Terabyte RAM Servers: Memory Bandwidth Benchmark and How to Boost RAM Bandwidth by 20% with a Single Command

Andrey Vladimirov
Stanford University
for Colfax International

January 16, 2012

Abstract

Colfax International produces servers capable of supporting up to 1 TB of RAM and up to 4 Intel Xeon CPUs. This paper reports the memory bandwidth benchmark of these servers obtained using the STREAM code.

Our benchmark includes comprehensive statistical data: the mean, standard deviation, extrema and the distribution of bandwidth measurements. The distribution of measurements reveals several modes of RAM performance, including an above-average bandwidth mode. By default, the mode realized by any given benchmark depends on an unpredictable runtime pattern of thread and memory binding to the physical cores. The paper shows how to optimize memory traffic for bandwidth and consistently achieve the fastest mode. This is done by controlling the code's thread affinity, and results in a bandwidth increase around 20% over the average unoptimized performance.

Without optimization, the measured RAM bandwidth with one thread is 5.79±0.06 GB/s (the 'copy' test), and it scales almost linearly with the number of threads until it peaks at 67±6 GB/s at 20 threads. Optimized code shows a maximum bandwidth up to 78.9±0.3 GB/s. A list of references for the NUMA architecture tools is provided.

Contents

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 Very Large Memory Servers

Colfax International makes servers capable of supporting up to 1 TB of RAM and up to 4 Intel Xeon CPUs[1]. Each CPU is connected directly to 2 out of 8 memory banks, but the NUMA architecture (*Non-Uniform Memory Access*)[2] with the Intel QPI interface[3] makes the machine a shared-memory system, in which any CPU may address RAM connected to any other CPU.

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 and higher bandwidth than *remote* memory, which needs to be fetched via a bus. On the other hand, the complex memory hierarchy is hidden from the user. That is, from the application's perspective, the machine appears to have a large amount of RAM, all of which can be allocated using standard programming tools, and no message passing between CPUs is required.

While today these machines may look exotic, if not for the novelty of the technology, then for the enormous amount of RAM, they have a variety of useful applications, ranging from virtualization and databases to memory-intensive scientific calculations. In a future publication, I will make an argument that these systems may be invaluable as compute nodes in HPC clusters.

Figure 1: Four Intel Xeon CPUs and 1 TB of RAM onboard.

In this report, I present a memory bandwidth benchmark of one of these machines. In my configuration, each of the 4 CPU sockets contained an Intel Xeon E7-4870 CPU[4] with 30 MB of L3 cache and 10 cores with hyper-threading technology, clocked at 2.40 GHz (i.e., a total of 40 cores and 120 MB of cache). The total amount of RAM installed in the system was 1 TB in 1066 GHz speed DDR3 memory modules. The system was running 64-bit CentOS Linux 6.0[5] with kernel version 2.6.32-71.el6.x86_64.

[1] http://www.colfax-intl.com/jlrid/SpotLight_more_Acc.asp?L=122&S=45&B=2329
[2] http://software.intel.com/en-us/articles/optimizing-software-applications-for-numa/
[3] http://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html
[4] http://ark.intel.com/products/53579/Intel-Xeon-Processor-E7-4870-(30M-Cache-2_40-GHz-6_40-GTs-Intel-QPI)
[5] http://www.centos.org/

2 Bandwidth vs Number of Threads

I measured the memory bandwidth using the popular STREAM benchmark code[6]. The STREAM code was compiled using the Intel C++ compiler version 12.0.4[7] with the working array size set to 500 MB (well in excess of the total amount of L3 cache in the system). The number of threads accessing the memory was determined by setting the environment variable `OMP_NUM_THREADS` to 1, 2, 4, 8, 16, 20, 40 and 80. Hyper-threading was enabled in this test. For each value of `OMP_NUM_THREADS`, the benchmark was run 1000 times. Each of these series of 1000 runs yielded the mean value, standard deviation, minimum and maximum of the bandwidth measurements.

Plots in Figure 2 show the bandwidth measured in these tests. The blue and red shaded regions delimit the minimal and maximal bandwidth observed in the 20 runs. The markers and error bars show the mean values and standard deviations of the bandwidth.

