Programming Assignment:- 1

Name of the Group:- Random

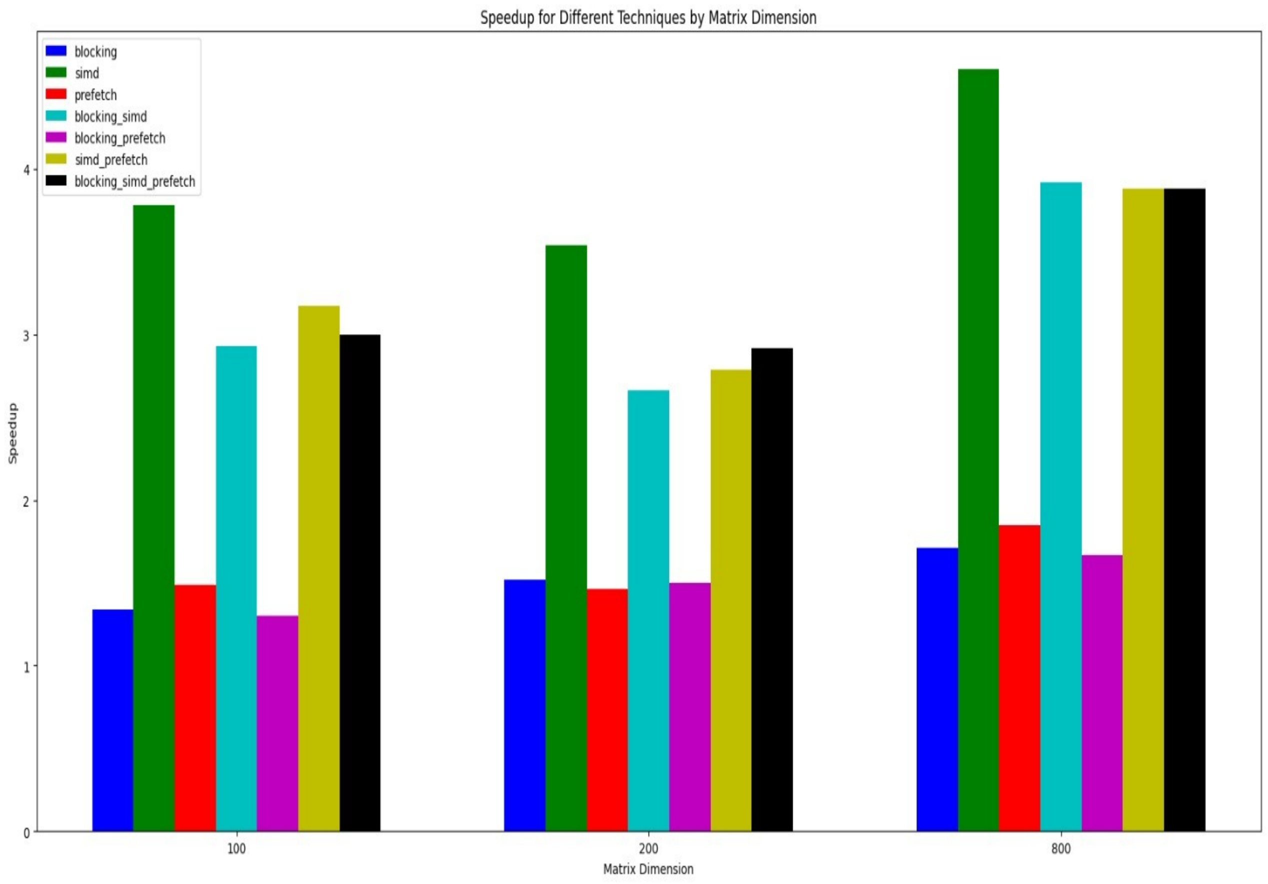
Members of the group:-

1. Ravi Kumar Choubey

2. Vedant Kalbande

3. Tarun Gogula

Graph of different Techniques:



Blocking Matrix Multiplication:

- In normal matrix multiplication each access to an element in matrix B gives a miss, even though a line of data is loaded to cache for every miss ,it is getting replaced by the time we access it.
- To exploit this we use Blocked matrix multiplication technique, here we try to use the data loaded into cache before it's getting replaced.
- We consider blocks of size BxB and perform matrix multiplication to utilise cache locality.

