Lab Report

ECPE 170 – Computer Systems and Networks – Spring 2016

Name: Drew Overgaard

Lab Topic: Performance Optimization (Memory Hierarchy) (Lab #: 07)

Question #1:
Describe how a two-dimensional array is stored in one-dimensional computer memory.
Answer:
A two dimensional array is stored in memory by first storing all the rows in sequential memory locations. For example, location 0 0 would be 0, location 0 1, would be 1, 0 2, 2 and so on then if we go to the second row 1, 0 would be 3. My program demonstrates how this is accomplished with both the two and three dimensional arrays.

Question #2:
Describe how a three-dimensional array is stored in one-dimensional computer memory.
Answer:
The tree dimensional array is stored similar to the two dimensional array. The only difference is that instead of 0 0, you have 0 0 0 for the first element in the array. So, 0 0 0 would be 0, 0 0 1, 0 1 1 would be 3.

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:
Now setting each space in the two dimensional array equal to a value.
Printing each spaces memory location aswell.
Now printing space: 0 0 Memory Address: 0x7fff0e27a4f0
Now printing space: 0 1 Memory Address: 0x7fff0e27a4f4
Now printing space: 0 2 Memory Address: 0x7fff0e27a4f8
Now printing space: 0 3 Memory Address: 0x7fff0e27a4fc
Now printing space: 0 4 Memory Address: 0x7fff0e27a500

Now printing space: 1 0 Memory Address: 0x7fff0e27a504
Now printing space: 1 1 Memory Address: 0x7fff0e27a508
Now printing space: 1 2 Memory Address: 0x7fff0e27a50c
Now printing space: 1 3 Memory Address: 0x7fff0e27a510
Now printing space: 1 4 Memory Address: 0x7fff0e27a514

Now printing space: 2 0 Memory Address: 0x7fff0e27a518
Now printing space: 2 1 Memory Address: 0x7fff0e27a51c
Now printing space: 2 2 Memory Address: 0x7fff0e27a520
Now printing space: 2 3 Memory Address: 0x7fff0e27a524
Now printing space: 2 4 Memory Address: 0x7fff0e27a528

Now setting each space in the three dimensional array equal to a value.
Printing each spaces memory location aswell.
Now printing space: 0 0 0 Memory Address: 0x7fff0e27a530
Now printing space: 0 0 1 Memory Address: 0x7fff0e27a534
Now printing space: 0 0 2 Memory Address: 0x7fff0e27a538
Now printing space: 0 0 3 Memory Address: 0x7fff0e27a53c
Now printing space: 0 0 4 Memory Address: 0x7fff0e27a540

Now printing space: 0 0 5 Memory Address: 0x7fff0e27a544
Now printing space: 0 0 6 Memory Address: 0x7fff0e27a548
Now printing space: 0 1 0 Memory Address: 0x7fff0e27a54c
Now printing space: 0 1 1 Memory Address: 0x7fff0e27a550
Now printing space: 0 1 2 Memory Address: 0x7fff0e27a554
Now printing space: 0 1 3 Memory Address: 0x7fff0e27a558
Now printing space: 0 1 4 Memory Address: 0x7fff0e27a55c
Now printing space: 0 1 5 Memory Address: 0x7fff0e27a560
Now printing space: 0 1 6 Memory Address: 0x7fff0e27a564
Now printing space: 0 2 0 Memory Address: 0x7fff0e27a568
Now printing space: 0 2 1 Memory Address: 0x7fff0e27a56c
Now printing space: 0 2 2 Memory Address: 0x7fff0e27a570
Now printing space: 0 2 3 Memory Address: 0x7fff0e27a574
Now printing space: 0 2 4 Memory Address: 0x7fff0e27a578
Now printing space: 0 2 5 Memory Address: 0x7fff0e27a57c
Now printing space: 0 2 6 Memory Address: 0x7fff0e27a580
Now printing space: 0 3 0 Memory Address: 0x7fff0e27a584
Now printing space: 0 3 1 Memory Address: 0x7fff0e27a588
Now printing space: 0 3 2 Memory Address: 0x7fff0e27a58c
Now printing space: 0 3 3 Memory Address: 0x7fff0e27a590
Now printing space: 0 3 4 Memory Address: 0x7fff0e27a594
Now printing space: 0 3 5 Memory Address: 0x7fff0e27a598
Now printing space: 0 3 6 Memory Address: 0x7fff0e27a59c
Now printing space: 0 4 0 Memory Address: 0x7fff0e27a5a0
Now printing space: 0 4 1 Memory Address: 0x7fff0e27a5a4
Now printing space: 0 4 2 Memory Address: 0x7fff0e27a5a8
Now printing space: 0 4 3 Memory Address: 0x7fff0e27a5ac
Now printing space: 0 4 4 Memory Address: 0x7fff0e27a5b0
Now printing space: 0 4 5 Memory Address: 0x7fff0e27a5b4
Now printing space: 0 4 6 Memory Address: 0x7fff0e27a5b8

