TP1 - Optimizing Memory Access

Imad Kissami

October, 2025

Exercise 1

- This exercise aims to explore the impact of memory access strides on the performance of a C program.
- The following program allocates an array of doubles, initializes it to 1.0, and then performs a summation while traversing the array with different strides.

```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; // Time in milliseconds
    rate = sizeof(double) * N * (1000.0 / msec) / (1024 * 1024);

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

Compilation

- Compile the program with O0 (without any optimization) :

```
gcc -O0 -o stride stride.c
```

— Compile the program with O2 (with level 2 optimization) :

```
gcc -O2 -o stride stride.c
```

Loop Optimizations

— Loop unrolling (partially) :

```c
for (int i = 0; i < N; i++) {
  sum += arr[i];
}
```

After unrolling :

```c
for (int i = 0; i < N; i += 4) {
  sum += arr[i] + arr[i + 1] + arr[i + 2] + arr[i + 3];
}
```

— Instruction scheduling : Reordering instructions to improve pipeline efficiency.

```
MUL R1, R2, R3 ; Multiply (long latency)
ADD R4, R1, R5 ; Add (depends on R1)
SUB R6, R7, R8 ; Independent subtraction
```

After reordering :

```
MUL R1, R2, R3 ; Multiply (long latency)
SUB R6, R7, R8 ; Independent subtraction (executed while MUL is running)
ADD R4, R1, R5 ; Add (by now, R1 is ready)
```

Execution and Analysis

— Run the code using -O0 and -O2 for different strides.
— Compare the execution times and bandwidths.
— Discuss the impact of loop unrolling.

Exercise 2

— Write `mxm.c` to implement the standard matrix multiplication using three nested loops.

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

— Modify the loop order (jk) to optimize cache usage and improve performance.
— Compute the execution time and memory bandwidth for both versions and compare the results.
— Explain the output.

Exercise 3

- Write `mxm_bloc.c` for block matrix multiplication.
- Compute the CPU time and memory bandwidth for different block sizes.
- Determine the optimal block size. Explain why it is the best choice.

Compilation

```
gcc -O2 -o mxm_block mxm_bloc.c
```

Execution and Analysis

- Run the program with different block sizes.
- Compare the CPU time and bandwidth for each block size.
- Identify the optimal block size and justify why it provides the best performance.

Instructions

- Modify the standard matrix multiplication algorithm to process submatrices (blocks) instead of individual elements.
- Use three nested loops, but ensure that matrix elements are accessed in blocks of size B x B.
- Follow this structure for blocking :
  - Divide matrices A, B, and C into blocks of size B x B.
  - Compute the result for each block before moving to the next.

Exercise 4 : Memory Management and Debugging with Valgrind

- Analyze the following C program, which allocates, initializes, prints, and duplicates an array.

```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;
```

```c
  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) {
  // add free memory line to fix the 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 on purpose
}
```

Compilation and Execution

— Compile the program with debugging symbols :

```
gcc -g -o memory_debug memory_debug.c
```

— Run the program using Valgrind to check for memory leaks :

```
valgrind --leak-check=full --track-origins=yes ./memory_debug
```

— Use Valgrind's Memcheck tool to detect memory leaks.
— Modify the program to fix memory leaks and re-run Valgrind to verify.