- Project 2 need to be done in a Linux machine which supports AVX2.

In this project, we investigate how to use Single Instructions Multiple Data (SIMD) to accelerate the program. There are different tools for using SIMD, such as AVX, CUDA, OpenMP or MPI. We focus on AVX in this project.

AVX stands for Advanced Vector Extensions. As its name, it allows CPU to execute multiple instructions in parallel in the SIMD fashion. To write program using the AVX, here is a tutorial. The manual from Intel can be found here. In short, consider the following code

```
void scalar_add(int64_t *a, int64_t *b, int64_t *c, int len) {
    for (int i = 0; i < len; ++i) {
        c[i] = a[i] + b[i];
    }
}
```

We can write the above function in the following SIMD way, they have equivalent effect of saving the addition results in the `c` array. But SIMD code executes addition in parallel.

```
#include <immintrin.h>
void simd_add(int64_t *a, int64_t *b, int64_t *c, int len) {
    for (int i = 0; i < len; i+=4) {
        __m256i a_vec = _mm256_loadu_si256((__m256i *)&a[i]);
        __m256i b_vec = _mm256_loadu_si256((__m256i *)&b[i]);
        __m256i c_vec = _mm256_add_epi64(a_vec, b_vec);
        _mm256_storeu_si256((__m256i *)&c[i], c_vec);
    }
}
```

Those `_mm256_loadu_si256`, `_mm256_add_epi64`, and `_mm256_storeu_si256` are intrinsics for AVX. AVX has different variants, e.g., AVX2 and AVX-512. AVX supports SIMD on data structure of 256 bits, while AVX-512 supports SIMD on data structure of 512bits. Some old Intel processors may only support AVX2 but not AVX-512. To compile a program with AVX instrincs, we use '`-m`' options while using `gcc` (shown in here). We can use `-mavx2` for AVX2 or `-mavx512f` for AVX-512. In AVX2, one 256-bit register can hold four 64-bit values and operate on the component values in parallel. In AVX-512, one 512-bit register can hold eight 64-bit values and operate on the component values in parallel. Intrinsics for AVX2 can be found here, and intrinsics for AVX-512 can be found here. You'll probably find intrinsics which contain the following names:

`setl`	Load all elements of a vector with a scalar value.
`cmp..`	Compare two registers and set a scalar bitmap as the result.
`srli`	Shift right. Handy for dividing by 2.
`mask_blend`	Combine data from two different registers into a third register depending on the mask bitmap.
`i64gather`	Load multiple values from different offsets from a base address.
`add,sub,...`	There are various arithmetic functions available.

Please check the template code `db.c` provided on Carmen. In this code, we have provided a basic binary search function `low_bin_search`. This function do a binary search to search the `target` item in the given array `data` and returns the index of the first element in the `data` array that is not less than the `target`. We will work step by step and implement different optimized versions of this `low_bin_search`. Please check here for the definitions of operators in C.

(a) **Implement** `low_bin_nb_arithmetic` The `low_bin_nb_arithmetic` achieves the same search functionality as the `low_bin_search`, but it does not use either `if` statement or the C ternary operator `? x : y`. The `low_bin_nb_arithmetic` use arithmetic operations to avoid `if` statements. Specifically, the operators used by `low_bin_nb_arithmetic` should only be the following: single equal "=", arithmetic operators, comparison operators, and bitwise logical operators. Note that in C, a logic true or false can be treated as 1 or 0 integer. You may use this integer to do some manipulation to avoid using branching statement.