- By performing normal matrix multiplication on those blocks and adding up results yields the better performance
- Added loop unrolling to increase performance by decreasing loop overhead
- We have to make sure that 3 B x B will fit into the L1 cache .
- L1 = 32KB ; double size = 8B ;
  So calculating Block size , B = 25

Execution Time:

N=100 : 4.6ms
N=200 : 28ms
N=800 : 1.55seconds

Performance Improvement:

N=100 : 1.34
N=200 : 1.52
N=800 : 1.71

Task 2: SIMD

We have implemented this using SIMD instructions such as _mm256_loadu_pd, _mm256_fmadd_pd, _mm256_setzero_pd, _mm256_storeu_pd.

- _mm256_loadu_pd : This function moves double-precision floating point values from unaligned memory location to a destination vector.

- _mm256_fmadd_pd(a,b,c) : Performs a set of SIMD multiply-add computation on packed double-precision floating-point values using three source vectors/operands, a, b, and c. Corresponding values in two operands, a and b, are multiplied and results are added to corresponding values in the third operand a.

- _mm256_setzero_pd : Sets registers to zero.

- _mm256_storeu_pd : Moves packed double-precision values from a vector to an unaligned memory location.

Method:
- Initially, we have copied column major sequence of matrix B into new matrix B_col using convertRowToColumnMajor(). We have done this because it optimizes cache usage by ensuring data locality during multiplication of matrix A and B_col which leads to fewer cache misses and better performance.

- Now in the innermost loop we are loading the values(4 at a time) of matrix A and B_col into vectors a, b respectively and then multplying vector values and adding the mulplication result into sum and storing final value into sum vector.
- After complete iterations of innermost loop we get the final values into sum and store it in result array and after adding all 4 values of result we get accurate value for C[i][j].This process is iterated for i=0 to dim and in that for j=0 to dim.

we also tried two methods before like: using setr_pd and using transpose of matrix but we get best result in the above method.

Matrix dim --> Iterations |	**100**	**200**	**800**
Iteration 1	3.89	3.25	4.69
Iteration 2	3.64	3.24	4.67
Iteration 3	3.65	3.29	4.68
Iteration 4	3.72	3.28	4.47
Iteration 5	3.81	3.30	4.45
Iteration 6	3.77	3.26	4.72
Iteration 7	3.90	3.23	4.60
Iteration 8	3.78	3.28	4.53
Iteration 9	3.89	3.27	4.65
Iteration 10	3.76	6.03	4.55
Average	3.78	3.54	4.6

Prefetching:-

Optimize matrix multiplication using software prefetching.
Let's break down the code and explain you step by step:-

1. double* B_col = (double*)malloc(sizeof(double) * dim * dim);
I used one temporary matrix "B_col" to copy the B matrix column elements into row manner. This optimization is made to optimize cache usage by ensuring better data locality during the multiplication of matrix A and B_col which leads to fewer caches misses and better performance (Cache Freiendly Data Access).
2. Loop reordering/interchanging :- This can lead to better cache locality

3.Precompute the values of 'temp' and 'temp1' outside the innermost loop. It helps compiler optimize the code better. It ensures that the same values aren't recalculated in every iteration of the innermost loop.

4. Prefetching Instructions:-
__builtin_prefetch(&A[i * dim + k + 4], 0, 3);
__builtin_prefetch(&B_col[j * dim + k + 4], 0, 3);

__builtin_prefetch --> prefetching instruction to load data into the cache ahead of CPU request. It prefetch sata for future iteration of the loop.

Parameters :- (_a_,__b_,__c_)
a. This is the memory address that i want to prefetch. Here, i am prefetching ahead of 4.
b. 0/1 -> indicate that whether prefetch operation for read/write form/to the memory. 0 -> read, 1-> write.
c. Provide hint to compile -> level of temporal locality. ‘0’ is for non-temporal locality and ‘3’ is high degree of locality. Temporal locality means data likely to be reused in the near future.

