Lab 3 Part C

Hemanshu Garg, 2022CS11090

Introduction

Matrix multiplication operations are at the heart of many linear algebra algorithms, and eficient matrix multiplication is critical for many applications within the applied sciences. In this report we explore minor various in algorithms of matrix mult and transpose and see their performance difference and try to explain them in context to the cache.

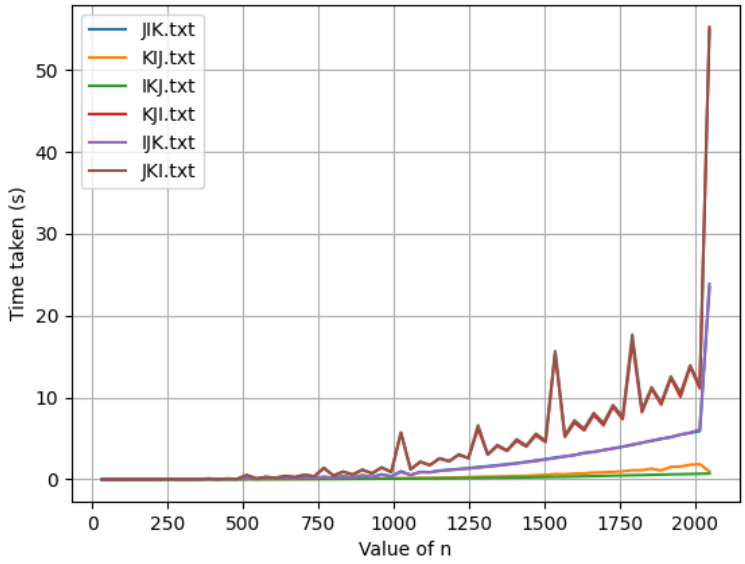
Approach for Multiplication

We use following matrix multiplication algorithms and benchmark their execution for values of n upto 2048 , averaging the execution time for each n 5 times:

- ```
  void matmultJIK(int *A, int *B, int *C, int n)
  { int i, j, k;
  for (j = 0; j < n; j++)
  for (i = 0; i < n; i++)
  for (k = 0; k < n; k++)
  C[i*n+j] += A[i*n+k] * B[k*n+j];  }
  ```
- ```
  void matmultKIJ(int *A, int *B, int *C, int n)
  { int i, j, k;
  for (k = 0; k < n; j++)
  for (i = 0; i < n; i++)
  for (j = 0; j < n; k++)
  C[i*n+j] += A[i*n+k] * B[k*n+j];  }
  ```
- ```
  void matmultKJI(int *A, int *B, int *C, int n)
  { int i, j, k;
  for (k = 0; k < n; j++)
  for (j = 0; j < n; i++)
  for (i = 0; i < n; k++)
  C[i*n+j] += A[i*n+k] * B[k*n+j];  }
  ```

And corresponding 3 more loop orders
  ```

Mult-benchmark graph



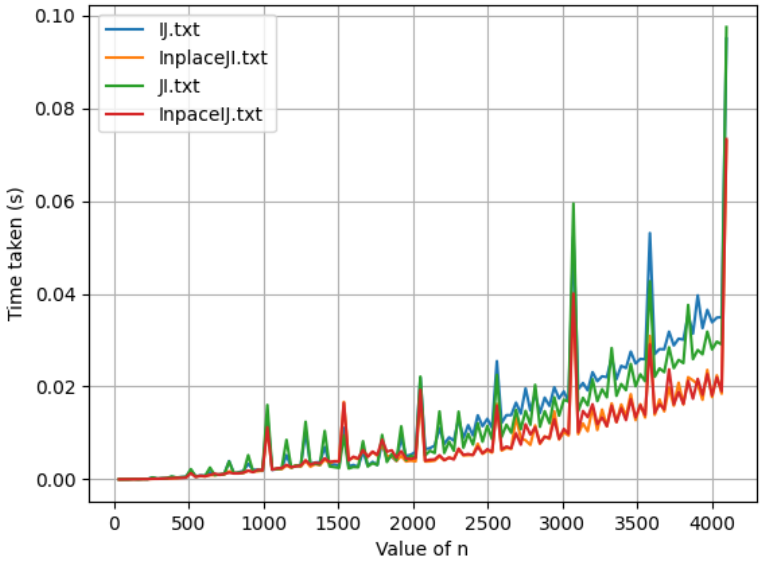
Approach for Transpose

We benchmark inplace and not inplace transpose algorithm and change order of the loops in those as well :

- ```
  void transposeIJ(int *A, int *C, int n)
  { int i, j;
  for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
  C[i*n+j] = A[j*n+i];
  }
  ```
- ```
  void transposeInplaceIJ(int *A, int n)
  { int i, j;
  for (i = 0; i < n; i++)
  for (j = i+1; j < n; j++)
  int temp = A[i*n+j];
  A[i*n+j] = A[j*n+i];
  A[j*n+i] = temp;
  }
  ```

This and their corresponding loop orders (2 more)

Transpose-benchmark graph



Observation and Conclusions

For multiplication it is clearly observed : Loops with innermost variable same have almost similar time. This makes sense as the critical step is data storing to the array, and so the outer loops order don't change frequency of that. There is however significant difference around n = 1250 among these three categories of loops, This can be explained quite easily,

```
C[i*n+j] += A[i*n+k] * B[k*n+j];
```

In the IJK/JIK loop order:- We access row of A, Column of B and single element of C at each iteration.

In the JKI /KJI loop order : - We access Columns in A and C , and single point in B

In the IKJ/KIJ loop order - We access single point in A and rows of C and B.

Clearly in loop order that access more columns are less efficient, this because they don't respect temporal locality and as we increase n, the chance of cache miss increases in column access.

For Transpose, We observe the loop order matters less (as we have atleast one column access in both), however inplace is faster than non inplace. This is because clearly we have less number of memory access in inplace , as loop takes less number of iterations to begin with.

About the Cache

We observe spikes in the the graphs of the benchmarks at interesting intervals.
For multiplication benchmark there are small spikes at intervals of 64 and big spikes at invervals of 256.
In Transpose we observe similar large and small spikes at regular intervals. Having spikes means therefore the cache follows write-back, write-allocate policy (as stores are getting cached). From these we can conclude block size of the cache of various levels. Having a spike at indicates that there is eviction/that data is not present in the block, major spikes at invtervals of 256 - each int takes 4 bytes, this corresponds to 1024 bytes. this could be possible with set size 8 and block size 128 bytes. Also the divergence in execution time for the three loops is around n = 1024 that is about 4096 bytes so some level of cache size is 4 KiB