Computer Architecture Assignment-1

Sohan Patidar, Megha Patidar

1. Introduction

Performance Monitoring Counters (PMCs) are specialized hardware registers in modern processors that track various events during program execution. These events include important metrics like cache hits and misses (L1, L2, LLC), Translation Lookaside Buffer (TLB) misses, branch mispredictions, instruction counts, page faults, etc. PMCs provide valuable insights into how efficiently a program utilizes the CPU, memory hierarchy, and other system resources. PMCs help in optimizing and improving overall system performance. They are commonly accessed using tools like PAPI or perf.

We have used "**perf**" for the experiment. **"perf"** is a powerful Linux-based performance analysis tool that allows developers to monitor and analyse the performance of their applications and the underlying system. It provides access to hardware Performance Monitoring Counters (PMCs) to track various events, such as CPU cycles, cache hits and misses, TLB misses, and context switches. It can also trace system calls and record detailed profiles of application behaviour, helping to identify performance bottlenecks and optimize resource usage. It is widely used for profiling and debugging in both user-space and kernel-space applications.

Here, we aim to evaluate the performance of six different versions of the matrix multiplication algorithm using the standard O(n^3) complexity. Matrix multiplication is a fundamental operation in many computational fields, making its optimization critical for high-performance computing. Specifically, we will assess the performance of the various loop orderings and the effect of memory hierarchy by analysing cache and Translation Lookaside Buffer (TLB) misses, page faults, and other performance metrics using "**perf**". The experiments will be conducted on two large matrix sizes—2048×2048 and 8192×8192 to capture how the algorithm scales with increasing problem sizes. We will first evaluate the basic loop order variations (i,j,k), (j,i,k), and (k,i,j).

Furthermore, we will experiment with using large pages (2MB) to determine whether they can reduce execution time by mitigating the overhead of TLB misses using "**mmap**". "**mmap**" is a system call in Unix-like OS that maps files or devices into memory, allowing a program to access them as if they were part of its memory space. When used for memory management, mmap can allocate large blocks of memory or directly map large pages (e.g., 2MB or 1GB pages) to reduce the overhead associated with memory paging and TLB misses.

Finally, we will create blocked or tiled versions of each loop variant, using a tile size of 64. Tiling (or blocking) is an optimization technique in computer algorithms. Instead of processing the entire dataset at once, tiling breaks it into smaller sub-blocks (or tiles) and process each one individually. This method improves data locality, as the working set of a tile can fit within faster levels of the memory hierarchy, such as the L1 or L2 cache.

The report will present the experimental methodology, performance results for each variant, and an analysis of the obtained performance metrics, and comparison between various versions and their performance counters.

2. Implementation

Our experiment focused on implementing different standard matrix multiplication with different variants and configurations for comparing the performance of different versions using "**perf**".

1. Memory allocation

For part (a) and (c), we utilized the **malloc** function for dynamic memory allocation. malloc allows us to allocate the required memory for the matrices A, B, and C at runtime based on the matrix size, ensuring flexibility in handling different matrix dimensions. By using malloc, we can manage large matrices such as 2048×2048 and 8192×8192, which would not fit in the stack's fixed-size memory. This allocation strategy also ensures that the memory is contiguous, providing efficient access patterns during computation.

For part (b) we incorporated the **mmap** system call in our matrix multiplication implementation to allocate memory for utilizing huge pages of size 2MB. The mmap function allows for direct mapping of the desired memory region, enabling efficient access patterns that are crucial for handling large matrices. This approach not only enhances memory access speed but also optimizes the use of system resources, as fewer, larger pages reduce fragmentation and improve cache coherence.

2. Input

For providing input data to our matrix multiplication, we employed a code to generate random numbers to create matrix elements within a specified range of 0 to 9 for 2048x2048 and 0 to 1 for 8192x8192. The generated input for matrices were stored in a file, enabling efficient access and retrieval during the computation process. Before performing the multiplication, we read the matrix data from the file, ensuring that the matrices A and B are populated with the generated values immediately after memory allocation. This method not only facilitates reproducibility of the experiments but also allows for quick adjustments to the input size and values, contributing to a more comprehensive analysis of the algorithm's performance across varying data sets.

