Memory Bound vs. Compute Bound: A Quantitative Study of Cache and Memory Bandwidth in High Performance Applications

Alex Hutcheson and Vincent Natoli
Stone Ridge Technology

January 2011

Abstract

High performance applications depend on high utilizations of bandwidth and computing resources. They are most often limited by either memory or compute speed. Memory bound applications push the limits of the system bandwidth, while compute bound applications push the compute capabilities of the processor. Hierarchical caches are standard components of modern processors, designed to increase memory bandwidth and decrease average latency, particularly when successive memory accesses are spatially local. This paper presents and analyzes measured bandwidths for various levels of CPU cache and varying numbers of processor cores. We have modified the STREAM benchmark[1] to include a more accurate timer and greater multicore capabilty in order to accurately measure bandwidth over a range of memory sizes that span from the L1 cache to main memory. While the STREAM benchmark is designed to test peak bandwidth for sequential memory access, this paper extends it to analyze the effects on bandwidth caused by random memory access and increased computational intensity, two common instances in High Performance Computing.

1 Introduction

In modern computer architectures, gains in processor performance, via increased clock speeds, have greatly outpaced improvements in memory performance, causing a growing discrepancy known as the "memory gap" or "memory wall". Therefore, as time has passed, high-performance applications have become more and more limited by memory bandwidth, leaving fast processors frequently idle as they wait for memory. In order to reduce this problem, it is now standard for small amounts of SRAM to be integrated onto the processor chip as various levels of cache memory. By location and design, this cache memory delivers much higher bandwidth and lower latency than is possible on main memory. This study will demonstrate the quantitative benefit in memory bandwidth caused by this cache memory. It will also demonstrate that the benefit of cache memory is dependent on the pattern of memory access as well as the computational intensity of the application.

Despite the importance of cache and memory performance, many practicioners have no more than a qualitative feel for the topic. With the emergence of multicore chips and shared caches, a truly quantitative understanding of the effect of bandwidth and compute capacity on performance becomes urgent for practitioners interested in extracting maximal performance from the system.

2 The Measured System

In this paper, the system being measured is an Appro 1U server containing two quad-core Intel Xeon X5560 "Gainestown" processors based on the Intel Nehalem microarchitecture. These processors have L1, L2, and L3 cache sizes of 32 KB, 256 KB, and 8192 KB, respectively. Each core has its own L1 and L2 cache, but the L3 cache is shared between the four processor cores. In the BIOS settings, "hardware prefetching" and "Intel TurboBoost" were disabled, in order to create a stable environment in which to run the tests. In addition,"simultaneous multi-threading" was disabled, to allow for the use of only the physical processor cores. "Hyper-Threaded" cores are not utilized. Further information is available at Intel's product page[2].

3 Compiler and Optimization

All code used for this paper was compiled on the Intel C Compiler, version 11.0 using the compiler flag -O2 for optimization. The optimization performed by the compiler includes the use of SSE2 instructions, which allow the processor to access the SSE vector registers. For programs in which only a single core was used, the `sched_setaffinity` function from the GNU C library was used to pin the program to a single core. For programs in which multiple cores were used, the OpenMP API was used to parallelize the program. When OpenMP was used, the environmental variables `KMP_AFFINITY` and `OMP_NUM_THREADS` were manipulated to control the assignment of processes to cores.

4 The STREAM Benchmark

The STREAM benchmark is a simple synthetic benchmark program developed by John D. McCalpin, and is freely available online[4]. The STREAM benchmark measures sustainable memory bandwidth by analyzing the amount of time that it takes to perform several simple vector operations, including copy, add, scalar, and triad, on arrays of double-precision floating point values.

Although the benchmark was developed to measure bandwidth to main memory, it can also be adapted to measure bandwidth to caches as well. This required several changes, including replacing the default timer with a more precise one. The built-in timer, `gettimeofday`, has a granularity that was too large to test smaller problem sizes that access lower level caches. This timer was replaced with the Read Time Stamp Counter (`rdtsc`) function, which has a much finer granularity and allows for the testing of much smaller problem sizes. In addition, the STREAM benchmark had to be modified to test a variety of memory sizes. This was achieved by modifying an environmental variable, recompiling the source code, and running the benchmark again. This activity was scripted, to allow for the rapid testing of a wide range of memory sizes. This modified version of the benchmark program was then used to test bandwidth in a variety of situations.

5 Results

For the purposes of this paper, several variants of the STREAM benchmark were developed. The most basic of these tested the bandwidth by accessing memory sequentially, and performing very basic computation with it. Additional versions of the program were designed to test the effects of random memory access and of increased computational intensity.

5.1 Bandwidth-Bound Problems

The initial modification of the benchmark was designed to simulate problems which are bandwidth-bound, meaning problems in which memory bandwidth is the limiting factor. In a bandwidth-bound situation, the processor is unable to utilize its full computational potential, because the memory controller is unable to keep up with its requests. Bandwidth bound problems include any problem of low computational intensity, defined as the number of operations performed per memory access.

