HAVERFORD

Influence of Array Storage and Access Methods on Performance of Multi-Dimensional Arrays Used in Programs with High Cache Reuse

Tian Jin and David Wonnacott

INTRODUCTION

Multi-dimensional arrays are ubiquitously used in many algorithms. They are segments of memory blocks, contiguous or scattered, indexed and accessed by its unique memory locations, which can be obtained by various ways. The performance of these arrays is usually closely related to the overall performance of the algorithm. Therefore, it is worth investigating which type of arrays leads to a better performance. Commonly, various types of arrays show little difference in terms of their performance; however, when arrays are extensively used in programs with high cache reuse, some intriguing performance patterns begin to emerge between these different types of arrays.

TYPES OF ARRAYS

Different types of arrays differ from each other in terms of their sites of allocation, memory layout and access patterns. For sites of allocation, an array could be situated in BSS segment, stack and heap. For memory layout, an array could be declared as a continuous block of memory or using multiple malloc calls to obtain several blocks of memory and connecting them using another layer of pointers. The actual difference between these two ways of memory layout should be trivial as consecutive multiple malloc calls usually return consecutive memory addresses pointing to physically adjacent memory spaces which are very close to a contiguous block of memory. Access pattern is an important factor as a determinant of array performance. Two methods of accessing array elements are widely used:

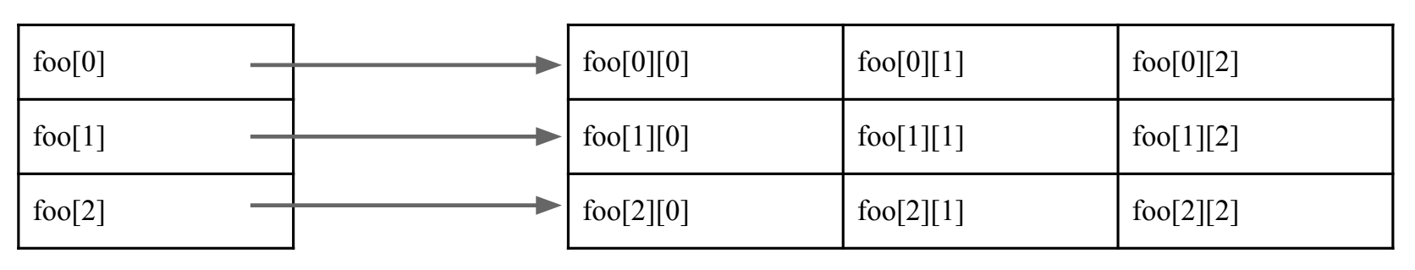
a). Calculation

In this case, arrays are usually allocated into a linear block of memory. Multi-dimensional arrays are mapped into a linear one by arithmetic transformation. To access specific element, integer calculation has to be performed to compute the exact memory location of it. An example formula for calculating the memory address of a two-by-three two-dimensional array is shown below:

$$Address = 3 \times column\ index + row\ index$$

b). Array of pointers

In this case, arrays are usually disjoint memory pieces linked together by a layer of pointers pointing to the starting position of each row. For a two-dimensional array, to access certain element, one has to retrieve the starting row position first and then add the result to the row index, which eventually gives the memory location of that element. An example of array of pointers is illustrated below where each block on the left (an array of pointers) points to the starting position of a row in the array.



EXPERIMENTAL PROCEDURE

Several factors are identified as likely contributors to the performance difference between access-by-calculation and access-by-array-of-pointers patterns:

(a) Rate of cache misses
(b) Site of allocation
(c) Dimensionality
(d) Row size (padding)
(e) The ratio of computation to memory traffic

The following body of code is modified in each case to test the effect of each factor on performance of the program:

```
for(int i=START; i<END; i+=T_SIZE)
  for(int j=START; j<END; j+=T_SIZE)
      for (int id=0; id<T_SIZE; id++)
          for (int jd=0; jd<T_SIZE; jd++)
              matrix(i+id,j+jd) = matrix(i+id-R,j+jd-R) OP
                                  matrix(i+id+R,j+jd-R) OP
                                  matrix(i+id-R,j+jd+R) OP
                                  matrix(i+id+R,j+jd+R)+1;
```

The knobs: T_SIZE refers to the size of the tile if the code is tiled. An appropriate configuration of tile size will lead to better cache efficiency during the runtime of the program. R refers to the access radius of the array elements used in this algorithm, which determines the memory locality of it. A large R means little memory locality and therefore little cache reuse. OP is the operator that defines the arithmetic operations to be carried out between each array elements used in the algorithm. OP is adjusted to have different computational heaviness.

