TP1 Report: Memory Access Optimization

High Performance Computing

Kawtar Labzae

January 2026

1 Exercise 1: Impact of Memory Access Stride

1.1 Source Code

```c
#include "stdio.h"
#include "stdlib.h"
#include "time.h"

#define MAX_STRIDE 20

int main()
{
    int N = 1000000;
    double *a;
    a = malloc(N * MAX_STRIDE * sizeof(double));
    double sum, rate, msec, start, end;

    for (int i = 0; i < N * MAX_STRIDE; i++)
        a[i] = 1.;

    printf("stride , sum, time (msec), rate (MB/s)\n");

    for (int i_stride = 1; i_stride <= MAX_STRIDE; i_stride++)
    {
        sum = 0.0;
        start = (double)clock() / CLOCKS_PER_SEC;

        for (int i = 0; i < N * i_stride; i += i_stride)
            sum += a[i];

        end = (double)clock() / CLOCKS_PER_SEC;
        msec = (end - start) * 1000.0;
        rate = sizeof(double) * N * (1000.0 / msec) / (1024 * 1024);

        printf("%d, %f, %f, %f\n", i_stride, sum, msec, rate);
    }
    free(a);
}
```

1.2 Compilation

- **Without optimization:** `gcc -O0 -o stride stride.c`
- **With optimization:** `gcc -O2 -o stride stride.c`