3. Working

2.3.1 Code working

2.3.1.a. In the first section of the experiment, we will be using the standard matrix multiplication algorithm which is implemented using the conventional three-loop structure. We have used the outer loop to iterate over the rows of matrix A and the columns of matrix B, and the innermost loop performs the dot product of corresponding elements from matrix A and matrix B to compute each element of the result matrix C. The general structure follows the loop order (i, j, k) but we have also implemented with other two loop orders- (j, i, k), and (k, i, j), using two different matrix sizes – 2048x2048 and 8192x8192, which will sum to a total variants of six different matrix multiplication variants.
Here is an example variant of loop order (i, j, k):

```
for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
                C[i][j] = 0;
                for (k = 0; k < N; k++) {
                // Matrix multiplication (C = A * B)
                        C[i][j] += A[i][k] * B[k][j];
                }
        }
}
```

By interchanging place of i,j,k we can implement other two variants and we will use this for N=2048 and N=8192 which will give a total of six variants for this standard matrix multiplication.

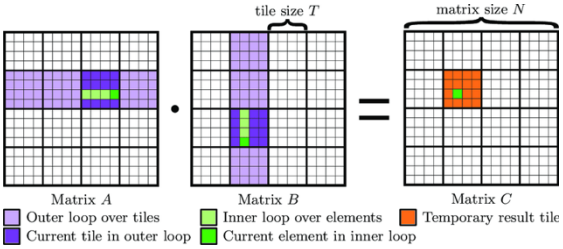
2.3.b. As instructed in the second section, we have used mmap system call for dynamic memory allocation, specifically allocating huge pages of size 2MB. We implemented six different variants of the matrix multiplication, exploring three loop orders—(i, j, k), (j, i, k), and (k, i, j)—to analyse the impact of memory access patterns on performance.

```
// Allocate matrices A, B, and C using mmap
int** A = allocate_matrix(SIZE);
int** B = allocate_matrix(SIZE);
int** C = allocate_matrix(SIZE);

// Clean up (unmap the allocated memory)
munmap(A, SIZE * sizeof(int*)+SIZE*SIZE * sizeof(int));
munmap(B, SIZE * sizeof(int*)+SIZE*SIZE * sizeof(int));
munmap(C, SIZE * sizeof(int*)+SIZE*SIZE * sizeof(int));

//Terminal command for changing page size to 2MB
sudo sysctl -w vm.nr_hugepages = required_pages
```

required_pages will be equal to atleast that number of pages which are needed to implement the matrix multiplication. For a 2048×2048 matrix, each of the three matrices will contain 2048×2048 elements, with each element occupying 4 bytes (the size of an integer). This results in a total memory requirement that will be divided across 2MB pages. So for matrix of size 2048x2048, least number of required_ pages are 24 and by applying same logic for 8192x8192 matrix, least number of required_ pages are 384.

2.3.c. For the third section, we have used tiling, a technique designed to optimize cache performance by processing subblocks of matrices instead of entire matrices at once. We defined a tile size of 64, allowing each subblock to fit within the cache, thus improving data locality and minimizing cache misses. This approach was applied to six different variants of the matrix multiplication algorithm, using different loop orders—(i, j, k), (j, i, k), and (k, i, j) for two different matrix sizes, starting with the conventional three-loop structure that iterated over the rows of matrix A and the columns of matrix B. For each of these variants, we implemented corresponding blocked versions that process smaller tiles, ensuring that the calculations for matrix C are performed on small data chunks.
Here is an example variant of loop order (i, j, k):

```
for(int i=0; i<SIZE; i=i+tile_size){
  for(int j=0; j<SIZE ;j=j+tile_size){
    for (int k=0; k < SIZE; k=k+tile_size){
      for(int ii=i; ii<i+tile_size; ii++){
        for(int jj=j; jj < j+tile_size; jj++){
          for(int kk=k; kk < k+tile_size; kk++){
            C[ii][jj] += A[ii][kk] * B[kk][jj];
          }}
      }}
}}}
```



2.3.1 Perf working

We need to run the following command by setting path to the code file, it will give specific performance counters.

$ perf stat -e task-clock,context-switches,cpu-migrations,page-faults, cycles, instructions, branches, branch-misses, cache-misses, cache-references, dTLB-loads, dTLB-load-misses, dTLB-stores, dTLB-store-misses, iTLB-loads, iTLB-load-misses, L1-dcache-loads, L1-dcache-load-misses, L1-icache-load-misses, L2_rqsts.miss, L2-loads, L2-load-misses,L2-stores, L2-store-misses, LLC-loads, LLC-load-misses,LLC-stores ./program_file_name

We can also run $ perf stat ./program_file_name , but it will only give some specific program counters.

