**1 Introduction**

This assignment focuses on aspects of optimizing and measuring performance.

* The first problem provides a baseline implementation for a function and asks for a more performant version which runs faster. To do so, one must exploit knowledge of the memory hierarchy and processor pipeline.
* The second problem provides a series of functions for classic search algorithms and requires that a main() function implementing a benchmark be implemented.

**2****Download Code and Setup**

Download the code pack linked at the top of the page. Unzip this which will create a project folder. Create new files in this folder. Ultimately you will re-zip this folder to submit it.

| **File** | **State** | **Notes** |
| --- | --- | --- |
| Makefile | Provided | Problem 1 & 2 Build file |
| A4-WRITEUP.txt | Edit | Fill in answers to assignment questions |
| mult\_benchark.c | Provided | Problem 1 main benchmark |
| matvec.h | Provided | Problem 1 header file |
| matvec\_util.c | Provided | Problem 1 utility functions for matrices/vectors |
| baseline\_matvec\_mult.c | Provided | Problem 1 baseline functions to beat |
| optimized\_matvec\_mult.c | Edit | Problem 1 create and fill in optimized function definition |
| search.h | Provided | Problem 2 header file |
| search\_funcs.c | Provided | Problem 2 search, setup, and cleanup functions |
| search\_benchark.c | Create | Problem 2 timing main() to create |

**3 Problem 1: Optimize Matrix-Vector Transpose Multiply**

**3.1 Overview**

A classic problem in numerical computing is to multiply a matrix by a vector. The standard algorithm for doing this is given below and follows a patter shown in the accompanying figure.

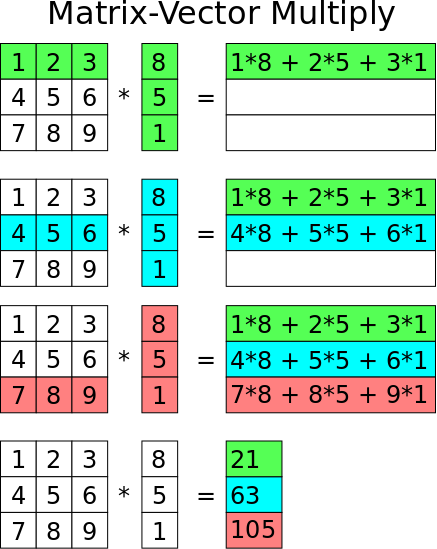


Figure 1: Matrix-vector multiply: iterate across rows. DO NOT OPTIMIZE THIS.

int baseline\_matrix\_mult\_vec(matrix\_t mat, vector\_t vec, vector\_t res){

...; // error checking

for(int i=0; i<mat.rows; i++){

VSET(res,i,0); // initialize res[i] to zero

for(int j=0; j<mat.cols; j++){

int elij = MGET(mat,i,j);

int vecj = VGET(vec,j);

int prod = elij \* vecj;

int curr = VGET(res,i);

int next = curr + prod;

VSET(res,i, next); // add on the newest product

}

}

return 0;

}

This algorithm is fairly efficient in terms of the memory system due to its iterating across rows in the matrix while multiplying and adding.

In contrast, multiplying a matrix-transpose by a vector involves traversing columns as in the implementation and diagram below.

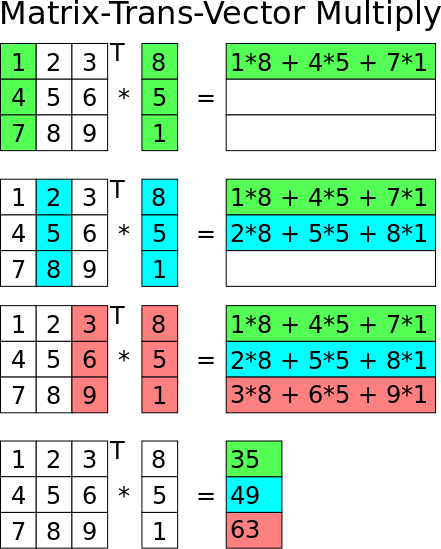


Figure 2: Matrix-transpose-vector multiply: iterate down columns. OPTIMIZE THIS.