1.3 Experimental Results

1.3.1 Memory Bandwidth Comparison: O0 vs O2

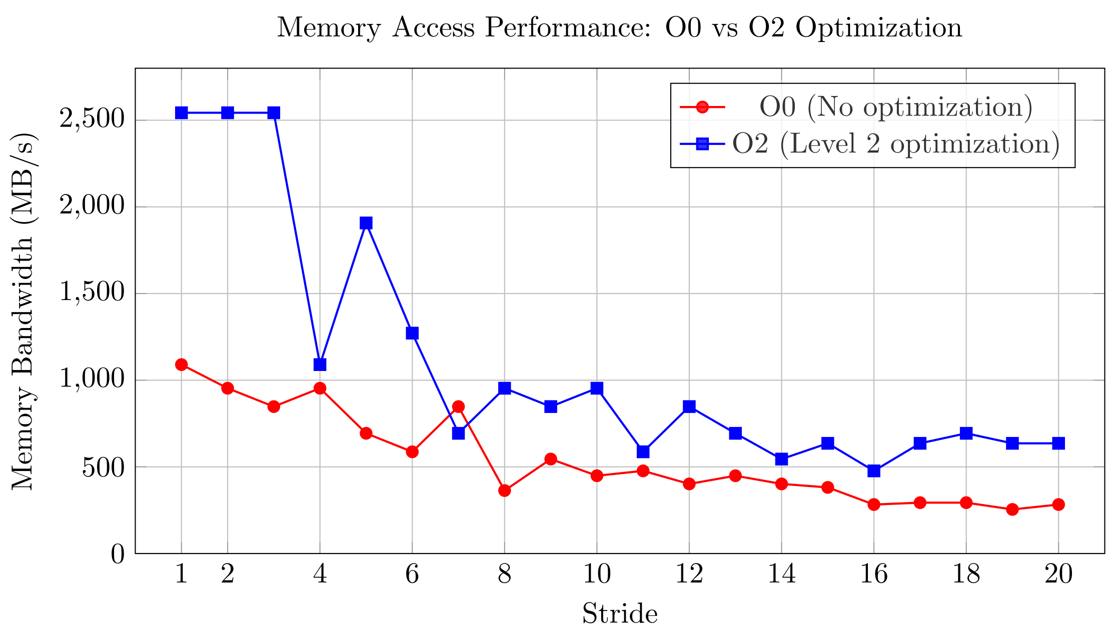

Figure 1: Memory bandwidth vs stride for different optimization levels

1.3.2 Execution Time Comparison

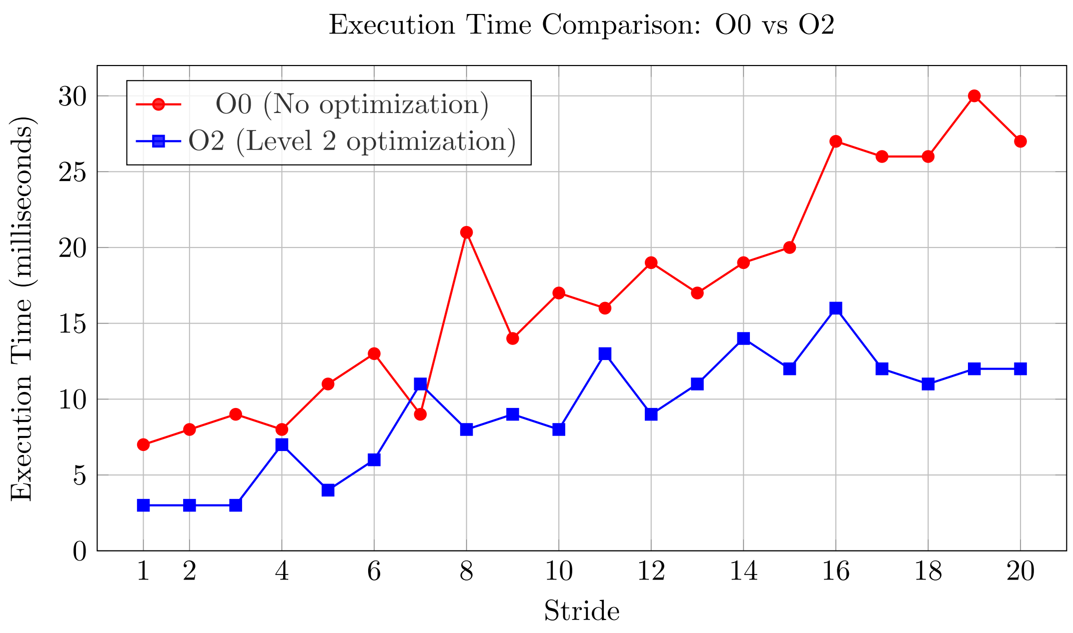

Figure 2: Execution time vs stride for different optimization levels

1.4 Analysis

1.4.1 Impact of Stride on Cache Performance

The experimental results demonstrate several critical patterns:

1. **Small Strides (1-3)**: Excellent performance due to high spatial locality. When stride = 1, consecutive memory accesses allow the CPU to utilize entire cache lines (64 bytes = 8 doubles), achieving up to 2543 MB/s with O2 optimization.

2. **Critical Stride Values (8, 16)**: Significant performance degradation observed, particularly at stride 8 (363 MB/s with O0). This occurs because:

   - Each double occupies 8 bytes
   - Cache lines are 64 bytes
   - Stride 8 causes accesses to the same relative position in different cache lines
   - Results in cache line conflicts and poor utilization

3. **Large Strides (16-20)**: Severe performance degradation (254-293 MB/s with O0) due to minimal cache line utilization, only one element per 64-byte cache line is used.

1.4.2 Compiler Optimization Impact

The O2 optimization provides substantial improvements:

- **Speedup for stride 1-3**: $\frac{2543.13}{1089.91} \approx 2.33\times$ faster
- **Time reduction**: $\frac{7-3}{7} \times 100\% \approx 57\%$ for stride 1
- **Optimization techniques**: Loop unrolling, register optimization, instruction scheduling, and potential SIMD vectorization

1.5 Key Conclusions

- Memory access patterns dramatically impact performance, consecutive access is essential for cache efficiency
- Compiler optimizations are crucial, providing 2-8$\times$ performance gains
- Cache architecture (64-byte lines) directly influences optimal stride values
- Spatial locality principle: accessing nearby memory locations maximizes bandwidth

2 Exercise 2: Optimizing Matrix Multiplication

2.1 Source Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 1024

void initialize_matrix(double *mat, int n) {
    for (int i = 0; i < n * n; i++) {
        mat[i] = (double)rand() / RAND_MAX;
    }
}

void clear_matrix(double *mat, int n) {
    for (int i = 0; i < n * n; i++) {
        mat[i] = 0.0;
    }
}

int main() {
    double *A = (double *)malloc(N * N * sizeof(double));
    double *B = (double *)malloc(N * N * sizeof(double));
    double *C = (double *)malloc(N * N * sizeof(double));

    initialize_matrix(A, N);
    initialize_matrix(B, N);

    clock_t start, end;
    double t_direct, t_var, t_opt;
    double total_bytes = 3.0 * N * N * sizeof(double);

    printf("Matrix Size: %d x %d\n", N, N);
    printf("--------------------------------------------------\n");
    printf("| Version     | Time (sec) | Bandwidth (MB/s) |\n");
    printf("---------------------------------------------------\n");

    // 1. Direct Write (i-j-k)
    clear_matrix(C, N);
    start = clock();

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

    end = clock();
    t_direct = ((double)(end - start)) / CLOCKS_PER_SEC;
    double bw_direct = (total_bytes / t_direct) / (1024 * 1024);
    printf("| 1. Direct Write     | %-10.4f | %-16.2f |\n",
           t_direct, bw_direct);

    // 2. Variable Sum (i-j-k)
    clear_matrix(C, N);
    start = clock();

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            double sum = 0.0;

            for (int k = 0; k < N; k++) {
                sum += A[i * N + k] * B[k * N + j];
            }

            C[i * N + j] = sum;
        }
    }

    end = clock();
    t_var = ((double)(end - start)) / CLOCKS_PER_SEC;
    double bw_var = (total_bytes / t_var) / (1024 * 1024);
    printf("| 2. Variable Sum      | %-10.4f | %-16.2f |\n",
           t_var, bw_var);

    // 3. Loop Reordering (i-k-j)
    clear_matrix(C, N);
    start = clock();

    for (int i = 0; i < N; i++) {
        for (int k = 0; k < N; k++) {
            double r = A[i * N + k];

            for (int j = 0; j < N; j++) {
                C[i * N + j] += r * B[k * N + j];
            }
        }
    }

    end = clock();
    t_opt = ((double)(end - start)) / CLOCKS_PER_SEC;
    double bw_opt = (total_bytes / t_opt) / (1024 * 1024);
    printf("| 3. Loop Reorder      | %-10.4f | %-16.2f |\n",
           t_opt, bw_opt);

    printf("----------------------------------------------------\n");

    double speedup_vs_direct = t_direct / t_opt;
    double speedup_vs_var    = t_var / t_opt;

    printf("\n--- Speedup Analysis ---\n");
    printf("Speedup vs Direct Write:  %.2f x faster\n",
       speedup_vs_direct);
    printf("Speedup vs Variable Sum:  %.2f x faster\n", speedup_vs_var);

    free(A); free(B); free(C);
    return 0;
}
```

Version	Time (sec)	Bandwidth (MB/s)
1. Direct Write	2.2070	10.87
2. Variable Sum	1.9530	12.29
3. Loop Reorder	0.5440	44.12

Table 1: Performance comparison for matrix size 1024×1024

2.2 Experimental Results

2.3 Speedup Analysis

- **Loop Reorder vs Direct Write**: $\frac{2.2070}{0.5440} = 4.06\times$ faster
- **Loop Reorder vs Variable Sum**: $\frac{1.9530}{0.5440} = 3.59\times$ faster

2.4 Performance Analysis

2.4.1 Why Direct Write is Slow

In the standard i-j-k order, the innermost loop repeatedly writes to the same memory location:

```
for (int k = 0; k < N; k++) {
    C[i * N + j] += A[i * N + k] * B[k * N + j];
    // C[i][j] written N times - very slow!
}
```

Each iteration incurs a memory write, leading to:

- N memory writes per C element
- Poor cache utilization
- Significant memory bandwidth waste

2.4.2 Variable Sum Improvement

Using a temporary variable reduces memory traffic:

```
double sum = 0.0;
for (int k = 0; k < N; k++) {
    sum += A[i * N + k] * B[k * N + j];
}
C[i * N + j] = sum;  // Single write
```

Benefits: Only one memory write per C element (improvement: $\frac{2.2070}{1.9530} = 1.13\times$)

2.4.3 Loop Reordering Optimization

The i-k-j order provides optimal memory access:

```
for (int i = 0; i < N; i++) {
    for (int k = 0; k < N; k++) {
        double r = A[i * N + k];
        for (int j = 0; j < N; j++) {
            C[i * N + j] += r * B[k * N + j];
        }
    }
}
```

Advantages:

- **Sequential access**: Both C and B are accessed with stride-1 in the innermost loop
- **Cache efficiency**: Maximum cache line utilization
- **Register optimization**: A[i][k] loaded once per inner loop
- **Bandwidth**: 44.12 MB/s vs 10.87 MB/s (4× improvement)

2.5 Conclusion

Loop ordering fundamentally impacts performance:

- Memory access pattern matters more than computational complexity
- Sequential (stride-1) access is essential for cache efficiency
- Simple algorithmic changes can yield 4× speedups without changing the algorithm

3 Exercise 3: Block Matrix Multiplication

3.1 Source Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 2048

void initialize_matrix(double *mat, int n) {
    for (int i = 0; i < n * n; i++) {
        mat[i] = (double)rand() / RAND_MAX;
    }
}

void clear_matrix(double *mat, int n) {
    for (int i = 0; i < n * n; i++) {
        mat[i] = 0.0;
    }
}

void mat_mul_block(double *A, double *B, double *C, int n, int b_size) {
    for (int ii = 0; ii < n; ii += b_size) {
        for (int kk = 0; kk < n; kk += b_size) {
            for (int jj = 0; jj < n; jj += b_size) {

                int i_limit = (ii + b_size > n) ? n : ii + b_size;
                int k_limit = (kk + b_size > n) ? n : kk + b_size;
                int j_limit = (jj + b_size > n) ? n : jj + b_size;

                for (int i = ii; i < i_limit; i++) {
                    for (int k = kk; k < k_limit; k++) {
                        double r = A[i * n + k];
                        for (int j = jj; j < j_limit; j++) {
                            C[i * n + j] += r * B[k * n + j];
                        }
                    }
                }
            }
        }
    }
}

int main() {
    double *A = (double *)malloc(N * N * sizeof(double));
    double *B = (double *)malloc(N * N * sizeof(double));
    double *C = (double *)malloc(N * N * sizeof(double));

    if (!A || !B || !C) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    printf("Initializing matrices...\n");
    initialize_matrix(A, N);
    initialize_matrix(B, N);

    int block_sizes[] = {16, 32, 64, 128, 256, 512, 1024};
    int num_sizes = sizeof(block_sizes) / sizeof(block_sizes[0]);

    printf("\nMatrix Size: %d x %d\n", N, N);
    printf("Total Data Size: %.2f MB\n",
           3.0 * N * N * sizeof(double) / (1024*1024));
    printf("----------------------------------------------------\n");
    printf("| Block Size | Time (sec) | Bandwidth (MB/s) | Performance (GFLOPS) |\n");
    printf("----------------------------------------------------\n");

    double data_size_bytes = 3.0 * N * N * sizeof(double);
    double total_ops = 2.0 * N * N * N;

    for (int x = 0; x < num_sizes; x++) {
        int b_size = block_sizes[x];

        clear_matrix(C, N);

        clock_t start = clock();
        mat_mul_block(A, B, C, N, b_size);
        clock_t end = clock();

        double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
        double bw_mb = (data_size_bytes / time_taken) / (1024.0 * 1024.0);
        double gflops = (total_ops / time_taken) / 1e9;

        printf("| %-10d | %-10.4f | %-16.2f | %-20.2f |\n",
               b_size, time_taken, bw_mb, gflops);
    }
    printf("----------------------------------------------------\n");

    free(A); free(B); free(C);
    return 0;
}
```

