Final Exam, Fall, 2022

Fudan University

☑ A ☐ B

Course title : _____ **Course number :** _____

Exam type : Closed book

Name : _____ **Student ID :** _____

Major : _____

REMINDER: *All the students, with a commitment to honesty and integrity, should obey the exam rules of Fudan University and shall NOT plagiarize or cheat in the exam. If there is any violation of the exam rules, the university will impose penalties according to corresponding regulations.*

Section	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Total Score**
Score								

Problem	**Title**	**Parts**	**Points**	**Score**	**Grader**
0	Name on Every Page	26	0		
1	True or False	4	8		
2	Row Sums	4	16		
3	Bit Hacks, Etc.	4	20		
4	Work/Span Analysis	5	20		
5	Caching	4	10		
6	Cache-Oblivious Algorithms	7	19		
7	Cache for Tableau Construction	3	7		
	Total		100		

1 True or False (4 parts, 8 points)

Circle the correct answer. **Incorrect answers will be penalized, so do not guess unless you are reasonably sure.** If you justify your answer or show your work, you will leave open the possibility of receiving partial credit should your answer be wrong.

1.1 (2 points)

Compressed sparse rows (CSR) is a good way to store a dense matrix.

True **False**

1.2 (2 points)

Semiconductor vendors ceased producing single-core chips and started to produce multicore chips, in part, because power limitations ended the scaling of clock frequency.

True **False**

1.3 (2 points)

If the parallelism of a program is much greater than the number of processor cores on a machine, a greedy scheduler produces superlinear speedup.

True **False**

1.4 (2 points)

Loop unrolling can negatively impact performance by increasing the number of registers used, causing registers to be spilled to memory.

True **False**

2 Row Sums (4 parts, 16 points)

Consider the execution of the following code on an x86-64 multicore computer with eight 3-GHz processing cores, each with AVX-512 vector units:

```
01 #define ROWS 500
02 #define COLS 100*1000
03
04
05 typedef double element_t;
06
07 void get_row_sums(element_t M[ROWS*COLS],
08                    element_t row_sums[ROWS]) {
09   cilk_for (int32_t i = 0; i < ROWS; i++) {
10     element_t sum = 0;
11     for (int32_t j = 0; j < COLS; j++) {
12       sum += M[i*COLS+j];
13     }
14     row_sums[i] = sum;
15   }
16 }
```

2.1 (4 points)

Perform a back-of-the-envelope calculation for the work of an execution of `get_row_sums()` on a 500-by-100,000 matrix when the program is compiled using the Open Cilk compiler as follows:

```
$ clang -O3 -fno-vectorize -fcilkplus get_row_sums.c
```

(The `-fno-vectorize` switch turns off loop vectorization.) To within a decimal order of magnitude, how long does `get_row_sums()` run? Please circle the letter of your answer.

A From 100 microseconds up to 1 millisecond.

B From 1 millisecond up to 10 milliseconds.

C From 10 milliseconds up to 100 milliseconds.

D From 100 milliseconds up to 1 second.

E None of the above.

(*Hint:* The hardware prefetcher can stream the matrix from memory to the processing cores so that the bottleneck is the scalar processing cores performing arithmetic.1 second = 1000 millisecond, 1 millisecond = 1000 microsecond.)

2.2 (4 points)

It turns out that the compiler does not vectorize the inner loop (lines 11–13) in the program, even when the `-fno-vectorize` flag is omitted from the command line, that is, when the program is compiled as follows:

```
$ clang -O3 -fcilkplus get_row_sums.c
```

Briefly explain why.

2.3 (4 points)

Suppose that in line 5 of the code, instead of defining `element_t` to be `double`, `element_t` is defined as `int64_t`. The compiler now vectorizes the inner loop (lines 11–13) when the program is compiled as in part 2.2. Explain briefly why the compiler can vectorize the inner loop in this case and how it might use the AVX-512 vector units to perform the inner-loop calculation.

2.4 (4 points)

Suppose that the original code (with `element_t` defined to be `double`) is compiled using the following command (with the `-ffast-math` flag added):

```
$ clang -O3 -ffast-math -fcilkplus get_row_sums.c
```

Briefly explain why the compiler can now vectorize the code. What disadvantage might there be to this approach for obtaining extra performance?

3 Bit Hacks, Etc. (4 parts, 20 points)

Circle the correct answer(s). **Incorrect answers will be penalized, so do not guess unless you are reasonably sure.** If you justify your answer or show your work, you will leave open the possibility of receiving partial credit if your answer is wrong.

3.1 (5 points)

The `popcount()` function returns the number of `1`'s in the bit representation of its input argument. For example, `popcount(5)` returns `2`, and `popcount(4)` returns `1`. What does the following code do?

```
01 uint64_t foo(uint64_t x) {
02   return popcount(x) - popcount(x+1) + (x != ~0);
03 }
```

A Returns the number of trailing `0`'s.[1]

B Returns the number of trailing `1`'s.