3. Observation

All the graphs have been plotted with the help of log scale.

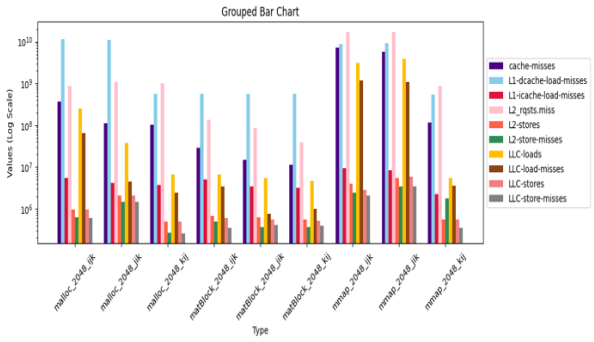
a. Statistics for 2048 variants

3.1.1. Cache statistics

Type	cache-misse	L1-dcache-loads	L1-dcache-load-mi	L1-icache-load-m	L2_rqsts.miss	L2-loads
malloc_2048_ijk	3.63E+08	1.90E+11	1.15E+10	5.40E+06	8.59E+08	2.52E+08
malloc_2048_jik	1.13E+08	1.90E+11	1.12E+10	4.10E+06	1.11E+09	3.78E+07
malloc_2048_kij	1.03E+08	1.90E+11	5.54E+08	3.71E+06	9.91E+08	6.78E+06
matBlock_2048_ijk	2.87E+07	2.00E+11	5.67E+08	5.11E+06	1.33E+08	6.48E+06
matBlock_2048_jik	1.46E+07	2.00E+11	5.63E+08	3.46E+06	8.33E+07	5.55E+06
matBlock_2048_kij	1.13E+07	2.00E+11	5.62E+08	3.23E+06	3.97E+07	4.62E+06
mmap_2048_ijk	7.18E+09	1.90E+11	8.70E+09	9.33E+06	1.67E+10	3.15E+09
mmap_2048_jik	5.74E+09	1.90E+11	9.21E+09	8.48E+06	1.71E+10	3.87E+09
mmap_2048_kij	1.14E+08	1.90E+11	5.50E+08	2.24E+06	8.61E+08	5.51E+06

Type	L2-load-miss	L2-stores	L2-store-misses	LLC-loads	LLC-load-miss	LLC-stores	LLC-store-miss
malloc_2048_ijk	6.46E+07	9.54E+05	6.34E+05	2.53E+08	6.48E+07	9.63E+05	6.06E+05
malloc_2048_jik	4.61E+06	2.09E+06	1.47E+06	3.72E+07	4.57E+06	2.09E+06	1.47E+06
malloc_2048_kij	2.65E+06	4.96E+05	2.62E+05	6.52E+06	2.46E+06	4.97E+05	2.56E+05
matBlock_2048_ijk	3.33E+06	6.73E+05	5.02E+05	6.49E+06	3.36E+06	5.93E+05	3.48E+05
matBlock_2048_jik	9.20E+05	6.18E+05	3.70E+05	5.49E+06	7.71E+05	5.63E+05	4.03E+05
matBlock_2048_kij	1.00E+06	5.61E+05	3.61E+05	4.60E+06	9.95E+05	5.19E+05	3.90E+05
mmap_2048_ijk	1.20E+09	3.96E+06	2.43E+06	3.13E+09	1.20E+09	2.81E+06	2.06E+06
mmap_2048_jik	1.11E+09	5.46E+06	3.42E+06	3.87E+09	1.11E+09	5.84E+06	3.48E+06
mmap_2048_kij	3.57E+06	5.55E+05	1.81E+06	5.52E+06	3.63E+06	5.52E+05	3.53E+05

