Multi-threading – Basic Examples + 2D Convolution

OBJECTIVES
- Learn the basics of multi-threading implementation using *pthreads* in C.
- Execute multi-threaded applications and measure the computation time.
- Compare multi-threaded applications against sequential (non-threaded) implementations.

USEFUL INFORMATION
- Refer to the Tutorial: Embedded Intel for the source files used in this Tutorial.
- Refer to the board website or the Tutorial: Embedded Intel for User Manuals and Guides.

BOARD SETUP (DE2I-150 TERASIC DEV. KIT) AND POWERING
- Connect the monitor (VGA or HDMI) as well as the keyboard and mouse.
- Connect the provided power cord to the power supply and plug the cord into a power outlet.
- Connect the supplied 12V DE2i-150 power adapter to the power connect (J1) on the DE2i-150 board. At this point, you should see the 12 V LED (D33) turn on.
  - ✓ Be careful not to plug the power adapter into the SATA power connector (see *DE2i-150 Getting Started Guide*, page 7).
- Click the **Power ON/OFF Button** (lower right corner) to boot the OS.
- The board should power on, emitting some beeps to indicate a successful load of the BIOS.

ACTIVITIES

FIRST ACTIVITY: SIMPLE PTHREADS EXAMPLES
- The following are simple examples that illustrates the use of pthreads.

FIRST EXAMPLE:
- Basic declaration of a group of threads.

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

void *execute_work (void *arg) { // thread function
  int i = *( (int *) arg); // arg: originally a pointer to integer. Then passed as a pointer to void
  printf("Thread %d: Started\n", i); printf("Thread %d: Ended\n", i);
  return 0; }

int main(int argc, char* argv[]) {
  int i, status, NUM_THREADS;   // NUM_THREADS: number of threads
  pthread_t *thread; // Declare threads' identifier: pointer to a group of threads
  int *thread_args; // Arguments for threads

  if (argc!=2) { printf("(main) Usage: %s number_of_threads\n",argv[0]); exit(-1); }
  NUM_THREADS = atoi(argv[1]);
  if (NUM_THREADS < 1) { printf ("(main) Incorrect number of threads!\n"); exit(-1); }

  thread_args = (int *) calloc (NUM_THREADS, sizeof(int)); // memory allocation: threads arguments
  thread = (pthread_t*) malloc(NUM_THREADS*sizeof(pthread_t)); // memory allocation: threads indices

  for (i = 0; i < NUM_THREADS; i++) { // creating all threads
    thread_args[i] = i; // unique argument per thread
    status = pthread_create (&thread[i], NULL, execute_work, (void *) &thread_args[i] );
    if (status != 0) { perror("Can't create thread"); free (thread); exit (-1); }  }

  // Wait for each thread to finish
  for (i = 0; i < NUM_THREADS; i++) pthread_join (thread[i], NULL);

  printf("(main) program has ended\n");
  free(thread_args); free(thread);
  return 0;
}
```

- ✓ ***thread start function***: `void *execute_work (void *arg)` Argument (passed): `&threads_args[i]`
  - The function (the same for all threads) prints out the thread id provided to the function when the thread was created.
  - This function specifies a return value (0). This is how we exit the threads.
- ✓ The program creates `NUM_THREADS` threads via `pthread_create`. Then, waits for them to finish by calling `pthread_join`.

- Application file: `pthreads_example.c`
- Compile this code: `gcc -Wall pthreads_example.c -o pthreads_example -lpthread` ↵
- Execute this application: `./pthreads_example <number of threads>` ↵
  - ✓ Example: `./pthreads 10` ↵
  - ✓ Fig. 1 shows the program execution. The program creates 10 threads and waits until they complete. Note that the threads are not created, executed, and completed in a consecutive fashion. Fig. 1 execution varies every time code is run.

Figure 1. Program execution with 10 threads. Note that thread 1 is created before thread 0. Also thread 9 ends before thread 3 ends. This execution differs every time we run the code.

SECOND EXAMPLE:

- Simple example illustrating mutex usage.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

void print(char* a, char* b) { // try uncommenting and commenting the mutex below and look at output
  pthread_mutex_lock(&mutex1); // comment out/uncomment
  printf("1: %s\n", a); sleep(1);
  printf("2: %s\n", b);
  pthread_mutex_unlock(&mutex1); } // comment out/uncomment

// These two functions will run concurrently:
void* print_i(void *ptr) { print("I am", " in i"); pthread_exit(NULL); }
void* print_j(void *ptr) { print("I am", " in j"); pthread_exit(NULL); }

int main() {
  pthread_t t1, t2;
  int status;
  status = pthread_create(&t1, NULL, print_i, NULL);
  status = pthread_create(&t2, NULL, print_j, NULL);
  status = pthread_join(t1, NULL);
  status = pthread_join(t2, NULL);
  return 0;
}
```