**** The exact performance gains will depend on the architecture of the CPU and the size of the matrices.**
**** My laptop does not have setting to disable the hardware prefetching (by set MSR bit) from the BIOS but i run my code in my friend laptop that has facility to set the MSR bit.**

Matrix dim --> Iterations |	**100**	**200**	**800**
Iteration 1	1.41	1.36	1.81
Iteration 2	1.36	1.37	1.82
Iteration 3	1.51	1.37	1.96
Iteration 4	1.58	1.39	1.87
Iteration 5	1.53	1.36	1.85
Iteration 6	1.49	2.30	1.85
Iteration 7	1.53	1.37	1.81
Iteration 8	1.49	1.46	1.87
Iteration 9	1.43	1.33	1.83
Iteration 10	1.54	1.34	1.82
Average	**1.49**	**1.46**	**1.85**

Blocking-prefetch Matrix Multiplication:

- Software prefetching loads the data into cache before it is needed by the program, to ultimately reduce memory latency time
- I have used __builtin_prefetch instruction to do prefetch
- In normal prefetching by the time we try to re-use the prefetched data, it’s getting replaced in cache.
- So by including blocking with prefetch we can use all the prefetched data effectively.

Limitation:

Hardware prefetcher is way ahead than software prefetcher, I tried disabling hardware prefetching but unable to do that. So there isn’t any visible performance improvement.

Execution Time:

N=100 : 5ms

N=200 : 30ms

N=800 : 1.56sec

Performance Improvement:

N=100 : 1.3

N=200 : 1.5

N=800 : 1.67

Task : Blocking + SIMD
Optimize matrix multiplication using blocking and Simd.

- Initially, we have stored column major sequence of matrix B into new matrix B_col using convertRowToColumnMajor(). We have done this because it optimizes cache usage by ensuring data locality during multiplication of matrix A and B_col which leads to fewer cache misses and better performance.
- The code divides the matrix multiplication into smaller blocks using nested loops to utilise cache locality. Here block size preferable is 20.
- The innermost loop contains Simd instructions for vectorized multiplication and additon. SIMD instructions allow multiple calculations to be performed in parallel generally 4 at a time , which can significantly speed up matrix multiplication by utilizing the processor's vector processing capabilities. This code benefits from multiple performance optimizations, including blocking and SIMD instructions, to maximize CPU performance during matrix multiplication.

The advantage here is that Blocking reduces cache misses by improving memory access patterns and SIMD instructions allows parallel processing of data. The benefits of above optimization helps us to gain significant performance improvements, especially when dealing with large matrices. The exact performance gain would depend on factors such as the CPU architecture, matrix size, and block size used, cache size.

Matrix dim --> Iterations |	**100**	**200**	**800**
Iteration 1	2.62	2.61	4.05

Iteration 2	2.76	2.73	4.02
Iteration 3	3.15	2.78	4.00
Iteration 4	2.84	2.75	3.80
Iteration 5	2.98	2.68	3.74
Iteration 6	3.25	2.66	4.03
Iteration 7	3.01	2.63	3.77
Iteration 8	2.95	2.57	3.73
Iteration 9	2.97	2.65	4.07
Iteration 10	2.75	2.54	3.95
Average	**2.93**	**2.66**	**3.92**

SIMD + PREFETCHING:-

Optimize matrix multiplication using SIMD and PREFETCHING.
Let's break down the code and explain you step by step:-

1. double* B_col = (double*)malloc(sizeof(double) * dim * dim);
I used one temporary matrix "B_col" to copy the B matrix column elements into row manner. This optimization is made to optimize cache usage by ensuring better data locality during the multiplication of matrix A and B_col which leads to fewer caches misses and better performance (Cache Freiendly Data Access).

2. SIMD Vectorization:-
```
__m256d sum = _mm256_setzero_pd();
...
__m256d a = _mm256_loadu_pd(&A[temp + k]);
__m256d b = _mm256_loadu_pd(&B_col[temp1 + k]);
sum = _mm256_fmadd_pd(a, b, sum);
```

In this code, i am using AVX instructions to perform vectorized operations on four double-precision floating-point numbers simultaneously. The '_mm256_load_pd' function loads four consecutive double-precision values into a 256 bit AVX register, and '_mm256_fmadd_pd' performs a fused multiply-add operation on these registers.
This SIMD approach allows for parallelization of arithmetic operations, which can result in significant speedup for large matrices.