Now printing space: 1 0 0 Memory Address: 0x7fff0e27a5bc
Now printing space: 1 0 1 Memory Address: 0x7fff0e27a5c0
Now printing space: 1 0 2 Memory Address: 0x7fff0e27a5c4
Now printing space: 1 0 3 Memory Address: 0x7fff0e27a5c8
Now printing space: 1 0 4 Memory Address: 0x7fff0e27a5cc
Now printing space: 1 0 5 Memory Address: 0x7fff0e27a5d0
Now printing space: 1 0 6 Memory Address: 0x7fff0e27a5d4
Now printing space: 1 1 0 Memory Address: 0x7fff0e27a5d8
Now printing space: 1 1 1 Memory Address: 0x7fff0e27a5dc
Now printing space: 1 1 2 Memory Address: 0x7fff0e27a5e0
Now printing space: 1 1 3 Memory Address: 0x7fff0e27a5e4
Now printing space: 1 1 4 Memory Address: 0x7fff0e27a5e8
Now printing space: 1 1 5 Memory Address: 0x7fff0e27a5ec
Now printing space: 1 1 6 Memory Address: 0x7fff0e27a5f0
Now printing space: 1 2 0 Memory Address: 0x7fff0e27a5f4
Now printing space: 1 2 1 Memory Address: 0x7fff0e27a5f8
Now printing space: 1 2 2 Memory Address: 0x7fff0e27a5fc
Now printing space: 1 2 3 Memory Address: 0x7fff0e27a600

Now printing space: 1 2 4 Memory Address: 0x7fff0e27a604
Now printing space: 1 2 5 Memory Address: 0x7fff0e27a608
Now printing space: 1 2 6 Memory Address: 0x7fff0e27a60c
Now printing space: 1 3 0 Memory Address: 0x7fff0e27a610
Now printing space: 1 3 1 Memory Address: 0x7fff0e27a614
Now printing space: 1 3 2 Memory Address: 0x7fff0e27a618
Now printing space: 1 3 3 Memory Address: 0x7fff0e27a61c
Now printing space: 1 3 4 Memory Address: 0x7fff0e27a620
Now printing space: 1 3 5 Memory Address: 0x7fff0e27a624
Now printing space: 1 3 6 Memory Address: 0x7fff0e27a628
Now printing space: 1 4 0 Memory Address: 0x7fff0e27a62c
Now printing space: 1 4 1 Memory Address: 0x7fff0e27a630
Now printing space: 1 4 2 Memory Address: 0x7fff0e27a634
Now printing space: 1 4 3 Memory Address: 0x7fff0e27a638
Now printing space: 1 4 4 Memory Address: 0x7fff0e27a63c
Now printing space: 1 4 5 Memory Address: 0x7fff0e27a640
Now printing space: 1 4 6 Memory Address: 0x7fff0e27a644

Now printing space: 2 0 0 Memory Address: 0x7fff0e27a648
Now printing space: 2 0 1 Memory Address: 0x7fff0e27a64c
Now printing space: 2 0 2 Memory Address: 0x7fff0e27a650
Now printing space: 2 0 3 Memory Address: 0x7fff0e27a654
Now printing space: 2 0 4 Memory Address: 0x7fff0e27a658
Now printing space: 2 0 5 Memory Address: 0x7fff0e27a65c
Now printing space: 2 0 6 Memory Address: 0x7fff0e27a660
Now printing space: 2 1 0 Memory Address: 0x7fff0e27a664
Now printing space: 2 1 1 Memory Address: 0x7fff0e27a668
Now printing space: 2 1 2 Memory Address: 0x7fff0e27a66c
Now printing space: 2 1 3 Memory Address: 0x7fff0e27a670
Now printing space: 2 1 4 Memory Address: 0x7fff0e27a674
Now printing space: 2 1 5 Memory Address: 0x7fff0e27a678
Now printing space: 2 1 6 Memory Address: 0x7fff0e27a67c
Now printing space: 2 2 0 Memory Address: 0x7fff0e27a680
Now printing space: 2 2 1 Memory Address: 0x7fff0e27a684
Now printing space: 2 2 2 Memory Address: 0x7fff0e27a688
Now printing space: 2 2 3 Memory Address: 0x7fff0e27a68c
Now printing space: 2 2 4 Memory Address: 0x7fff0e27a690
Now printing space: 2 2 5 Memory Address: 0x7fff0e27a694
Now printing space: 2 2 6 Memory Address: 0x7fff0e27a698
Now printing space: 2 3 0 Memory Address: 0x7fff0e27a69c
Now printing space: 2 3 1 Memory Address: 0x7fff0e27a6a0
Now printing space: 2 3 2 Memory Address: 0x7fff0e27a6a4
Now printing space: 2 3 3 Memory Address: 0x7fff0e27a6a8
Now printing space: 2 3 4 Memory Address: 0x7fff0e27a6ac
Now printing space: 2 3 5 Memory Address: 0x7fff0e27a6b0
Now printing space: 2 3 6 Memory Address: 0x7fff0e27a6b4
Now printing space: 2 4 0 Memory Address: 0x7fff0e27a6b8
Now printing space: 2 4 1 Memory Address: 0x7fff0e27a6bc
Now printing space: 2 4 2 Memory Address: 0x7fff0e27a6c0

Now printing space: 2 4 3 Memory Address: 0x7fff0e27a6c4
Now printing space: 2 4 4 Memory Address: 0x7fff0e27a6c8
Now printing space: 2 4 5 Memory Address: 0x7fff0e27a6cc
Now printing space: 2 4 6 Memory Address: 0x7fff0e27a6d0

As can be seen from the output, each location in memory is allocated sequentially.
So the entire row is allocated first then the next row and the next.
The memory address of the location where each sequential array element is stored increments by 4.