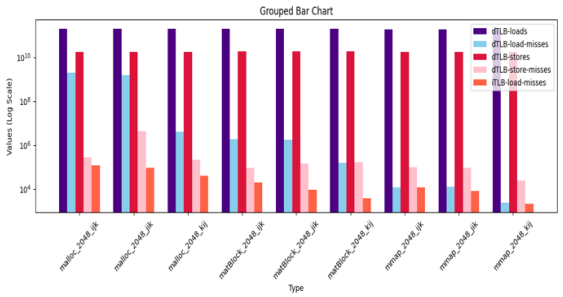
- **Graph for cache data :-**



3.1.2. TLB statistics

Type	dTLB-loads	dTLB-load-misse	dTLB-stores	dTLB-store-misse	iTLB-load-miss
malloc_2048_ijk	1.90E+11	1.91E+09	1.78E+10	271319.00	115424.00
malloc_2048_jik	1.90E+11	1.52E+09	1.77E+10	4287852.00	88877.00
malloc_2048_kij	1.90E+11	4.03E+06	1.78E+10	211164.00	39377.00
matBlock_2048_ijk	2.00E+11	1.85E+06	1.80E+10	89240.00	19339.00
matBlock_2048_jik	2.00E+11	1.72E+06	1.80E+10	141224.00	8840.00
matBlock_2048_kij	2.00E+11	1.54E+05	1.80E+10	160920.00	3710.00
mmap_2048_ijk	1.90E+11	1.19E+04	1.77E+10	96372.00	11773.00
mmap_2048_jik	1.90E+11	1.27E+04	1.76E+10	92425.00	7917.00
mmap_2048_kij	1.90E+11	2.42E+03	1.76E+10	23914.00	2078.00

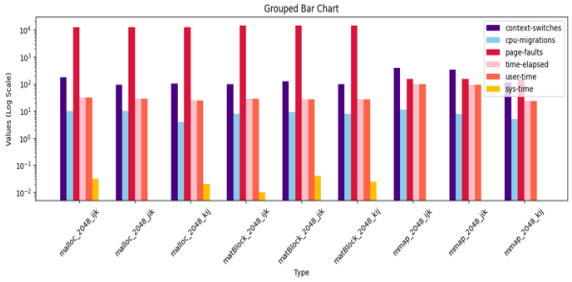
- **Graph for TLB data :-**



3.1.3. Other important statistics

Type	context-swit	cpu-migrations	page-faults	time-elapsed	user-time	sys-time
malloc_2048_ijk	173.00	10.00	12450.00	31.00	31.36	0.03
malloc_2048_jik	92.00	10.00	12450.00	28.00	28.00	0.00
malloc_2048_kij	103.00	4.00	12450.00	24.07	24.00	0.02
matBlock_2048_ijk	99.00	8.00	14504.00	28.34	28.33	0.01
matBlock_2048_jik	127.00	9.00	14505.00	27.66	27.62	0.04
matBlock_2048_kij	98.00	8.00	14506.00	27.61	27.59	0.02
mmap_2048_ijk	383.00	11.00	154.00	98.51	98.50	0.00
mmap_2048_jik	332.00	8.00	155.00	92.09	92.00	0.00
mmap_2048_kij	114.00	5.00	153.00	23.61	23.61	0.00

- **Graph for other data :-**



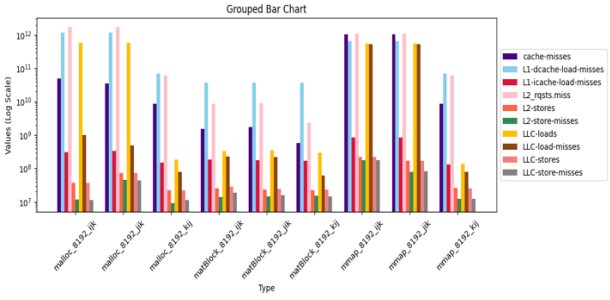
b. Statistics for 8192 variants

3.2.1. Cache statistics

Type	cache-misses	L1-dcache-loa	L1-dcache-lo	L1-icache-loa	L2_rqsts.mis	L2-loads
malloc_8192_ijk	5.02E+10	1.21E+13	1.20E+12	3.01E+08	1.74E+12	5.75E+11
malloc_8192_jik	3.50E+10	1.21E+13	1.17E+12	3.34E+08	1.75E+12	5.73E+11
malloc_8192_kij	8.64E+09	1.21E+13	6.89E+10	1.51E+08	6.19E+10	1.82E+08
matBlock_8192_ijk	1.54E+09	1.27E+13	3.61E+10	1.86E+08	8.84E+09	3.28E+08
matBlock_8192_jik	1.73E+09	1.27E+13	3.62E+10	1.79E+08	9.21E+09	3.46E+08
matBlock_8192_kij	5.72E+08	1.27E+13	3.59E+10	1.73E+08	2.32E+09	2.89E+08
mmap_8192_ijk	1.05E+12	1.21E+13	6.55E+11	8.46E+08	1.07E+12	5.50E+11
mmap_8192_jik	1.03E+12	1.21E+13	6.55E+11	8.30E+08	1.09E+12	5.50E+11
mmap_8192_kij	8.75E+09	1.21E+13	6.85E+10	1.30E+08	6.15E+10	1.40E+08

