



Project Report

ADVANCED COMPUTER ARCHITECTURE

TOPIC

MULTIPROCESSORS MEDIAN FILTER USING JAVA THREAD

Advisor

Nguyen Duc Minh

Group Member

Cao Anh Quan - USTHBI4 - 125

Do Son Tung – USTHBI4 -157

Do Gia Khang – USTHBI4 -

Hanoi 4 – 2015

CONTENT

CONTENT.....	1
Part 1. Overview of Multiprocessor.....	4
1.1 Multiprocessing Challenge.....	4
1.2 Multiprocessor's Types.....	4
1.2.1 Centralized Shared Memory.....	4
1.2.2 Distributed Shared Memory.....	5
1.3 Problem of Multiprocessors and Resolution.....	5
Part 2. Implement Median Filter using Java Thread.....	7
2.1. Introduction to Median Filter.....	7
2.1.1. What is Median Filter?	7
2.1.2. Median Filter Algorithm	7
2.2. Program Description.....	7
2.2.1. Functions of Program	7
2.2.2. Specifications	8
2.3. Experiment.....	8
2.3.1. Tools.....	8
2.3.2. Program on PC	10
2.3.2.1. Testing System.....	10
2.3.2.2. Test Cases.....	10
2.3.2.3. Charts.....	14
2.3.3. Program on Android Device.....	16
2.3.3.1. Testing System.....	16
2.3.3.2. Test Cases.....	16
2.3.3.3. Charts.....	17
2.3.4. Profiling Information	17
2.3.5. Summary	21
2.3.5.1. Expectation.....	21
2.3.5.2. Results.....	21

2.3.5.3. Result Images.....	22
2.3.5.4. Explanation.....	26

PART 1

Overview of Multiprocessor

1.1 Multiprocessing Challenge

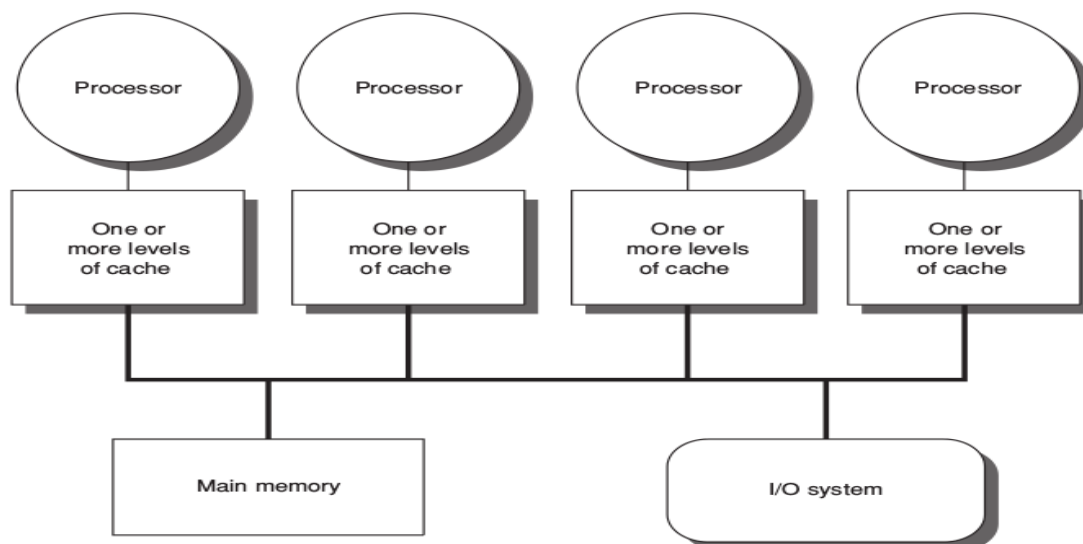
Multiprocessing is the use of two or more CPUs within a single computer system. It has the ability of a system to support more than one processor and/or the ability to allocate tasks between them. Multiprocessing is sometimes used to refer to the execution of multiple concurrent processes in a system as opposed to a single process at any one instant. So that, in doing multiprocessing we often have some challenge:

- Running parallel programs where threads must communicate to complete the task
- The large latency of remote access in a parallel processor

