Lab Report

ECPE 170 – Computer Systems and Networks – Fall 2012

Name: Erich Viebrock

Lab Topic: Performance Optimization (Memory Hierarchy) (Lab #: 6)

Lab

Question #1:
Describe how a two-dimensional array is stored in one-dimensional computer memory.
Answer:
If `array[row][col]` is created, it is stored in one-dimensional computer memory by starting at `array[0][0]`, then increment the column, then increment the row. For example, the array would be stored as such:

First stored element: `array[0][0]`
Second stored element: `array[0][1]`
Third stored element: `array[1][0]`
Fourth stored element: `array[1][1]`

Question #2:
Describe how a three-dimensional array is stored in one-dimensional computer memory.
Answer:
A three-dimensional array uses the same concept as the two-dimensional array, but with an extra dimension, of course. The computer sees the rightmost element, increments that index to its maximum size, then increments the index to the left of the rightmost element, and so on.

Question #3:
Copy and paste the output of your program into your lab report, and be sure that the source code and Makefile is included in your Mercurial repository.
Answer:

```
In this program, a two dimensional (2x2) array is declared. Arrays
are declared by array[row][column], which we will step
through with a pointer and reveal the memory address of each
individual element. We will first increment the column, then
increment the row.

Below are the memory addresses of the array:

Address of array[0][0]: 0xbfe65760
Address of array[0][1]: 0xbfe65764
Address of array[1][0]: 0xbfe65768
Address of array[1][1]: 0xbfe6576c

As we suspected, a two dimensional array stores elements by
incrementing the column, then incrementing the row. Notice how
each memory address increments the least significant bit by a value
of 4 as we move through the array. This is because we
declared an integer array, which stores elements in 4 bytes each.
Also note when we have a least significant bit of 8, the
next memory address's least significant bit is 'c'. This is because
memory addresses are stored in hexadecimal, which uses
```

```
a = 10, b = 11, c = 12, d = 13 e = 14, and f = 15.

Now, let's do the same thing, but with a three dimensional (2x2x2)
array.

Below are the memory address of the array:

Address of array[0][0][0]: 0xbfe65740
Address of array[0][0][1]: 0xbfe65744
Address of array[0][1][0]: 0xbfe65748
Address of array[0][1][1]: 0xbfe6574c
Address of array[1][0][0]: 0xbfe65750
Address of array[1][0][1]: 0xbfe65754
Address of array[1][1][0]: 0xbfe65758
Address of array[1][1][1]: 0xbfe6575c

Notice how memory is stored in the same fashion as the two
dimensional array. Therefore, we can conclude the rightmost
element in the array is incremented first, then increments from right
to left. Although we only tested two and three
dimensional arrays, the same concept should apply to an array of any
dimension.
```

Question #4:
Provide an Access Pattern table for the `sumarrayrows()` function assuming ROWS=2 and COLS=3.
Answer:

Memory Address	0	4	8	12	16	20
Memory Contents	a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]
Program Access Order	1	2	3	4	5	6

Note: Address assumes array starts at memory address 0.

Question #5:
Does `sumarrayrows()` have good temporal or spatial locality?
Answer:
The function `sumarrayrows()` has good spatial locality, because the elements are accessed sequentially. Also, because the elements are only accessed once, the function has bad temporal locality.

Question #6:
Provide an Access Pattern table for the `sumarraycols()` function assuming ROWS=2 and COLS=3.
Answer:

Memory Address	0	4	8	12	16	20
Memory Contents	a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]
Program Access Order	1	3	5	2	4	6

Note: Address assumes array starts at memory address 0.

Question #7:
Does `sumarraycols()` have good temporal or spatial locality?
Answer:
The function `sumarrayrow()` has bad spatial locality, because the elements are not accessed sequentially. Also, because the elements are only accessed once, the function has bad temporal locality.

