# ECE 4900 Final

V1.0

Generated by Doxygen 1.8.20

# Chapter 1

# ECE-4900 - Fall 2021 - Final Project:

#### 1.0.0.1 Multi-threaded BLOB Analysis for Video Processing

by Matthew Hait

## 1.1 Prerequisites

This application has the following dependencies:

- **libavcodec-dev**
- **libavformat-dev**
- **libswscale-dev**
- **libtbb-dev**

## 1.2 Clone

```
git clone **yet to be added...**
```

## 1.3 Build

```
mkdir cmake
cd cmake
cmake ..
make
```

## 1.4 Description

This repository is a demonstration for Threaded Building Blocks (TBB) for image BLOB recognition in thermal imaging. This application reads images, video, or binary images and returns BLOB centroids with the BLOB's average value.

```
./ECE-4900_FINAL_M_Hait [OPTIONS] -i INPUT
```

```
## Options
```bash
Usage:
  ./ECE-4900_FINAL_M_Hait [OPTION...]
 Required options:
  -i SOURCE  File [*.jpg/*.bin] or folder input
 Optional options:
  -b, --benchmark   Tests performance of arbitrary white 25x25 to 4K
                    images.
  -c                Compare TBB parallel and sequential
  -h, --help        Print help
  -o, --output arg  Output image map of BLOB centroids
                            1: Output binary output file [*.bof]
                            2: Output image file [*.jpg] (default: 1)
  -s                Run Sequential
 *.BIF Input required options:
  -x arg  Image X size
  -y arg  Image Y size
```
```

# Chapter 2

# File Index

## 2.1 File List

Here is a list of all files with brief descriptions:

# Chapter 3

# File Documentation

## 3.1 CMakeLists.txt File Reference

### Functions

- cmake_minimum_required (VERSION 3.17) set(PROJECT_NAME ECE-4900_FINAL_M_Hait) set(CMAKE_R↩ UNTIME_OUTPUT_DIRECTORY "$
- bin set (CMAKE_CXX_STANDARD 17) add_definitions(-D_GLIBCXX_USE_CXX17_ABI=0) set(CMAKE_CX↩ X_FLAGS "-D__STDC_CONSTANT_MACROS") project($
- list (APPEND SOURCE_FILES main.cpp include/utils/img_utils.c include/utils/vision_utils.cpp include/utils/ffmpeg↩ _utils.cpp include/utils/ffmpeg_utils.h include/utils/cxxopts.hpp) include_directories(include/utils) add_↩ subdirectory(include/FFmpeg) add_executable($

### 3.1.1 Function Documentation

#### 3.1.1.1 cmake_minimum_required()

```
cmake_minimum_required (
            VERSION 3.  17 )
```

Definition at line 1 of file CMakeLists.txt.

#### 3.1.1.2 list()

```
list (
            APPEND SOURCE_FILES main.cpp include/utils/img_utils.c include/utils/vision_utils.cpp
include/utils/ffmpeg_utils.cpp include/utils/ffmpeg_utils.h include/utils/cxxopts.  hpp )
```

Definition at line 10 of file CMakeLists.txt.

#### 3.1.1.3 set()

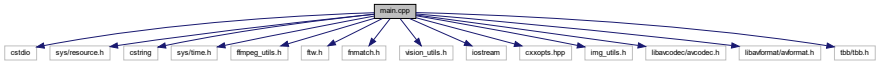
```
bin set (
             CMAKE_CXX_STANDARD 17 )  [pure virtual]
```

Definition at line 6 of file CMakeLists.txt.