C Returns the number of `1`'s in `x`.

D Returns the number of `1`'s in `x+1`.

E None of the above.

[1] The number of consecutive `0`'s starting from the least significant bit.

3.2 (5 points)

What does the following code do?

```
01 uint8_t foobar(uint64_t x) {
02   uint8_t n = 0;
03   if (!(x & 0xFFFFFFFF)) { n += 32; x >>= 32; }
04   if (!(x &     0xFFFF)) { n += 16; x >>= 16; }
05   if (!(x &       0xFF)) { n +=  8; x >>=  8; }
06   if (!(x &        0xF)) { n +=  4; x >>=  4; }
07   if (!(x &        0x3)) { n +=  2; x >>=  2; }
08   return n + ((x & 1) ^ 1) + (x == 0);
09 }
```

A Returns the number of trailing `0`'s.

B Returns the number of trailing `1`'s.

C Returns the number of `1`'s in `x`.

D Returns the number of `1`'s in `x+1`.

E None of the above.

3.3 (5 points)

What does the following code do?

```
01 uint8_t bar(uint64_t x) {
02   return (x == 0) ^ (x == (x & -x));
03 }
```

A Returns `0`.

B Returns `1`.

C Returns `1` if `x` is odd.

D Returns `1` if `x` is an exact power of 2.

E None of the above.

3.4 (5 points)

Consider the following function declaration, where `my_t` is a struct and `A` and `B` are arrays:

```
01 void foo(my_t* restrict A, my_t* restrict B);
```

The `restrict` keyword does what? (Circle all that apply.)

A Causes a compilation error if the arrays pointed to by `A` and `B` could overlap.

B Allows the compiler to assume that the fields in the structs `A[i]` and `B[j]` do not alias for any values of `i` and `j`.

C Allows the compiler to assume that `A[i]` and `B[j]` do not alias for any values of `i` and `j`.

D Causes a compilation error if, for some values of `i` and `j`, the fields in the structs `A[i]` and `B[j]` alias.

E None of the above.

4 Work-Span Analysis (5 parts, 20 points)

Analyze the asymptotic work, span, and parallelism of each of the following snippets of code. Express your answers as a function of n, and use Θ-notation. Show your work to receive possible partial credit should your answer be wrong.

4.1 (4 points=1+2+1)

```
01 cilk_for (int32_t i = 0; i < n; i++) {
02   cilk_for (int32_t j = i; j < n; j++) {
03     A[i*n+j]++;
04   }
05 }
```

Work:

Span:

Parallelism:

4.2 (4 points=1+2+1)

```
01 cilk_for (int32_t i = 0; i < n; i++) {
02   for (int32_t j = i; j < n; j++) {
03     A[i*n+j]++;
04   }
05 }
```

Work:

Span:

Parallelism:

4.3 (4 points=1+2+1)

```
01 for (int32_t i = 0; i < n; i++) {
02   cilk_for (int32_t j = i; j < n; j++) {
03     A[i*n+j]++;
04   }
05 }
```

Work:

Span:

Parallelism:

4.4 (4 points=1+2+1)

```
01 void increment(int64_t matrix[n*n], int32_t i, int32_t j){
02   matrix[i*n+j]++;
03 }
04
05 cilk_for (int32_t i = 0; i < n; i++) {
06   for (int32_t j = i; j < n; j++) {
07     cilk_spawn increment(A, i, j);
08   }
09   cilk_sync;
10 }
```

Work:

Span:

Parallelism:

4.5 (4 points=1+2+1)

```
01 void myfunc(int64_t* A, int64_t start, int64_t end){
02   cilk_for (int64_t i = start; i < end; i++) {
03     A[i]++;
04   }
05   if (end-start >= 3) {
06     cilk_spawn myfunc(A, start, start+(end-start)/3);
07     cilk_spawn myfunc(A, start+(end-start)/3, start+2*(end-start)/3);
08     myfunc(A, start+2*(end-start)/3, end);
09     cilk_sync;
10   }
11 }
12
13 myfunc(A, 0, n);
```

Work:

Span:

Parallelism:

5 Caching in Matrix Computation (4 parts, 10 points)

5.1 (4 points)

Ben Bitdiddle is trying to run the following code on 32×32 matrices, A and B, of `double` elements. Ben's machine has a 32KB (32,768 byte) L1 data cache with 64-byte cache lines.

```
01 #define N 32
02
03 // Assume A and B are matrices with 32 x 32 entries.
04 // A starts at address a, where a % 32K = 0, and B starts at address a + 36K.
05
06 double add_matrices() {
07   double sum = 0;
08   for (int32_t i = 0; i < N; i++) {
09     for (int32_t j = 0; j < N; j++) {
10       sum += A[i*N+j] + B[i*N+j];
11     }
12   }
13   return sum;
14 }
```

Suppose that A and B are stored in row-major order. Therefore, each matrix has size $2^{10} \cdot 8 = 8192$ bytes. Further, assume that the starting memory address of A, *addr*, is a multiple of $2^{15} = 32{,}768$ and that the starting memory address of B is $addr + 2^{15} + 2^{12} = addr + 36{,}864$.