Question #8:
Provide an Access Pattern table for the `sumarray3d()` function assuming ROWS=2, COLS=3, and DEPTH=4.
Answer: *array[rows][cols][depth]*

Memory Address	0	4	8	12	16	20
Memory Contents	a[0][0][0]	a[0][0][1]	a[0][0][2]	a[0][0][3]	a[0][1][0]	a[0][1][1]
Program Access Order	1	3	5	7	9	11

Memory Address	24	28	32	36	40	44
Memory Contents	a[0][1][2]	a[0][1][3]	a[0][2][0]	a[0][2][1]	a[0][2][2]	a[0][2][3]
Program Access Order	13	15	17	19	21	23

Memory Address	48	52	56	60	64	68
Memory Contents	a[1][0][0]	a[1][0][1]	a[1][0][2]	a[1][0][3]	a[1][1][0]	a[1][1][1]
Program Access Order	2	4	6	8	10	12

Memory Address	72	76	80	84	88	92
Memory Contents	a[1][1][2]	a[1][1][3]	a[1][2][0]	a[1][2][1]	a[1][2][2]	a[1][2][3]
Program Access Order	14	16	18	20	22	24

Question #9:
Does `sumarray3d()` have good temporal or spatial locality?
Answer:
The function `sumarray3d()` has bad spatial locality, because the elements are not accessed sequentially. Also, because the elements are only accessed once, the function has bad temporal locality.

Question #10:
Imagine that the memory system **only** had main memory (no cache was present). Is temporal or spatial locality important for performance here when repeatedly accessing an array with 8-byte elements? Why or why not?
Answer:
Spatial locality is more important for performance since the memory system only has one level of memory. Therefore, all data would have equal temporal locality, whereas spatial locality can vary.

Question #11:
Imagine that the memory system had main memory **and a 1-level cache**, but each cache line size was **only 8 bytes** (64 bits) in size. Is temporal or spatial locality important for performance here when repeatedly accessing an array with 8-byte elements? Why or why not?
Answer:
Temporal locality is more important for performance because each element can be saved in an individual cache line. By saving each element in cache, the data can be accessed quicker.

Question #12:
Imagine that the memory system had main memory and a 1-level cache, and the cache line size was **64 bytes**. Is temporal or spatial locality important for performance here when repeatedly accessing an array with 8-byte elements? Why or why not?
Answer:
Temporal locality is more important for performance because each element can be saved in an individual cache line. By saving each element in cache, the data can be accessed quicker.

Question #13:
Imagine your program accesses a 100,000 element array (of 8 byte elements) once from beginning to end with stride 1. The memory system has a 1-level cache with a line size of 64 bytes. No pre-fetching is implemented. How many cache misses would be expected in this system?
Answer:
No cache misses would be expected to miss in this system, assuming the cache has 100,000 lines. Otherwise, there would be 100,000 – (total cache lines) misses.