## 3.2 main.cpp File Reference

```
#include <cstdio>
#include <sys/resource.h>
#include <cstring>
#include <sys/time.h>
#include <ffmpeg_utils.h>
#include <ftw.h>
#include <fnmatch.h>
#include "vision_utils.h"
#include <iostream>
#include <cxxopts.hpp>
#include "img_utils.h"
#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include "tbb/tbb.h"
```

Include dependency graph for main.cpp:




### Macros

- #define ntoken 6
- #define KERN_X 7
- #define KERN_Y 7

### Functions

- static int count_explore (const char ∗fpath, const struct stat ∗sb, int typeflag)
- static int list_explore (const char ∗fpath, const struct stat ∗sb, int typeflag)
- static int myCompare (const void ∗a, const void ∗b)
- int main (int argc, char ∗∗argv)

## Variables

- bool CONTACT_KERNEL [KERN_X ∗KERN_Y]
- size_t batch_count = 0
- size_t list_count = 0
- char ∗∗ batch_list
- uint8_t globalThresh = 200
- int globalSave = 0

### 3.2.1 Macro Definition Documentation

#### 3.2.1.1 KERN_X

```
#define KERN_X 7
```

Definition at line 19 of file main.cpp.

#### 3.2.1.2 KERN_Y

```
#define KERN_Y 7
```

Definition at line 20 of file main.cpp.

#### 3.2.1.3 ntoken

```
#define ntoken 6
```

Definition at line 18 of file main.cpp.

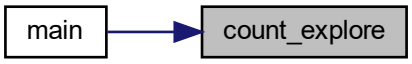
### 3.2.2 Function Documentation

#### 3.2.2.1 count_explore()

```
static int count_explore (
             const char * fpath,
             const struct stat * sb,
             int typeflag )  [static]
```

Definition at line 37 of file main.cpp.

Here is the caller graph for this function:

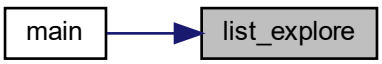



#### 3.2.2.2 list_explore()

```
static int list_explore (
             const char * fpath,
             const struct stat * sb,
             int typeflag )  [static]
```

Definition at line 46 of file main.cpp.

Here is the caller graph for this function:

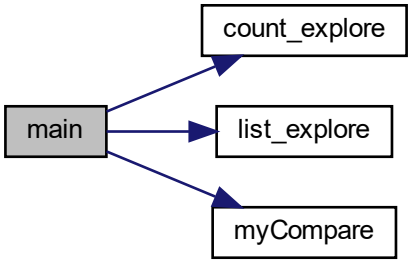


#### 3.2.2.3 main()

```
int main (
            int argc,
            char ** argv )
```

Definition at line 75 of file main.cpp.

Here is the call graph for this function:

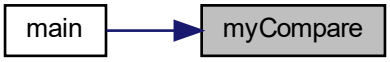



#### 3.2.2.4 myCompare()

```
static int myCompare (
            const void * a,
            const void * b )  [static]
```

Definition at line 62 of file main.cpp.

Here is the caller graph for this function:



### 3.2.3 Variable Documentation

#### 3.2.3.1 batch_count

```
size_t batch_count = 0
```

Definition at line 29 of file main.cpp.

#### 3.2.3.2 batch_list

```
char** batch_list
```

Definition at line 31 of file main.cpp.

#### 3.2.3.3 CONTACT_KERNEL

```
bool CONTACT_KERNEL[KERN_X *KERN_Y]
```

**Initial value:**

```
= {false, false, false, true, false, false, false,
                                    false, false, true, true, true, false, false,
                                    false, true, true, true, true, true, false,
                                    true, true, true, true, true, true, true,
                                    false, true, true, true, true, true, false,
                                    false, false, true, true, true, false, false,
                                    false, false, false, true, false, false, false }
```

Definition at line 22 of file main.cpp.

#### 3.2.3.4 globalSave

```
int globalSave = 0
```

Definition at line 72 of file main.cpp.

#### 3.2.3.5 globalThresh

```
uint8_t globalThresh = 200
```

Definition at line 71 of file main.cpp.

#### 3.2.3.6 list_count

```
size_t list_count = 0
```

Definition at line 30 of file main.cpp.