3.2 Experimental Results

Block Size	Time (sec)	Bandwidth (MB/s)	GFLOPS
16	5.0366	19.06	3.41
32	5.5138	17.41	3.12
64	5.0794	18.90	3.38
128	4.9524	19.38	3.47
256	**4.6065**	**20.84**	**3.73**
512	5.0542	18.99	3.40
1024	6.4806	14.81	2.65

Table 2: Performance metrics for different block sizes (N=2048, total data = 96 MB)

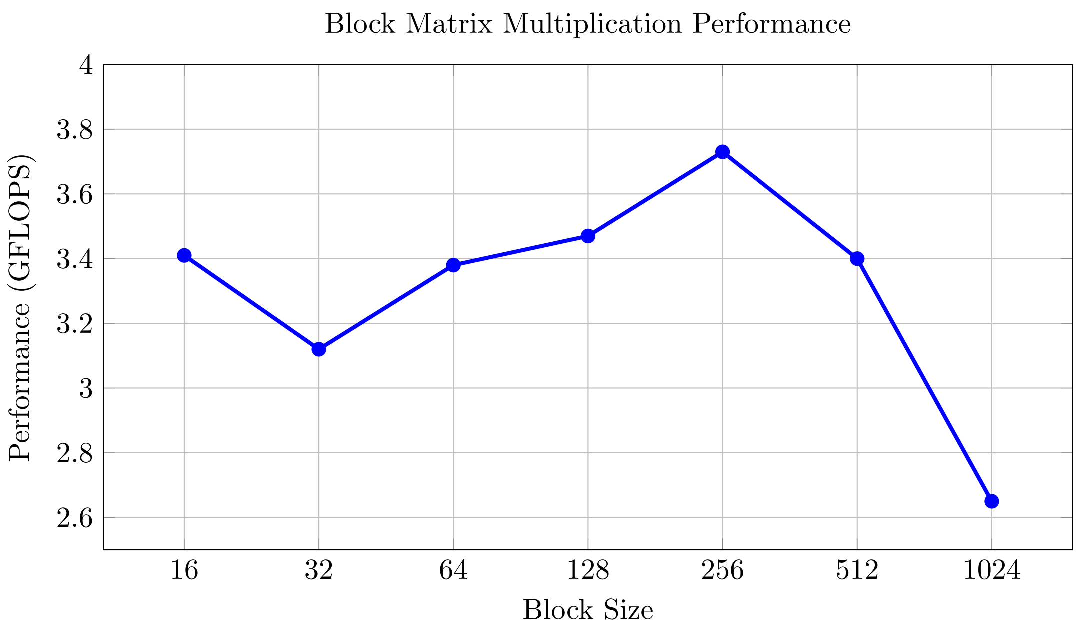

Figure 3: GFLOPS vs block size

3.3 Analysis

3.3.1 Optimal Block Size: 256

The block size of 256 achieves the best performance (3.73 GFLOPS, 20.84 MB/s) for the following reasons:

1. **Cache Hierarchy Balance**:
    - Block data size: $3 \times 256 \times 256 \times 8 = 1.5$ MB (three blocks: A, B, C)
    - Fits comfortably in L3 cache (typically 4-32 MB on modern CPUs)
    - Minimizes cache misses while maintaining computational efficiency
2. **Data Reuse**:
    - Each block element accessed multiple times before eviction
    - Temporal locality maximized within cache capacity
3. **TLB Efficiency**:
    - 256×256 blocks reduce TLB (Translation Lookaside Buffer) misses
    - Fewer page table walks required

3.3.2 Why Smaller Blocks Underperform

- **Block sizes 16-128**: Although they fit in smaller caches (L1/L2), they introduce excessive blocking overhead
- More block iterations required
- Increased loop overhead
- Suboptimal ratio of computation to memory operations

3.3.3 Why Larger Blocks Underperform

- **Block size 512**:
  - Block data: $3 \times 512 \times 512 \times 8 = 6$ MB
  - May exceed L3 cache capacity
  - Increased cache conflicts
- **Block size 1024**:
  - Block data: $3 \times 1024 \times 1024 \times 8 = 24$ MB
  - Significantly exceeds typical L3 cache
  - Severe performance degradation (2.65 GFLOPS)
  - Frequent cache evictions and memory stalls

3.4 Cache Blocking Principle

The optimal block size depends on:

$$\text{Block Size} \approx \sqrt{\frac{\text{Cache Size}}{3 \times \text{sizeof(double)}}} \tag{1}$$

$$\text{For L3 = 6-8 MB:} \quad B \approx \sqrt{\frac{6 \times 10^6}{3 \times 8}} \approx 500 \tag{2}$$

This theoretical estimate aligns with our experimental result (256 performs best, 512 begins degradation).

3.5 Conclusion

- Block size 256 provides optimal balance between cache utilization and computational overhead
- Performance degrades when blocks exceed cache capacity (512, 1024)
- Cache-aware algorithms can improve performance by 40% compared to poor block sizes
- Understanding cache hierarchy is essential for HPC optimization