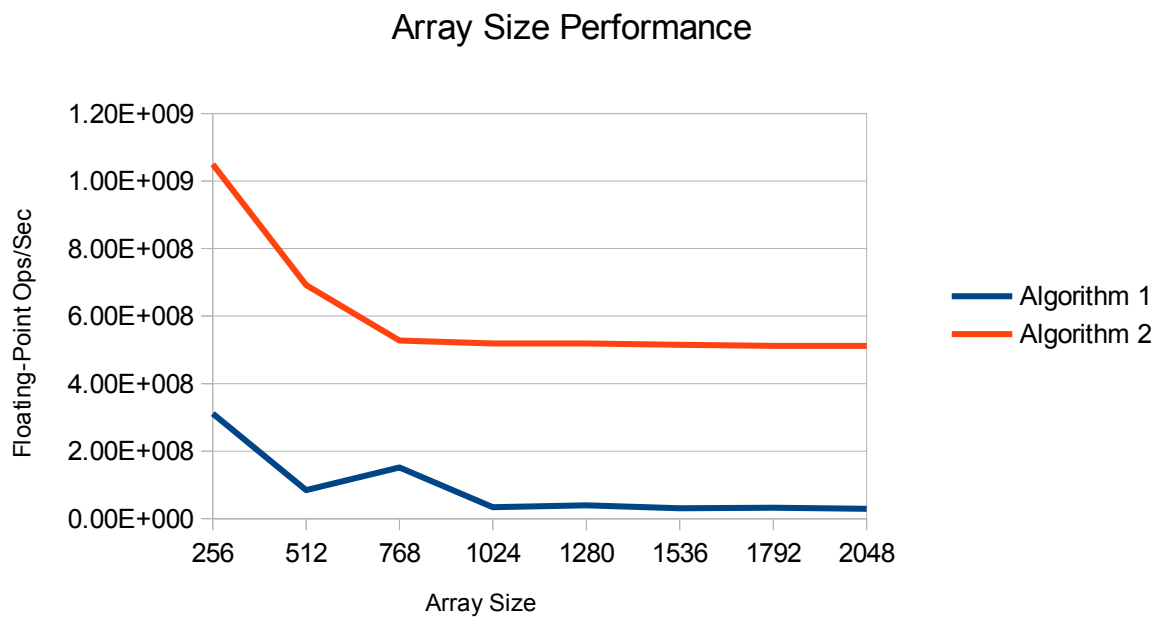
Question #14:
Imagine your program accesses a 100,000 element array (of 8 byte elements) once from beginning to end with stride 1. The memory system has a 1-level cache with a line size of 64 bytes. A hardware prefetcher is implemented. In the *best-possible case*, how many cache misses would be expected in this system? Feel free to make assumptions like "the hardware prefetcher runs fast enough to stay ahead of the user program."
Answer:
No cache misses would be expected to miss in this system, assuming the cache has 100,000 lines. Otherwise, there would be 100,000 – (total cache lines) misses. A hardware prefetcher would do nothing to enhance cache misses, but would improve spatial locality. Therefore, the overall performance of the system would improve.

Question #15:
Inspect the provided source code. Describe how the *two*-dimensional arrays are stored in memory, since the code only has one-dimensional array accesses like: `a[element #]`.
Answer:
The arrays are created with a call to `malloc`, which creates heap memory. Then, the two dimensional array is created by the user input of `n`. Because the number of indexes is equal between rows and columns, `n` defines the size of rows and columns.

Question #16:
After running your experiment script, create a table that shows floating point operations per second for both algorithms at the array sizes listed in Table 2.
Answer:

Algorithm 1		Algorithm 2	
Array Size	Floating-point ops/sec	Array Size	Floating-point ops/sec
256	3.11E+008	256	1.05E+009
512	8.47E+007	512	6.92E+008
768	1.52E+008	768	5.28E+008
1024	3.40E+007	1024	5.19E+008
1280	3.96E+007	1280	5.19E+008
1536	3.13E+007	1536	5.15E+008
1792	3.28E+007	1792	5.12E+008
2048	2.96E+007	2048	5.12E+008

Question #17:
After running your experiment script, create a graph that shows floating point operations per second for both algorithms at the array sizes listed in Table 2.
Answer:



Question #18:
Be sure that the script source code is included in your Mercurial repository.
Answer:
OK.

Post-Lab

Question #1:
Place the output of `/proc/cpuinfo` in your report. *(I only need to see one processor core, not all the cores as reported).*
Answer:

```
processor : 0
vendor_id : GenuineIntel
cpu family      : 6
model           : 13
model name      : Intel(R) Pentium(R) M processor 2.00GHz
stepping  : 8
microcode : 0x20
cpu MHz         : 800.000
cache size      : 2048 KB
fdiv_bug  : no
hlt_bug         : no
f00f_bug  : no
coma_bug  : no
fpu       : yes
fpu_exception  : yes
cpuid level    : 2
wp        : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge mca
cmov clflush dts acpi mmx fxsr sse sse2 ss tm pbe nx up bts est tm2
bogomips  : 1596.07
clflush size   : 64
cache_alignment     : 64
address sizes  : 32 bits physical, 32 bits virtual
power management:
```