(b) **Implement** `low_bin_nb_mask` The `low_bin_nb_mask` achieves the same search functionality as the given `low_bin_search`, but it does not use either `if` statement or the C ternary operator `? x : y`. It is a further optimized version of `low_bin_nb_arithmetic` as the `low_bin_nb_mask` uses bitwise operations to avoid `if` statements. The `low_bin_nb_mask` does not use multiplication operators either. Specifically, the operators used by `low_bin_nb_mask` should only be the following: single equal "=", addition operator "+", subtraction operator "-", division operator "/", comparison operators, and bitwise logical operators. If you have finished the part (a), you may find out that the place to use multiplication will only be multiplying a value by 1 or 0. And multiplying a value by 1 or 0 can be represented through bitwise operations. Also note that the 64-bit representation of -1 is 0xFFFFFFFFFFFFFFFF. Try hard to see how to reduce the number of addition operators, subtraction operators, and division operators as much as possible.

(c) **Implement** `low_bin_nb_4x` This function tries to do 4 binary searches in one function call, hoping that the underlying system will overlap the cache miss latencies for concurrent searches. Inside this routine, there should be an outer while loop to check a termination condition, and an inner for loop to iterate through each of the 4 searches. The compiler may unroll the inner for loop. The inner loop should use the code from `low_bin_nb_mask`. These is a subtlety in this "parallelization" in that some of the 4 searches may need more iterations than others. In such a case your loop should continue until all searches are complete and should make sure that "extra" loops of a completed search do not change the search result.

(d) **Implement SIMD binary search** Finally, you should write a function `low_bin_nb_simd` that uses AVX2 to do the binary search in SIMD. Some comment in the template code can be helpful. This function should achieve the same functionality as `low_bin_nb_4x`, but the CPU will be able to do 4 searches in parallel.

(e) **Implement band join** You will use the `low_bin_nb_4x` or `low_bin_nb_mask` method you previously developed to implement an in-memory band join function. Band-joins are useful when the join predicate involves inequalities. For example, consider the SQL query

```
SELECT *
FROM Jobs J, Candidates C
WHERE J.salary >= C.request-10000
  AND J.salary <= C.request+10000
```

The above SQL query performs a band join. Suppose we sort the candidates by their request values. Then we can use binary search to find the first possible candidate for a job by searching for `J.salary-10000`. We can then scan through the ordered candidates one by one, outputing each as a match with the `Jobs` record, until we find a candidate with `C.request > J.salary+10000`.

Please finish your implementation in the `band_join` function in the given `db.c`. In the given `band_join` template, the `outer` can be treated as the `request` column of `Candidates` while the `inner` can be treated as the `salary` column of `Jobs`. For each matched inner and outer pair, the function outputs indices of the inner items into the `inner_result` and outputs indices of the outer items into the `outer_result`. As mentioned in the code template, you need to make sure you don't exceed the size of the array allocated for the output results (the `result_size` parameter). If you hit that size limit, you should stop there, and output what you have so far, which should be a prefix of the full band join results. You may choose to ouput the results in any order. There are examples of band join output in later.

Your band_join should use `low_bin_nb_4x` to search 4 records in one function call in order to overlap the latencies between each search. If the number of records to be searched is not a multiple of 4, the `band_join` should call `low_bin_nb_mask` to search for the remaining records.

(f) **Implement SIMD band join** This part is required for 3-person group, but optional for other groups. You should use SIMD to implement a band join function. This function achieves the same functionality as in (e) but it should be faster. You may call the previous `low_bin_nb_simd` in this function. You may choose to use AVX2. As AVX2 supports 256-bit register and each record is a 64-bit integer, Your algorithm should process 4 records at a time. When the outer table cardinality is not a multiple of 4, there will be a few leftover records that you should handle with `low_bin_nb_mask`. Please finish your implementation in the `band_join_simd` function in the given `db.c`.

After finishing the code, you may compile the `db.c` using

```
gcc -O3 -mavx2 -o db db.c
```

After compiling, you can run the code by `./db` Please use this program to profile the implemented functions and finish a PDF report as described in (g) and (h).