Question #4:
Provide an Access Pattern table for the sumarrayrows() function assuming ROWS=2 and COLS=3.
The table should be sorted by ascending memory addresses, not by program access order.
Answer:

Memory addresses	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	0	1	2	3	4	5

Question #5:
Does sumarrayrows() have good temporal or spatial locality?
For your answer to receive full credit, you must discuss the locality of both the array itself, and the scalar variables such as i that are present in the function.
Answer:
The variables I, j and , sum all have temporal locality since their accessed on each pass of the loop. There is no way to know if these variables have good spatial locality because there scalar variables. The array itself has good spatial locality because elements are accessed in order. Like the other function, elements are only accessed once so this one also has poor 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[1][0]	a[1][1]	a[2][0]	a[2][1]
Program Access Order	1	2	3	4	5	6

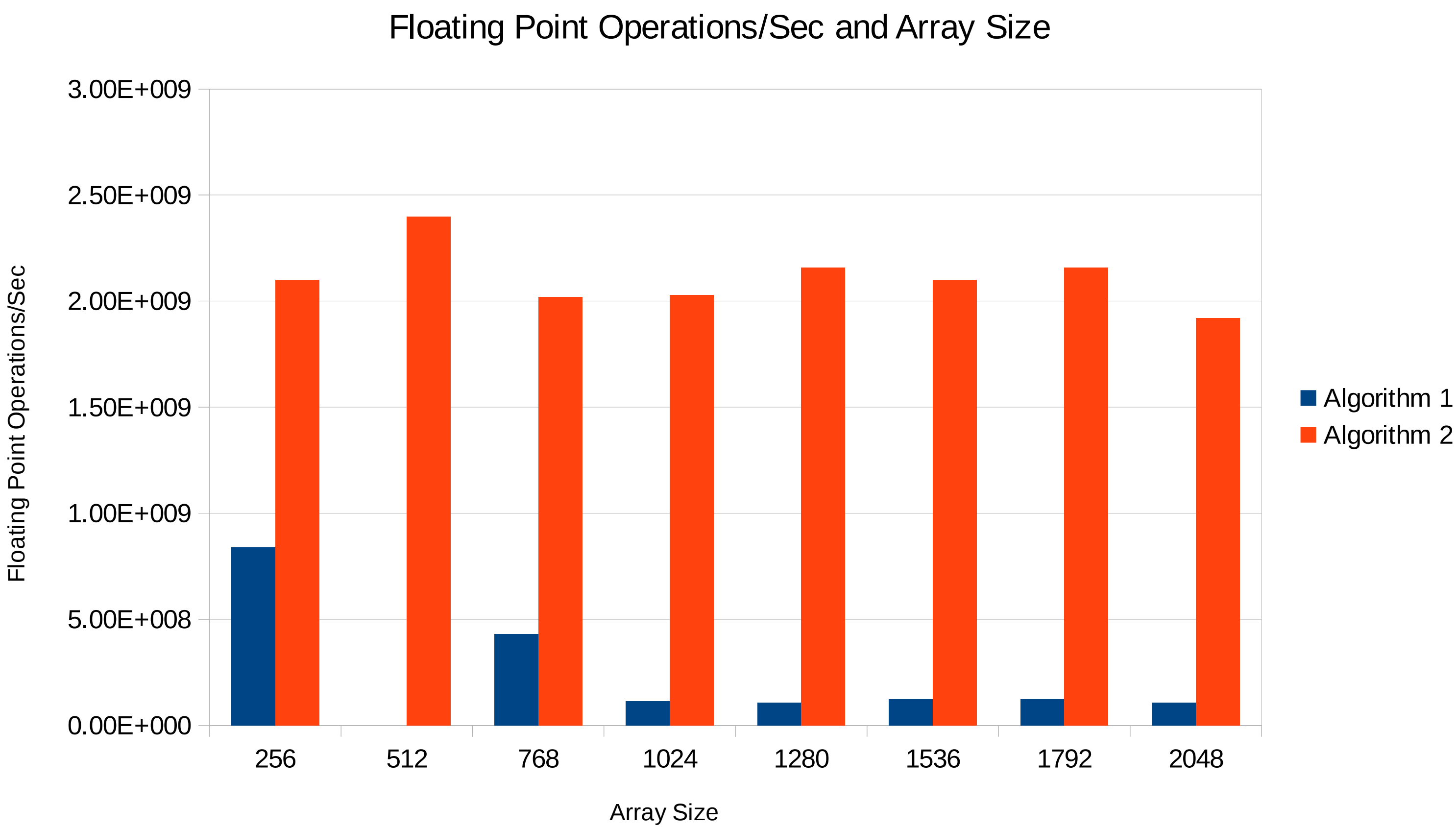
Question #7:
Does sumarraycols() have good temporal or spatial locality?
For your answer to receive full credit, you must discuss the locality of both the array itself, and the scalar variables such as i that are present in the function.
Answer:
The variables I, j and , sum all have temporal locality since their accessed on each pass of the loop. There is no way to know if these variables have good spatial locality because there scalar variables. The array itself has bad spatial locality because elements are not accessed in order. Like the other function, elements are only accessed once so this one also has poor temporal locality.