int baseline\_matrix\_trans\_mult\_vec(matrix\_t mat, vector\_t vec, vector\_t res){

...; // error checking

for(int j=0; j<mat.cols; j++){

VSET(res,j,0); // initialize res[j] to zero

for(int i=0; i<mat.rows; i++){

int elij = MGET(mat,i,j);

int veci = VGET(vec,i);

int prod = elij \* veci;

int curr = VGET(res,j);

int next = curr + prod;

VSET(res,j, next); // add on the newest product

}

}

return 0;

}

Notice that the routine does not actually transpose the matrix. No changes are made to it. Instead the iteration pattern is simply adjusted to compute the correct answer for the multiply. However, it should be apparent based on experience with the memory system that this change will have performance implications.

**3.2 Optimize matrix\_trans\_mult\_vec()**

The purpose of this problem is to write optimized\_matrix\_trans\_mult\_vec() which is a faster version of the provided baseline\_matrix\_trans\_mult\_vec() to perform a matrix transpose multiplied by a vector.

Write you code in the file optimized\_matvec\_mult.c.

Keep the following things in mind.

1. You will need to acquaint yourself with the functions and types related to matrices and vectors provided in the matvec.h and demonstrated in the baseline functions.
2. The goal of optimized\_matrix\_trans\_mult\_vec() is to exceed the performance of baseline\_matrix\_trans\_mult\_vec() (run faster) and ideally exceed the performance of baseline\_matrix\_mult\_vec().
3. To achieve this goal, several optimizations must be implemented and suggestions are given in a later section.
4. You will need to document your optimizations in the file A4-WRITEUP.txt and provide timing results of running the optimized version.
5. Part of your grade will be based on the speed of the optimized code on the machine apollo.cselabs.umn.edu. The main routine mult\_benchmark.c will be used for this.

Some details are provided in subsequent sections.

**3.3 Evaluation on apollo**

The provided file mult\_benchmark.c provides a benchmark for the speed of matrix vector multiplication functions. It will be used by graders to evaluate the submitted code and should be used during development to gauge performance improvements.

The machine apollo.cselabs.umn.edu will be used for evaluation and the scoring present in mult\_benchmark.c is "tuned" to apollo. That means that codes should be **tested on apollo** so that no unexpected results occur after submission. Results reported should be from apollo.

The output of the mult\_benchmark is shown below.

* SIZE: the size of the matrix being multiplied. The benchmark always uses square matrices
* Runtimes for the 3 functions
  + BASE: the time it takes for baseline\_matrix\_trans\_mult\_vec() to complete.
  + NORM: the time it takes for baseline\_matrix\_mult\_vec() to complete.
  + OPT: the time it takes for optimized\_matrix\_trans\_mult\_vec() to complete.
* Speedups with higher numbers being better
  + BSPDUP: the speedup provided of optimized\_matrix\_trans\_mult\_vec() over baseline\_matrix\_trans\_mult\_vec() which is BASE / OPT.
  + NSPDUP: the speedup provided of optimized\_matrix\_trans\_mult\_vec() over baseline\_matrix\_mult\_vec() which is NORM / OPT.
* POINTS: which are earned according to the following formula
* points = (int) (BSPDUP\*0.5 + NSPDUP - 1.0 - (NORM / BASE) - 0.5);

This scheme, while a little involved, means that unless actual optimizations are implemented, 0 points will be scored.

Below are several demonstration runs of mult\_benchmark.

# RUN ON NON-APOLLO MACHINE: NOTE WARNINGS

computer01> ./mult\_benchmark

WARNING: expected host 'csel-apollo' but got host 'computer01'