Type	L2-load-misses	L2-stores	L2-store-misses	LLC-loads	LLC-load-misses	LLC-stores	LLC-store-misses
malloc_8192_ijk	1.02E+09	3.62E+07	1.16E+07	5.75E+11	1.02E+09	3.62E+07	1.15E+07
malloc_8192_jik	4.91E+08	7.40E+07	4.51E+07	5.73E+11	4.90E+08	7.27E+07	4.46E+07
malloc_8192_kij	7.78E+07	2.26E+07	9.17E+06	1.82E+08	7.78E+07	2.27E+07	1.11E+07
matBlock_8192_	2.27E+08	2.53E+07	1.37E+07	3.29E+08	2.28E+08	2.91E+07	1.86E+07
matBlock_8192_	2.18E+08	2.28E+07	1.48E+07	3.44E+08	2.17E+08	2.38E+07	1.56E+07
matBlock_8192_	6.33E+07	2.23E+07	1.52E+07	2.89E+08	6.26E+07	2.33E+07	1.43E+07
mmap_8192_ijk	5.37E+11	2.23E+08	1.80E+08	5.50E+11	5.36E+11	2.18E+08	1.74E+08
mmap_8192_jik	5.36E+11	1.68E+08	7.87E+07	5.50E+11	5.36E+11	1.70E+08	8.10E+07
mmap_8192_kij	7.87E+07	2.64E+07	1.24E+07	1.40E+08	7.87E+07	2.53E+07	1.24E+07

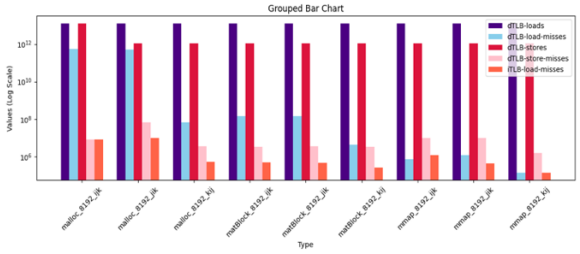
- **Graph for cache data :-**



3.2.2. TLB statistics

Type	dTLB-loads	dTLB-load-miss	dTLB-stores	dTLB-store-m	iTLB-load-m
malloc_8192_ijk	1.21E+13	5.42E+11	1.21E+13	8.27E+06	8.38E+06
malloc_8192_jik	1.21E+13	5.16E+11	1.11E+12	6.85E+07	9.71E+06
malloc_8192_kij	1.21E+13	6.92E+07	1.11E+12	3.54E+06	5.37E+05
matBlock_8192_ijk	1.27E+13	1.46E+08	1.13E+12	3.42E+06	5.01E+05
matBlock_8192_jik	1.27E+13	1.44E+08	1.13E+12	3.47E+06	4.58E+05
matBlock_8192_kij	1.27E+13	4.18E+06	1.13E+12	3.31E+06	2.55E+05
mmap_8192_ijk	1.21E+13	7.35E+05	1.11E+12	9.75E+06	1.22E+06
mmap_8192_jik	1.21E+13	1.23E+06	1.11E+12	1.00E+07	4.40E+05
mmap_8192_kij	1.21E+13	1.38E+05	1.11E+12	1.54E+06	1.41E+05