## 3.3 main.cpp

```
00001 #include <cstdio>
00002 #include <sys/resource.h>
00003 #include <cstring>
00004 #include <sys/time.h>
00005 #include <ffmpeg_utils.h>
00006 #include <ftw.h>
00007 #include <fnmatch.h>
00008 #include "vision_utils.h"
00009 #include <iostream>
00010 #include <cxxopts.hpp>
00011 extern "C" {
00012     #include "img_utils.h"
00013     #include <libavcodec/avcodec.h>
00014     #include <libavformat/avformat.h>
00015 }
00016 #include "tbb/tbb.h"
00017 using namespace tbb;
00018 #define ntoken 6
00019 #define KERN_X  7   // @brief Contact Kernel x len
00020 #define KERN_Y  7   // @brief Contact Kernel y len
00021 //bool CONTACT_KERNEL[9] = { false, true, false, true, true, true, false, true, false};   // @brief
       Contact Kernel for Grass Fire algorithm
00022 bool CONTACT_KERNEL[KERN_X*KERN_Y] = {false, false, false, true, false, false, false,
00023                                       false, false, true, true, true, false, false,
00024                                       false, true, true, true, true, true, false,
00025                                       true, true, true, true, true, true, true,
00026                                       false, true, true, true, true, true, false,
00027                                       false, false, true, true, true, false, false,
00028                                       false, false, false, true, false, false, false };
00029 size_t batch_count = 0; //@brief Number of images loaded under batch op.
00030 size_t list_count = 0;  //@brief Number of images loaded under batch op.
00031 char **batch_list;
00032
00033 /* @brief Counts number of images in a directory.
00034  * @param   *fpath      Path to file
00035  * @param   *sb
00036  * @return  typeFlag    Flag for file*/
00037 static int count_explore( const char *fpath, const struct stat *sb, int typeflag ) {
00038     if (typeflag == FTW_F) {
00039         if (fnmatch("*.jpg", fpath, FNM_CASEFOLD) == 0) {
00040             batch_count++;
00041         }
00042     }
00043     return 0;
00044 }
00045
00046 static int list_explore( const char *fpath, const struct stat *sb, int typeflag ) {
00047     if (typeflag == FTW_F) {
00048         if (fnmatch("*.jpg", fpath, FNM_CASEFOLD) == 0) {
00049             int len = (int)strlen(fpath);
00050             batch_list[list_count] = (char*)malloc(len*sizeof(char)+1); // Add 1 for \0
00051             strcpy(batch_list[list_count],fpath);
```

```
00052             list_count++;
00053         }
00054     }
00055     return 0;
00056 }
00057
00058 /* @brief Function to compare strings for sorting.
00059  * @param   a   String 1
00060  * @param   b   String 2
00061  * @return  */
00062 static int myCompare(const void* a, const void* b) {
00063     return strcmp(*(const char**)a, *(const char**)b);
00064 }
00065
00066 // Setups statics for tbb batch operation
00067 bool *tbb_img_batch::k = CONTACT_KERNEL;
00068 uint8_t tbb_img_batch::kX = KERN_X;
00069 uint8_t tbb_img_batch::kY = KERN_Y;
00070
00071 uint8_t globalThresh = 200; // @brief Global input pixel threshold to mask to.
00072 int     globalSave = 0;     // @brief Global save logic, 1: save *.jpg, 2: save *.bof
00073
00074
00075 int main (int argc ,char ** argv) {
00076     struct  timeval start{},end{};  // @brief Time struct for tracking computation.
00077     long tUs;                       // @brief uS for time calculations
00078     rastImage rgbInput, grayOut;
00079     char fOutName[11];
00080
00081     struct rlimit old_lim, lim, new_lim;
00082     // Get old limits
00083     if( getrlimit(RLIMIT_NOFILE, &old_lim) != 0)
00084         fprintf(stderr, "Get1 %s\n", strerror(errno));
00085
00086     lim.rlim_cur = 104857600;
00087     lim.rlim_max = 104857600;
00088
00089     // Set limits
00090     if(setrlimit(RLIMIT_STACK, &lim) == -1) {
00091         printf("ERROR expanding stack. Try running as admin.\n");
00092         fprintf(stderr, "\nSet: %s\n", strerror(errno));
00093     }
00094     // Get new limits
00095     if( getrlimit(RLIMIT_STACK, &new_lim) != 0)
00096         fprintf(stderr, "Get2 %s\n", strerror(errno));
00097
00098     try {
00099         // Setting Up application Options
00100         cxxopts::Options options(argv[0], "Finds heat sources in thermal images and returns centroids and