- ✓ ***thread start function***:
  - Thread `t1: void* print_i (void *ptr))` Argument that is passed: `NULL`
  - Thread `t2: void* print_j (void *ptr))` Argument that is passed: `NULL`
  - The threads use `pthread_exit` to exit.

✓ With a mutex, the two threads (when created) execute concurrently. However, when a thread (`t1` or `t2`) starts executing, it will lock a portion of the code. Only when that portion is completed, it is unlocked so that the other thread can execute.

- Application file: `mutex_exam.c`
- Compile this code: `gcc -Wall mutex_exam.c -o mutex_exam -lpthread ↵`
- Execute this application: `./mutex_exam ↵`

✓ Program output (with mutex):

```
1: I am
2:  in i
1: I am
2:  in j
```

THIRD EXAMPLE:

- Dot product using a mutex. Vectors' length: `VECLEN×NUMTHRDS`.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
// Source: https://computing.llnl.gov/tutorials/pthreads/
typedef struct {
   double      *a;
   double      *b;
   double     sum;
   int     veclen;
 } DOTDATA;

/* Define globally accessible variables and a mutex */
#define NUMTHRDS 4
#define VECLEN 100000 // This is the length of the vector each thread operates on
   DOTDATA dotstr; // global structure
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;

void *dotprod (void *arg) { // thread start function
   int i, start, end, len;
   long offset;
   double tsum, *x, *y;
   offset = (long)arg; // argument passed from main()

   len = dotstr.veclen; x = dotstr.a;        y = dotstr.b;
   start = offset*len;  end = start + len; // [start, end): range on which this thread operates

   tsum = 0; // sum computed by a single thread
   for (i = start; i < end; i++) { tsum += (x[i] * y[i]); } // perform dot product

   // Lock a mutex prior to updating the value in the shared structure, and unlock it upon updating
   pthread_mutex_lock (&mutexsum);
   dotstr.sum += tsum;
   printf("Thread %ld did %d to %d:  tsum=%f global sum=%f\n",offset,start,end-1,tsum,dotstr.sum);
   pthread_mutex_unlock (&mutexsum);
   pthread_exit((void*) 0);
}

int main (int argc, char *argv[]) {
   long i;
   double *a, *b;
   void *status;
   a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
   b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));

   for (i=0; i < VECLEN*NUMTHRDS; i++) { a[i]=1;   b[i]=a[i]; } // input data (just 1's)

   dotstr.veclen = VECLEN; dotstr.a = a; dotstr.b = b; dotstr.sum=0; // initializing global variables
   pthread_mutex_init(&mutexsum, NULL); // mutex: dynamic initialization