In order to simulate this situation, the program accesses an array of values sequentially, performing very basic operations on the values it receives. Because the processor can perform the operations much faster than the memory controller can satisfy requests for memory, the majority of the program's runtime is spent waiting for memory requests to be satisfied. This latency is greatly reduced when the array in question is small enough to fit into one of the levels of the processor's cache memory, resulting in a large increase in bandwidth. Therefore, one can expect the smallest problem sizes to yield the greatest bandwidth, with marked decreases in bandwidth when the cache size is overrun.

In order to simplify the collection and presentation of data, we limited our measurements to only include the "add" routine present in the original STREAM benchmark. It should be noted that, in our tests, the "triad" routine produced near-identical results. Results from the "copy" and "scalar" operations were complicated by compiler optimizations, and are not presented here.

5.1.1 Single Core

Running this benchmark on one CPU core, and varying the amount of memory allocated, yielded the results shown in Figure 1.

Figure 1: Bandwidth results for sequential access, one core

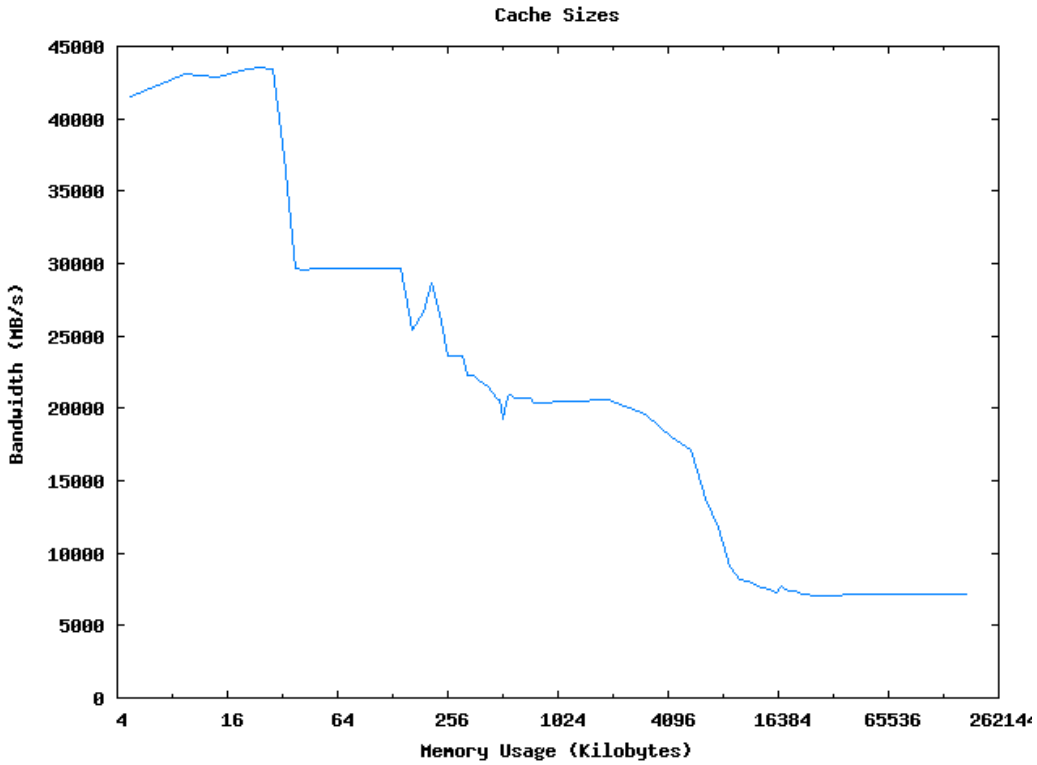


Figure 1 demonstrates a clear pattern of decreasing bandwidth with each sequential cache level. When the memory that is allocated exceeds the current cache size, a bandwidth penalty is apparent. These sharp decreases in bandwidth correspond with the advertised cache sizes for this device. For memory sizes less than 32 KB, corresponding to the size of the L1 cache, measured effective bandwidth is between 40 GB/s and 45 GB/s, with most of the measurements indicating a bandwidth of approximately 43 GB/s. This is very close to our ideal predicted value of 44.8 GB/s, which we derived from the 128 bit read/write capabilty of the L1 cache multiplied by our processor's 2.8 GHz clock frequency. For memory sizes between 32 KB and 256 KB, which would place the data in the L2 cache, measured effective bandwidth is approximately 30 GB/s. For memory sizes from 256 KB to 8 MB, corresponding to L3 cache, effective bandwidth is measured to be approximately 20 GB/s. For memory sizes greater than 8 MB, indicating main memory

access, bandwidth is measured to be relatively constant at around 7 GB/s. This data indicates that effective bandwidth from L1 cache is 5 to 7 times greater than from main memory.