      average values of heat BLOBS.\n");
00101         options
00102                 .positional_help("-i SOURCE [<args>]")
00103                 .show_positional_help()
00104                 .set_width(70)
00105                 .set_tab_expansion()
00106                 .allow_unrecognised_options()
00107                 .add_options("Required")
00108                         ("i", "File [*.jpg/*.bin] or folder input",
      cxxopts::value<std::vector<std::string»(), "SOURCE")
00109                         ("t", "Pixel threshold saturation value",
      cxxopts::value<uint8_t>(globalThresh)->default_value("200"));
00110         options
00111                 .add_options("Optional")
00112                         ("b,benchmark", "Tests performance of arbitrary 25x25 to 4K images.")
00113                         ("c", "Compare TBB parallel and sequential")
00114                         ("h,help", "Print help")
00115                         ("o,output", "Output image map of BLOB centroids\n\t1: Output image file
      [*.jpg]\n\t2: Output binary output file [*.bof]", cxxopts::value<int>(globalSave))
00116                         ("s", "Run Sequential");
00117         options.add_options("*.BIF Input required")
00118                 ("x", "Image X size", cxxopts::value<size_t>())
00119                 ("y", "Image Y size", cxxopts::value<size_t>());
00120         auto result = options.parse(argc, argv);
00121         if (result.count("help")) {
00122             std::cout « options.help({"Required","Optional","*.BIF Input required"}) « std::endl;
00123             exit(0);
00124         }
00125         if (argc == 1){
00126             std::cout « options.help({"Required","Optional","*.BIF Input required"}) « std::endl;
00127             exit(0);
00128         }
```

```
00129         if (result.count("b")) {
00130 // Run Benchmark
00131             //8k: 7680x4320
00132             rastImage       benchImage, benchMap;
00133             rastImageBin    benchMask;
00134             printf("Testing 100 levels from 76x43 to 7680x4320. All Units [us]\n");
00135             printf("-----------------------------------------------------------------------\n");
00136             printf("Res\t");
00137             //printf("%5.5s","     ");
00138             printf("\tS.Mask\tP.Mask\tP.RGF\tS.RGF\tS.BAVG\tP.BAVG\tP.BAVG2\tS.BCNT\tP.BCNT\tP.BCNT2\n");
00139             printf("-----------------------------------------------------------------------\n");
00140             for (int x = 77, y = 43; x <= 7680; x+=77, y+=43) {
00141                 // Setup Data
00142                 benchImage.width = x; benchImage.height = y;
00143                 benchImage.size = x*y; benchImage.ch = 1;
00144                 benchImage.image = (uint8_t*)malloc(benchImage.size*sizeof(uint8_t));
00145                 if (benchImage.image == nullptr) {printf("Unable to allocate mem (benchImage)\n");
      fflush(stdout); return-1;}
00146                 // Set all pixels to 255 (White)
00147                 for (size_t i = 0; i < benchImage.size; i++) {
00148                     benchImage.image[i] = 255;
00149                 }
00150                 // Copy grey metadata to mask
00151                 benchMask.width = benchImage.width; benchMask.height = benchImage.height; benchMask.ch =
      1; benchMask.size = benchMask.width * benchMask.height;
00152                 benchMask.image = (bool *)malloc((benchMask.size)*sizeof(bool));
00153                 if (benchMask.image == nullptr) {printf("Unable to allocate mem (benchMask)\n");
      fflush(stdout); return-1;}
00154 
00155                 // Copy mask metadata to map
00156                 benchMap.width = benchMask.width; benchMap.height = benchMask.height; benchMap.ch = 1;
      benchMap.size = benchMap.width * benchMap.height;
00157                 benchMap.image = (uint8_t*)calloc((benchMap.size),sizeof(uint8_t));
00158                 if (benchMap.image == nullptr) {printf("Unable to allocate mem (benchMap)\n");
      fflush(stdout); return-1;}
00159 
00160                 size_t  blob_cnt = 0;
00161                 // Print Resolution
00162                 printf("%dx%d,\t",x,y);
00163                 if(x == 77 || x == 154) printf("\t");
00164                 // S.Mask