   // create and join threads
   for (i=0; i < NUMTHRDS; i++) pthread_create(&callThd[i], NULL, dotprod, (void *)i);
   for (i=0; i < NUMTHRDS; i++) pthread_join(callThd[i], &status);

   printf ("Sum =  %f \n", dotstr.sum); // print out results
   free (a); free (b);
   pthread_mutex_destroy(&mutexsum);
   pthread_exit (NULL);
}
```

- ✓ ***thread start function***: `void *dotprod (void *arg)` Argument that is passed: `void *i`
  - It includes the mutex lock and unlock.
  - It uses `pthread_exit` to exit.
- ✓ The program creates `NUMTHRDS` threads and then waits for all threads to finish by calling `pthread_join` on each thread.
- ✓ Fig. 2 shows the thread strategy for 4 threads and vector length of 8. Each thread gets an index i (called **`offset`** in the ***thread start function***) and processes data over the range $[i \times len, (i+1) \times len - 1]$, where `len=VECLEN`. Then, it locks a mutex, updates the global variable `dotstr.sum`, and unlocks the mutex.
  - The mutex is necessary. Otherwise, the dot product result might be updated by a different thread than the one generating a partial product. This will cause an incorrect value (`tsum` at the wrong time) to be added to the result.

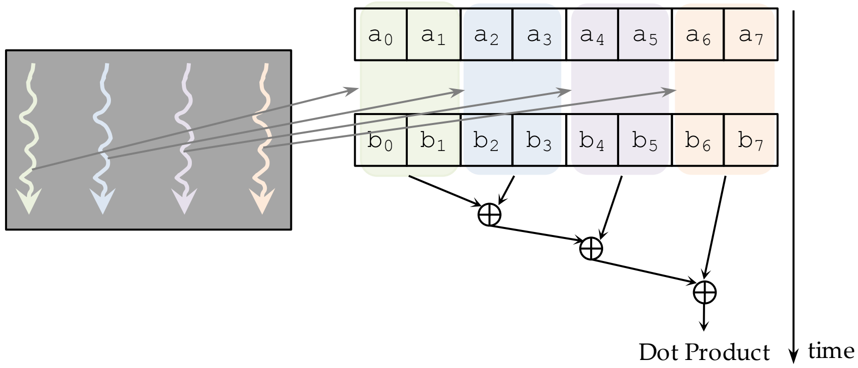

Figure 2. Task allocation for 4 threads and vector length of 8 (VECLEN = 2). Thread 0 computes partial dot product for elements 0 and 1. Thread 1 operates on elements 2 and 3. Thread 2 operates on elements 4 and 5. Thread 3 operates on elements 6 and 7. The computation of each partial dot product occurs concurrently. However, updating the global variable `dotstr.sum` (the result) occurs sequentially. This is enforced via the mutex.

- Application file: `dotprod.c`
- Compile this code: `gcc -Wall dotprod.c -o dotprod -lpthread ↲`
- Execute this application: `./dotprod ↲`
  - ✓ Program output:

```
thread 1: did 100000 to 199999: tsum = 100000 global sum = 100000
thread 0: did 0 to 99999: tsum = 100000 global sum = 200000
thread 2: did 200000 to 299999: tsum = 100000 global sum = 300000
thread 3: did 300000 to 399999: tsum = 100000 global sum = 400000
```

SECOND ACTIVITY: MULTIPLE 2D CONVOLUTIONS (SEQUENTIAL IMPLEMENTATION)

- Refer to Tutorial # 2 for details of the 2D convolution operation.
  - ✓ In a 2D convolution, the input matrix I is of size SX×SY (SX columns, SY rows), the kernel is of size KX×KY, while the output (considering only the central part of the convolution output) is of size SX×SY.
- Here, we will apply multiple convolutions (different kernels) to one input matrix. The purpose of this exercise is to have a sequential implementation to compare against a multi-threaded implementation (Third Activity).
- In this implementation, we read an input matrix, three different kernels, and generate three different output matrices. Two examples are shown:
  - ✓ Small input matrix: SX=SY=4, KX=KY=3. This is shown in Fig. 3. The input matrix is read from a text file, and the output matrix is written as a text file.
  - ✓ Grayscale image: SX=640, SY=480, KX=KY=3. This is shown in Fig. 4. Input matrix: read as a binary file. Output matrix: written as a binary file.

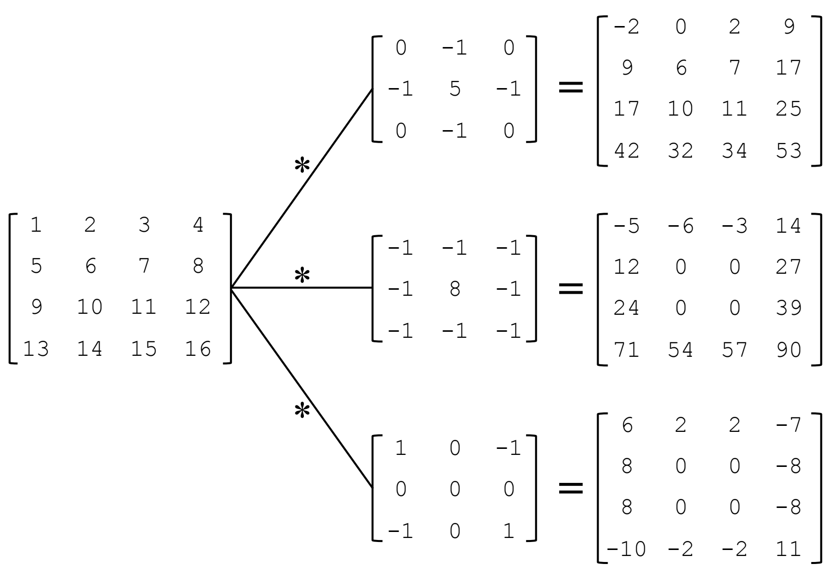

Figure 3. Applying three 2D convolution kernels to one input matrix. SX=SY=4, KX=KY=3. Output size is the same as input size.

- General Procedure (sequential execution of convolutions):