Question #8:
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:
Two dimensional arrays are stored sequentially in memory. That's how the arrays in the program will be stored.

Question #9:
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:

Array Size	Algorithm 1 Floating-point ops/sec	Algorithm 2 Floating-point ops/sec
256	8.39E+08	2.10E+09
512	3.69E+08	2.40E+09
768	4.31E+08	2.02E+09
1024	1.13E+08	2.03E+09
1280	1.08E+08	2.16E+09
1536	1.23E+08	2.10E+09
1792	1.23E+08	2.16E+09
2048	1.09E+08	1.92E+09

Question #10:
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 #11:
Be sure that the script source code is included in your Mercurial repository.
Answer:
I have included the script source code in the Lab 3 folder. It's named “script.py”.

Question #12:
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 : 69
model name : Intel(R) Core(TM) i5-4300U CPU @ 1.90GHz
stepping : 1
microcode : 0x1d
cpu MHz : 2501.000
cache size : 3072 KB
physical id : 0
siblings : 1
core id : 0
cpu cores : 1
apicid : 0
initial apicid : 0
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts mmx fxsr sse sse2 ss syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts nopl xtopology tsc_reliable nonstop_tsc aperfmperf eagerfpu pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm ida arat epb pln pts dtherm fsgsbase tsc_adjust bmi1 avx2 smep bmi2 invpcid xsaveopt
bugs :
bogomips : 5002.00
clflush size : 64
cache_alignment : 64
address sizes : 42 bits physical, 48 bits virtual
power management:

Question #13:
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!
Answer:
(a) L1 instruction cache size
2 x 32 KB 8-way set associative instruction caches
(b) L1 data cache size
2 x 32 KB 8-way set associative data caches
(c) L2 cache size
2 x 256 KB 8-way set associative caches
(d) L3 cache size

3 MB 12-way set associative shared cache
(e) What URL did you obtain the above specifications from?
http://www.cpu-world.com/CPUs/Core_i5/Intel-Core%20i5-4300U%20Mobile%20processor.html