Question #2:
Based on the processor type reported, obtain the following specifications for your CPU from cpu-world.com or cpudb.stanford.edu
You might have to settle for a close processor from the same family. Make sure the frequency and L3 cache size **match** the results from /proc/cpuinfo!
(a) L1 instruction cache size
(b) L1 data cache size
(c) L2 cache size
(d) L3 cache size
(e) What URL did you obtain the above specifications from?
Answer:
(a) 32
(b) 32
(c) 2048
(d) None, lol.

(e) Intel Pentium M "Dothan", 2000.0 MHz

Question #3:
Why is it important to run the test program on an idle computer system?
Explain what would happen if the computer was running several other active programs in the background at the same time, and how that would impact the test results.
Answer:
As displayed on the memory mountain, the red areas indicate intensive memory usage. In order to assure the most accurate test results of a program measuring bandwidth, the computer system can not be bottlenecked of memory because of other programs running. If the computer was running several programs in the background at the same time, this could cause our program to run short of main memory, and need to write to the hard drive in order to continue computing. This process significantly decreases efficiency, and produces inaccurate results.

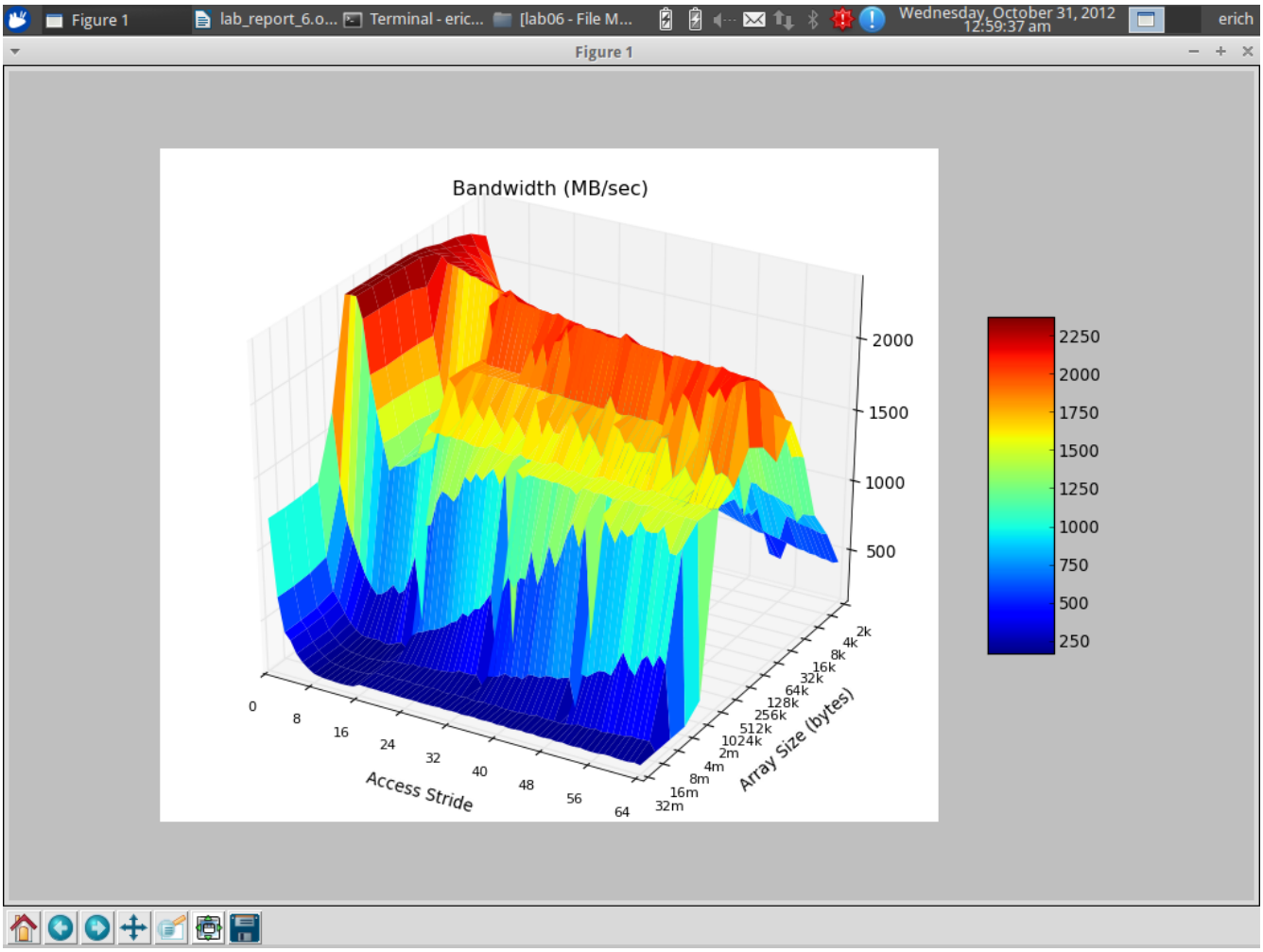
Question #4:
What is the size (in bytes) of a data element read from the array in the test?
Answer:
The size of a data element is 8 bytes.