WARNING: timing results / scoring will not reflect actual scoring

WARNING: run on host 'csel-apollo' for accurate results

SIZE BASE NORM OPT BSPDUP NSPDUP POINTS

512 1.9750e-03 1.4700e-03 7.6100e-04 2.60 1.93 1

1024 1.3969e-02 2.2660e-03 1.2210e-03 11.44 1.86 6

2048 4.7152e-02 8.1340e-03 4.0550e-03 11.63 2.01 6

4096 1.9722e-01 3.2944e-02 1.7052e-02 11.57 1.93 6

8192 9.8621e-01 1.3456e-01 6.8141e-02 14.47 1.97 7

RAW POINTS: 26

TOTAL POINTS: 26 / 35

# LITTLE CREDIT RUN

apollo> ./mult\_benchmark

SIZE BASE NORM OPT BSPDUP NSPDUP POINTS

512 1.2200e-03 1.0410e-03 1.2580e-03 0.97 0.83 0

1024 1.7830e-02 4.2570e-03 1.7480e-02 1.02 0.24 0

2048 2.4486e-01 1.7139e-02 2.4501e-01 1.00 0.07 0

4096 1.0155e+00 6.8476e-02 1.0159e+00 1.00 0.07 0

8192 4.1294e+00 2.7372e-01 4.1300e+00 1.00 0.07 0

RAW POINTS: 0

TOTAL POINTS: 0 / 35

# PARTIAL CREDIT RUN

apollo> ./mult\_benchmark

SIZE BASE NORM OPT BSPDUP NSPDUP POINTS

512 1.2010e-03 1.0880e-03 8.8800e-04 1.35 1.23 0

1024 1.7780e-02 4.2520e-03 3.5800e-03 4.97 1.19 2

2048 2.5421e-01 1.7171e-02 1.4701e-02 17.29 1.17 8

4096 1.0151e+00 6.8376e-02 5.8682e-02 17.30 1.17 8

8192 4.1297e+00 2.7372e-01 2.3425e-01 17.63 1.17 8

RAW POINTS: 26

TOTAL POINTS: 26 / 35

# FULL CREDIT RUN

apollo> ./mult\_benchmark

SIZE BASE NORM OPT BSPDUP NSPDUP POINTS

512 1.2130e-03 1.0500e-03 5.3100e-04 2.28 1.98 1

1024 1.8842e-02 4.2570e-03 2.2040e-03 8.55 1.93 4

2048 2.4402e-01 1.7148e-02 9.2740e-03 26.31 1.85 13

4096 1.0155e+00 6.8455e-02 3.6902e-02 27.52 1.86 14

8192 4.1297e+00 2.7365e-01 1.4942e-01 27.64 1.83 14

RAW POINTS: 46

TOTAL POINTS: 35 / 35

Note that it is possible to exceed the score associated with maximal performance but no more than the final reported points will be given for the performance portion of the problem.

**3.4****Optimization Suggestions and Documentation**

Previous labs have covered several kinds of optimizations which are useful for this to improve the speed of optimized\_matrix\_trans\_mult\_vec(). These include

* Re-ordering memory accesses to be as sequential as possible which favors cache
* Increasing potential processor pipelining by adjusting the destinations of arithmetic operations.

These should be sufficient to gain full credit though you are free to explore additional optimizations.

The file A4-WRITEUP.txt has several questions that should be answered in a similar fashion to lab write-ups. These document the optimizations used in optimized\_matrix\_trans\_mult\_vec() require a justification for their use.

**3.5 Grading Criteria for Problem 1 (55%)   GRADING**

| **Weight** | **Criteria** |
| --- | --- |
|  | optimized\_matvec\_mult.c |
| 35 | Performance of optimized\_matrix\_trans\_mult\_vec() on apollo.cselabs.umn.edu |
|  | As measured by the provided mult\_bench |
| 5 | Clean and well-documented code for optimized\_matrix\_trans\_mult\_vec() |
| \* | DEDUCTIONS: Up to 5 point loss for memory errors which can be detected using |
|  | make p1-valgrind |
|  | A4-WRITEUP.txt |
| 2 | Answer 1A A4-WRITEUP.txt (source code) |
| 3 | Answer 1B A4-WRITEUP.txt (timing table) |
| 10 | Answer 1C A4-WRITEUP.txt (optimizations description) |

**4 Problem 2: Timing Search Algorithms**

**4.1 Overview**

This problem center on timing several algorithms to measure their performance. This will require use of C timing functions which have been demonstrated in various parts of the class including the previous problem.

You will measure the performance of 4 search functions which simply determine whether an integer query is present in an associated data structure. All of these along with associated setup functions are provided in the search\_funcs.c file. The search algorithms are as follows.