(g) **Profiling the code of binary search** You will profile the code using the provided `bulk_bin_search` and `bulk_bin_search_4x` functions. Simply uncomment the variant of the code that you want to measure within `bulk_bin_search` and `bulk_bin_search_4x`. You will run the code by invoking

```
./db N X Y Z R
```

For this part of the project you can ignore `X`, `Y` and `Z` and supply dummy values. `N` is the number of elements in the array being searched, which is also the same as the number of probes. For this test, the provided code searches the array for each of its elements in a random order. `R` is the number of repeats of the experiment within the timing loop, which you should set to an appropriate number to deliver stable results. (If this number is too low, then various overheads of setting up the search may dominate the measured time; if this number is too high, your experiment will take too long.)

Your should measure the performance of `low_bin_search`, `low_bin_nb_arithmetic`, `low_bin_nb_mask`,

low_bin_nb_4x, and low_bin_nb_simd as you vary N from 10 to 10^7. Please show the performance results in the report. You report should describe the CPU and cache information of the machine you have used (see /proc/cpuinfo or use lscpu) and analyze how the CPU and cache affect the results. You may notice as N increases, the time per search of the five different binary search functions will change. Please explain why the time per search changes as N increases.

(h) **Profiling the code of band join** You can profile the band join when using the following code

```
./db N X Y Z
```

For band join, R is not need as in previous part (g). N is the size of the inner (sorted) table of the band-join, and X is the size of the outer table. Y is the size of the output result being allocated, and Z is the bound being used in the band computation. In other words, the outer key must be within Z (inclusive) of the inner table value. The outer table values are generated by the system as 31-bit random numbers that are independent of the values in the inner table. So, for small Z values, you probably won't get any matches.

Please measure and analyze the performance of band_join function and include the measured results in the PDF report. Pick an interesting value for N and X and justify in a sentence or two why you think they are interesting. Choose a Y value that is large enough to hold the results, except perhaps for the very largest join results, but not so large that you exceed the physical memory. For these particular choices, vary Z to get a range of output sizes. You may notice as Z increases, the time of band join per outer record will change. Please explain why this timer of band join per outer record change due to the Z value.

For 3-person group, please also measure and analyze the performance of band_join_simd implemented in (f) and include the measured results in the PDF report. The report of 3-person group should analyze the difference between the band_join_simd and band_join.

Example output

Here are two examples when running a finished db.c which is compiled with the # define DEBUG. It can be helpful to check the correctness and better understand the band join.

```
$ ./db 8 5 5 100000000 1
data: 424238336 719885387 846930887 1649760493 1681692778 1714636916 1804289384 1957747794
queries: 1804289383 846930886 1681692777 1714636915 1957747793 424238335 719885386 1649760492
outer: 596516649 1189641421 1025202362 1350490027 783368690
Searching for 1804289383...
Result is 6
Searching for 846930886...
Result is 2
Searching for 1681692777...
Result is 4
Searching for 1714636915...
Result is 5
Searching for 1957747793...
Result is 7
Searching for 424238335...
Result is 0
Searching for 719885386...
Result is 1
Searching for 1649760492...
Result is 3
Time in bulk_bin_search loop is 208 microseconds or 26.000000 microseconds per search
Searching for 1804289383 846930886 1681692777 1714636915  ...
Result is 6 2 4 5  ...
Searching for 1957747793 424238335 719885386 1649760492  ...
Result is 7 0 1 3  ...
Time in bulk_bin_search_4x loop is 60 microseconds or 7.500000 microseconds per search
Band join result size is 2 with an average of 0.400000 matches per output record
```

```
Time in band_join loop is 0 microseconds or 0.000000 microseconds per outer record
band_join results: (4,1) (4,2)
```