  1. Read input matrix I (from a text file or binary file)
  2. Read kernel matrices K_i from text files.
  3. **for** i = 0 to 2 **do**
     Compute convolution with kernel K_i.
     Store result on output matrix O_i and then save it on a text file or binary file.
  4. **end for**

- Application files: `conv2m.c, conv2m_fun.c, conv2m_fun.h, Makefile`
  ✓ Note that we measure the processing time (us) using `gettimeofday()`.

- Compile this application: `make conv2m` ↵
  ✓ You can also do: `make all` ↵

- Execute this application: `./conv_2m <modifier>` ↵
  ✓ Two execution possibilities:
    - `./conv2m 1`: Input matrix I read as a text file; output matrix O is stored as a text file. See Fig. 3 for the matrices. Fig. 5 depicts the execution on the Terasic DE2i-150 Board.
    - `./conv2m 2`: Input matrix I read as a binary file; output matrix O is stored as a binary file. See Fig. 4 for the images. Fig. 6 depicts the execution on the Terasic DE2i-150 Board.
  ✓ You can use MATLAB® to verify that your results (output images) are correct.
    - Files: `img_op_m.m, iss.jpg.`

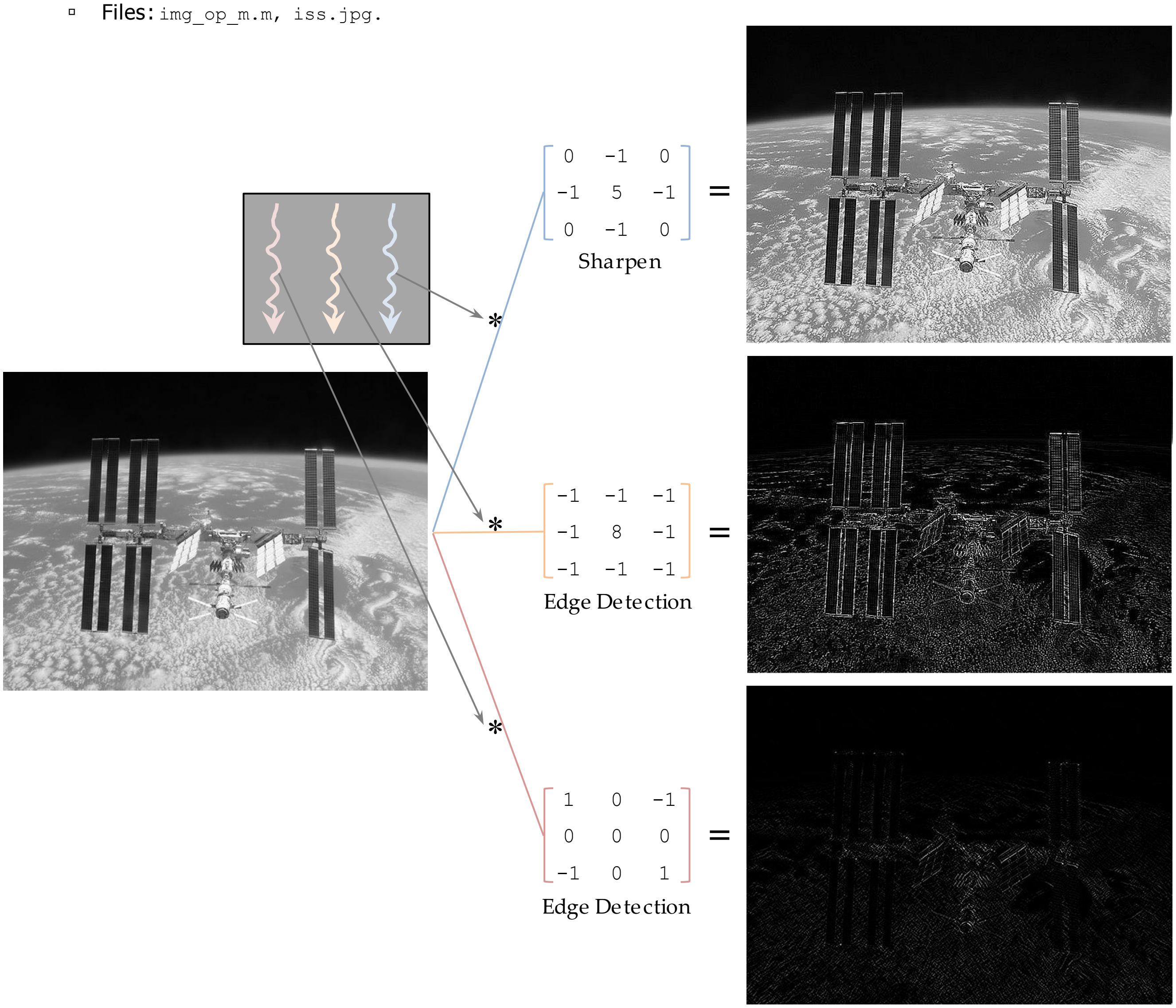

Figure 4. 2D convolution for a grayscale image. SX=640, SY=480, KX=KY=3. Output image: the pixel values might fall outside the [0,255] bounds. When displaying, it is customary to restrict the pixel values to [0, 255]. This figure also shows the strategy when implementing this with threads: each thread computes one convolution concurrently.