**Linear Search in an Array**

The array need not be sorted and is searched from beginning to end.

**Linear Search in a Linked List**

Nodes are linked together and the list is searched for a query from beginning to end.

**Binary Search in a Sorted Array**

The classic divide and conquer algorithm which repeatedly halves the search space.

**Binary Search in a Tree**

A binary search tree enables searching for a query by following left/right branches.

If you do not have a sense of the relative computational complexities of these algorithms, you should review these as they should factor into your analysis of the timings of the algorithms.

As was the case for problem 1, you can develop your timing program on any platform but the analysis should be conducted on apollo.cselabs.umn.edu to ensure comparability.

**4.2 main() in search\_benchmark.c**

The requirements from this problem is that you provide a main() function in the file search\_benchmark.c with the following features.

1. Runs on a range of sizes that can be specified on the command line. A typical approach is to allow one to specify a minimum and maximum size of the search data structures and repeatedly double starting at the minimum and ending at the max.
2. Create "sequential" data in the structures using the provided functions.
   * make\_sequential\_array() for arrays
   * make\_sequential\_list() for linked lists
   * make\_sequential\_trees() for trees
3. Perform a variable number of searches on the data structures. This should be done the following way.
   * On the command line the number of repetitions of searches should be specified on the command line.
   * Search for every element in the data structures the given number of repetitions.
   * Search an equal number of times for elements NOT in the data structure.
   * Perform the search in a way that does not favor cache layout of any particular data structure.

A typical example method for this is as follows. For a size 100 data structure perform

* + An outer loop over the number of repetitions
  + A first inner loop over successful queries 0, 50, 1,51,2,52, …,49,99
  + A second inner loop over failing queries -1,-51,-2,-52,…,-50,-100

This pattern of searches does not favor any particular cache pattern of the data structures.

1. Enable any combination of algorithms to be tested by specifying which are to be run on the command line. The required way to do this is to accept a command line argument which enables which search types should be done. The associated characters with each algorithm are
   * a for linear array search
   * b for binary search
   * l for linked list search
   * t for tree search

Specify a command line argument which enables the searches:

* + l: run linked list search only
  + ab: run linear array search and binary array search
  + alt: run all but the binary array search
  + ablt: run all algorithms
  + If no string is specified, run all algorithms

1. Ensure that an equal number of searches is done for each algorithms.
2. Ensure that the timing that is done is ONLY for searching and not for setup and cleanup for the data structures.
3. Ensure that there are no memory leaks or other problems in setup and cleanup for the searches.

**4.3 Sample main() Runs**

Below are some sample runs of the main() function and output that is produced. Note that the times have intentionally been set to all identical times. Your exact output may vary some but main() must use the command line options as indicated below. These arguments are

1. Minimum data size (power of 2)
2. Maximum data size (power of 2)
3. Number of repeats
4. (Optional) Characters specifying which search algorithms to run. If this is omitted, run all algorithms.

> ./search\_benchmark

usage: ./search\_benchmark <minpow> <maxpow> <repeats> [which]

which is a combination of:

a : Linear Array Search

l : Linked List Search

b : Binary Array Search

t : Binary Tree Search

(default all)