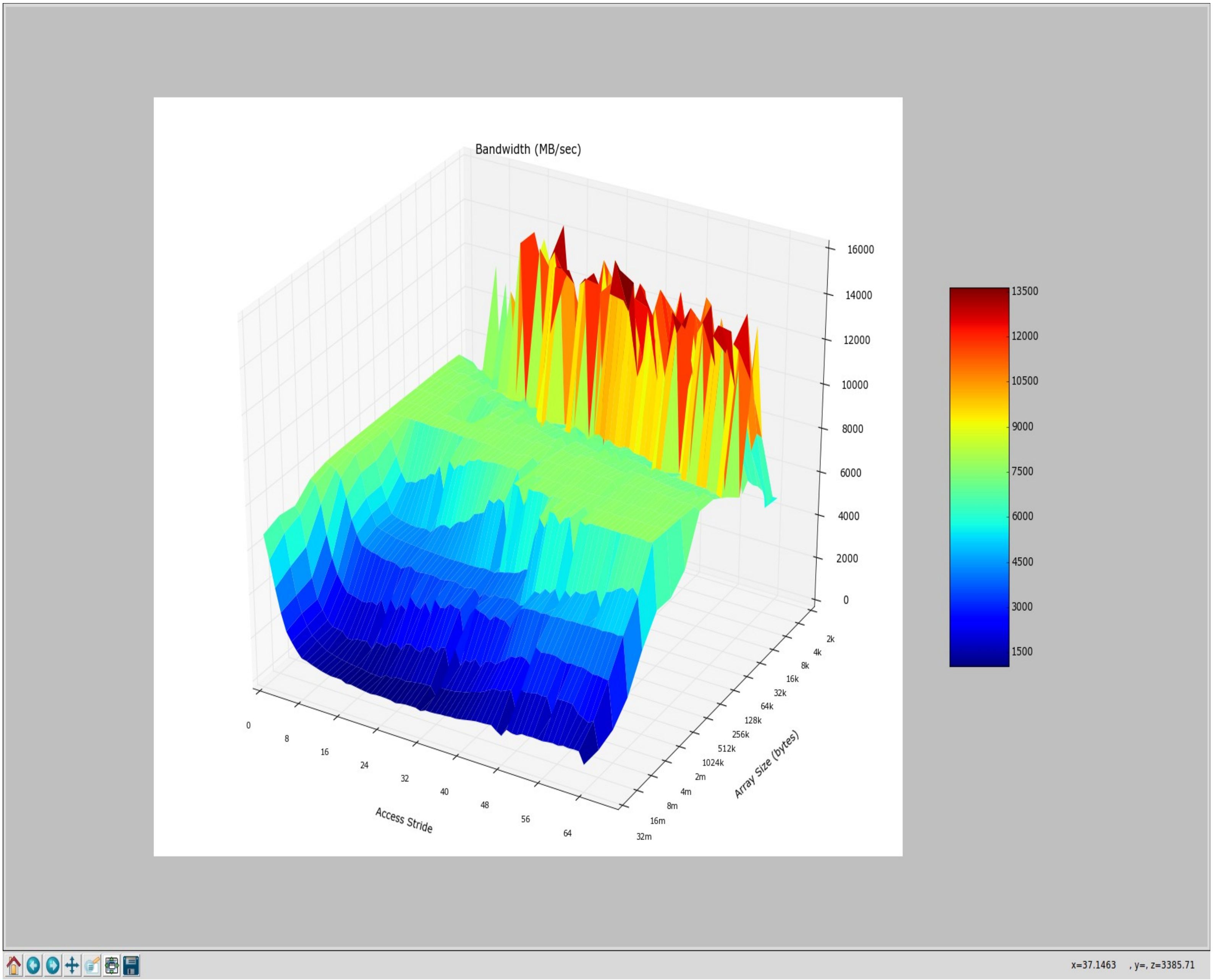
Question #14:
Why is it important to run the test program on an idle computer system?
Explain what would happen if the computer was running several other active programs in the background at the same time, and how that would impact the test results.
Answer:
It's important to run the test on a machine that is not running any other software because running software in the background takes memory and cpu power, which in a perfect environment with nothing running in th background would not be used and we would see better results. The results are skewed due to running the test inside of a virtual machine with programs and a host OS running.

Question #15:
What is the size (in bytes) of a data element read from the array in the test?
Answer:
The size (in bytes) of a data element read from the array in the test is 32MB.

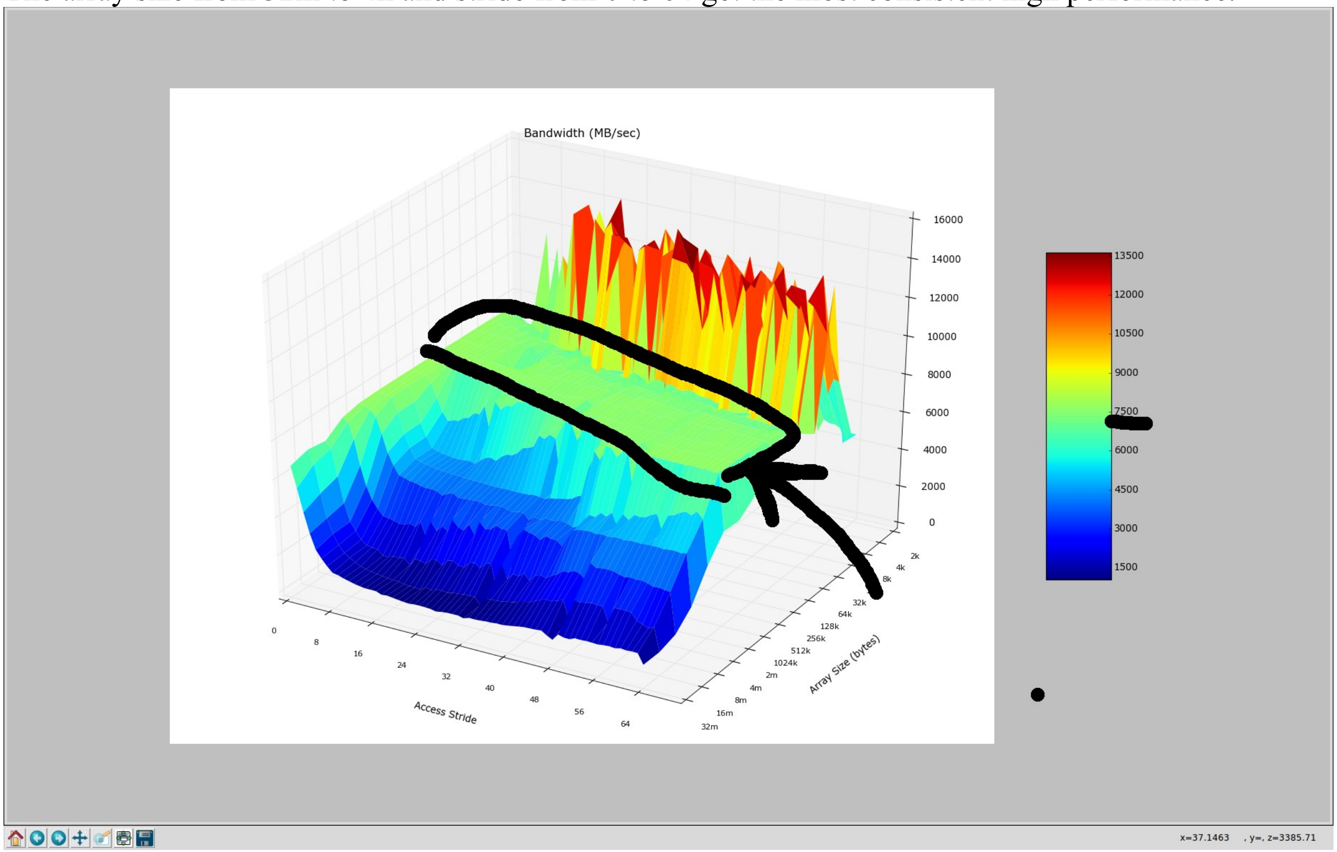
Question #16:
What is the range (min, max) of *strides* used in the test?
Answer:
The range of strides used in the test is 0 to 64.