Caution with GCC: for statically allocated arrays, which are default to be accessed by calculation, GCC may not be optimizing the program fully. Programs using statically allocated arrays therefore have to use the following code as a workaround:

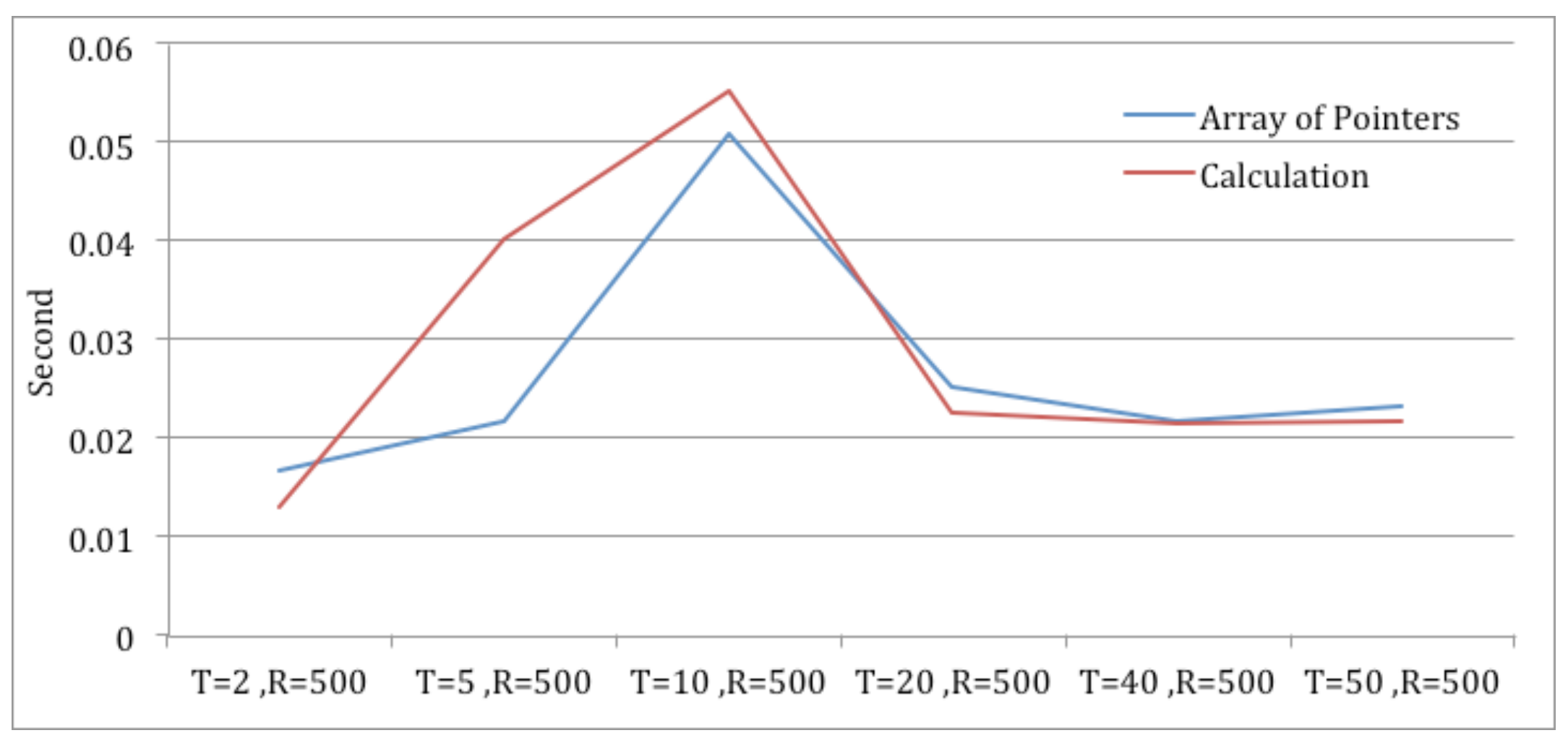
```
int* matrix=(int*)&_matrix[0];
#define matrix(i,j) matrix[(i)*5000+(j)];
```

In the above code, a pointer is obtained pointing to the starting position of the memory chunk and access-by-calculation pattern is explicitly defined as macros. In this case, GCC is capable of optimizing it to a similar extent as it does to programs using access-by-array-of-pointers pattern. This might very well be an issue with GCC as similar performance slowdown is also seen in other programs with high cache reuse apart from our experimental ones.