If Ben's machine has a **direct-mapped cache** with least-recently-used (LRU) replacement and that the code executes on a single processing core, how many cache sets in the L1 data cache are occupied after a call to the function `add_matrices`?

A 128

B 192

C 256

D 1024

E 1536

F 2048

G None of the above.

Ben now wants to run the code below with different statements:

```
// define STATEMENT here

double matrix_sum(double A[N*N]) {
  double sum = 0;
  for(int32_t i = 0; i < N; i++) {
    for(int32_t j = 0; j < N; j++) {
      for(int32_t k = 0; k < N; k++) {
        STATEMENT;
      }
    }
  }
  return sum;
}
```

If Ben's machine has a **fully associative cache** with an LRU replacement policy, what are the asymptotic number of cache misses of `matrix_sum` for each of the following definitions of `STATEMENT`? Express your answers in terms of N, the cache block size $\mathcal{B}$, and the cache size $\mathcal{M}$, using Θ-notation. Assume that $N \gg \mathcal{M}$ and that the matrix is stored in row-major order.

5.2 (2 points)

```
#define STATEMENT (sum += A[i*N+j] * k)
```

5.3 (2 points)

```
#define STATEMENT (sum += A[i*N+k] * j)
```

5.4 (2 points)

```
#define STATEMENT (sum += A[k*N+j] * i)
```

6 Cache-oblivious Algorithms (7 parts, 19 points)

The ***one-dimentional convolution*** algorithm takes as input a row array A with $r + w$ elements and a weight array B with w elements, and outputs results into an output array C with r elements. For all $i = 0, 1, \ldots, r - 1$ and $j = 0, 1, \ldots, w - 1$, we want to compute some function `foo`$(A[i + j], B[j])$ and accumulate the result into $C[i]$.

Below, you will analyze the work and cache complexity of a looping implementation and a recursive implementation of the one-dimensional convolution. Assume that we have a fully associative cache with a least-recently-used (LRU) replacement policy. The cache has size $\mathcal{M}$ and cache line size $\mathcal{B}$. You may assume that a call to the function `foo` incurs $\Theta(1)$ work and cache misses.

```
01 void loop_convolution(int64_t* A, int64_t* B, int64_t* C, int64_t r, int64_t w) {
02   for(int64_t i = 0; i < r; i++) {
03     for(int64_t j = 0; j < w; j++) {
04       C[i] += foo(A[i + j], B[j]);
05     }
06   }
07 }
```

6.1 (1 points)

The work of `loop_convolution` is $W(r,w) = \Theta(rw)$. Please explain why.

6.2 (1 points)

When $r + w < \mathcal{M}/100$, the cache complexity of `loop_convolution` is $Q(r,w) = \Theta((r+w)/\mathcal{B})$. Please explain why.

6.3 (1 points)

When $r > \mathcal{M}$ and $w > \mathcal{M}$, the cache complexity of `loop_convolution` is $Q(r,w) = \Theta(rw/\mathcal{B})$. Please explain why.

```
01 void rec_convolution(int64_t* A, int64_t* B, int64_t* C,
02                      int64_t r, int64_t A_index, int64_t w, int64_t B_index) {
03   if (r == 1 && w == 1) {
04     C[A_index] += foo(A[A_index + B_index], B[B_index]);
05   } else {
06     int64_t r_half = r / 2;
07     int64_t w_half = w / 2;
08     if (r > w) {
09       rec_convolution(A, B, C, r_half, A_index, w, B_index);
10       rec_convolution(A, B, C, r_half, A_index + r_half, w, B_index);
11     } else {
12       rec_convolution(A, B, C, r, A_index, w_half, B_index);
13       rec_convolution(A, B, C, r, A_index, w_half, B_index + w_half);
14     }
15   }
16 }
```

6.4 (4 points)

The work of `rec_convolution` is $W(r,w) = \Theta(rw)$. Please explain why.

6.5 (3 points)

The cache complexity of `rec_convolution` can be described by the following recurrence:

$$Q(r,w) \leq \begin{cases} 2Q(r/2,w) + \Theta(1) & \text{if } r \geq w \text{ and } r + w > \alpha\mathcal{M}, \\ 2Q(r,w/2) + \Theta(1) & \text{if } r < w \text{ and } r + w > \alpha\mathcal{M}, \\ \Theta(\mathcal{M}/\mathcal{B}) & \text{if } r + w \leq \alpha\mathcal{M}, \end{cases}$$

for some constant $\alpha < 1$. Please explain why.