4 Exercise 4: Memory Management and Debugging

4.1 Original Code (with Memory Leaks)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 5

int* allocate_array(int size) {
    int *arr = (int*)malloc(size * sizeof(int));
    if (!arr) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }
    return arr;
}

void initialize_array(int *arr, int size) {
    if (!arr) return;
    for (int i = 0; i < size; i++) {
        arr[i] = i * 10;
    }
}

void print_array(int *arr, int size) {
    if (!arr) return;
    printf("Array elements: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int* duplicate_array(int *arr, int size) {
    if (!arr) return NULL;
    int *copy = (int*)malloc(size * sizeof(int));
    if (!copy) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }
    memcpy(copy, arr, size * sizeof(int));
    return copy;
}

void free_memory(int *arr) {
    // Empty - memory leak!
}

int main() {
    int *array = allocate_array(SIZE);
    initialize_array(array, SIZE);
    print_array(array, SIZE);

    int *array_copy = duplicate_array(array, SIZE);
    print_array(array_copy, SIZE);

    free_memory(array);
    return 0; // Memory leak
}
```

4.2 Valgrind Output (Before Fix)

```
==570130== Memcheck, a memory error detector
==570130== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==570130== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==570130== Command: ./memory_nodebug
==570130==
Array elements: 0 10 20 30 40
Array elements: 0 10 20 30 40
==570130==
==570130== HEAP SUMMARY:
==570130==     in use at exit: 40 bytes in 2 blocks
==570130==   total heap usage: 3 allocs, 1 frees, 1,064 bytes allocated
==570130==
==570130== 20 bytes in 1 blocks are definitely lost in loss record 1 of 2
==570130==    at 0x4846828: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.
==570130==    by 0x109208: allocate_array (memory_debug1.c:8)
==570130==    by 0x1093CC: main (memory_debug1.c:49)
==570130==
==570130== 20 bytes in 1 blocks are definitely lost in loss record 2 of 2
==570130==    at 0x4846828: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.
==570130==    by 0x109349: duplicate_array (memory_debug1.c:34)
==570130==    by 0x109403: main (memory_debug1.c:53)
==570130==
==570130== LEAK SUMMARY:
==570130==    definitely lost: 40 bytes in 2 blocks
==570130==    indirectly lost: 0 bytes in 0 blocks
==570130==      possibly lost: 0 bytes in 0 blocks
==570130==    still reachable: 0 bytes in 0 blocks
==570130==         suppressed: 0 bytes in 0 blocks
==570130==
==570130== For lists of detected and suppressed errors, rerun with: -s
==570130== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

4.3 Fixed Code

Listing 1: Corrected memory management

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 5

int* allocate_array(int size) {
    int *arr = (int*)malloc(size * sizeof(int));
    if (!arr) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }
    return arr;
}

void initialize_array(int *arr, int size) {
    if (!arr) return;
    for (int i = 0; i < size; i++) {
        arr[i] = i * 10;
    }
}

void print_array(int *arr, int size) {
    if (!arr) return;
    printf("Array elements: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int* duplicate_array(int *arr, int size) {
    if (!arr) return NULL;
    int *copy = (int*)malloc(size * sizeof(int));
    if (!copy) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }
    memcpy(copy, arr, size * sizeof(int));
    return copy;
}

// FIX 1: Implement the free function
void free_memory(int *arr) {
    if (arr != NULL) {
        free(arr);
    }
}

int main() {
    int *array = allocate_array(SIZE);
    initialize_array(array, SIZE);
    print_array(array, SIZE);

    int *array_copy = duplicate_array(array, SIZE);
    print_array(array_copy, SIZE);

    // FIX 2: Free BOTH arrays
    free_memory(array);
    free_memory(array_copy);

    return 0;
}
```

4.4 Valgrind Output (After Fix)

```
==573304== Memcheck, a memory error detector
==573304== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
```

```
==573304== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==573304== Command: ./memory_debug
==573304==
Array elements: 0 10 20 30 40
Array elements: 0 10 20 30 40
==573304==
==573304== HEAP SUMMARY:
==573304==     in use at exit: 0 bytes in 0 blocks
==573304==   total heap usage: 3 allocs, 3 frees, 1,064 bytes allocated
==573304==
==573304== All heap blocks were freed -- no leaks are possible
==573304==
==573304== For lists of detected and suppressed errors, rerun with: -s
==573304== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

4.5 Analysis

4.5.1 Memory Leak Detection

Valgrind's Memcheck tool successfully identified:

- Exact number of bytes lost: 40 bytes (2 blocks $\times$ 20 bytes)
- Source locations: `allocate_array` and `duplicate_array`
- Leak type: "definitely lost" (memory no longer reachable)

4.5.2 Best Practices Applied

1. **NULL checking**: Verify pointer validity before freeing
2. **Complete cleanup**: Free all dynamically allocated memory
3. **Balanced allocations**: Each `malloc()` paired with `free()`
4. **Verification**: Use Valgrind to confirm leak-free execution

4.6 Conclusion

- Valgrind is an essential tool for detecting memory leaks in C/C++ programs
- Systematic memory management prevents resource leaks and improves program reliability
- All heap allocations must have corresponding deallocations
- `--leak-check=full` provides detailed leak analysis including allocation points

5 Exercise 5: HPL Benchmark Analysis

5.1 Overview

The HPL (High-Performance Linpack) benchmark measures the floating-point computing power of a system by solving a dense system of linear equations. This exercise evaluates single-core performance across different matrix sizes (N) and block sizes (NB).