Question #5:
What is the range (min, max) of *strides* used in the test?
Answer:
The minimum number of strides in the test is 1, while the maximum number of strides is 64.

Question #6:
What is the range (min, max) of *array sizes* used in the test?
Answer:
The minimum number array size in the test is 2,000, while the maximum array size is 32,000,000.

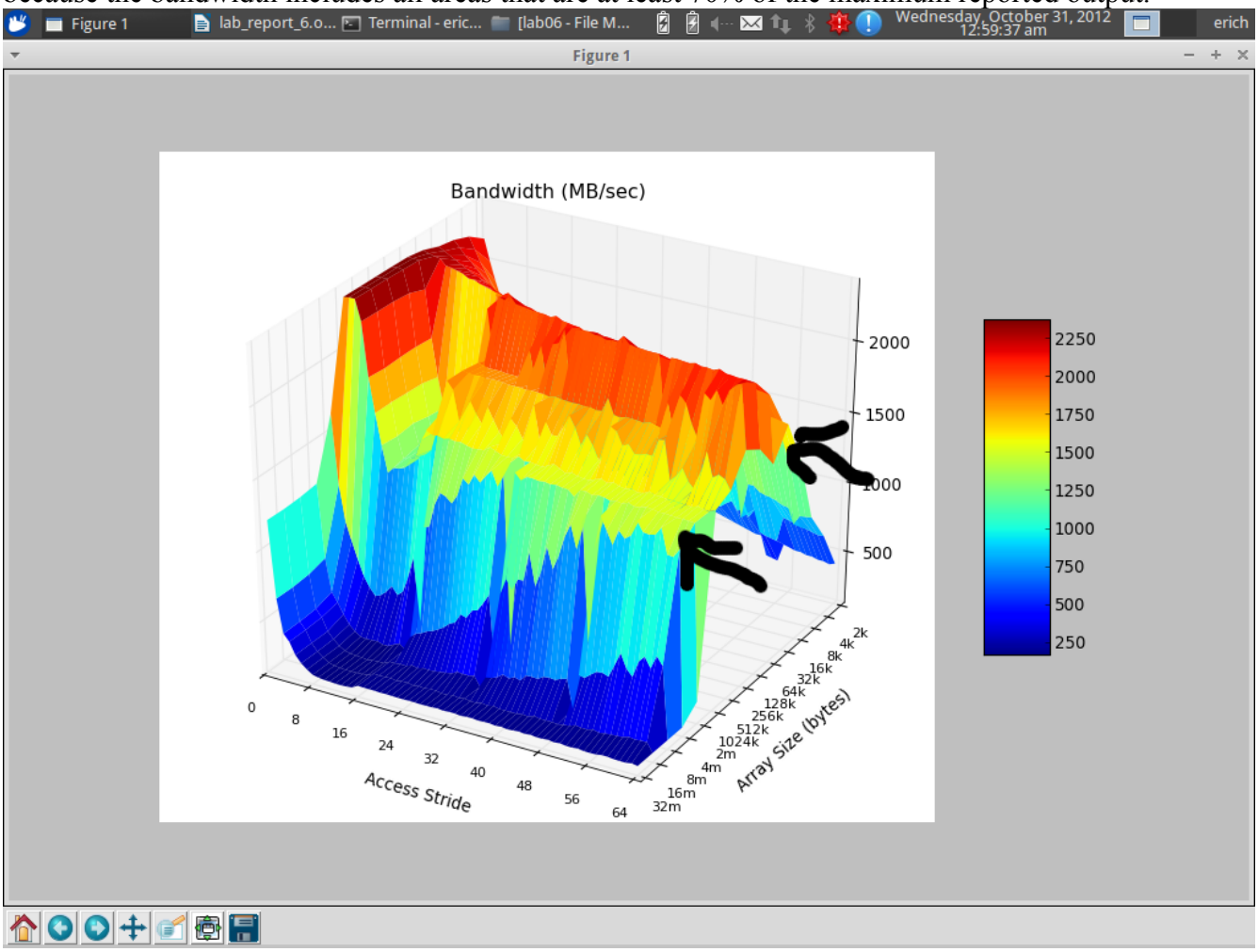
Question #7:
Take a screen capture of the displayed "memory mountain" (maximize the window so it's sufficiently large to read), and place the resulting image in your report