1.2 Multiprocessor's Types

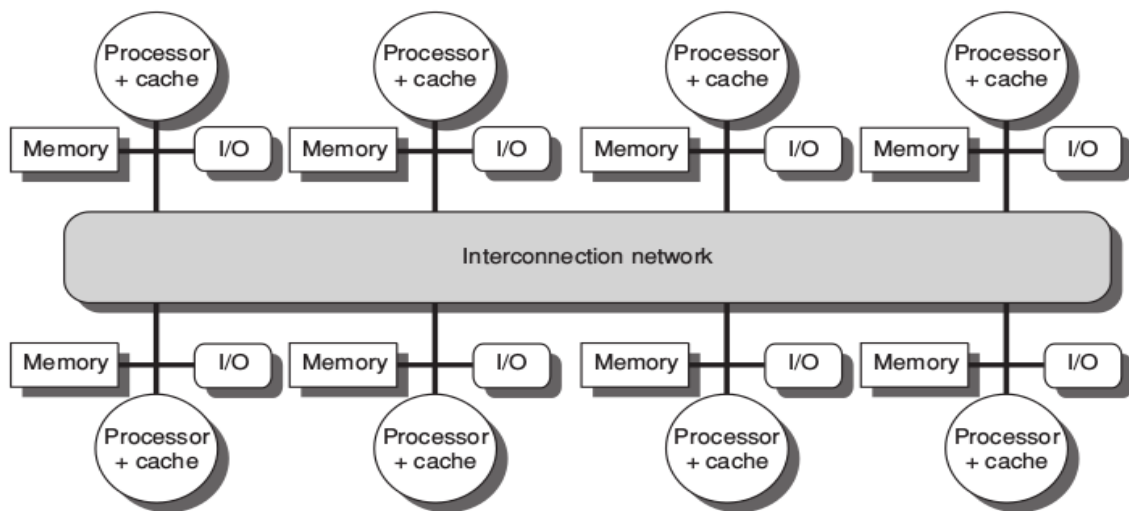
1.2.1 Centralized Shared Memory

- It's a multiprocessing design where several processors access globally shared memory.
- Multiple processor will access into the same main memory.
- Basic architecture



1.2.2 Distributed Shared Memory

- It's a form of memory architecture where the (physically separate) memories can be addressed as one (logically shared) address space/
- Basic architecture: Each individual nodes containing a processor, some memory, typically some I/O, and an interface to an interconnection network that connects all the nodes



1.3 Problem of Multiprocessors and Resolution

Cache Coherence

- Phenomenon: 2 different processors have 2 different values from the same shared location
- Reason: existing 2 states: global (defined by main memory) & local (defined by individual caches)
- Resolution:

- Snooping protocol

A protocol for maintaining cache coherency in symmetric multiprocessing environments. In a snooping system, all caches on the bus monitor the bus to determine if they have a copy of the block of data that is requested on the bus. Every cache has a copy of the sharing status of every block of physical memory it has. Multiple copies of a document in a multiprocessing environment typically can be read without any coherence problems; however, a processor must have exclusive access to the bus in order to write

➤ There are two main types of snooping protocol:

1. Write-invalidate

The processor that is writing data causes copies in the caches of all other processors in the system to be rendered invalid before it changes its local copy. The local machine does this

by sending an invalidation signal over the bus, which causes all of the other caches to check for a copy of the invalidated file. Once the cache copies have been invalidated, the data on the local machine can be updated until another processor requests it.

2. Write-update

The processor that is writing the data broadcasts the new data over the bus (without issuing the invalidation signal). All caches that contain copies of the data are then updated. This scheme differs from write-invalidate in that it does not create only one local copy for writes.