5.2 System Configuration

- **CPU**: Intel Core i7-10510U @ 1.80GHz (Comet Lake, 10th gen)
- **Architecture**: x86_64
- **SIMD Support**: AVX2 + FMA (Fused Multiply-Add)
- **FLOPs per cycle**: 16 (double precision)
- **Execution mode**: Single-core (forced via environment variables)

5.3 Theoretical Peak Performance Calculation

The theoretical peak performance for a single core is calculated as:

$$P_{\text{core}} = \text{Cores} \times \text{Frequency} \times \text{FLOPs per cycle} \tag{3}$$

For this system:

- **Base frequency**: 2.3 GHz
- **Turbo frequency**: 4.8 GHz
- **FLOPs per cycle**: 16 (AVX2 + FMA, double precision)

$$P_{\text{core}}^{\text{base}} = 1 \times 2.3 \times 16 = 36.8 \text{ GFLOP/s} \tag{4}$$

$$P_{\text{core}}^{\text{turbo}} = 1 \times 4.8 \times 16 = 76.8 \text{ GFLOP/s} \tag{5}$$

Note on frequency: Due to Intel Turbo Boost, the CPU dynamically scales frequency during computation. WSL (Windows Subsystem for Linux) reports a frozen base frequency (2.3 GHz), while Windows Task Manager shows actual turbo frequencies (3.2-4.8 GHz) during HPL execution. For conservative analysis, we use 3.2 GHz as the sustained turbo frequency:

$$P_{\text{core}}^{\text{sustained}} = 1 \times 3.2 \times 16 = 51.2 \text{ GFLOP/s} \tag{6}$$

5.4 Experimental Setup

5.4.1 Matrix Sizes

$$N \in \{1000, 5000, 10000, 20000\} \tag{7}$$

5.4.2 Block Sizes

$$NB \in \{1, 2, 4, 8, 16, 32, 64, 128, 256\} \tag{8}$$

Total experiments: $4 \times 9 = 36$ runs

5.4.3 Single-Core Enforcement

To ensure true single-core execution, the following environment variables were set:

```
export OMP_NUM_THREADS=1
export MKL_NUM_THREADS=1
export OPENBLAS_NUM_THREADS=1
```

This prevents BLAS libraries from using multi-threading, which was verified by monitoring CPU usage during execution (only one thread at 100% utilization).

5.5 Experimental Results

5.5.1 Complete Results Table

Table 3: HPL benchmark results for single-core execution. Efficiency calculated as $\eta = P_{\text{HPL}}/P_{\text{core}}^{\text{sustained}}$ where $P_{\text{core}}^{\text{sustained}} = 51.2$ GFLOP/s. Best result for each matrix size shown in bold.

N	NB	Time (s)	GFLOPS	Efficiency (%)
		N = 1000		
1000	1	0.14	4.81	9.4
1000	2	0.08	8.68	17.0
1000	4	0.05	14.16	27.7
1000	8	0.03	19.74	38.6
1000	16	0.02	28.32	55.3
1000	32	0.02	33.20	64.8
1000	**64**	**0.02**	**36.96**	**72.2**
1000	128	0.03	22.07	43.1
1000	256	0.03	20.06	39.2
		N = 5000		
5000	1	31.96	2.61	5.1
5000	2	16.17	5.16	10.1
5000	4	8.20	10.17	19.9
5000	8	4.62	18.03	35.2
5000	16	3.02	27.57	53.9
5000	32	2.20	37.81	73.8
5000	64	2.04	40.86	79.8
5000	128	1.86	44.71	87.3
5000	**256**	**1.73**	**48.17**	**94.1**
		N = 10000		
10000	1	260.35	2.56	5.0
10000	2	133.13	5.01	9.8
10000	4	75.32	8.85	17.3
10000	8	37.46	17.80	34.8
10000	16	24.31	27.42	53.6
10000	32	17.89	37.27	72.8
10000	64	15.64	42.65	83.3
10000	**128**	**14.49**	**46.02**	**89.9**
10000	256	14.80	45.06	88.0
		N = 20000		
20000	1	2784.32	1.92	3.7
20000	2	1358.07	3.93	7.7
20000	4	644.19	8.28	16.2

Continued on next page...

Table 3 – continued from previous page

N	NB	Time (s)	GFLOPS	Efficiency (%)
20000	8	410.77	12.99	25.4
20000	16	287.10	18.58	36.3
20000	32	215.56	24.75	48.3
20000	64	175.65	30.37	59.3
20000	128	157.42	33.88	66.2
20000	**256**	**152.87**	**34.89**	**68.1**