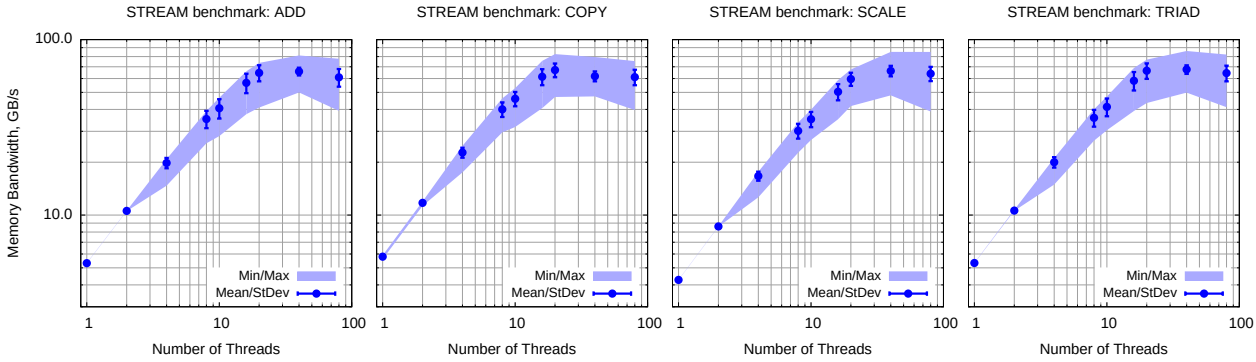

Figure 2: STREAM benchmark results

The data show that the best performance in all tests is achieved with 20-40 threads. With fewer than 20 threads, the bandwidth scales almost linearly with the number of threads. The standard deviation of multi-threaded runs is of order 10% of the mean value. For example, the 'Copy' test with 20 threads had a mean of 67 GB/s and a standard deviation of 6 GB/s. In this test, the bandwidth measurements in 1000 runs ranged from 47 to 83 GB/s. We will see in the next section that the probability distribution of the bandwidth measurements is not normal (i.e., Gaussian), but multimodal.

[6] http://www.cs.virginia.edu/stream/
[7] http://software.intel.com/en-us/articles/intel-parallel-studio-xe/

3 Distribution of bandwidth measurements

For some performance critical applications, one may want to know the minimal and maximal bandwidth expected in any given memory access. In NUMA-enabled systems, the variation of bandwidth from run to run is not guaranteed to be normal, and this section illustrates and explains this fact

Figures 3 and 4 show the numbers of runs out of 1000, in which the measured bandwidth fell into the respective bins of the histogram. The width of the bins is 1 GB/s.