00165                 gettimeofday(&start, nullptr);
00166                 mask_blobs(&benchImage, &benchMask, globalThresh);
00167                 gettimeofday(&end, nullptr);
00168                 tUs = (end.tv_sec - start.tv_sec) * 1000000 + end.tv_usec - start.tv_usec;
00169                 printf("%ld,\t", tUs); fflush(stdout);
00170                 //if(x == 77 || x == 154 || x == 231) printf("\t");
00171 
00172                 // P.Mask
00173                 gettimeofday(&start, nullptr);
00174                 tbb_mask_blobs(&benchImage, &benchMask, globalThresh);
00175                 gettimeofday(&end, nullptr);
00176                 tUs = (end.tv_sec - start.tv_sec) * 1000000 + end.tv_usec - start.tv_usec;
00177                 printf("%ld,\t", tUs); fflush(stdout);
00178 
00179 
00180                 // P.RGF
00181                 gettimeofday(&start, nullptr);
00182                 blob_cnt = tbb_recursive_grass_fire(&benchMask, &benchMap, CONTACT_KERNEL, KERN_X,
      KERN_Y);
00183                 gettimeofday(&end, nullptr);
00184                 tUs = (end.tv_sec - start.tv_sec) * 1000000 + end.tv_usec - start.tv_usec;
00185                 printf("%ld,\t", tUs); fflush(stdout);
00186 
00187                 // Reallocate to do RGF again...
00188                     free(benchMask.image);
00189                     free(benchMap.image);
00190 
00191                     // Copy grey metadata to mask
00192                     benchMask.width = benchImage.width; benchMask.height = benchImage.height; benchMask.ch
      = 1; benchMask.size = benchMask.width * benchMask.height;
00193                     benchMask.image = (bool *)malloc((benchMask.size)*sizeof(bool));
00194                     if (benchMask.image == nullptr) {printf("Unable to allocate mem (benchMask)\n");
      return-1;}
00195                     mask_blobs(&benchImage, &benchMask, globalThresh);
00196                     // Copy mask metadata to map
00197                     benchMap.width = benchMask.width; benchMap.height = benchMask.height; benchMap.ch = 1;
      benchMap.size = benchMap.width * benchMap.height;
00198                     benchMap.image = (uint8_t*)calloc((benchMap.size),sizeof(uint8_t));
00199                     if (benchMap.image == nullptr) {printf("Unable to allocate mem (benchMap)\n");
      return-1;}
```

```
00200
00201
00202                 // S.RGF
00203                 gettimeofday(&start, nullptr);
00204                 blob_cnt = recursive_grass_fire(&benchMask, &benchMap, CONTACT_KERNEL, KERN_X, KERN_Y);
00205                 gettimeofday(&end, nullptr);
00206                 tUs = (end.tv_sec - start.tv_sec) * 1000000 + end.tv_usec - start.tv_usec;
00207                 printf("%ld,\t", tUs); fflush(stdout);
00208
00209                 // S.BAVG
00210                 BLOB *blob_meta = (BLOB*)malloc(sizeof(BLOB)*blob_cnt);
00211                 gettimeofday(&start, nullptr);
00212                 for (size_t i = 1; i < blob_cnt; i++) {
00213                     blob_meta[i-1].b_avg = seq_BLOB_avg_value(&benchMap,&benchImage,i);
00214                 }
00215                 gettimeofday(&end, nullptr);
00216                 tUs = (end.tv_sec - start.tv_sec) * 1000000 + end.tv_usec - start.tv_usec;
00217                 printf("%ld,\t", tUs); fflush(stdout);
00218
00219                 // P.AVG
00220                 gettimeofday(&start, nullptr);
00221                 tbb_BLOB_avg_value(&benchMap, blob_meta, &benchImage, blob_cnt);
00222                 gettimeofday(&end, nullptr);
00223                 tUs = (end.tv_sec - start.tv_sec) * 1000000 + end.tv_usec - start.tv_usec;
00224                 printf("%ld,\t", tUs); fflush(stdout);
00225
00226                 // P.AVG2
00227                 gettimeofday(&start, nullptr);
00228                 tbb_BLOB_avg_value_reduce(&benchMap, blob_meta, &benchImage, blob_cnt);
00229                 gettimeofday(&end, nullptr);
00230                 tUs = (end.tv_sec - start.tv_sec) * 1000000 + end.tv_usec - start.tv_usec;
00231                 printf("%ld,\t", tUs); fflush(stdout);
00232
00233
00234                 // S.BCNT
00235                 gettimeofday(&start, nullptr);
00236                 for (size_t i = 1; i < blob_cnt; i++) {