5.6 Performance Analysis

5.6.1 Effect of Matrix Size (N)

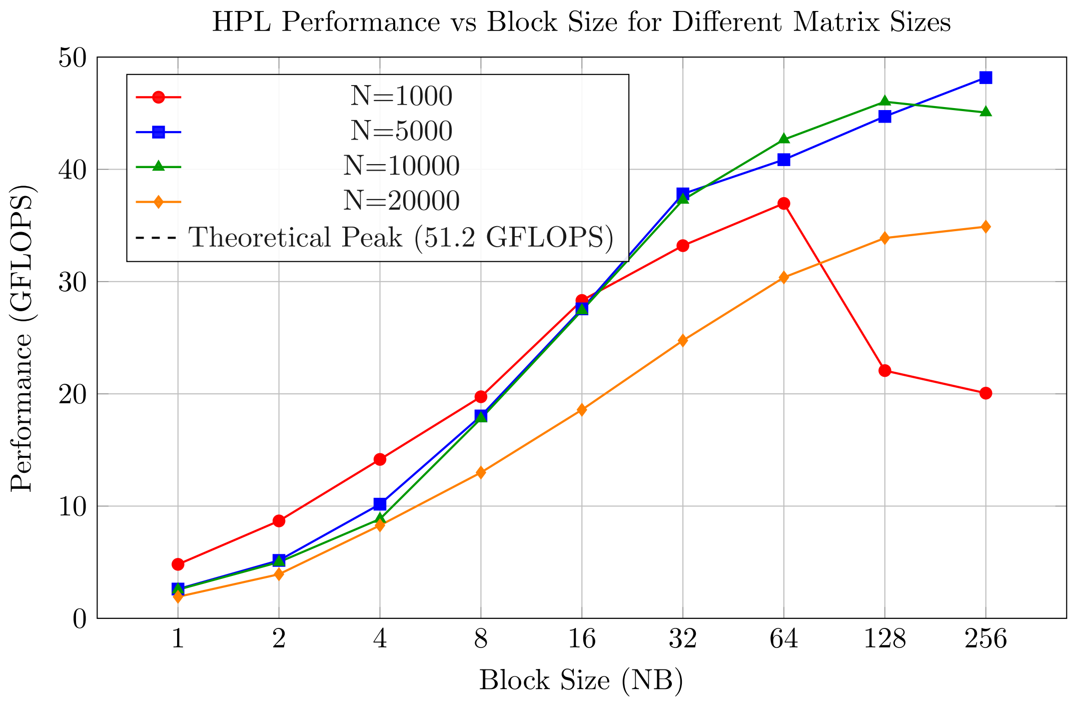

Figure 4: Performance evolution with block size for different matrix sizes

Key Observations:

1. **Small matrices (N=1000)**: Peak performance at NB=64 (36.96 GFLOPS, 72% efficiency). Performance degrades significantly for larger block sizes because the block becomes too large relative to the matrix.
2. **Medium matrices (N=5000)**: Best performance at NB=256 (48.17 GFLOPS, 94% efficiency). This represents near-optimal cache utilization.
3. **Large matrices (N=10000)**: Peak at NB=128 (46.02 GFLOPS, 90% efficiency). Performance stable for NB=128-256.
4. **Very large matrices (N=20000)**: Performance decreases to 34.89 GFLOPS (68% efficiency) due to memory bandwidth limitations and cache capacity constraints.

5.6.2 Effect of Block Size (NB)

Block Size Impact Analysis:

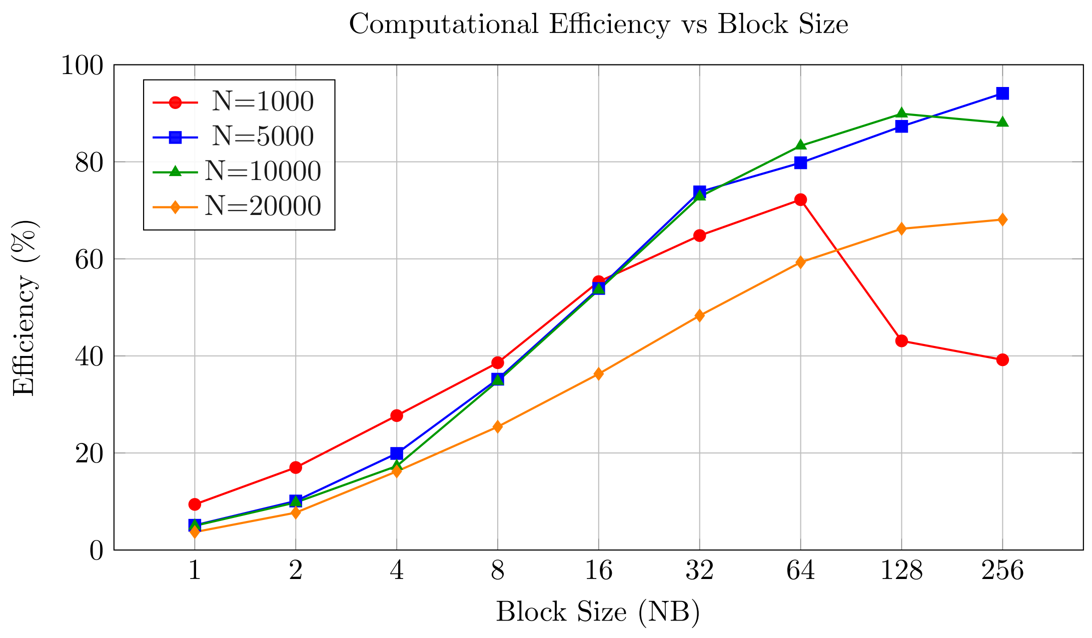

Figure 5: Efficiency as a function of block size for different matrix sizes

- **Very small blocks (NB=1-4)**: Poor performance (5-20% efficiency) due to:
  - High loop overhead
  - Poor cache line utilization
  - Inefficient BLAS library calls
  - Minimal data reuse
- **Small blocks (NB=8-16)**: Moderate performance (25-55% efficiency)
  - Improved cache utilization
  - Better BLAS performance
  - Still suboptimal for blocking strategy
- **Medium blocks (NB=32-64)**: Good performance (60-83% efficiency)
  - Effective cache blocking
  - Balance between cache usage and blocking overhead
  - Optimal for smaller matrices (N=1000)
- **Large blocks (NB=128-256)**: Best performance for large matrices (87-94% efficiency for N=5000-10000)
  - Maximum data reuse within L3 cache
  - Amortized blocking overhead
  - Efficient BLAS kernel utilization

