ and $T_{1000} = 9 \times 10^3$.
4. With 16 GB, regardless of the number of CPUs, a calculation with $N = 2^{31}$ would not fit in memory, and each transform would have to be computed in distributed memory, further losing efficiency.

And considering that the scientific project motivating this research may require up to 10^6 of transforms $N = 2^{33}$ in size, a system with the amount of RAM as large as this one (1 TB) is an indispensable tool.

Modern computing clusters could tremendously benefit from including very large memory compute nodes in the grid. These nodes serve several purposes:

1. For problems that do not require the whole amount of the onboard RAM, the node can be used to run several concurrent smaller-memory tasks. Efficiency of resource sharing can be boosted by optimizing the NUMA policy or thread affinity setting. Large amount of memory per core allows the user to reduce instruction-level parallelism in favor of task-level parallelism, which improves performance, especially for poorly scalable algorithms. One of possible applications of these systems to concurrent processing (1D FFTs) was demonstrated in this paper.
2. Similarly, for problems that are memory-bound and executed in distributed memory, a large amount of RAM decreases the required number of compute nodes. This also boosts the efficiency of calculations by reducing inter-node communication overhead.
3. For memory-bound problems that cannot be or have not been efficiently parallelized, the utility of very large memory nodes is apparent.

Acknowledgements

I am grateful to the UC Santa Cruz-based team that carries out the blind pulsar search project²⁰ for bringing my attention to this interesting computational challenge and for a fruitful exchange of ideas and expertise on the subject.

Please visit <http://research.colfaxinternational.com/> to learn more about the Colfax Research project, comment on this article, and subscribe for updates.

²⁰<http://scipp.ucsc.edu/>, <http://scipp.ucsc.edu/~pablo/>