5.1.2 Multiple Cores

Using OpenMP, it is possible to parallelize the benchmark and thereby obtain results for running the benchmark on several cores. The unmodified STREAM benchmark includes OpenMP instructions, however, the creation of the multiple threads was placed within the timing loop. This created an overhead time cost that interfered with results, particularly in small problem sizes. To resolve this, we modified the code to create the threads outside of the timing loop, and thus produce more accurate data. The program was tested using two, four, and eight cores. The number of threads and the placement of threads on processor cores were controlled using the environmental variables `OMP_NUM_THREADS` and `KMP_AFFINITY`, respectively.

Our initial test measured the resultant bandwidth when the benchmark was run on two cores, on the same processor. Results of this test are shown in Figure 2.

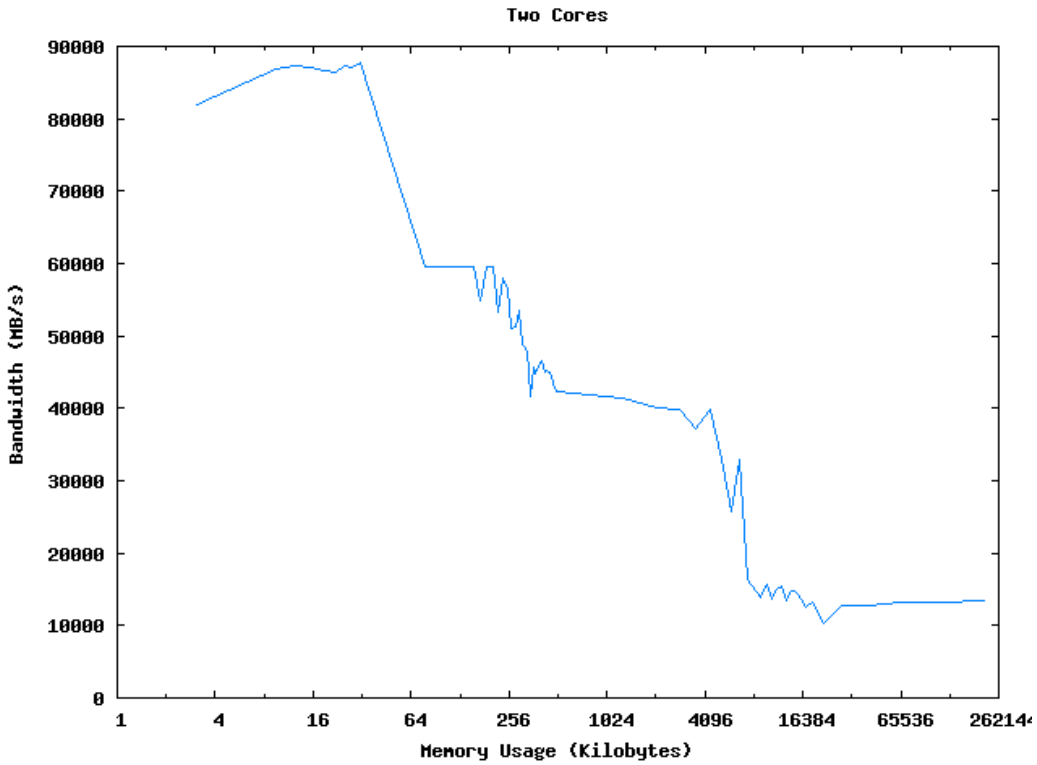
The results displayed in Figure 2 demonstrate the clear performance benefit that can be achieved through parallelization. Measured bandwidths to cache levels L1, L2, and L3 were approximately 86 GB/s, 60 GB/s, and 40 GB/s, respectively. Effective bandwidth to main memory is measured to be approximately 14 GB/s. In each case, the bandwidth performance represents a factor of two increase over single core performance. Table 1 provides a quick comparison of these results.

Memory Hierarchy	Single-Core Bandwidth	Two-Core Bandwidth
L1	43 GB/s	86 GB/s
L2	30 GB/s	60 GB/s
L3	20 GB/s	40 GB/s
Main Memory	7 GB/s	14 GB/s

Our second test measured the effective bandwidth of four cores on the same processor. These measurements are illustrated in Figure 3.

The results of this test help to illustrate how bandwidth is affected as the number of cores on a given processor is increased. At the L1 and L2 cache levels, the expected twofold increase over the two-core results is observed, yielding effective bandwidths of approximately 172 GB/s to L1 and