➤ Pros:

- Fast
- All request/response can be seen by all processors

➤ Cons:

- Broadcast the value all over the bus
- Not scalable (Use much bandwidth)

○ Directory Protocol

Directory-based protocols were invented as a means of dealing with cache coherence in systems containing more processors than can be accommodated on a single bus. Each node will have one directory to store a vector bit which remembers the sharing status of each memory block

Pros:

- Don't use much bandwidth (P2P not broadcast)
- Good for large scale system (> 64 processors)

Cons:

- Longer Latency

PART 2

IMPLEMENT MEDIAN FILTER USING JAVA THREADS

2.1. Introduction to Median Filter:

2.1.1. What is Median Filter?

- Median filtering is a nonlinear process useful in reducing impulsive, or salt-and-pepper noise. It is also useful in preserving edges in an image while reducing random noise.
- Impulsive or salt-and pepper noise can occur due to a random bit error in a communication channel. In a median filter, a window slides along the image, and the median intensity value of the pixels within the window becomes the output intensity of the pixel being processed.
- For example, suppose the pixel values within a window are 5,6, 55, 10 and 15, and the pixel being processed has a value of 55. The output of the median filter at the current pixel location is 10, which is the median of the five values.

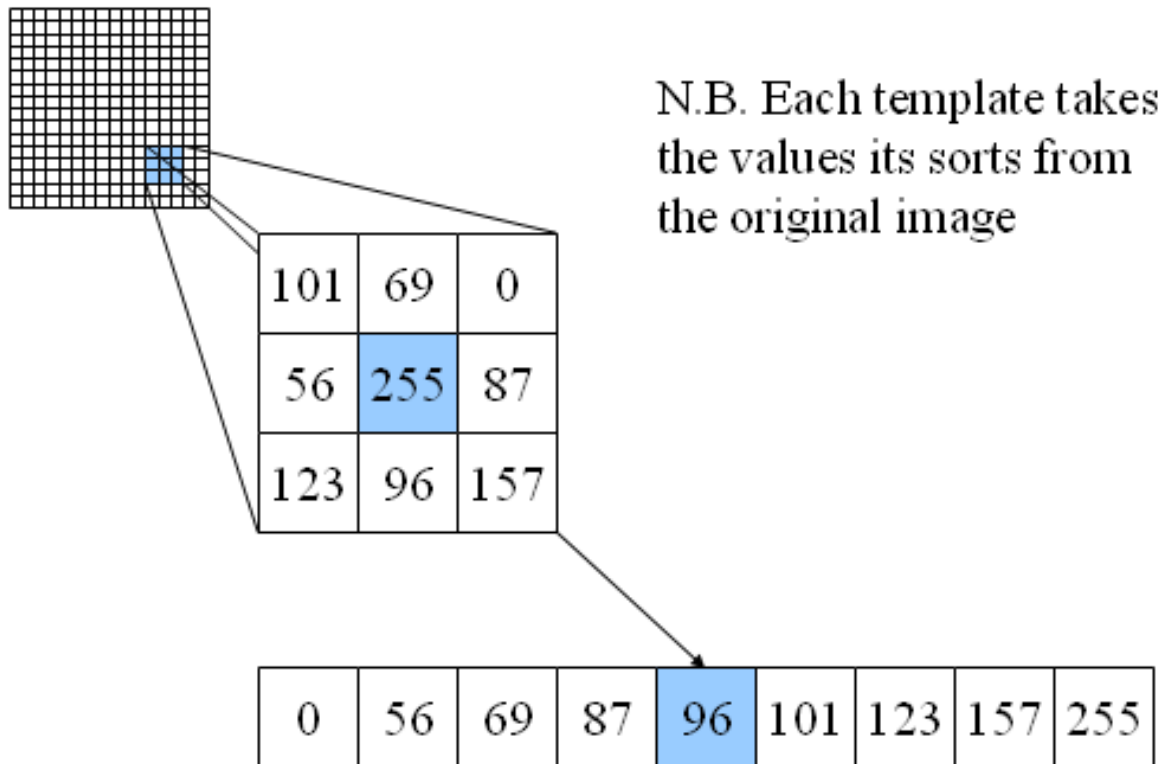
2.1.2. Median Filter Algorithm:

- Run through the signal entry by entry, replacing each entry with the median of neighboring entries.
- The pattern of neighbors is called the "window", which slides, entry by entry, over the entire signal.
- For 1D signals, the most obvious window is just the first few preceding and following entries, whereas for 2D (or higher-dimensional) signals such as images, more complex window patterns are possible (such as "box" or "cross" patterns).