```
ece4900@atom: ~/work_ubuntu/pthreads/conv2m
(write_txtfile) Output Matrix
-2      0       2       9
9       6       7       17
17      10      11      25
42      32      34      53
(write_txtfile) Output Matrix
-5      -6      -3      14
12      0       0       27
24      0       0       39
71      54      57      90
(write_txtfile) Output Matrix
6       2       2       -7
8       0       0       -8
8       0       0       -8
-10     -2      -2      11
start: 329691 us
end: 329703 us
Elapsed time (only convolutions (3) computation): 12 us
ece4900@atom:~/work_ubuntu/pthreads/conv2m$
```

Figure 5. Execution of three matrix convolutions of size 4x4 on the Terasic DE2i-150 FPGA Development Kit. This is a sequential (non-threaded) implementation

```
ece4900@atom: ~/work_ubuntu/pthreads/conv2m
ece4900@atom:~/work_ubuntu/pthreads/conv2m$ ./conv2m 2
(read_binfile) Input binary file 'iss.bif': # of elements read = 3
07200
(read_binfile) Size of each element: 1 bytes
Kernel 0:
0       -1      0
-1      5       -1
0       -1      0
Kernel 1:
-1      -1      -1
-1      8       -1
-1      -1      -1
Kernel 2:
1       0       -1
0       0       0
-1      0       1
(write_binfile) Output binary file 'iss_a.bof': # of elements writ
ten = 307200
(write_binfile) Output binary file 'iss_b.bof': # of elements writ
ten = 307200
(write_binfile) Output binary file 'iss_c.bof': # of elements writ
ten = 307200
start: 181824 us
end: 336392 us
Elapsed time (only convolutions (3) computation): 154568 us
ece4900@atom:~/work_ubuntu/pthreads/conv2m$
```

Figure 6. Execution of three image convolutions of size 640x480 on the Terasic DE2i-150 FPGA Development Kit. This is a sequential (non-threaded) implementation

THIRD ACTIVITY: 2D CONVOLUTION WITH PTHREADS

- Using *pthreads* leverages the parallel computing capabilities of the microprocessor. Here, parallelism is achieved by distributing the operations (ideally evenly) among threads than run in parallel.
- Though an individual 2D convolution could be parallelized, we prefer to execute several 2D convolutions in parallel. This has an important application in the development of Convolutional Neural Networks (CNNs).
- Here, we distributed the operation in terms of individual convolutions.

STRATEGY

- To compute a group of 2D convolutions, a group of threads is generated, where each thread computes an individual 2D convolutions. All these threads simultaneously compute the 2D convolutions.
- If the number of threads is given by *nthreads*, then the index *i* represents each thread form *0* to *nthreads-1*. Thread *i* computes convolution *i*.
- Fig. 7 depicts the strategy for 3 threads, each in charge of computing an individual computation on a 4x4 matrix. The same strategy is depicted in Fig. 4 for a 640x480 image.

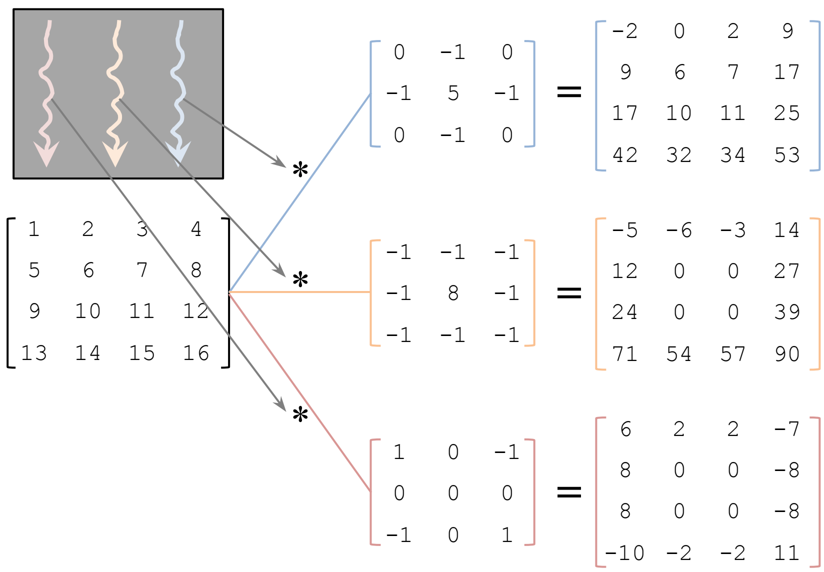

Figure 7. 2D convolution example. SX=SY=4, KX=KY=3. Output size is the same as input size.

- Application files: `conv2m_pthreads.c`, `conv2m_fun.c`, `conv2m_fun.h`, `Makefile`
  - ✓ Note that we measure the processing time (us) using `gettimeofday()`. In Fig. 8 and 9, the measurements include the printing of the messages: "computing slice or thread i".
  - ✓ Code structure:
    - Thread generation and initialization of arguments.
    - Initialization of input I, and the kernels Ka, Kb, Kc by reading from binary files and/or text files
    - Create *nthreads* threads, where each thread *i* computes an individual 2D convolution.
    - Wait until threads complete, merge all the results.
    - Store result (output matrices) on text files or binary files (one per convolution output).

- Compile this application: `make conv2m_pthreads` ↵
- Execute this application: `./conv2m_pthreads <modifier>` ↵
  - ✓ Fig. 8 and Fig. 9 display the execution on the Terasic DE2i-150 Board for the 4x4 matrix (`./conv2m_pthreads 1`) and the 640x480 grayscale image (`./conv2m_pthreads 2`) respectively.

PERFORMANCE COMPARISON

- Table I shows the comparison of the computation time between the sequential implementation and the multi-threaded implementation. When processing images, we see a reduction in the execution time (as this is a relatively large computation). Note that when processing the small matrix, the execution time using threads is larger than the basic sequential implementation. This is because the setup overhead is longer than the actual computation time.

TABLE I. EXECUTION TIME (US) COMPARISON BETWEEN MULTI-THREADED AND NON-THREADED IMPLEMENTATIONS

	Implementation	
Application	Sequential (non-threaded)	Multi-threaded (3 threads)
4x4 image (Fig. 5)	12	432
640x480 images (Fig. 2)	154568	84810