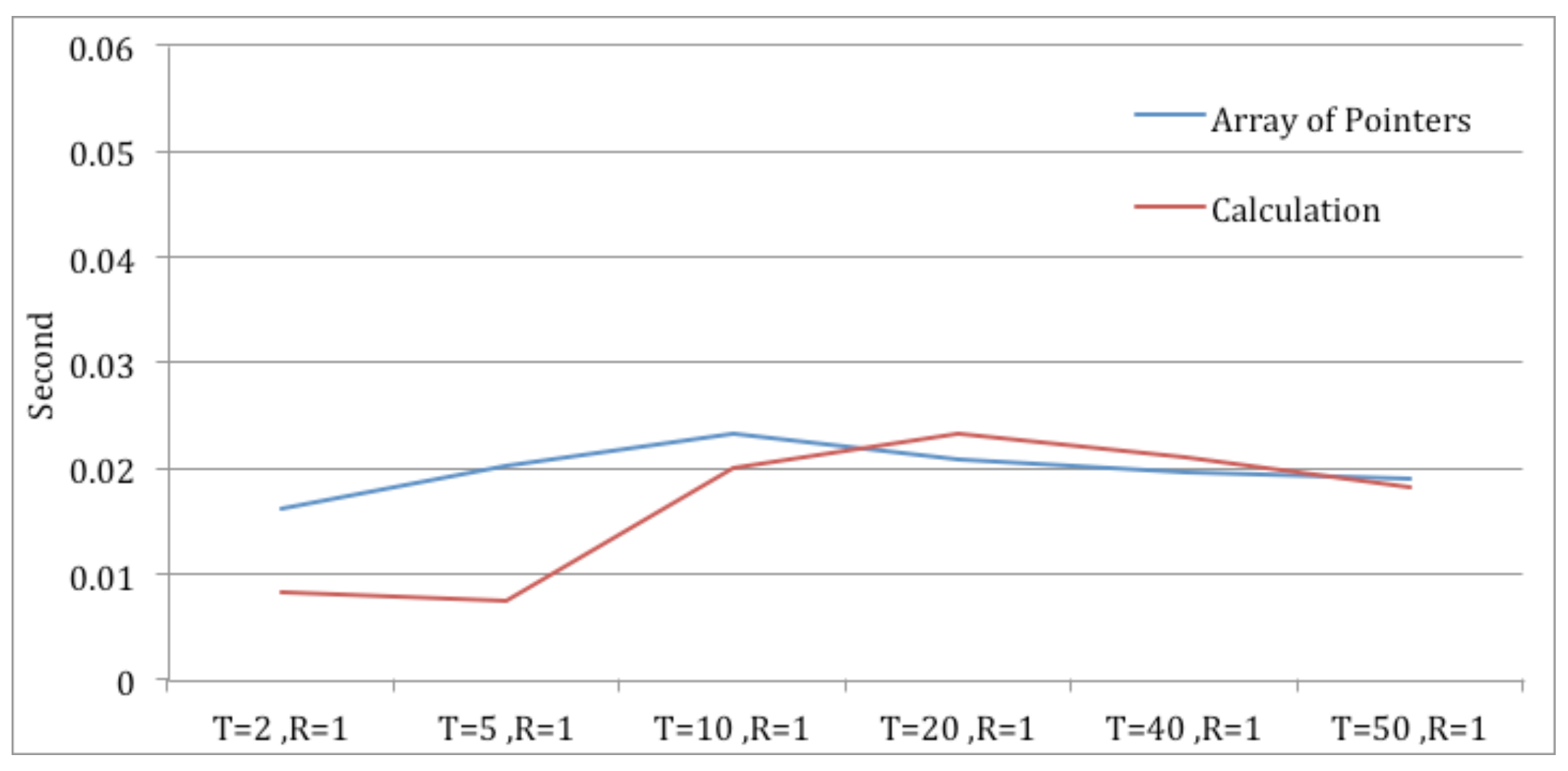
RESULTS AND DISCUSSION

a). Rate of cache misses

- Rate of cache misses measures how many of the memory operations are accessing memory locations not present in L1 CPU cache. Regardless of which ways of access pattern an algorithm employs, its performance is highly dependent on the rate of cache misses as long as there exist memory operations somewhere within the algorithm.
- However, access-by-array-of-pointers pattern is reasonably more sensitive to the rate of cache misses because for access-by-array-of-pointers pattern, memory access to each array element is sometimes preceded by at least another memory access to retrieve the starting position of that row (although loop invariant hoisting could happen) and thus will consume relatively more CPU cycles doing memory operations.
- By varying tile size and access radius, one can get a sense of how performances of these two access patterns vary as rate of cache misses changes.
- Below is the benchmarking results when access radius is relatively large and memory locality is relatively low:



- And another benchmarking results when access radius is relatively small and memory locality is relatively high:



- As seen from the experimental data, for low memory locality, two ways of memory access pattern have similar performance results; however, as memory locality becomes higher, an interesting pattern emerges: as tile size becomes bigger and bigger, access-by-array-of-pointers eventually outperforms access-by-multiplication.
- The fact that as cache misses gets fewer and fewer, access-by-multiplication will be decreasingly advantageous is obvious; however, we could not find evidence supporting the observation that at certain point, access-by-array-of-pointers outperforms access-by-calculation.

b). Sites of allocation:

Multi-dimensional arrays could be allocated in the BSS segment or on the heap. The experimental program is ran with different rate of cache misses to see the effect of different sites of allocation on the performance of the arrays.

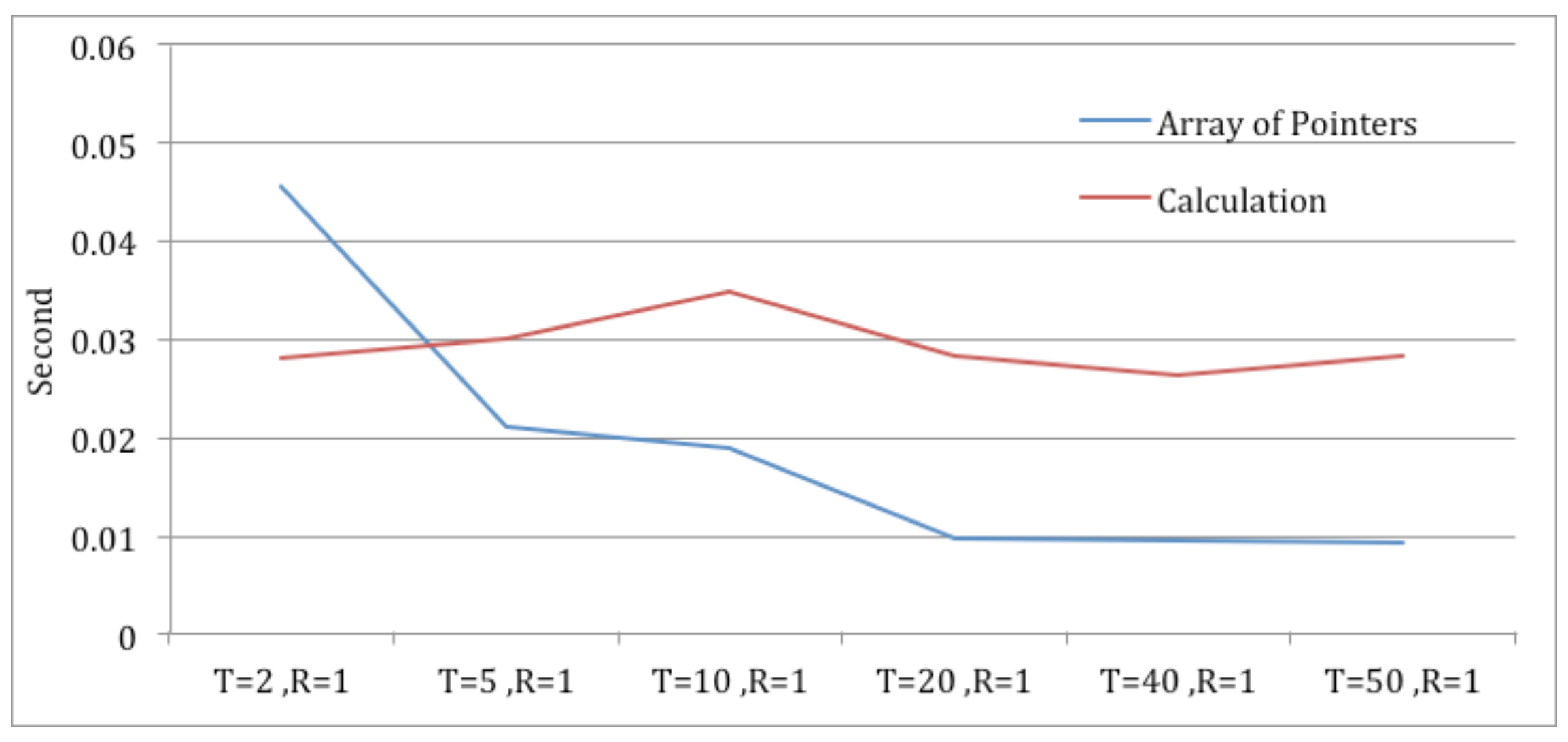
- D_HEAP: Arrays are allocated as discontiguous blocks in the heap.
- C_HEAP: Arrays are allocated as a continuous block in the heap
- D_BSS: Arrays are allocated as discontiguous blocks in the BSS segment
- C_BSS: Arrays are allocated as a contiguous block in the BSS segment