2.2. Program Description

2.2.1. Function:

- The program allow you to select the image, then select the number of runs, select the number of threads to run, then run median filter with the image, display the output image.Show the run time.The program can automatically run all the test.



2.2.2. Specifications:

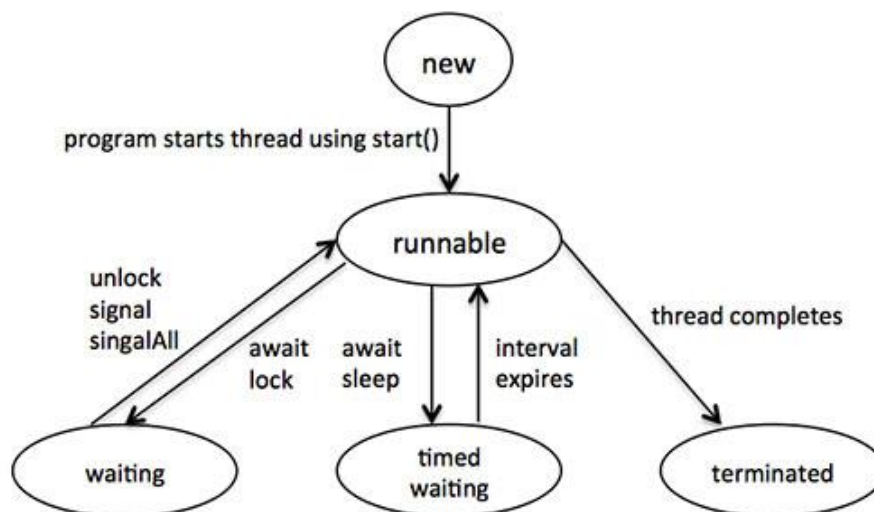
- The program successfully filter the image.
- The program allow the user to choose the image, number of threads and number of runs.
- The program filter the image with the give number of threads and given number of runs.
- The program show the output image to user.
- The program show the run time correctly to user.
- The program can automatically run all the tests at once then display the runtime (for program on PC)

2.3. Experiments:

2.3.1. Tools:

- Profiling tool: YourKit Java Profiler 2015 build 15050.**
- IDE: Eclipse 4.4.2 (Luna) with plugin ADT-23.0.6 (for developing android program)**
- Language: Java –Multitasking (JDK 1.8.0_40)**
 - Java is a multithreaded programming language which means we can develop multithreaded program using Java.

- A multithreaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.
- By definition **multitasking** is when multiple processes share common processing resources such as a CPU.
- **Multithreading** extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel.
- The OS divides processing time not only among different applications, but also among each thread within an application. Multithreading enables you to write in a way where multiple activities can proceed concurrently in the same program.
- A thread, in the context of Java, is the path followed when executing a program. All Java programs have at least one thread, known as the main thread, which is created by the JVM at the program's start, when the main() method is invoked with the main thread.
- In Java, creating a thread is accomplished by implementing an i



d

extending a class. Every Java thread is created and controlled by the java.lang.Thread class.

- When a thread is created, it is assigned a priority. The thread with higher priority is executed first, followed by lower-priority threads. The JVM stops executing threads under either of the following conditions:
 - If the exit method has been invoked and authorized by the security manager

- All the daemon threads of the program have die

2.3.2. Program on PC:

2.3.2.1. Testing System:

Device name	Dell vostro 3560
Operating system	Window 8 Enterprise
Memory	8GiB
Processor	Intel® Core™ i5-3230M CPU @ 2.60GHz x 4
Physical cores	2
Logical cores	4 (hyperthreading technology)
JDK	1.8.0_40
Graphic	Intel® Ivybridge Mobile
Image	800 * 600