Question #17:
What is the range (min, max) of *array sizes* used in the test?
Answer:
The range of array sizes used in the test is 32m to 2k.

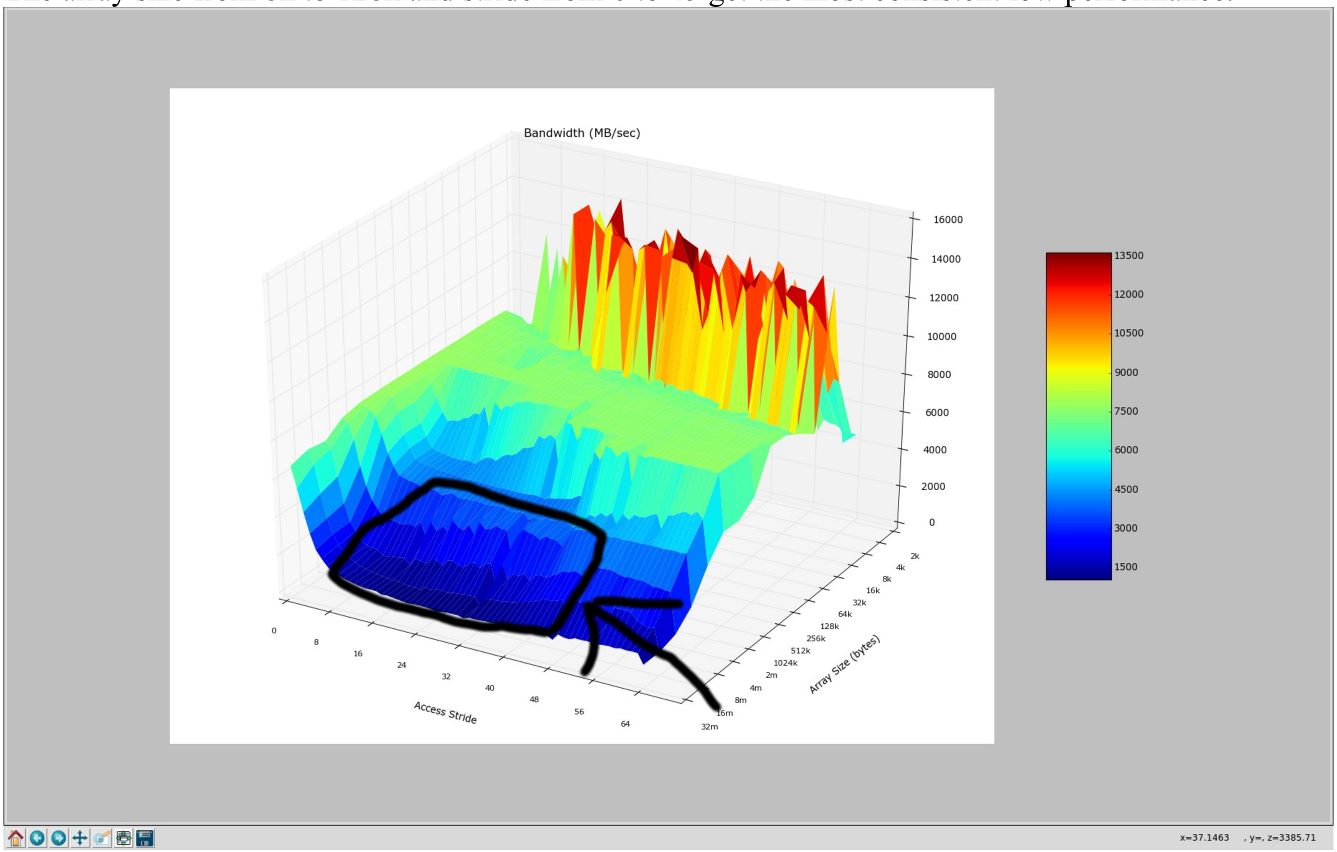
Question #18:
Take a screen capture of the displayed "memory mountain" (maximize the window so it's sufficiently large to read), and place the resulting image in your report
Answer:



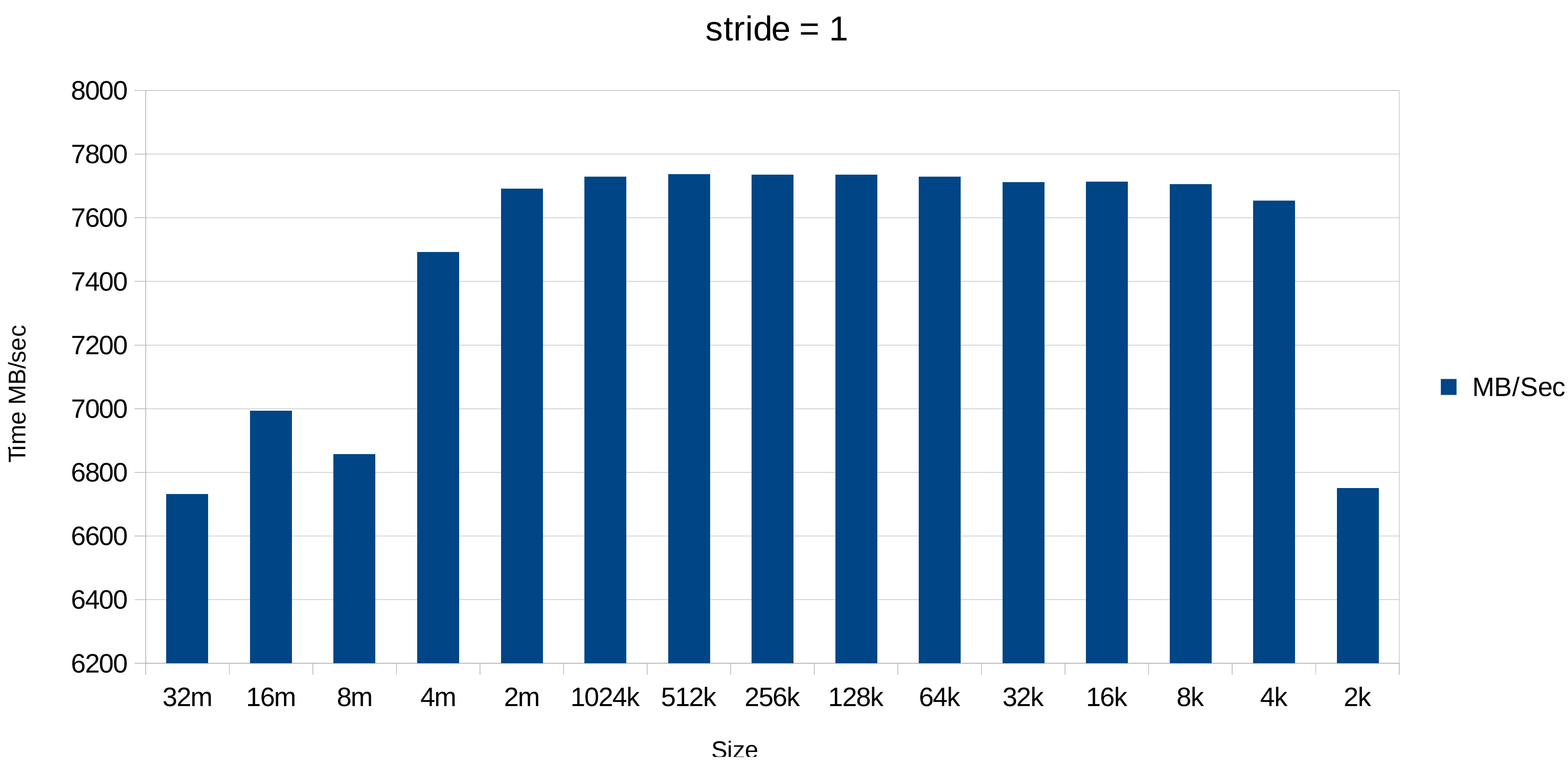
Question #19:
What region (array size, stride) gets the most **consistently** high performance? (Ignoring spikes in the graph that are noisy results...) What is the read bandwidth reported? Annotate your figure by drawing an arrow on it.
Answer:
The array size from 32m to 4k and stride from 0 to 64 get the most consistent high performance.