Figure 2: Bandwidth results for sequential access, two cores using OpenMP

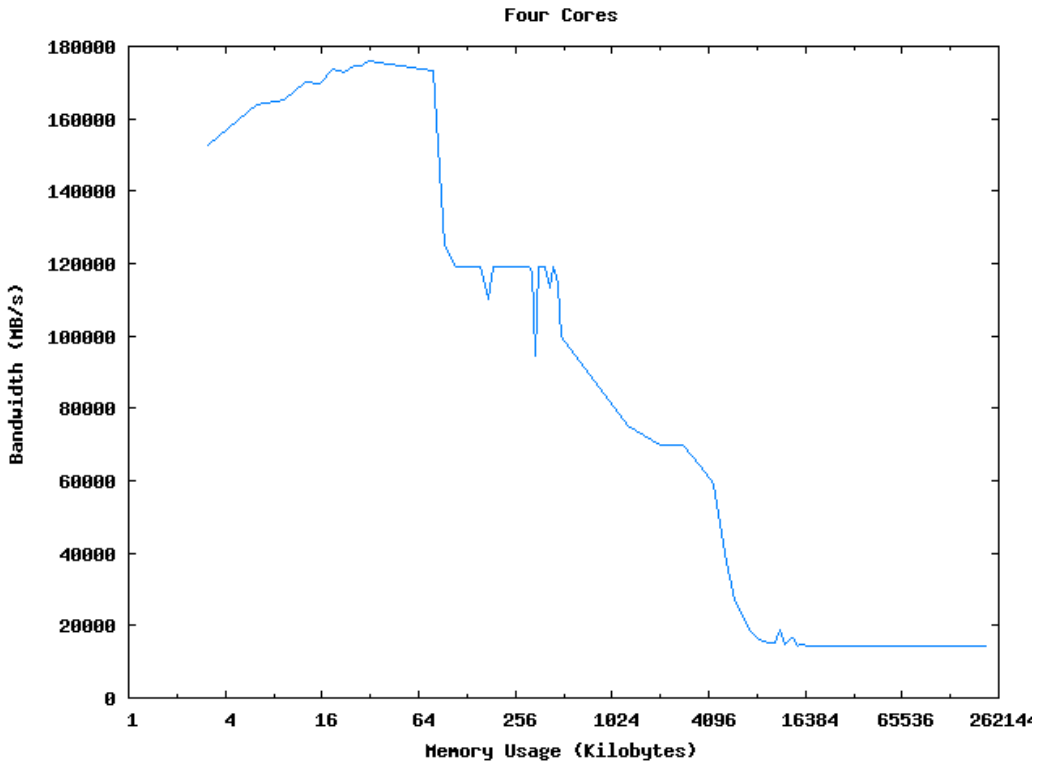


120 GB/s to L2. However, bandwidth to L3 and main memory do not exhibit the same increase. Effective bandwidth to L3 is measured to be approximately 60 GB/s, only a 1.75x increase over the two-core results. In addition, effective bandwidth to main memory is stagnant, yielding the same 14 GB/s measurement as the two-core test. Table 2 compares the results of the two tests.

Memory Hierarchy	Two-Core	Four-Core	Scaling Factor
L1	86 GB/s	172 GB/s	2x
L2	60 GB/s	120 GB/s	2x
L3	40 GB/s	70 GB/s	1.75x
Main Memory	14 GB/s	14 GB/s	1x

These results can be explained through an analysis of the memory architecture. Both the L1 and L2 cache levels are not shared, each core has its own L1 and L2 caches. Because of this, we can expected bandwidth per-