Config	D_HEAP(ms)	C_HEAP(ms)	D_BSS(ms)	C_BSS(ms)
T=2 ,R=500	16.8	13.1	16.3	12.7
T=5 ,R=200	22.1	33.1	22.6	32.1
T=10 ,R=100	42.6	46.3	43.9	45.9
T=20 ,R=50	20.9	18.8	21.0	18.9
T=40 ,R=25	17.3	20.4	16.3	20.0

- The data collected clearly suggests that for the same rate of cache misses and access patterns, the performance difference between arrays allocated in the heap and BSS segment is insignificant.
- However, it is worth noticing that accessing uninitialized arrays declared in BSS segment usually costs more than those declared in the heap as zeros may be filled on demand to the array upon access.

c). Dimensionality

Another way to further increase the data locality is through declaring the operation array as a four-dimensional one, making each of the iteration dimension an actual storage dimension. Therefore memory operations within one iteration dimension will actually stay in a close physical region. Using this four-dimensional array, performance results for two access patterns are gathered as follow:



As the graph clearly indicates, as data locality increases, access-by-array-of-pointers pattern starts to gain significant advantage over access-by-calculation pattern.

d). Row size (padding)

- The access-by-calculation pattern performs integer multiplication while calculating the memory location of each memory address.
- Multiplication is a relatively heavy computation whereas if row size is a power of two, shift instructions could be performed rather than multiplication.
- However changing row size to a power of two runs the risk of high cache misses as the size of CPU cache is often a power of two.
- Therefore the row size is changed to sums of powers of two to test its effect (Numbers are in milliseconds):

~Size	Pointers	Calculation	Pointers(SPT)	Calculation(SPT)
4400	26.1	34.7	25.9	35.0
4600	27.6	35.1	27.5	36.9
5100	29.4	36.0	29.9	37.3
6100	27.5	34.2	30.2	35.3

As seen from the experimental data, there is no significant gain in terms of performance by padding the row size to make it a sum of powers of two.

b). The ratio of computation to memory traffic

This ratio provides a measure of how important memory operation is as a factor for performance consideration. It also helps confirm that the performance difference is caused by memory access speed by ruling out the possibility that such difference is a result of computational factors. The operator in the experimental program body is changed to addition, multiplication and division to increase the "heaviness" of computation. The results for two access patterns are shown below:

Operation	Static(ms)	Dynamic(ms)	Difference
Addition	34.3	29.2	17%
Multiplication	45.5	42.4	7.4%
Division	103	103	0%

As clearly demonstrated, as computation instructions increases from addition to division, the performance gap quickly closes up and eventually no significant performance difference could be seen. This corroborates the hypothesis that such difference is not caused by any computations but purely a result of memory access patterns.

CONCLUSION

- Rate of cache misses is critical in determining which method of allocation is better in each case. For low rate of cache misses, access-by-array-of-pointers usually outperforms access-by calculation pattern.
- The site of allocation is not a significant factor affecting performance for multi-dimensional arrays that are previously initialized.
- Arrays with higher dimensionality, which increases data locality, will make the effect of rate of cache misses more dramatic.
- Row size does not matter even when cheaper shift instructions may be used instead of more expensive multiplication instructions.
- The ratio of computation to memory traffic is an important factor that determines the extent to which one access pattern is better than the other. A higher ratio of computation to memory traffic will make the performance difference between two access patterns less obvious.