TP1 - Optimizing Memory Access

Imad Kissami

January, 2026

Exercise 1 : Impact of Memory Access Stride

- 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
```

Execution and Analysis

1. Run the code using -O0 and -O2 for different strides.
2. Compare the execution times and bandwidths (plot).
3. Discuss the impact of loop unrolling.

Exercise 2 : Loop Optimizations

Consider the following program that computes the sum of an array of double-precision values :

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

#define N 1000000

int main() {
   double *a = malloc(N * sizeof(double));
   double sum = 0.0;
   double start, end;

   for (int i = 0; i < N; i++)
      a[i] = 1.0;

   start = (double)clock() / CLOCKS_PER_SEC;
   for (int i = 0; i < N; i++)
      sum += a[i];
   end = (double)clock() / CLOCKS_PER_SEC;

   printf("Sum = %f, Time = %f ms\n", sum, (end - start) * 1000);

   free(a);
   return 0;
}
```

— Manually unroll the summation loop using different unrolling factors :

$$U = 1,\ 2,\ 3,\ 4,\ \ldots,32$$

— Example (unrolling factor $U = 4$) :

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

Execution and Analysis

1. Implement unrolling factors $U = 1, 2, 3, 4, ..., 32$
2. Measure execution time for each version
3. Identify the unrolling factor that gives the best performance
4. Compare manual unrolling at `-O0` with compiler optimization at `-O2`
5. Determine whether manual unrolling is still beneficial with `-O2`
6. Repeat 1, 2, 3 by replacing the array type and accumulator with (float, int, short)

Exercise 3 : Instruction Scheduling

— 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)
```

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

#define N 100000000

int main() {
    double a = 1.1, b = 1.2;
    double x = 0.0, y = 0.0;
    clock_t start, end;

    start = clock();
    for (int i = 0; i < N; i++) {
        x = a * b + x; // stream 1
        y = a * b + y; // independent stream 2
    }
    end = clock();

    printf("x = %f, y = %f, time = %f s\n",
    x, y, (double)(end - start) / CLOCKS_PER_SEC);
    return 0;
}
```

```
gcc -O0 -S exercise3.c -o exercise3.s

cat exercise3.s
```

— O0 compilation :

```
gcc -O0 -fno-omit-frame-pointer -S -masm=intel exercice3.c -o O0.s

cat O0.s
```

```
.L3:
movsd  xmm0, QWORD PTR -32[rbp]     ; load a
mulsd  xmm0, QWORD PTR -24[rbp]     ; xmm0 = a*b
movsd  xmm1, QWORD PTR -48[rbp]     ; load x
addsd  xmm0, xmm1                   ; xmm0 = a*b + x
movsd  QWORD PTR -48[rbp], xmm0     ; store x

movsd  xmm0, QWORD PTR -32[rbp]     ; load a (encore)
mulsd  xmm0, QWORD PTR -24[rbp]     ; xmm0 = a*b (encore)
movsd  xmm1, QWORD PTR -40[rbp]     ; load y
addsd  xmm0, xmm1                   ; xmm0 = a*b + y
movsd  QWORD PTR -40[rbp], xmm0     ; store y

add    DWORD PTR -52[rbp], 1        ; i++
```

```c
for (i = 0; i < 100000000; i++) {
    corps;
}
```

```c
i = 0;
goto L2;        // go test the condition first

L3:             // loop body
    corps;
    i = i + 1;

L2:             // // loop test
    if (i <= 99999999) goto L3;
    // otherwise exit the loop
```

— O2 compilation :

```
gcc -O2 -fno-omit-frame-pointer -S -masm=intel exercice3.c -o O2.s

cat O2.s
```

```
mov   eax, 100000000   ; initial counter
.L2:
addsd xmm1, xmm0
addsd xmm1, xmm0       ; 2 additions per loop iteration
sub   eax, 2           ; decrement by 2
jne   .L2              ; loop while eax != 0
```

```c
for (i = 0; i < 100000000; i++) {
    corps;
}
```

```c
int n = 100000000;
do {
    // body (does 2 logical iterations per loop)
    x += c;
    x += c;
    n -= 2;
} while (n != 0);
```

Execution and Analysis

1. Compare the CPU time using `O0` and `O2`.
2. Print the assembly code, and understand the output.
3. Now, implement an optimized version of the code, compile it using `O0`, and compare it with `O2`.

Exercise 4 : Optimizing matrix multiplication

```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];
```

Execution and Analysis

1. Write `mxm.c` to implement the standard matrix multiplication using three nested loops.
2. Modify the loop order (jk) to optimize cache usage and improve performance.
3. Compute the execution time and memory bandwidth for both versions and compare the results.
4. Explain the output.

Exercise 5 : Matrix multiplication per bloc

1. Write `mxm_bloc.c` for block matrix multiplication.
2. Compute the CPU time and memory bandwidth for different block sizes.
3. Determine the optimal block size. Explain why it is the best choice.
4. Run the program with different block sizes.
5. Compare the CPU time and bandwidth for each block size.
6. Identify the optimal block size and justify why it provides the best performance.

Compilation

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

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 6 : 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;
  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.

Exercise 7 : HPL benchmark

The objective of this exercise is to run HPL [1] for several matrix sizes and several block sizes, then record and analyze the results (timing, achieved GFLOP/s, and numerical validation).

1. https ://www.netlib.org/benchmark/hpl/hpl-2.3.tar.gz

Hardware Architecture

The experiments are performed on a compute node equipped with **Intel Xeon Platinum 8276L** processors. Each node contains :

- 2 CPU sockets
- 28 cores per socket
- Base frequency : 2.2 GHz
- AVX-512 vector instructions with fused multiply-add (FMA)

Each CPU core can perform up to 32 double-precision floating-point operations per clock cycle.

Theoretical Peak Performance (Single Core)

The theoretical peak performance of a single CPU core is :

$$P_{\text{core}} = 1 \times 2.2 \times 10^9 \times 32 = 70.4 \text{ GFLOP/s}$$

Experimental Setup (Single MPI Process)

For all experiments, HPL is executed with :

- Number of MPI processes : 1
- Process grid : $P = 1$, $Q = 1$
- Number of OpenMP threads : 1
- The benchmark is launched using :

```
export OMP_NUM_THREADS=1
mpirun -np 1 ./xhpl
```

Parameters to Explore

Matrix Sizes

Run HPL with the following matrix sizes :

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

Block Sizes

For each matrix size N, run HPL with the following block sizes :

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

Note : This produces $4 \times 9 = 36$ executions in total.

What to Record (After Each Execution)

After each run, record :

- Execution time (seconds)
- Achieved performance P_{HPL} (GFLOP/s)
- Numerical validation status (`PASSED`)

Execution and Analysis

1. For each run, compare the measured performance P_{HPL} with the theoretical peak performance of one core P_{core}.
2. Compute the efficiency for each experiment :

$$\eta = \frac{P_{\text{HPL}}}{P_{\text{core}}}$$

3. Analyze the influence of N and NB :
   — How does performance evolve when N increases ?
   — What is the effect of NB on time and GFLOP/s ?
   — Identify (approximately) the block size(s) giving the best performance.
4. Explain why the measured performance is lower than the theoretical peak.