Here we generate 8 random number as data (the inner array). We also generate 5 number as outer array. In the binary search part, as we search for the number 1804289383, its index in the data array is 6 (index starts from 0), so the result is 6. In the band join part, for the number 783368690 (index is 4) in the outer array, we check if any number in the data array (the inner array) is in the range $[783368690 - 100000000, 783368690 + 100000000]$ and they are 719885387 (index is 1) and 846930887 (index is 2). For other number p in the outer array, we does not find matched number in the data array which is in the range $[p - 100000000, p + 100000000]$. So we output the results as `(4, 1), (4, 2)`.

```
$ ./db 4  1  2 10000000000 1
data: 846930887 1681692778 1714636916 1804289384
queries: 1804289383 846930886 1681692777 1714636915
outer: 1957747793
Searching for 1804289383...
Result is 3
Searching for 846930886...
Result is 0
Searching for 1681692777...
Result is 1
Searching for 1714636915...
Result is 2
Time in bulk_bin_search loop is 134 microseconds or 33.500000 microseconds per search
Searching for 1804289383 846930886 1681692777 1714636915  ...
Result is 3 0 1 2  ...
Time in bulk_bin_search_4x loop is 56 microseconds or 14.000000 microseconds per search
Band join result size is 2 with an average of 2.000000 matches per output record
Time in band_join loop is 1 microseconds or 1.000000 microseconds per outer record
band_join results: (0,0) (0,1)
```

Here we generate 4 numbers for the data array (the inner array), and 1 number for the outer array. For the number 1957747793, all numbers in the number array are in the range of $[1957747793 - 10000000000, 1957747793 + 10000000000]$. However, the output result size is just 2, so we only output two results `(0, 0) (0, 1)`. It is also correct if just outputting `(0, 2) (0, 3)`.

Please note in the above two examples, the output measured time should not accurately reflect the real performance. To accurate measure the performance, we should comment the `# define DEBUG` and does not let the program print out anything during the execution parts to be measured. We should also properly choose the `R` or `Z` number during measurement. For example, after the `# define DEBUG` is commented and we recompile the program and run it again.

```
$ ./db 10000000  100000000  10000000  10000000 10
Time in bulk_bin_search loop is 25787870 microseconds or 0.257879 microseconds per search
Time in bulk_bin_search_4x loop is 15585010 microseconds or 0.155850 microseconds per search
Band join result size is 10000000 with an average of 92592.592593 matches per output record
Time in band_join loop is 45071 microseconds or 0.000451 microseconds per outer record
```

Items to be submitted for each group:

- The source code implemented with the functionalities described previously. The source code should be in C. Please do not use any external library to implement any functionality described from part (a) to (f).
- A README file illustrate how to compile your code. It is optional to include a Makefile for compiling your code.
- A PDF file of your report. In the report, please describe each group member's contribution if there are more than one person in the group.

Here are some comments for this project:

- When measuring the performance in part (g) and (h), please make sure that `#define DEBUG` is commented. The program should not print out anything during the execution part to be measured. As `printf` has a much larger overhead compared to binary search or band join.
- Implementing every part in small step is a good way to avoid mistakes. Make sure that the binary search functions can work correctly before implementing complicated band join.
- If any bugs happen, please add `-g` when using gcc to compile the code and use gdb to debug. A short introduction of using gdb can be found here. More can be found at here.
- If it is too challenging to come up with solutions from (a) to (e), please make as much effort as you can and make sure the submitted code can successfully compile. You can profile your program based on `low_bin_search` in your submitted report. The grades will be made based on the effort shown in the final submission.

Bonus points for this project:

- If you are not a 3-person group, but you have implemented the part (f), you may get up to 5 bonus points.
- If you implement another binary search variant using AVX-512 and use AVS-512 to implement another band join variant, you may get up to 5 bonus points. The AVX-512 variant binary search should search eight 64-bit records in parallel. You may consider profiling this AVX-512 variant similar to part (h) and and showing the performance in the PDF report. If you decided to use AVX-512, please replace "`-mavx2`" with "`-mavx512f`" in the compilation command.
- If you have done a solid analyze in part (g) and (h), you may get up to 5 bonus points. You may consider analyzing the assembly code of the program and comparing the performance of executable compiled by different versions of gcc.