Figure 3: Bandwidth results for sequential access, four cores using OpenMP

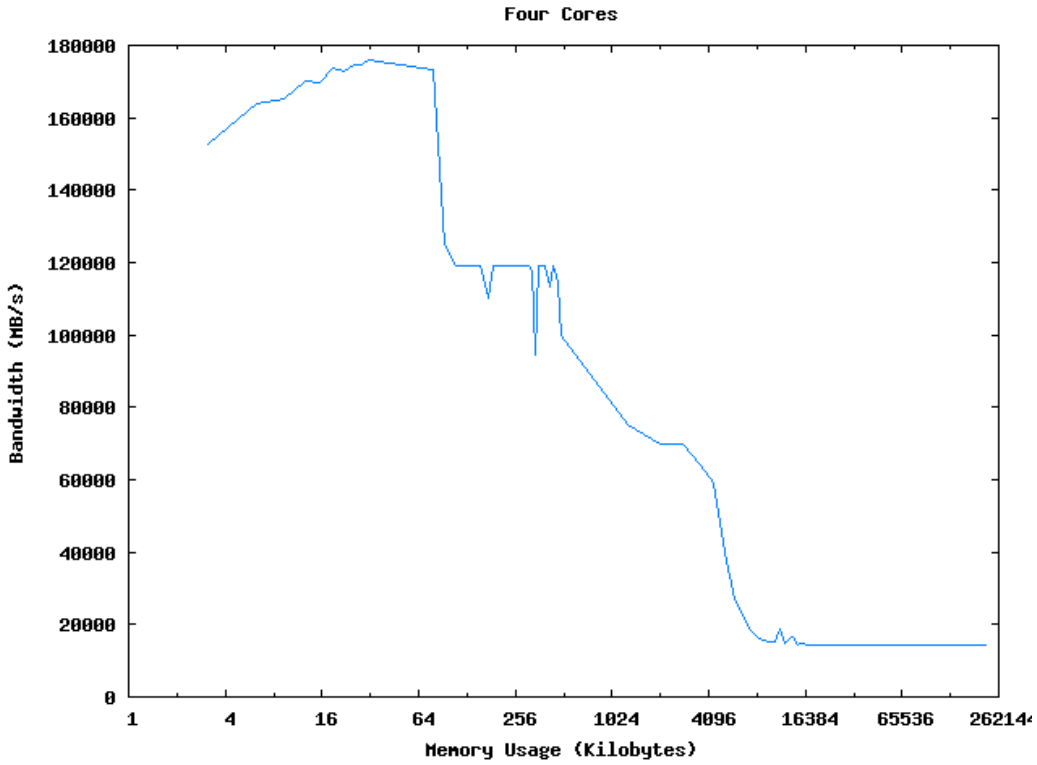


formance to these levels to scale in a linear fashion as the number of active cores is increased. In contrast, the L3 cache is shared between all of the cores, and main memory is shared throughout the system. As the number of active cores on the processor is increased, we can once again expect bandwidth performance to scale in a linear fashion, but only until the resource becomes saturated and is operating at peak performance. Once this peak performance is achieved, additional cores will have no impact on bandwidth performance, it is effectively constant from that point on. This effect is visible in our test. Main memory bandwidth appears to peak at 14 GB/s, while L3 peaks at 60 GB/s.

If this explanation is correct, we would expect that distributing the four cores across both processors would result in increase performance in L3 cache, because each processor would utilize its own L3 cache and memory controller. In order to test this, we performed another test, in which we used four cores

total, with two on each processor. The results are shown in Figure 4 .

Figure 4: Bandwidth results for sequential access, four cores, distributed between two processors, using OpenMP



As predicted, distributing the active cores between both processors yielded a greater effective bandwidth from the L3 level. At this point, neither of the processors' L3 caches have reached peak performance, so they are able to scale as additional processors are brought online. Table 3 summarizes the results. Note that bandwidth performance to memory was unaffected, because access to memory is shared across the entire system.

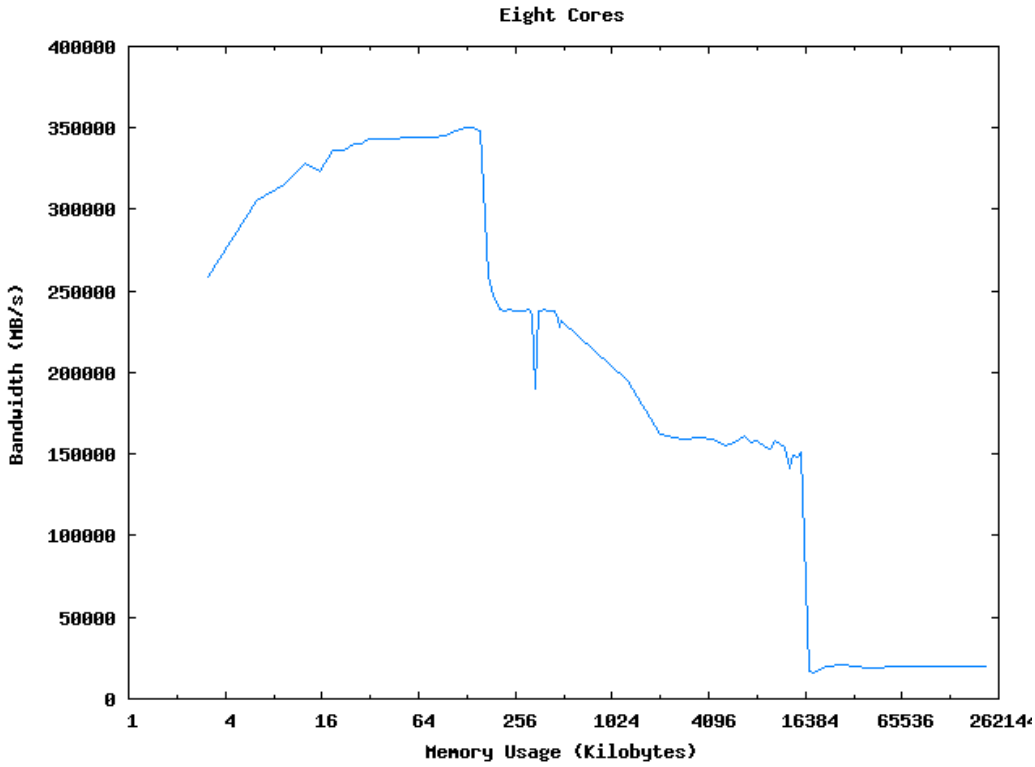
Memory Hierarchy	Two-Core	Four-Core Distributed	Scaling Factor
L1	86 GB/s	172 GB/s	2x
L2	60 GB/s	120 GB/s	2x
L3	40 GB/s	80 GB/s	2x
Main Memory	14 GB/s	14 GB/s	1x

Given the data we now have, it is possible to assess the approximate

peak performance of each element of the system. From our trials, we have determined that maximum effective bandwidth to L1 is approximately 43 GB/s per core, to L2 is 30 GB/s per core, to L3 is approximately 70 GB/s per processor, and to main memory is approximately 14 GB/s for the system. With these values, we would therefore estimate the total peak bandwidth of our two quad-core processor system to be approximately 344 GB/s to L1, 240 GB/s to L2, 140 GB/s to L3, and 14 GB/s to main memory.