# run all algorithms, single repetition of searches

> ./search\_benchmark 9 14 1

LENGTH SEARCHES array list binary tree

512 1024 1.2345e+06 1.2345e+06 1.2345e+06 1.2345e+06

1024 2048 1.2345e+06 1.2345e+06 1.2345e+06 1.2345e+06

2048 4096 1.2345e+06 1.2345e+06 1.2345e+06 1.2345e+06

4096 8192 1.2345e+06 1.2345e+06 1.2345e+06 1.2345e+06

8192 16384 1.2345e+06 1.2345e+06 1.2345e+06 1.2345e+06

16384 32768 1.2345e+06 1.2345e+06 1.2345e+06 1.2345e+06

# note that SEARCHES is 2\*LENGTH as 1 successful and 1 unsuccessful

# search is run for each element in the data structure

# run linear array, linked list, binary array search algorithms, 10 repetition of searches

> ./search\_benchmark 5 10 10 alb

LENGTH SEARCHES array list binary

32 640 1.2345e+06 1.2345e+06 1.2345e+06

64 1280 1.2345e+06 1.2345e+06 1.2345e+06

128 2560 1.2345e+06 1.2345e+06 1.2345e+06

256 5120 1.2345e+06 1.2345e+06 1.2345e+06

512 10240 1.2345e+06 1.2345e+06 1.2345e+06

1024 20480 1.2345e+06 1.2345e+06 1.2345e+06

# run linear binary array and tree search algorithms, 2 repetition of searches

> ./search\_benchmark 14 19 2 bt

LENGTH SEARCHES binary tree

16384 65536 1.2345e+06 1.2345e+06

32768 131072 1.2345e+06 1.2345e+06

65536 262144 1.2345e+06 1.2345e+06

131072 524288 1.2345e+06 1.2345e+06

262144 1048576 1.2345e+06 1.2345e+06

524288 2097152 1.2345e+06 1.2345e+06

**4.4 Proper Setup and Cleanup for Searches**

Each search algorithm requires setup and cleanup which is described below. All these follow the same patter which can enable somewhat more elegant setup/search/cleanup which is discussed in the section on bonus credit.

**Linear and Binary Array Search**

Use the function make\_sequential\_array() to create an appropriately sized int array for these functions. Call free() on this array when finished with it.

**Linked List Search**

Use the function make\_sequential\_list() to create an appropriately sized list\_t. Call list\_free() on this list when finished with it.

**Binary Tree Search**

Use the function make\_sequential\_tree() to create an appropriately sized bst\_t. Call bst\_free() on this list when finished with it.

**4.5 What to Measure**

The reason for the requirements mentioned above is to on to study the performance of different search algorithms and answer associated questions in the A4-WRITEUP.txt. The main goals of these questions are to elucidate.

1. To compare the linear and logarithmic search complexities to see if one or the other is superior at small and large input sizes
2. To compare the contiguous memory (array) approaches to the linked memory (list/tree) approaches to see if one or the other is superior at small and large sizes.

To that end make sure to answer the thoroughly answer questions provided.

**4.6 MAKEUP CREDIT: Code Layout in search\_benchmark.c**

"Makeup" credit will allows the score on this assignment to exceed 100% but will not allow the overall score on the project portion of the grade to exceed the weight specified in the syllabus. It is designed to help "make up" for lost credit on previous assignments.

Makeup credit is available in this assignment for implementing the selection of which search functions to run in an "elegant" fashion. Likely the best method for this is to use a table of function pointers. This style is demonstrated in Lab09 but must be expanded upon in this lab to reach its full potential.

The main purpose to using such a table is to avoid a large if/else or switch/case block. For example, a simple approach to doing different search types is something like the following.

int main(...){

int do\_linear\_array = 1;

int do\_linked\_list = 1;

...;

for(all sizes){

if(do\_linear\_array){

// setup array

// start timer

// do searches

// stop timer

// print output

// free the array

}

if(do\_linked\_list){

// setup list

// start timer

// do searches

// stop timer

// print output

// free the list

}

...;

}

}

This formulation obviously involves a much redundant code. A good way to avoid this is to parameterize the repeated parts as functions and iterate over the table invoking functions appropriate to the different types of searches.

To get a sense of how this might work, here is an incomplete example setup used in one solution.

// Table of search algorithms

searchalg\_t algs[] = {

{"Linear Array Search", "array", 'a', 1,

(search\_t) linear\_array\_search, (setup\_t) make\_sequential\_array, (cleanup\_t) free},

{"Linked List Search", "list", 'l', 1,

(search\_t) linkedlist\_search, (setup\_t) make\_sequential\_list, (cleanup\_t) list\_free},

...

{NULL}

};

None of the types are given in the above but several notable things are present.

1. The types of searches are described in an array (table) of structs
2. Each field pertains to a description or function for the search
3. Some of the functions are for setup, others for cleanup, and the first is the actual search function.
4. Casting is required to get the different function prototypes to "fit" into the same kind of struct.
5. All of the searches are enabled by default but fields can changed to disable them.
6. One only needs to iterate through the array invoking appropriate functions. This avoids the need for a large if/else style program.