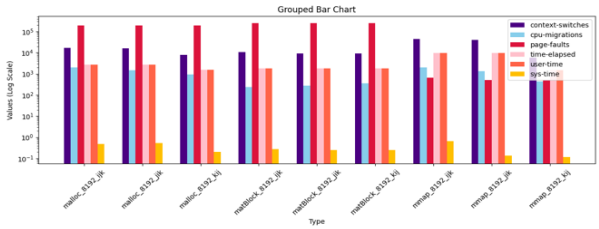
- **Graph for TLB data :-**



3.2.3. Other important statistics

Type	context-switc	cpu-migration	page-faults	time-elapsed	user-time	sys-time
malloc_8192_ijk	17336.00	1968.00	196877.00	2747.53	2746.88	0.48
malloc_8192_jik	16285.00	1441.00	197209.00	2763.07	2762.36	0.52
malloc_8192_kij	7982.00	925.00	196877.00	1519.99	1519.72	0.20
matBlock_8192_ijk	10819.00	233.00	254253.00	1769.00	1769.14	0.27
matBlock_8192_jik	9264.00	271.00	254253.00	1766.08	1765.78	0.25
matBlock_8192_kij	8973.00	358.00	254253.00	1765.37	1765.08	0.25
mmap_8192_ijk	44,801	2,038	640	9510.39628	9509.32	0.66
mmap_8192_jik	40,715	1,344	515	9651.14	9650.7445	0.132
mmap_8192_kij	5,964	434	514	1510.02826	1509.8698	0.12

- **Graph for other data :-**



4. Results and Conclusion

We have tried to get a detailed comparison between three memory allocation strategies: malloc, mmap and tiling each applied to array sizes of 2048x2048 and 8192x8192 with access patterns (i,j,k), (j,i,k), and (k,i,j).

Starting with the part(a) which uses **malloc** with standard matrix multiplication, we observe that its performance scales variably with both array size and access pattern. For instance, in the case of the larger array size, i.e. 8192x8192, the (i,j,k) access pattern consumes a significantly higher task clock and cycle count, indicating more intensive computational load and inefficient cache usage. The malloc_8192_ijk configuration cycles indicate that it the slowest among the malloc configurations. The same pattern is visible in its IPC and cache misses. In contrast, malloc_2048_kij demonstrates more efficient performance, suggesting that the (k,i,j) access pattern might align better with memory locality, leading to fewer cache misses and faster execution.

For part(b), which uses **mmap** method, used for large memory regions(huge pages), presents a different performance. **mmap** allows for the management of large data, it incurs significant operational overhead, especially in larger array sizes. For instance, the mmap_8192_jik configuration result values for branch misses and dTLB loads, suggesting that excessive paging and memory management operations severely affect its performance. The extreme number of cache misses and branch mispredictions in these configurations indicates that mmap struggles to handle the same workloads as efficiently as malloc or tiling. The access pattern seems to have a profound effect as well, while (i,j,k) and (j,i,k) configurations exhibit higher task clocks, (k,i,j) remains a relatively better choice across all. However, even with (k,i,j), the operational cost of mmap remains high due to the frequent context switches. Also, page faults decreasing significantly but still not helping much in performance as other computations are being trade off. The high number of dTLB loads and cache misses suggests that while mmap is efficient in managing large memory regions, its memory access patterns and paging behavior result in significant penalties.

Finally for part(c), tiling approach, designed to optimize memory access in blocked matrix operations, shows a different trend. It exhibits relatively balanced performance across most configurations, with the 2048_kij access pattern standing out as the most efficient and giving task clock which is lowest in the entire dataset. This suggests that tiling, when paired with smaller arrays and (k,i,j) access pattern, is highly effective at managing cache utilization and minimizing memory latency. On the other hand, tiling for 8192_ijk demonstrates the highest cycle count and task clock, indicating that larger memory sizes can overwhelm this strategy unless combined with an optimal access pattern. This highlights that while tiling can be efficient, especially in smaller arrays, its efficiency diminishes with larger datasets unless access patterns are structured to take advantage of cache locality.

In conclusion, the overall analysis shows that no single memory allocation strategy outperforms the other across all parameters. Rather, the selection of a memory allocation approach should be carefully aligned with the data size and the particular access pattern which will be employed. For smaller to moderate memory regions, both malloc and tiling demonstrate better performance, with (k,i,j) access patterns consistently leading to fewer cycles and cache misses. However, for larger datasets, while mmap allows for efficient memory mapping, it incurs substantial performance penalties due to paging and memory management overhead. Therefore, care selection of the appropriate memory allocation strategy based requirements of application, keeping in mind the trade-offs between computational efficiency and memory management overhead.

5. References

[1] Linux Kernel Profiling with perf. Available at: https://perf.wiki.kernel.org/index.php/Tutorial

[2] Large Page Support in the Linux kernel: https://linuxgazette.net/155/krishnakumar.html

[3] mmap-Linux manual page https://man7.org/linux/manpages/man2/mmap.2.html

[4] https://chatgpt.com/

[5] https://www.google.com/

[6] https://github.com/