In order to test this prediction, we ran a final test, with all eight cores active. The results are shown in Figure 5.

Figure 5: Bandwidth results for sequential access, eight cores on two processors using OpenMP



As the figure demonstrates, the results confirmed our predictions regarding the systems maximum effective bandwidth at each level. Table 4 reviews our experimentally determined values for peak bandwidth performance on our system.

Memory Hierarchy	Unit Bandwidth	Maximum Total Bandwidth
L1	43 GB/s per core	344 GB/s
L2	30 GB/s per core	240 GB/s
L3	70 GB/s per processor	150 GB/s
Main Memory	14 GB/s total	14 GB/s

5.2 Random Memory Access

Although many computing applications utilize sequential or strided access to memory, this is not always the case. Some problems can only be solved by accessing memory in a random fashion. Examples of high-performance computing applications that require random memory access include sparse matrix algebra and hash table lookups. To simulate such a situation, another version of the benchmark was developed. In this version, the memory is not accessed sequentially, but instead random array indices are used.

When memory is requested, the memory controller delivers not only the specific bits of data that are requested, but also the "cache line" consisting of the spatially local bits of memory surrounding it. If memory is being accessed sequentially, it therefore becomes very likely that the next requested piece of data will already be in cache memory, having been included in the most recent cache line. The program receives a performance boost, because it no longer must wait for a lengthy call from main memory. Instead, it is able to access the data from the much faster cache memory.

When data is being accessed randomly, however, the program loses this benefit. In this situation, it becomes very unlikely that the two successive pieces of data will be located close enough together to be on the same cache line. As a result, almost every time the program needs data, it will be forced to make a call to memory for another cache line.

The consequence of this type of data access is that it does not allow for the proper utilization of the hierarchal cache memory on larger arrays, resulting in a much greater percentage of calls to main memory. The system is accessing cache lines at its peak bandwidth, but is only using a small part of each cache line. Although data transfer is occurring at very high speeds, a large percentage of the data is neither needed nor utilized. As a result, one would expect to see a marked decrease in performance in effective bandwidth when this pattern of memory access is employed.

5.2.1 Benchmark

In order to simulate a random memory access scenario, we made additional modifications to our benchmark. Our standard benchmark consists of three double-precision floating point arrays, identified by the letters `a`, `b`, and `c`. The `add` operation iterated through the arrays by ascending index value, starting at index value 0 and ending at the last elements of the array. For each index value `i`, the function would read the contents of `b[i]` and `c[i]`, compute their sum, and write that value to `a[i]`.

When modified for random memory access, the benchmark would also create a fourth array, known as `r`, to be filled with random integer index values. The `add` operation was then modified, so that rather than reading from and writing to index value `i`, it instead used a random index value drawn from `r` by calling `r[i]`.

The results of this benchmark are complicated by the fact that array `r` is accessed sequentially, while the other arrays are accessed randomly, but, overall, the modification models a random memory access situation.

5.2.2 Results

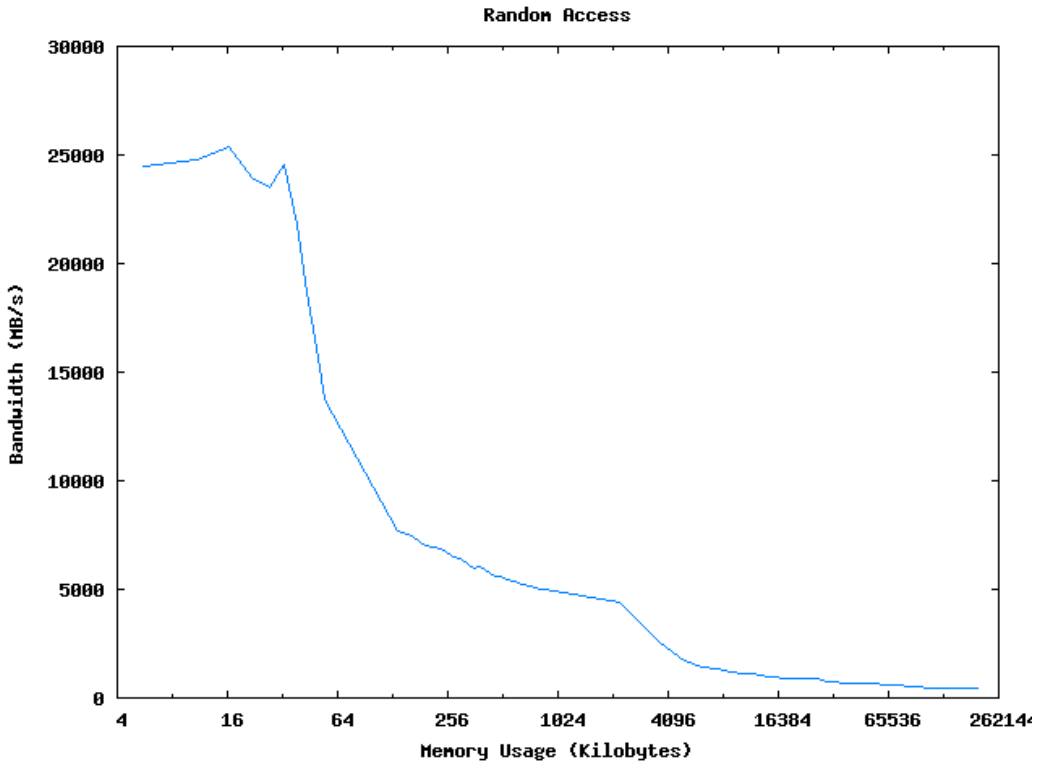
The results shown in Figure 6 were obtained when the random access benchmark was run on a single core.

