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 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 contiguous 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 an array 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-dimensional array is shown below:

$$\text{Address} = \text{column index} \times \text{row index}$$

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) Size 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:

```c
for(int i=START; i<END; i++)
for(int j=START; j<END; j++)
matrix[i][j] = matrix[i][j-1] + matrix[i][j+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. The rate of cache misses is an important factor determining the performance of the array. 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 an array 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-dimensional array is shown below:

$$\text{Address} = \text{column index} \times \text{row index}$$

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.

Another way to further increase the data locality is through declaring the 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:

<table>
<thead>
<tr>
<th>Rate of cache misses</th>
<th>Access-by-calculation</th>
<th>Access-by-multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>42.6%</td>
<td>36.8%</td>
</tr>
<tr>
<td>High</td>
<td>34.3%</td>
<td>26.5%</td>
</tr>
</tbody>
</table>

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

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 performance 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:

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.

b). 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 means 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.

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.

The data collected clearly suggests that the same rate of cache misses is more dramatic. Row size does not matter even when shorter shift instructions may be used instead of 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.

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.

CONCLUSION

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:

<table>
<thead>
<tr>
<th>Rate of cache misses</th>
<th>Access-by-calculation</th>
<th>Access-by-multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>42.6%</td>
<td>36.8%</td>
</tr>
<tr>
<td>High</td>
<td>34.3%</td>
<td>26.5%</td>
</tr>
</tbody>
</table>