Answer:



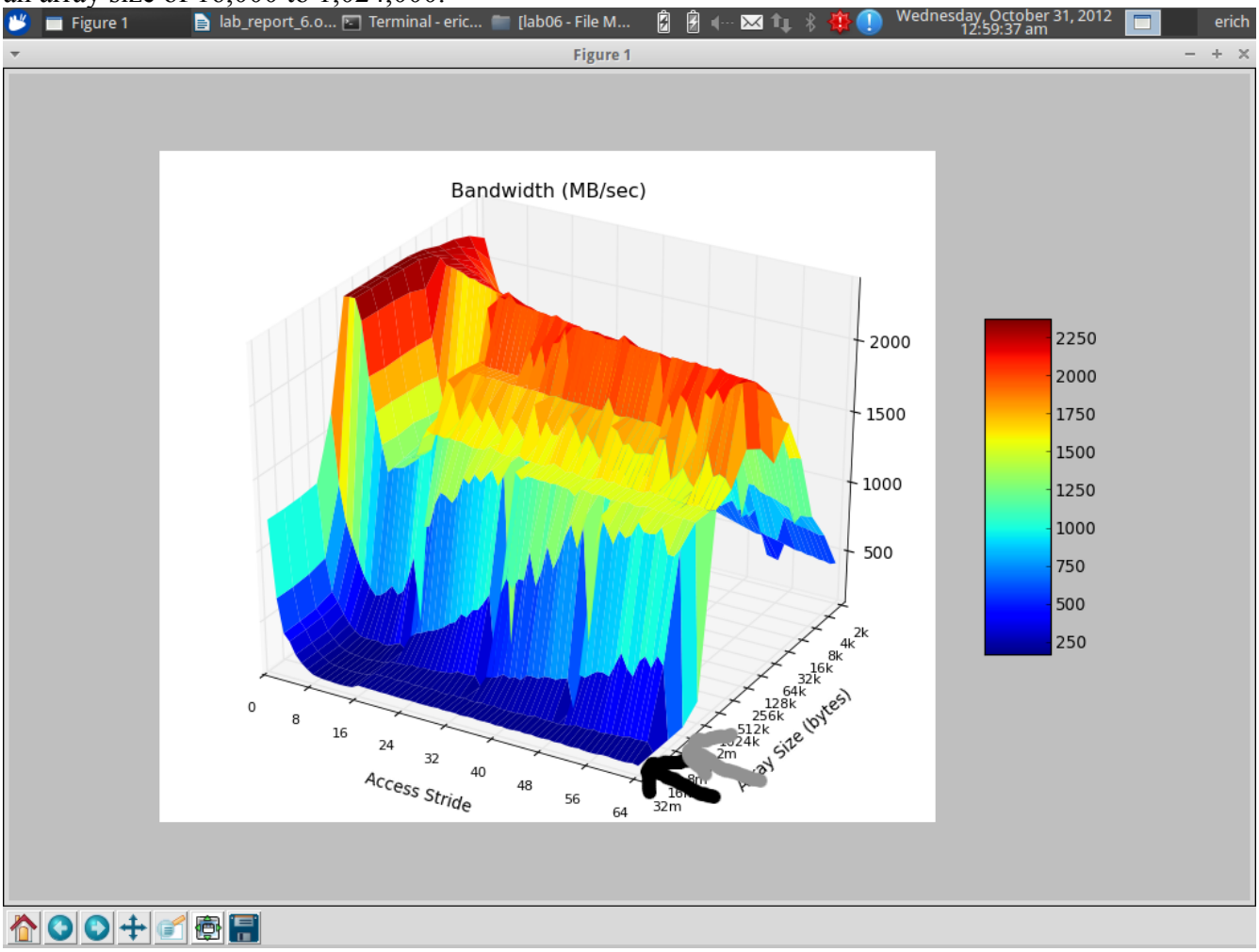
Question #8:
What region (array size, stride) gets the most **consistently** high performance? (Ignoring spikes in the graph that are noisy results...) What is the read bandwidth reported? Annotate your figure by drawing an arrow on it.

Answer:
The region of 8 strides and an array size of about 16,000 to 1,024,000 gets the most consistent high performance. However, the read bandwidth is reported as a region of 8 to 64 strides, and an array size of 16,000 to 1,024,000. The bandwidth is different than the most consistent high performance region, because the bandwidth includes all areas that are at least 70% of the maximum reported output.