2.3.2.2. Test Cases:

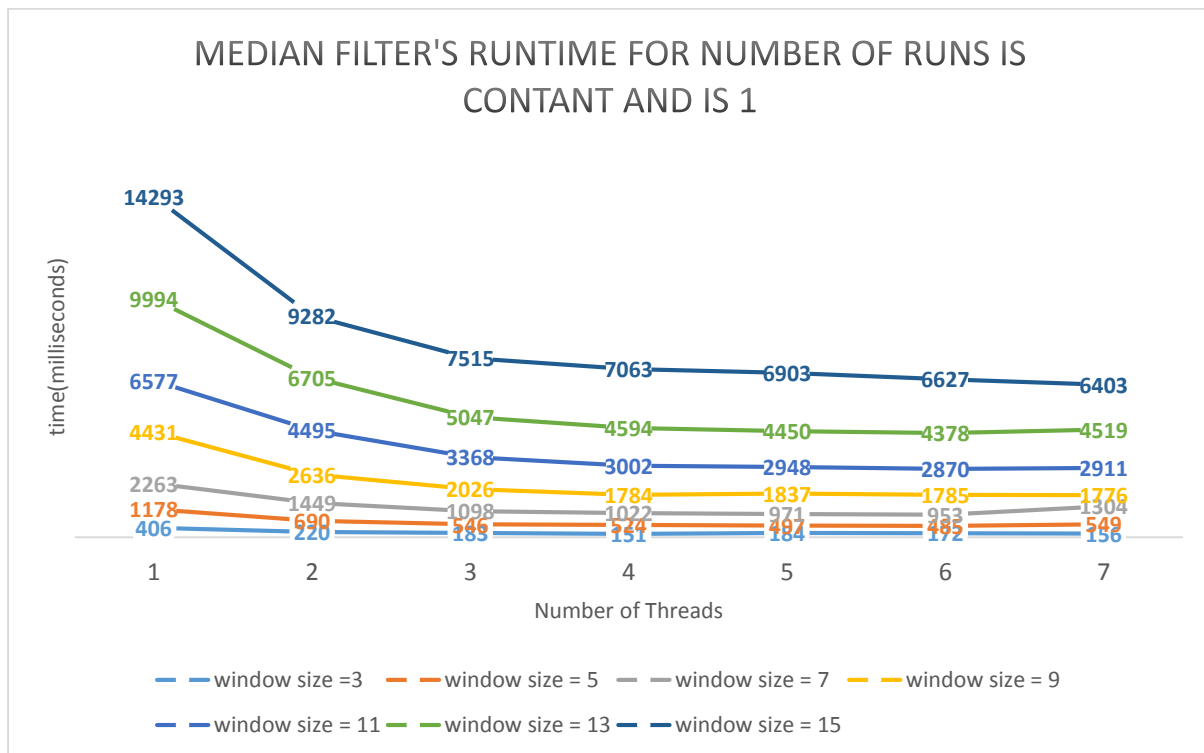
No.	Threads	Runs	Window Size	Runtime
1	1	1	3	406
2	2	1	3	220
3	3	1	3	183
4	4	1	3	151
5	5	1	3	184
6	6	1	3	172
7	7	1	3	156
8	1	1	5	1178
9	2	1	5	690
10	3	1	5	546
11	4	1	5	524
12	5	1	5	497
13	6	1	5	485
14	7	1	5	549
15	1	1	7	2263
16	2	1	7	1449
17	3	1	7	1098
18	4	1	7	1022
19	5	1	7	971
20	6	1	7	953