6.6 (5 points=2+2+1)

Provide the following information about the recursion tree defined by the recurrence for the cache complexity of `rec_convolution`.

Height =

Number of leaves =

Misses per leaf =

6.7 (4 points)

Provide a closed form solution for the cache complexity of `rec_convolution` (in O-notation):

$Q(r,w) =$

7 Cache for Tableau Construction (3 parts, 7 points)

The tableau construction problem involves filling an $N \times N$ tableau, where each entry of the tableau is calculated as a function of some of its neighbors. To be specific, the equation to fill an element of the tableau would take the form

$$A[i][j] = f(A[i-1][j-1], A[i][j-1], A[i-1][j])$$

where f is an arbitrary function.

7.1 (2 points)

Consider the code snippet in Figure 1 below.

```
01 #define A(i, j) A[N + (i) - (j) - 1]
02
03 void tableau(double *A, size_t N) {
04   for (size_t i = 1; i < N; i++) {
05     for (size_t j = 1; j < N; j++) {
06       A(i, j) = f(A(i-1, j-1), A(i, j-1), A(i-1, j));
07     }
08   }
09 }
```

Figure 1: A simple, iterative loop for filling a tableau.

In this problem, we are only interested in computing the final value of the tableau, stored in `A(N-1,N-1)`, and hence we really only need $2N-1$ amount of space during computation. Thus, the algorithm declares `A` as an array of size $2N-1$.

The algorithm initializes the first row and first column of the tableau, and invokes the `tableau` function as shown in Figure 2.

```
10 for (size_t i = 0; i < N; i++) {
11   A(i, 0) = INIT_VAL;
12 }
13 for (size_t j = 0; j < N; j++) {
14   A(0, j) = INIT_VAL;
15 }
16 tableau(A, N);
17 res = A(N - 1, N - 1);
```

Figure 2: Initializing and calling the iterative `tableau` function.

Recall the tall cache assumption, which states that $\mathcal{B}^2 < \alpha\mathcal{M}$, where $\mathcal{B}$ is the size of the cache line, $\mathcal{M}$ is the size of the cache, and $\alpha \leq 1$ is a constant.

Assuming that an optimal replacement strategy holds and that the cache is tall, give a tight upper bound on the cache complexity $Q(n)$ for each of the following cases using O notation, where $c \leq 1$ is a sufficiently small constant:

1. $n \geq c\mathcal{M}$
2. $n < c\mathcal{M}$

7.2 (2 points)

Now consider the code snippet for a recursive tableau implementation, as shown in Figure 3.

```
18 #define A(i, j) A[N + (i) - (j) - 1]
19
20 void recursive_tableau(double *A, size_t rbegin, size_t rend, size_t cbegin,
21                         size_t cend) {
22   if (rend-rbegin == 1 && cend-cbegin == 1) {
23     size_t i = rbegin, j = cbegin;
24     A(i, j) = f(A(i-1, j-1), A(i, j-1), A(i-1, j));
25   } else {
26     size_t rmid = rend-rbegin > 1 ? (rbegin + (rend-rbegin) / 2) : rend;
27     size_t cmid = cend-cbegin > 1 ? (cbegin + (cend-cbegin) / 2) : cend;
28     recursive_tableau(A, rbegin, rmid, cbegin, cmid);
29     if (cend > cmid)
30       recursive_tableau(A, rbegin, rmid, cmid, cend);
31     if (rend > rmid)
32       recursive_tableau(A, rmid, rend, cbegin, cmid);
33     if (rend > rmid && cend > cmid)
34       recursive_tableau(A, rmid, rend, cmid, cend);
35   }
36 }
```

Figure 3: A recursive implementation for filling in a tableau.

This algorithm similarly uses only $2N - 1$ amount of space, initializes the array `A`, and invokes the `recursive_tableau` function as shown in Figure 4.

```
37 for (size_t i = 0; i < N; i++) {
38   A(i, 0) = INIT_VAL;
39 }
40 for (size_t j = 0; j < N; j++) {
41   A(0, j) = INIT_VAL;
42 }
43 if (N > 1) {
44   recursive_tableau(A, 1, N, 1, N);
45 }
46 res = A(N-1, N-1);
```

Figure 4: Initializing and calling the `recursive_tableau` function.

This recursive algorithm divides the tableau into four quadrants to compute.

Assuming that an optimal replacement strategy holds and that the cache is tall, give a tight upper bound on the cache complexity $Q(n)$ using O notation.

7.3 Comparison (3 points)

Explain why $Q(n)$ for the recursive formulation is better than the one for the iterative formulation.