00237                     seq_BLOB_center(&benchMap, &blob_meta[i-1], i);      // Find BLOB center
00238                 }
00239                 gettimeofday(&end, nullptr);
00240                 tUs = (end.tv_sec - start.tv_sec) * 1000000 + end.tv_usec - start.tv_usec;
00241                 printf("%ld,\t", tUs); fflush(stdout);
00242
00243                 // P.BCNT
00244                 gettimeofday(&start, nullptr);
00245                 tbb_BLOB_center(&benchMap, blob_meta, blob_cnt);
00246                 gettimeofday(&end, nullptr);
00247                 tUs = (end.tv_sec - start.tv_sec) * 1000000 + end.tv_usec - start.tv_usec;
00248                 printf("%ld,\t", tUs); fflush(stdout);
00249
00250                 // P.BCNT2
00251                 gettimeofday(&start, nullptr);
00252                 tbb_BLOB_center_reduce(&benchMap, blob_meta, blob_cnt);
00253                 gettimeofday(&end, nullptr);
00254                 tUs = (end.tv_sec - start.tv_sec) * 1000000 + end.tv_usec - start.tv_usec;
00255                 printf("%ld,\t", tUs); fflush(stdout);
00256
00257                 printf("\n");
00258                 free(blob_meta);
00259                 free(benchMask.image);
00260                 free(benchMap.image);
00261                 free(benchImage.image);
00262             }
00263
00264             return 1;
00265         }
00266         if (result.count("i")) {
00267             if (!result.count("t")) {
00268                 printf("No threshold value (t) entered, defaulting to 200.\n");
00269             }
00270             auto& ff = result["i"].as<std::vector<std::string>>();
00271             // JPG Input
00272             if (ff.back().substr(ff.back().size()-4) == ".jpg") {
00273                 printf("*.jpg detected. Importing image... ");
00274                 // Using stb to load image directly from *.jpg file.
00275                 if(img_load(&rgbInput, ff.back().c_str()) != 1) {
00276                     return(-1);
00277                 }
00278                 printf("Loaded image with a width of %dpx, a height of %dpx. The original image had %d
    channels.\n", rgbInput.width, rgbInput.height, rgbInput.ch);
00279                 if(rgbInput.ch == 3) {
```

```
00280                     // RGB image loaded. Convert to grayscale.
00281                     if(img_c_to_g(&rgbInput, &rgbInput) != 1) {
00282                         printf("broke");
00283                         return(-1);
00284                     }
00285                 }
00286                 // Grayscale image loaded. Copy contents of rgbInput into grayOut. This just makes code
      consistent later.
00287                 // Don't need to check for one channel since img_load a few lines above checks it.
00288                 grayOut = rgbInput;
00289                 grayOut.image = (uint8_t *)malloc(grayOut.size);        // Need to reallocate memory
00290                 if (grayOut.image == nullptr) {
00291                     printf("Ope. Unable to allocate memory for comp.\n");
00292                     return (0);
00293                 }
00294                 memcpy(grayOut.image, rgbInput.image, rgbInput.size);
00295                 img_free(&rgbInput);    // Free source file as it is no longer needed.
00296
00297                 long tUs_seq;
00298                 // Sequential Batch image load.
00299                 if (result.count("c") || result.count("s")) {
00300                     printf("Running sequential... ");
00301                     gettimeofday(&start, nullptr);
00302                     single_image_process(ff.back().c_str(),globalSave,CONTACT_KERNEL, KERN_X, KERN_Y,
      &grayOut, globalThresh, true);
00303                     gettimeofday(&end, nullptr);
00304                     tUs = (end.tv_sec - start.tv_sec) * 1000000 + end.tv_usec - start.tv_usec;
00305                     tUs_seq = tUs;
00306                     printf("Comp. Time:\t\t%ld [us],\t\n", tUs);
00307                 }
00308                 // Parallel (Default)
00309                 if(!result.count("s")) {
00310                     printf("Running parallel pipeline... ");
00311                     gettimeofday(&start, nullptr);
00312                     single_image_process(ff.back().c_str(),globalSave,CONTACT_KERNEL, KERN_X, KERN_Y,