Measured bandwidth results from random access were drastically lower than results obtained from sequential access. At the L1 cache level, effective bandwidth was measured to be approximately 24 GB/s, as compared to the 43 GB/s measured using sequential access. Beyond L1, a steep performance drop is visible. Bandwidth performance in L3 is measured to be approximately 5 GB/s, and main memory bandwidth is measured to be only about 0.9 GB/s. Table 5 compares these results to those measured using sequential access.

Memory Hierarchy	Sequential Access	Random Access
L1	43 GB/s	24 GB/s
L3	20 GB/s	5 GB/s
Main Memory	7 GB/s	0.9 GB/s

As the results in Table 5 demonstrate, random memory access yields significantly lower bandwidth than sequential access. In fact, for main memory, effective bandwidth is reduced by an approximate factor of 8. This means

Figure 6: Bandwidth results for random access, one core



that, when accessing memory randomly, the system is using only about 13% of its potential single-core bandwidth.

This loss in performance results from the fact that, when memory is accessed randomly, the system is unable to properly utilize the CPU cache. When the processor's memory controller makes a request for memory, it returns not only the requested value, but an entire 64 byte cache line. Because we are using double-precision floating point numbers, which are 8 bytes in size, this cache line consists of 8 values.

When the memory is being accessed sequentially, this process is beneficial, because subsequent calls to memory will be satisfied by data that is already in the L1 cache. In this situation, the memory controller only needs to make one call to main memory for every eight values that are needed. However, when memory is accessed randomly, this performance benefit disappears. Out of the 64 bytes that are copied to the cache, only 8 bytes are actually

used by the processor. This yields an effective bandwidth that is lower by a factor of 8.

In addition, a random access pattern also decreases effective bandwidth to the L1 cache. The interface between the processor and L1 cache accomodates 128 bits of data in both the read and write directions. When the processor requests a value from L1, the cache provides the processor with 128 bits of data, which would be two double-precision numbers. When data is being accessed sequentially, the "extra" number can be used in the following iteration. When data is accessed randomly, however, only the first number is used, resulting an a factor of two decrease in effective bandwidth, as is observed in Table 5.

5.3 Compute Bound Problems

Problems not bound by bandwidth are often compute bound, meaning performance is limited by the rate at which the processor can perform arithmetic operations. In this situation, the processor is unable to perform the necessary operations on data as fast as the memory can deliver new data, resulting in a bottleneck in the compuational stage. This is commonly the case when a large amount of computation needs to be performed on a limited field of data. Common high performance computing taks that are considered compute-bound include random number generation and dynamic programming.

Because programs are typically bound by either bandwidth or computational ability, it is valuable for the programmer to have knowledge of the border between the two, so that they can determine the limiting factor in their algorithms and optimize for it. As such, we decided to modify our benchmark so that it could be used to determine this cutoff.