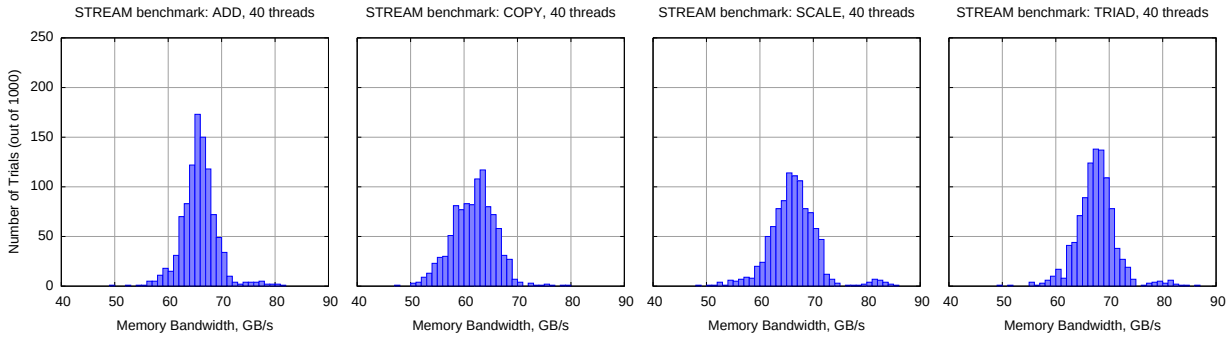

Figure 3: Distribution of memory bandwidth measurements with 40 threads

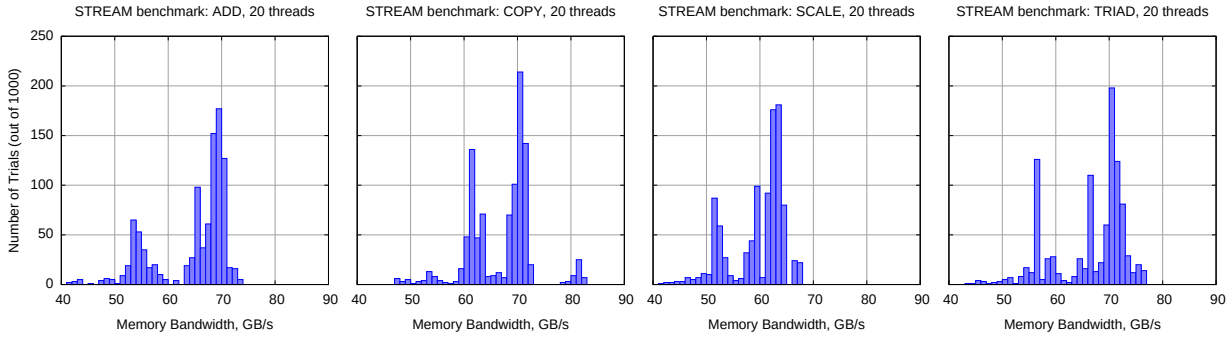

Figure 4: Distribution of memory bandwidth measurements with 20 threads

In some cases (e.g., the ‘Add’ test with 40 threads), the distribution is consistent with the normal distribution. However, in other cases, a second, high bandwidth, mode is observed around 82 GB/s (e.g., the ‘Scale’ test with 40 threads). There was even a case (the ‘Copy’ test with 20 threads), where the distribution had the form of several distinct, narrow peaks around 53, 62, 71 and 82 GB/s.

The reason for such multimodality is that the bandwidth depends on the distribution of threads and allocated memory across the physical cores in the system. Some ”lucky” runs had distribution favorable for bandwidth, while other had suboptimal operating conditions. For many applications, such behavior is acceptable, as it strikes the balance between memory bandwidth and latency. However, for applications sensitive to the speed of data streaming from RAM to the CPU, it is possible to tweak this behavior and always use the highest bandwidth mode. We discuss this optimization in the next section.

4 Optimizing for Memory Bandwidth with Thread Affinity

In his white paper 'An NUMA API for Linux'[8], A. Kleen states that with NUMA memory architecture,

> To get more bandwidth the memory controllers of multiple nodes can be used in parallel. This is similar to how RAID can improve disk IO performance spreading IO operations over multiple hard disks.

At the same time, he notes:

> Most programs seem to prefer lower latency, but there are a few exceptions that want bandwidth. Using node local memory has the best latency.

We do not investigate memory latency in this paper. However, for applications that require high bandwidth, and in which the pattern of memory access is regular, it is easy instruct the OpenMP library to optimize the thread affinity policy for memory bandwidth.

Intel's OpenMP library supports thread affinity control via the environment variable KMP_AFFINITY[9]. In particular, the `scatter` type of affinity distributes threads in a round-robin pattern across the packages (i.e., CPUs). This results in highly parallel access to memory, utilizing all available memory controllers.

To test this approach, I added the following command line to the shell script executing the benchmark:

```
export KMP_AFFINITY=scatter
```

The result is illustrated in Figure 5, where the data before and after the optimization are plotted.

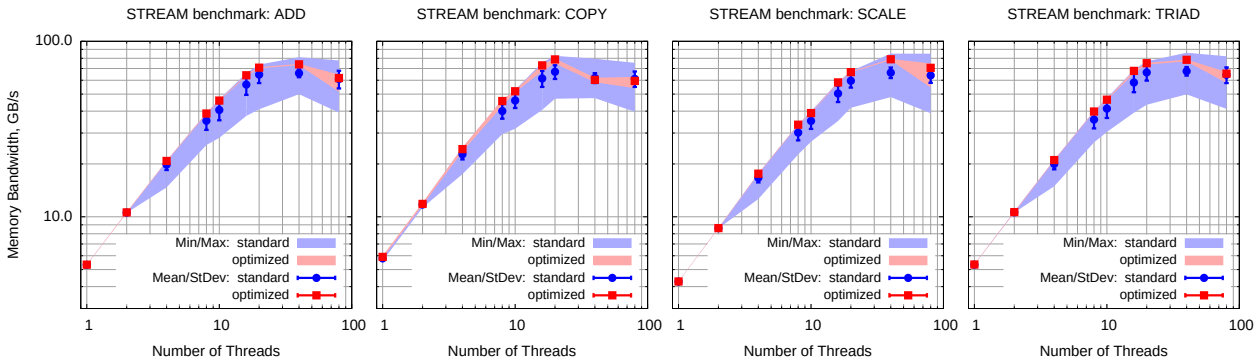

Figure 5: STREAM benchmark results with and without bandwidth optimization

It is evident that with `KMP_AFFINITY=scatter`, the variation of bandwidth from run to run is considerably smaller, and the value of bandwidth is consistently higher than in the unoptimized case.

Figures 6 and 7 show the histograms of binned frequencies of bandwidth measurements before and after the thread affinity optimization. In all cases except the 'Copy' test with 40 threads, the measurements with optimization are concentrated in the high-bandwidth mode.