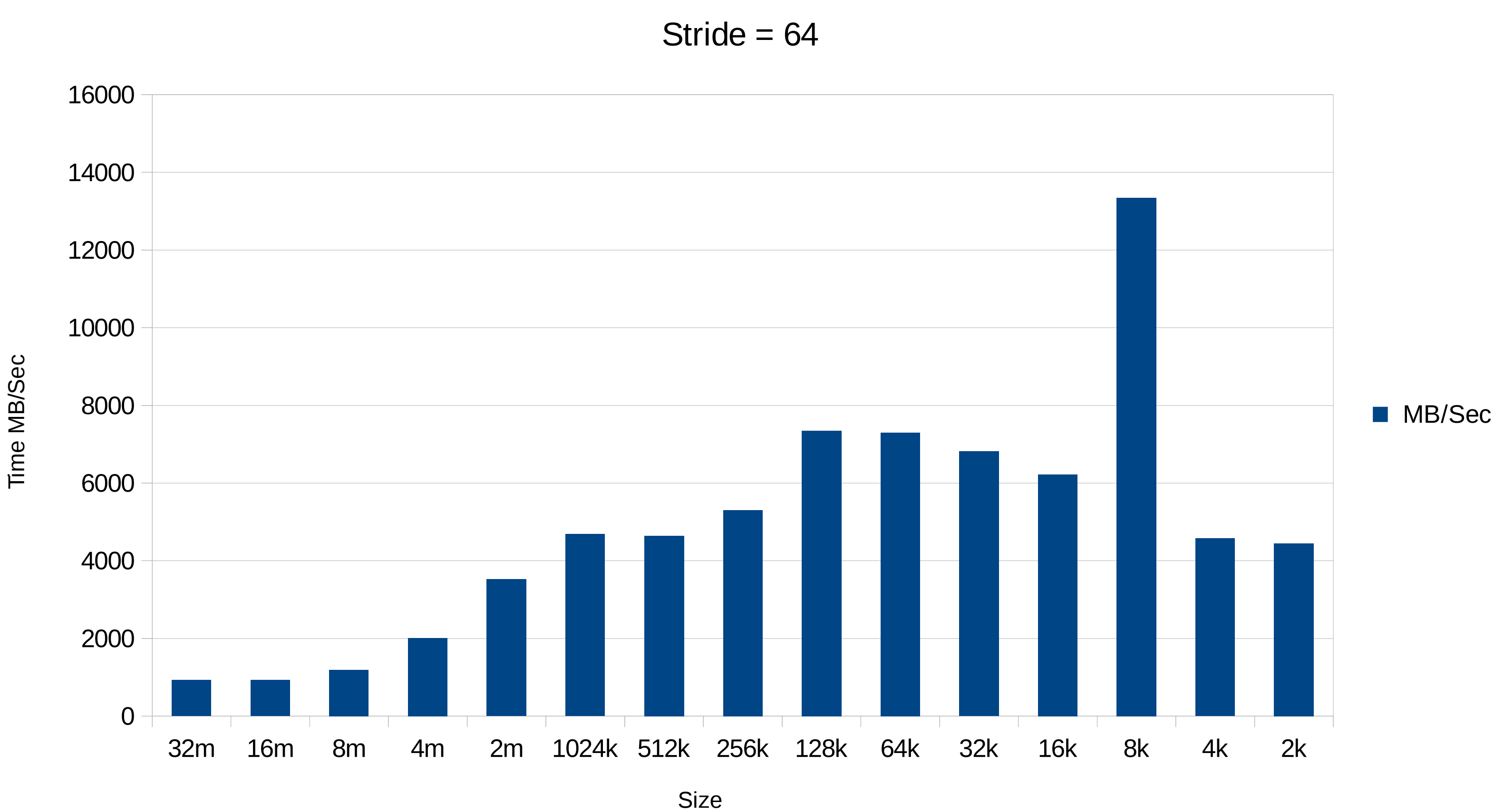


Question #20:
What region (array size, stride) gets the most **consistently** low performance? (Ignoring spikes in the graph that are noisy results...) What is the read bandwidth reported? Annotate your figure by drawing an arrow on it.
Answer:
The array size from 8k to 128k and stride from 6 to 45 get the most consistent low performance.



Question #21:
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.
Answer:





Question #22:
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 due to the temporal locality of the data in the array. If the temporal locality is bad then the stride will be lower as can be seen from the graphs. Hardware pref etching applies to this scenario because instructions that are loaded into the program can affectively predict which instructions are needed and save those.

Question #23:
What is temporal locality? What is spatial locality?
Answer:
Temporal locality is the idea that when a memory address that is accessed once it will most likely need to be accessed again in the future. Spatial locality is the idea that memory addresses next to the address that was accessed will likely need to be accessed.

Question #24:
Adjusting the total array size impacts temporal locality - why? Will an increased array size increase or decrease temporal locality?
Answer:
The total array size will not affect temporal locality, because temporal locality only deals with one memory address. If one specific address is accessed in the program it is likely to be accessed again in the future.

Question #25:
Adjusting the read *stride* impacts spatial locality - why? Will an increased read stride increase or decrease spatial locality?
Answer:
An increase in read stride will decrease spatial locality because there will be more space in between each individual array element. And spatial locality is all about accessing nearby array elements.

Question #26:
As a software designer, 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:
One guideline would be to make sure that the memory is freed after it is done being used. Another guideline to follow is to allocate memory concurrently so that the temporal locality is good. As a software designer you also want to make sure spatial locality is enforced in programs so accessing frequently used locations in memory is quick and easy.