```
ece4900@atom: ~/work_ubuntu/pthreads/conv2m
Creating 3 Threads
Computing slice (or thread) 0
Computing slice (or thread) 1
Computing slice (or thread) 2
(write_txtfile) Output Matrix
-2      0       2       9
9       6       7       17
17      10      11      25
42      32      34      53
(write_txtfile) Output Matrix
-5      -6      -3      14
12      0       0       27
24      0       0       39
71      54      57      90
(write_txtfile) Output Matrix
6       2       2       -7
8       0       0       -8
8       0       0       -8
-10     -2      -2      11
start: 767012 us
end: 767696 us
Elapsed time (only convolutions (3) computation): 684 us
ece4900@atom:~/work_ubuntu/pthreads/conv2m$
```

Figure 8. Execution of three matrix convolutions of size 4x4 on the Terasic DE2i-150 FPGA Development Kit. This is a multi-threaded (3 threads) implementation.

```
ece4900@atom: ~/work_ubuntu/pthreads/conv2m
0       -1      0
-1      5       -1
0       -1      0
Kernel 1:
-1      -1      -1
-1      8       -1
-1      -1      -1
Kernel 2:
1       0       -1
0       0       0
-1      0       1


Creating 3 Threads
Computing slice (or thread) 0
Computing slice (or thread) 2
Computing slice (or thread) 1
(write_binfile) Output binary file 'iss_a.bof': # of elements written = 307200
(write_binfile) Output binary file 'iss_b.bof': # of elements written = 307200
(write_binfile) Output binary file 'iss_c.bof': # of elements written = 307200
start: 868407 us
end: 955453 us
Elapsed time (only convolutions (3) computation): 87046 us
ece4900@atom:~/work_ubuntu/pthreads/conv2m$
```

Figure 9. Execution of three image convolutions of size 640x480 on the Terasic DE2i-150 FPGA Development Kit. This is a multi-threaded (3 threads) implementation.