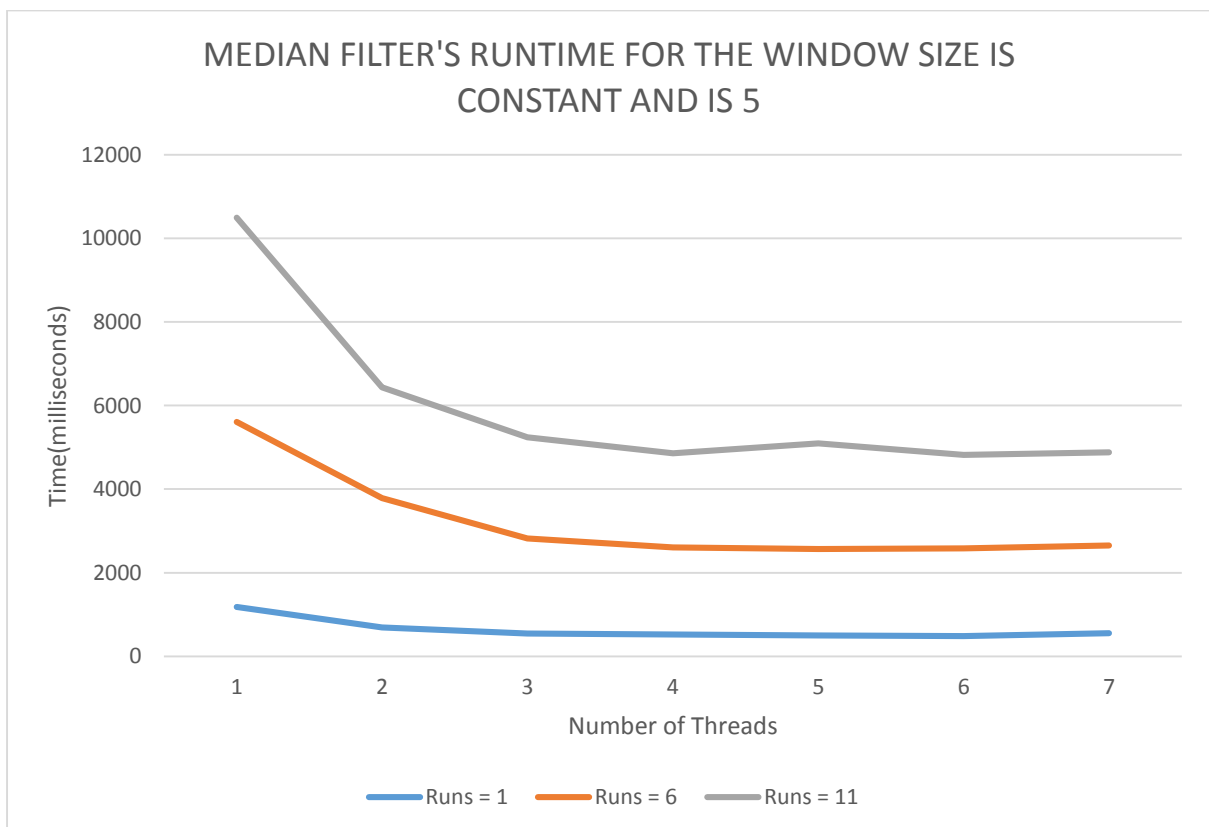
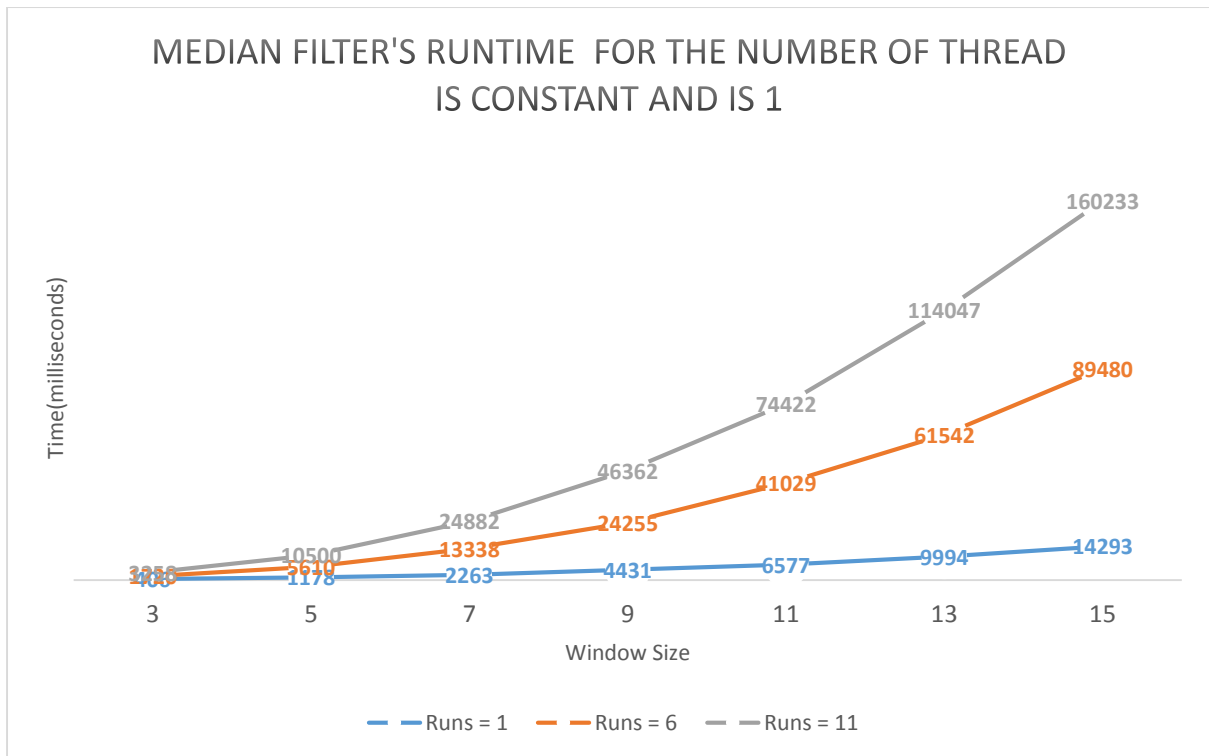
21	7	1	7	1304
22	1	1	9	4431
23	2	1	9	2636
24	3	1	9	2026
25	4	1	9	1784
26	5	1	9	1837
27	6	1	9	1785
28	7	1	9	1776
29	1	1	11	6577
30	2	1	11	4495
31	3	1	11	3368
32	4	1	11	3002
33	5	1	11	2948
34	6	1	11	2870
35	7	1	11	2911
36	1	1	13	9994
37	2	1	13	6705
38	3	1	13	5047
39	4	1	13	4594
40	5	1	13	4450
41	6	1	13	4378
42	7	1	13	4519
43	1	1	15	14293
44	2	1	15	9282
45	3	1	15	7515
46	4	1	15	7063
47	5	1	15	6903
48	6	1	15	6627
49	7	1	15	6403
50	1	6	3	1720
51	2	6	3	1144
52	3	6	3	896
53	4	6	3	860
54	5	6	3	834
55	6	6	3	805
56	7	6	3	844
57	1	6	5	5610
58	2	6	5	3785
59	3	6	5	2823
60	4	6	5	2603
61	5	6	5	2568
62	6	6	5	2586