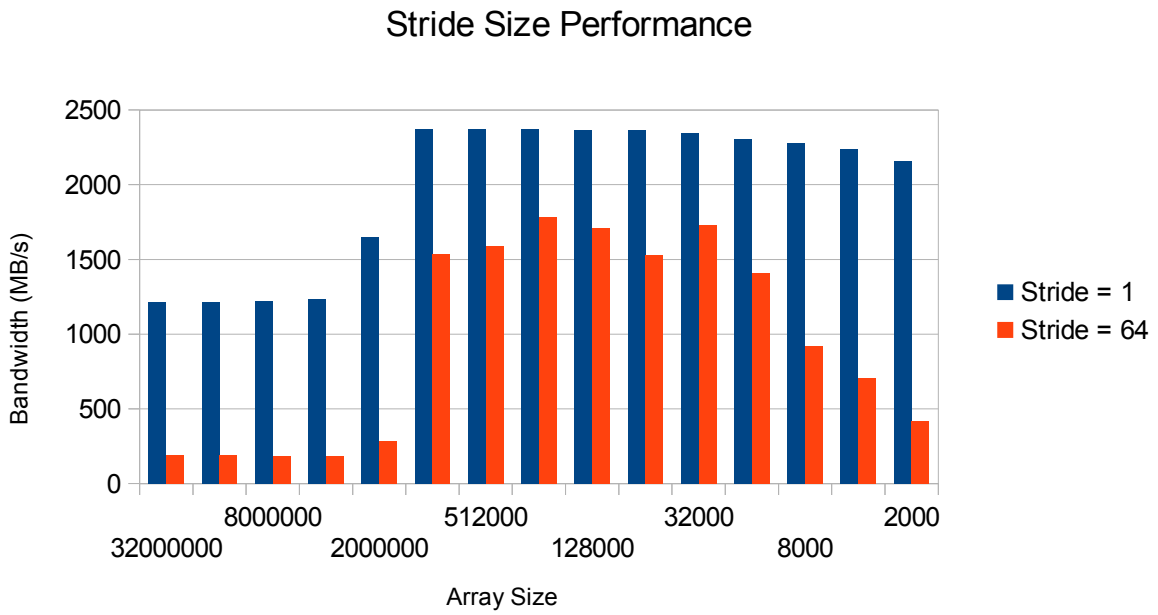


Question #9:
What region (array size, stride) gets the most **consistently** low performance? (Ignoring spikes in the graph that are noisy results...) What is the read bandwidth reported? Annotate your figure by drawing an arrow on it.

Answer:
The region of 8 to 64 strides and an array size of about 4,000,000 to 16,000,000 gets the most consistent low performance. However, the read bandwidth is reported as a region of 8 to 64 strides, and an array size of 16,000 to 1,024,000.



Question #10:
Using LibreOffice calc, create two new bar graphs: One for stride=1, and the other for stride=64. Place them side-by-size in the report.
No credit will be given for sloppy graphs that lack X and Y axis labels and a title.
You can obtain the raw data from results.txt. Open it in gedit and turn off *Text Wrapping* in the preferences. (Otherwise, the columns will be a mess)
Answer:



Question #11:
When you look at the graph for stride=1, you (should) see relatively high performance compared to stride=64. This is true even for large array sizes that are much larger than the L3 cache size reported in /proc/cpuinfo.
How is this possible, when the array cannot possibly all fit into the cache? Your explanation should include a brief overview of hardware prefetching as it applies to caches.
Answer:
This is possible because the prefetcher loads the cache lines into cache before the program runs. Because the array cannot possibly fit into the cache, the most accessed elements are stored in cache, whereas the remainder elements are stored in main memory.

Question #12:
What is temporal locality? What is spatial locality?
Answer:
Temporal locality is placing a frequently accessed data element in a quickly accessible location. Spatial locality is placing frequently accessed data elements in a sequential fashion, to make them easily and quickly accessible. In order to have a memory optimized program, both techniques must be used.