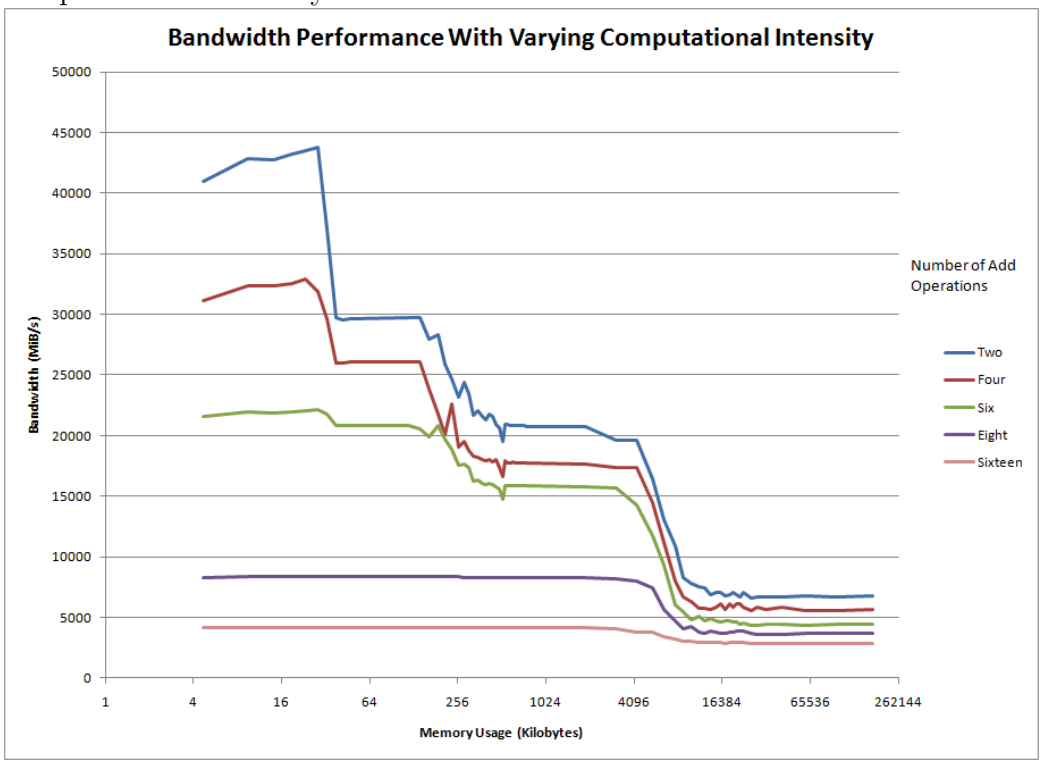
5.3.1 The Benchmark

To simulate this situation, the bandwidth-bound benchmark was progressively modified to reflect a more computationally intense kernel. In order to do this, the "add" routine from the stream benchmark was modified to include more double-precision floating point addition operations. The initial case of this modified benchmark featured two double-precision addition operations, with two more appended with each successive iteration.

5.3.2 Results

The modified benchmark was used to measure the bandwidth resulting from up to sixteen double-precision addition operations. The results of this test are reflected in Figure 7.

Figure 7: Bandwidth results for sequential access, one core, with increasing computational intensity



The results demonstrated in Figure 6 clearly show a decrease in bandwidth as the number of arithmetic operations increases. As the processor attempts to perform more and more operations, it eventually is unable to keep up with data delivered by the memory subsystem. From here, performance will continue to decrease with additional operations, but it will do so uniformly, regardless of problem size. At this point, the limiting factor is no longer memory bandwidth, but the computational speed of the processor.

As the measurements illustrate, even a relatively small number of computational operations will quickly erode the performance benefit provided by

the cache memory. As soon as eight or more operations are being performed, the benefit from the cache is nonexistent, and the program is effectively compute bound. The processor is no longer able to handle the data as fast as the main memory can provide it. Our benchmark allowed us to effectively observe the transition between a bandwidth-limited process and a compute-limited process.

6 Conclusions

CPU caching allows the Intel Nehalem architecture to achieve a much higher bandwidth than is otherwise possible, by enabling much quicker access to memory that is spatially local to previously accessed data. The bandwidth benefit from caching is greater when memory is accessed sequentially. Using OpenMP, programs can be parallelized, so that more than one core or processor are utilized at once. The bandwidth benefit from L1 and L2 caching scales in a linear fashion as the number of cores is increased, and the bandwidth benefit from L3 scales linearly as the number of processors is increased. Therefore, parallelization of an algorithm can yield a significant increase in effective bandwidth. Random memory access eliminates most of the benefit of caching, resulting in a decrease in bandwidth. As the computational intensity of a program increases, it becomes less limited by memory bandwidth and more limited by the processor's arithmetic performance. The greater the computational intensity becomes, the less benefit is experienced from caching, until a main memory call eventually takes less time than the processing of the data.

A working knowledge of the caching process, as well as the factors that limit the speed of an algorithm, including bandwidth and computational intensity, can allow a programmer to better optimize their code to maximize use of limited system resources. With this knowledge, strategies such as parallelization can be used effectively to improve algorithms and achieve faster runtimes.

7 Acknowledgement

This paper greatly benefitted from the prior work of Robert Schöne, Wolfgang E. Nagel, and Stefan Püger[3], as well as from the personal advice of Robert

Schoene at the Technische Universität Dresden. Stone Ridge Technology employees Dave Dembeck and Ken Esler also deserve credit for valuable ideas and input. This work would not have been possible without the prior work of John D. McCalpin in developing the STREAM benchmark.

References

[1] J. D. McCalpin, *Memory bandwidth and machine balance in current high performance computers*. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, 1995.

[2] Intel Corporation, *Intel Xeon Processor X5560 Website*, `http://ark.intel.com/Product.aspx?id=37109`

[3] Robert Schöne, Wolfgang E. Nagel, and Stefan Püger, *Analyzing Cache Bandwidth on the Intel Core 2 Architecture*. Parallel Computing: Architectures, Algorithms and Applications , 2007.

[4] J. D. McCalpin, *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. `http://www.cs.virginia.edu/stream/`