3. Loop Unrolling:-
The loop over 'k' is unrolled by a factor of 4. This means that four iterations of the loop are combined into a single iteration, allowing for better pipelining and reduced loop overhead. This can improve performance by reducing loop control overhead.

4. Prefetching Instructions:-
```
__builtin_prefetch(&A[i * dim + k + 4], 0, 3);
__builtin_prefetch(&B_col[j * dim + k + 4], 0, 3);
```

__builtin_prefetch --> prefetching instruction to load data into the cache ahead of CPU request. It prefetch sata for future iteration of the loop.

Parameters :- (_a_,__b_,__c_)
a. This is the memory address that i want to prefetch. Here, i am prefetching ahead of 4.
b. 0/1 -> indicate that whether prefetch operation for read/write form/to the memory. 0 -> read, 1-> write.
c. Provide hint to compile -> level of temporal locality. ‘0’ is for non-temporal locality and ‘3’ is high degree of locality. Temporal locality means data likely to be reused in the near future.

5. Storing and Accumulating Results:-
After SIMD vectorized calculation, the result is stored in an array ‘result’, and the elements of ‘result’ are then accumulated into final result ‘C’. This accumulation step ensures that the correct values are stored in the output matrix ‘C’.

Matrix dim --> Iterations |	**100**	**200**	**800**
Iteration 1	3.23	2.77	3.89
Iteration 2	3.04	2.53	3.95
Iteration 3	3.08	2.70	3.83
Iteration 4	3.40	2.85	3.80
Iteration 5	3.29	2.80	3.91
Iteration 6	3.14	2.78	3.85
Iteration 7	3.13	2.94	3.88
Iteration 8	3.12	2.81	3.87
Iteration 9	2.95	3.05	3.90
Iteration 10	3.33	2.70	3.91
Average	**3.17**	**2.79**	**3.88**

Task : Blocking + SIMD + Prefetching
Optimize matrix multiplication using blocking, Simd and Software prefetching.

- Initially, we have stored column major sequence of matrix B into new matrix B_col using convertRowToColumnMajor(). We have done this because it optimizes cache usage by ensuring data locality during multiplication of matrix A and B_col which leads to fewer cache misses and better performance.
- Now, we divide the matrix multiplication into smaller blocks using nested loops to utilise cache locality. Here block size preferable is 20.
-
- The innermost loop contains Simd instructions for vectorized multiplication and accumulation. SIMD instructions allow multiple calculations to be performed in parallel generally 4 at a time , which can significantly speed up matrix multiplication by utilizing the processor's vector processing capabilities.
- Along with the blocking and Simd we introduces prefetching using the __builtin_prefetch function. to prefetch data into the CPU cache, helping to reduce memory latency.

- In this case, it prefetches data from matrices A and B_col before they are accessed in the innermost loop.

The benefits we got here is that Blocking reduces cache misses by improving memory access patterns and SIMD instructions allows parallel processing of data and Prefetching benefits performance by reducing memory access latency. When data is prefetched into the cache before it's actually used, the CPU can continue executing instructions without waiting for memory reads, potentially hiding memory latency and improving throughput. The exact performance improvement from prefetching and other optimizations would depend on factors such as the specific CPU architecture, memory hierarchy, and the size of matrices being multiplied.

Matrix dim --> Iterations |	**100**	**200**	**800**
Iteration 1	2.85	2.53	3.86
Iteration 2	3.12	2.61	3.86
Iteration 3	3.17	2.52	3.91
Iteration 4	2.89	2.79	3.75
Iteration 5	2.92	2.56	3.81
Iteration 6	3.01	3.30	4.05
Iteration 7	2.88	3.88	3.71
Iteration 8	3.04	2.58	3.96
Iteration 9	3.00	2.58	3.94
Iteration 10	3.13	3.88	3.94
Average	**3**	**2.92**	**3.88**