Question #13:
Adjusting the total array size impacts temporal locality - why? Will an increased array size increase or decrease temporal locality?
Answer:
Adjusting the total array size impacts temporal locality because the initial array size will be saved sequentially, but will most likely have other data elements saved in the next slots of memory afterwards. Then when the array size is adjusted, the new indexes will have no choice but to be saved elsewhere in memory, therefore causing the entire array to not be saved sequentially. Thus, increasing array size decreases temporal locality.

Question #14:
Adjusting the read *stride* impacts spatial locality - why? Will an increased read stride increase or decrease spatial locality?
Answer:
Adjusting the read stride depicts how much information the CPU can read from memory in a single instance. As read stride increases, the probability of reading data elements sequentially decreases. Therefore, having a low read stride will significantly increase program performance.

Question #15:
As a software design, describe at least 2 broad "guidelines" to ensure your programs run in the high-performing region of the graph instead of the low-performing region.
Answer:
The two guidelines to follow to ensure a high-performing program would be to highly emphasize temporal and spatial locality. According to the graph, this is done by keeping access strides low (about 4-8), and an array size of about 16,000 to 1,024,000.

Post-Lab Wrapup

Question #1:
What was the best aspect of this lab?
Answer:
Unveiling the mystery of how memory and cache memory works.

Question #2:
What was the worst aspect of this lab?
Answer:
Very extensive.

Question #3:
How would you suggest improving this lab in future semesters?
Answer:
Make lab more concise.