Analyzing Programs with Valgrind

Assignment 5

Idea

In this problem, we compare the performance of two programs which initialize and multiply two matrices. If a matrix is stored in row major order, then the rows are stored contiguous segments of memory. If a matrix is stored in column major order, then the columns are stored in contiguous segments of memory. Let us take a A and B to be the two matrices under consideration. Then $C = A \times B = [c_{ij}]_{n \times n}$ is such that $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$.

Scenario #1: Here, we multiply a row major ordered matrix with another row major ordered matrix. In this case the general order in which we access the matrices corresponds with the way they are held in memory.

```
for(int i=0; i<1024; i++) {
     for(int j=0; j<1024; j++) {
          a[i][j]=rand()%11;
          b[i][j]=rand()%11;
     }
}
```

Here, we initialize the elements in the typical row major order- we fill up the elements of A and B row by row in memory.

```
for(int i=0; i<1024; i++)
     for(int j=0; j<1024; j++)
          for(int k=0; k<1024; k++)
               c[i][j]+= a[i][k]*b[k][j];
```

In this snippet of code, we obtain c_{ij} by multiplying the i-th row of A with the j-th column of B. The problem here is that we do not access elements of double dimensional array ‘b’ which occur contiguously in memory. Hence, every time we access b[i][j] we need to get a new block of memory into cache.

Scenario #2: Here, we multiply a row major ordered matrix with a column major ordered matrix. The initialization of the elements can happen the same way- though conceptually, this means for the double dimensional array ‘b’ we are filling in elements column by column.

```
for(int i=0; i<1024; i++)
     for(int j=0; j<1024; j++)
          for(int k=0; k<1024; k++)
               c[i][j]+= a[i][k]*b[j][k];
```

The multiplication is a bit different though. In this version, we can see that the elements of ‘b’ are accessed so that two consecutively accessed elements occur at contiguous memory locations. We are multiplying a row of ‘a’ with a column of ‘b’; both of them are stored contiguously in memory. This means that there is a **spatial locality** which can be exploited by the cache. Every time we access b[i][j] there is a good chance there is a block with it already!

Implementation Details: We use global variable instead of local variables to better highlight the difference between Scenario #1 and Scenario #2. Global variables are, however, slower as they are stored in heap which is slower than stack. Further, we initialize the elements of ‘c’ as 0 beforehand, though it would be beneficial to do so immediately before updating them.

System Configuration:

2.3 GHz 8-Core Intel Core i9
16 GB 2667 MHz DDR4
L1 cache: 32KB per core, 8 way associative.
L2 cache: 256KB per core, 4 way associative
L3 cache: 2MB per core, 16 way associative
Source: wikichip

Events	Scenario #2	Scenario #1
Runtime (s)	6.413209	32.113809
Number of instructions (Ir)	1,098,593,645	5,455,426,716
Data references	556,791,548	2,167,404,349
D1 misses	67,385,713	1,076,192,623
LLd misses	201,088	201,093
D1 miss rate (%) read+write	12.3% + 1.4%	49.8% + 12.1%
LLd miss rate (%) read+write	0.0% + 1.4%	0.0% + 1.4%
LL references	67,390,506	1,076,197,439
LL misses	204,220	204,216
LL miss rate (write) (%) r+w	0.0% + 1.4%	0.0% + 1.4%

Clearly, based on the discussion earlier, the miss rate in Scenario #1 is generally larger than the miss rate for Scenario #2.

Also, the performance in terms of time is quite markedly better for Scenario #2.