To complete this part, document your code with comments and also describe you design using function pointers/structs in A4-WRITEUP.txt.

**4.7 Grading Criteria for Problem 2 (45%)   GRADING**

| **Weight** | **Criteria** |
| --- | --- |
|  | search\_benchmark.c |
| 5 | Accepts parameters that control the min/max data sizes and number of repeats |
| 5 | Proper searching for success and fail elements in an order which doesn't favor any algorithms |
| 5 | Timing is clearly done on ONLY the search functions, not on setup or cleanup |
| 5 | Clean and well-documented code for main() |
| \* | DEDUCTIONS: Up to 5 point loss for memory errors which can be detected using |
|  | make p2-valgrind |
| 10 | OPTIONAL MAKEUP CREDIT for using a table of function pointers effectively |
|  | Must describe this design in 2D of A4-WRITEUP.txt |
|  | A4-WRITEUP.txt |
| 10 | Answer 2A A4-WRITEUP.txt (min size for differences) |
| 10 | Answer 2B A4-WRITEUP.txt (list vs array) |
| 5 | Answer 2C A4-WRITEUP.txt (tree vs array)) |

**5 Writeup**

This assignment involves answering questions in the file A4-WRITEUP.txt which is included in the code pack and pasted below.

\_\_\_\_\_\_\_\_\_\_\_\_

A4 WRITEUP

\_\_\_\_\_\_\_\_\_\_\_\_

- Name: (FILL THIS in)

- NetID: (THE kauf0095 IN kauf0095@umn.edu)

Answer the questions below according to the assignment

specification. Write your answers directly in this text file and submit

it along with your code.

PROBLEM 1: optimized\_matrix\_trans\_mult\_vec()

============================================

Do your timing study on apollo.cselabs.umn.edu

(A) Paste Source Code

~~~~~~~~~~~~~~~~~~~~~

Paste a copy of your source code for the function

optimized\_matrix\_trans\_mult\_vec() below.

(B) Timing on Apollo

~~~~~~~~~~~~~~~~~~~~

Paste a copy of the results of running `mult\_bench' on

apollo.cselabs.umn.edu in the space below which shows how your

performance optimizations improved on the baseline codes.

(C) Optimizations

~~~~~~~~~~~~~~~~~

Describe in some detail the optimizations you used to speed the code

up. THE CODE SHOULD CONTAIN SOME COMMENTS already to describe these

but in the section below, describe in English the techniques you used

to make the code run faster. Format your descriptions into discrete

chunks such as.

Optimization 1: Blah bla blah... This should make run

faster because yakkety yakeety yak.

Optimization 2: Blah bla blah... This should make run

faster because yakkety yakeety yak.

... Optimization N: Blah bla blah... This should make run

faster because yakkety yakeety yak.

Full credit solutions will have a least two optimizations.

PROBLEM 2: Timing Search Algorithms

===================================

Do your timing study on apollo.cselabs.umn.edu

(A) Min Size for Algorithmic Differences

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Determine size of input array does one start to see a measurable

difference in the performance of the linear and logarithmic

algorithms. Produce a timing table which includes all algorithms

which clearly demonstrates an uptick in the times associated with some

while others remain much lower. Identify what size this appears to be

a occur.

(B) List vs Array

~~~~~~~~~~~~~~~~~

Determine whether the linear array and linked list search remain

approximately at the same performance level as size increases to large

data or whether one begins to become favorable over other. Determine

the approximate size at which this divergence becomes obvious. Discuss

reasons WHY this difference arises.

(C) Tree vs Array

~~~~~~~~~~~~~~~~~

Compare the binary array search and binary tree search on small to

very large arrays. Determine if there is a size at which the

performance of these two begins to diverge. If so, describe why this

might be happening based on your understanding of the data structures

and the memory system.

(D) OPTIONAL MAKEUP CREDIT

~~~~~~~~~~~~~~~~~~~~~~~~~~

If you decided to make use of a table of function pointers/structs

which is worth makeup credit, describe your basic design for this

below.