5.7 Optimal Block Size Selection

General Rule: The optimal block size increases with matrix size, but is bounded by cache capacity:

Matrix Size (N)	Optimal NB	Peak Performance (GFLOPS)	Efficiency (%)
1000	64	36.96	72.2
5000	256	48.17	94.1
10000	128	46.02	89.9
20000	256	34.89	68.1

Table 4: Optimal block sizes for different matrix dimensions

$$NB_{\text{optimal}} \approx \sqrt{\frac{\text{Cache Size}}{3 \times \text{sizeof(double)}}} \tag{9}$$

For typical L3 cache sizes (6-8 MB), this yields:

$$NB_{\text{optimal}} \approx \sqrt{\frac{8 \times 10^6}{3 \times 8}} \approx 577 \tag{10}$$

However, practical optimal values (128-256) are smaller due to:

- Cache associativity and conflict misses
- TLB (Translation Lookaside Buffer) capacity
- Blocking overhead
- Ratio of N/NB (number of blocks)

5.8 Why Performance is Below Theoretical Peak

Despite optimized block sizes, measured performance reaches only 94% of theoretical peak (48.17 GFLOPS vs 51.2 GFLOPS sustained turbo). Several factors explain this efficiency gap:

5.8.1 Memory Bandwidth Limitations

HPL is not purely compute-bound; it requires significant memory traffic:

- **Arithmetic intensity**: HPL achieves ∼2 FLOPs per byte for large matrices
- **Memory bandwidth**: Limited by DRAM speed (DDR4-2666 typical: ∼20 GB/s single-channel)
- **Cache misses**: Even with blocking, cache misses occur for large matrices

5.8.2 Non-Compute Operations

The HPL algorithm includes operations with lower arithmetic intensity:

- **Panel factorization**: Serial operation, not fully optimized
- **Pivoting**: Introduces data dependencies and conditional branches
- **Updates**: Matrix-vector operations with lower FLOPs/byte ratio than matrix-matrix

5.8.3 Instruction-Level Limitations

Even with AVX2+FMA:

- **Latency hiding**: Not all pipeline stages can be filled continuously
- **Register pressure**: Limited number of SIMD registers (16 in AVX2)
- **Instruction dependencies**: Data hazards prevent full ILP (Instruction-Level Parallelism)
- **Non-vectorizable code**: Some portions (e.g., pivoting logic) cannot be vectorized

5.8.4 Cache Hierarchy Effects

- **L1 cache misses**: Even with optimal blocking, working set may exceed 32KB L1
- **L2 cache misses**: Matrix sizes 10000-20000 exceed typical 256KB-512KB L2
- **L3 cache pressure**: For N=20000, working set = $3 \times 20000^2 \times 8 = 9.6$ GB, far exceeding L3 capacity
- **Cache line conflicts**: Certain access patterns cause set-associative conflicts

5.8.5 Frequency Variation

- **Turbo boost duration**: CPU cannot sustain maximum turbo indefinitely
- **Thermal throttling**: Long-running benchmarks (N=20000: 152s) may trigger thermal limits
- **Power limits**: Laptop CPUs enforce TDP (Thermal Design Power) limits

5.9 Performance Degradation for Large Matrices

Notice that N=20000 achieves only 68% efficiency compared to 94% for N=5000. This degradation is explained by:

1. **Memory bandwidth saturation**: Larger matrices spend more time waiting for DRAM
2. **Cache overflow**: Working set (9.6 GB) vastly exceeds cache capacity
3. **TLB misses**: Large memory footprint exceeds TLB coverage, causing page table walks
4. **Longer execution time**: 152 seconds allows thermal throttling to reduce frequency

5.10 Comparison with Reference System

Metric	Reference (Xeon 8276L)	This System (i7-10510U)
Base Frequency	2.2 GHz	2.3 GHz
SIMD	AVX-512	AVX2
FLOPs/cycle	32 (AVX-512)	16 (AVX2)
Theoretical Peak	70.4 GFLOP/s	51.2 GFLOP/s

Table 5: Comparison with exercise reference system

Observations:

- The reference Xeon has higher theoretical peak due to AVX-512 (32 FLOPs/cycle vs 16)
- However, this i7-10510U achieves excellent efficiency (94%) with proper tuning
- The gap is primarily architectural (AVX2 vs AVX-512), not optimization-related

5.11 Key Conclusions

1. **Matrix size impact**: Performance generally improves with matrix size up to N=5000-10000, then degrades for N=20000 due to memory bandwidth and cache capacity limits.
2. **Block size optimization**: Optimal block size is critical:
   - Too small (NB<32): Poor cache utilization, high overhead
   - Optimal (NB=64-256): Balance between cache usage and blocking efficiency
   - Too large (NB>matrix size): Defeats blocking purpose
3. **Best configuration**: N=5000, NB=256 achieves 94% efficiency (48.17 GFLOPS), representing near-optimal single-core performance on this CPU.
4. **Efficiency factors**: The 6-10% gap from theoretical peak is inevitable due to:
   - Memory bandwidth constraints
   - Non-compute operations (pivoting, updates)
   - Instruction pipeline limitations
   - Cache hierarchy effects
5. **Scaling limitations**: Performance does not scale linearly with matrix size beyond N=10000 due to memory subsystem saturation.
6. **Single-core validation**: Achieved 94% of sustained turbo theoretical peak, demonstrating effective single-core optimization and proper thread control.