I do not show the results of a benchmark with `KMP_AFFINITY=compact`, in which threads are distributed compactly on the minimal possible number of CPUs. In this case, the system will try to allocate memory on the local node, and memory latency should be optimized. However, I notice for the record that in this case, the bandwidth with 20 threads drops to ≈20 GB/s.

This concludes the exploratory part of this paper, but read on for a list of helpful resources.

[8] http://halobates.de/numaapi3.pdf

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

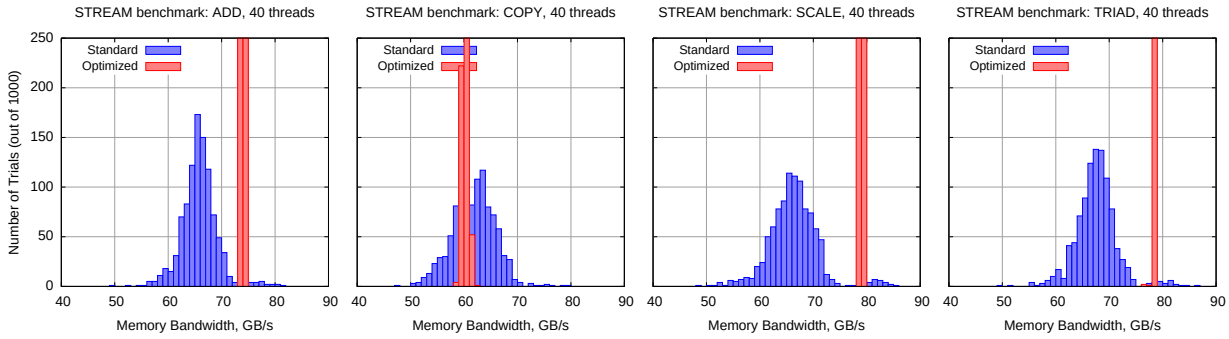

Figure 6: Distribution of memory bandwidth measurements with 40 threads

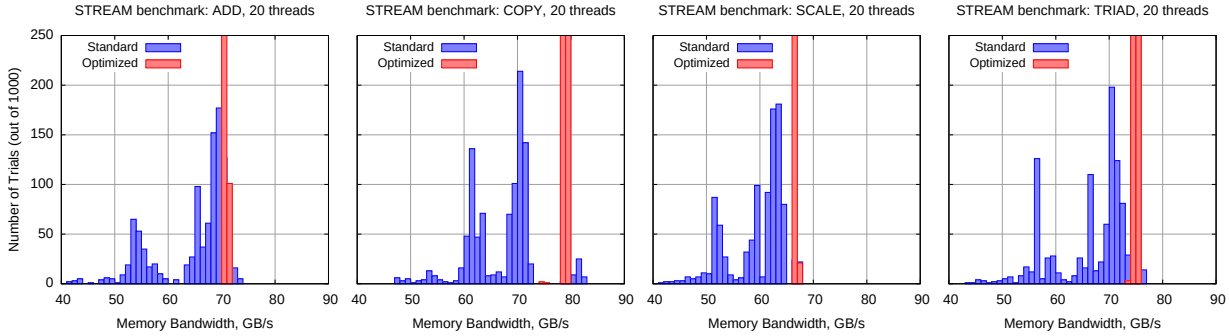

Figure 7: Distribution of memory bandwidth measurements with 20 threads

5 Additional resources and closing words

In addition to the control that the `KMP_AFFINITY` variable gives over thread affinity, programmers can take advantage of Linux system calls in NUMA-aware kernels: `mbind`[10], `mmap`[11], `set_mempolicy`[12], and related calls listed in the documentation. These calls allow fine-grained NUMA policy control.

Another tool that may be useful for fine-tuning memory intensive application performance is the library `numactl`[13], which implements a simple NUMA policy support. Command-line tools included with `numactl` allow to set NUMA scheduling or policy memory placement policy for a process.

Stay tuned for upcoming posts, where I describe how we are using these machines to perform very large Fourier transforms for astrophysical calculations, and argue for the utility of these systems as compute nodes in HPC clusters. Please visit `http://research.colfaxinternational.com/` to learn more about the Colfax Research project, comment on this article, and subscribe for updates.

Special thanks to Troy Porter[14] of Stanford University for a productive discussion of this project.

[10] http://www.kernel.org/doc/man-pages/online/pages/man2/mbind.2.html
[11] http://www.kernel.org/doc/man-pages/online/pages/man2/mmap.2.html
[12] http://www.kernel.org/doc/man-pages/online/pages/man2/set_mempolicy.2.html
[13] http://linux.die.net/man/8/numactl and http://oss.sgi.com/projects/libnuma/
[14] http://www.stanford.edu/~tporter/