      &grayOut, globalThresh, false);
00313                     gettimeofday(&end, nullptr);
00314                     tUs = (end.tv_sec - start.tv_sec) * 1000000 + end.tv_usec - start.tv_usec;
00315                     printf("Comp. Time:\t%ld [us],\t", tUs);
00316                 }
00317                 printf("\n");
00318                 if(result.count("c")) {
00319                     printf("-----------------------------------------\n");
00320                     float perc = (1 - ((float) tUs / (float) tUs_seq)) * 100;
00321                     printf("Total savings: %f%%\n", perc);
00322                 }
00323                 img_free(&grayOut);
00324                 return 1;
00325             }
00326             // BIF Input
00327             if (ff.back().substr(ff.back().size()-4) == ".bif") {
00328                 if (!result.count("x") || ! result.count("y")) {
00329                     printf("error, *.bif files require -x & -y sizes\n");
00330                     exit(0);
00331                 }
00332                 grayOut.width = result["x"].as<int>();
00333                 grayOut.height = result["y"].as<int>();
00334                 grayOut.size = grayOut.width*grayOut.height;    // Don't need to consider channels since
      it should be grayscale.
00335                 grayOut.ch = 1;                                 // We'll set the channels just in case we
      need it later.
00336                 printf("*.bif detected. Importing data... ");
00337                 FILE *file_i = fopen(ff.back().c_str(),"rb");
00338                 if (file_i == nullptr) {
00339                     printf("\n Error: Unable to open \"%s\".\n",ff.back().c_str());
00340                     return -1;
00341                 }
00342                 fseek(file_i, 0L, SEEK_END);    // Go to end of file
00343                 // Check if file size matches user input
00344                 if(grayOut.size != ftell(file_i)) {
00345                     printf("\n Error: file size of %ld bytes does not match user x:y size of %ld bytes.
      Check file and \"-x\" & \"-y\" inputs.\n",ftell(file_i),grayOut.size);
00346                     return -1;
00347                 }
00348                 rewind(file_i);         // Return to start of file before grabbing data
00349                 printf("File is %ld bytes... ",grayOut.size);
00350                 grayOut.image = (uint8_t *)malloc(grayOut.size);
00351                 if(grayOut.image == nullptr) {
00352                     printf("Ope. Unable to allocate memory for comp.\n");
00353                     return(0);
00354                 }
```

```
00355                 if(fread(grayOut.image,grayOut.size,1,file_i) != 1) {
00356                     printf("Error: Unable to read entire input file.\n");
00357                     free(grayOut.image);
00358                     return -1;
00359                 }
00360                 fclose(file_i);
00361                 long tUs_seq;
00362                 // Sequential Batch image load.
00363                 if (result.count("c") || result.count("s")) {
00364                     printf("Running sequential... ");
00365                     gettimeofday(&start, nullptr);
00366                     single_image_process(ff.back().c_str(),globalSave,CONTACT_KERNEL, KERN_X, KERN_Y,
      &grayOut, globalThresh, true);
00367                     gettimeofday(&end, nullptr);
00368                     tUs = (end.tv_sec - start.tv_sec) * 1000000 + end.tv_usec - start.tv_usec;
00369                     tUs_seq = tUs;
00370                     printf("Comp. Time:\t\t%ld [us],\t\n", tUs);
00371                 }
00372                 // Parallel (Default)
00373                 if(!result.count("s")) {
00374                     printf("Running parallel pipeline... ");
00375                     gettimeofday(&start, nullptr);
00376                     single_image_process(ff.back().c_str(),globalSave,CONTACT_KERNEL, KERN_X, KERN_Y,
      &grayOut, globalThresh, false);
00377                     gettimeofday(&end, nullptr);
00378                     tUs = (end.tv_sec - start.tv_sec) * 1000000 + end.tv_usec - start.tv_usec;
00379                     printf("Comp. Time:\t%ld [us],\t", tUs);
00380                 }
00381                 printf("\n");
00382                 if(result.count("c")) {
00383                     printf("-----------------------------------------\n");
00384                     float perc = (1 - ((float) tUs / (float) tUs_seq)) * 100;
00385                     printf("Total savings: %f%%\n", perc);
00386                 }
00387                 img_free(&grayOut);
00388                 return 1;
00389             }
00390             // MOV Input
00391             if (ff.back().substr(ff.back().size()-4) == ".mov") {
00392                 int frame_width, frame_height;
00393                 unsigned char* frame_data;
00394                 // TODO: ADD mov to frame func
00395                 if (!load_frame(ff.back().c_str(), &frame_width, &frame_height, &frame_data)) {
00396                     printf("Error\n");
00397                     return 0;
00398                 }
00399             }
00400             // Folder Batch input
00401             if((char)ff.back().back() == '/') {
00402                 // Load and sort file names into list:
00403                 ftw(ff.back().c_str(),count_explore, 8);                            // Get number of
      files
00404                 batch_list = (char**)malloc(batch_count*sizeof(char*));
00405                 if(batch_list == nullptr) {printf("Cannot allocate memory."); return -1;}
00406                 ftw(ff.back().c_str(),list_explore, 8);
00407                 qsort(batch_list, batch_count, sizeof(const char*), myCompare);         // Sort by file
      name
00408                 // Grab first image data for console info
00409                 rastImage testFirst;
00410                 img_load(&testFirst,batch_list[0]);
00411                 printf("Loaded %ld images with a width of %dpx, a height of %dpx. The original images had
      %d channels.\n", batch_count, testFirst.width, testFirst.height, testFirst.ch);
00412                 img_free(&testFirst);
00413                 long tUs_seq;
00414                 // Sequential Batch image load.
00415                 if (result.count("c") || result.count("s")) {
00416                     printf("Running sequential... ");
00417                     gettimeofday(&start, nullptr);
00418                     seq_img_batch(CONTACT_KERNEL, KERN_X, KERN_Y, batch_count, globalThresh, batch_list);
00419                     gettimeofday(&end, nullptr);
00420                     tUs = (end.tv_sec - start.tv_sec) * 1000000 + end.tv_usec - start.tv_usec;
00421                     printf("Comp. Time:\t\t%ld [us],\t", tUs);
00422                     printf("AVG. Frame Time: %ld [us].\n", tUs/batch_count);
00423                     tUs_seq = tUs;
00424                 }
00425                 // Parallel (Default)
00426                 if(!result.count("s")) {
00427                     printf("Running parallel pipeline... ");
00428                     fflush(stdout);
00429                     gettimeofday(&start, nullptr);
00430                     tbb_img_batch batch_job(batch_count, batch_list, globalThresh);
```

```
00431                     batch_job(ntoken);
00432                     gettimeofday(&end, nullptr);
00433                     tUs = (end.tv_sec - start.tv_sec) * 1000000 + end.tv_usec - start.tv_usec;
00434                     printf("Comp. Time:\t%ld [us],\t", tUs);
00435                     printf("AVG. Frame Time: %ld [us].\n", tUs / batch_count);
00436                 }
00437                 if(result.count("c")) {
00438                     printf("-----------------------------------------\n");
00439                     float perc = (1 - ((float) tUs / (float) tUs_seq)) * 100;
00440                     printf("Total savings: %f%%\n", perc);
00441                 }
00442                 // Cleanup
00443                 for (size_t i = 0; i < batch_count; i++) {
00444                     free(batch_list[i]);
00445                 }
00446                 free(batch_list);
00447             }
00448         } else {
00449             printf("No source file listed\n");
00450             exit(0);
00451         }
00452     }
00453     catch (const cxxopts::OptionException& e) {
00454         std::cout « "error parsing options: " « e.what() « std::endl;
00455         exit(1);
00456     }
00457     return 0;
00458 }
```

## 3.4 README.md File Reference

# Index