63	7	6	5	2650
64	1	6	7	13338
65	2	6	7	8512
66	3	6	7	6750
67	4	6	7	5912
68	5	6	7	5948
69	6	6	7	5828
70	7	6	7	5858
71	1	6	9	24255
72	2	6	9	15828
73	3	6	9	12249
74	4	6	9	10928
75	5	6	9	11059
76	6	6	9	10750
77	7	6	9	10836
78	1	6	11	41029
79	2	6	11	26369
80	3	6	11	20581
81	4	6	11	18487
82	5	6	11	18338
83	6	6	11	17920
84	7	6	11	17760
85	1	6	13	61542
86	2	6	13	40084
87	3	6	13	33426
88	4	6	13	29329
89	5	6	13	27817
90	6	6	13	27149
91	7	6	13	27123
92	1	6	15	89480
93	2	6	15	56921
94	3	6	15	44488
95	4	6	15	40329
96	5	6	15	39256
97	6	6	15	38446
98	7	6	15	39025
99	1	11	3	3258
100	2	11	3	1954
101	3	11	3	1645
102	4	11	3	1423
103	5	11	3	1663
104	6	11	3	1526

105	7	11	3	1606
106	1	11	5	10500
107	2	11	5	6439
108	3	11	5	5244
109	4	11	5	4855
110	5	11	5	5093
111	6	11	5	4820
112	7	11	5	4884
113	1	11	7	24882
114	2	11	7	15913
115	3	11	7	12649
116	4	11	7	11024
117	5	11	7	10930
118	6	11	7	10725
119	7	11	7	10707
120	1	11	9	46362
121	2	11	9	29638
122	3	11	9	22911
123	4	11	9	20072
124	5	11	9	19696
125	6	11	9	19268
126	7	11	9	19553
127	1	11	11	74422
128	2	11	11	48080
129	3	11	11	37551
130	4	11	11	33216
131	5	11	11	32538
132	6	11	11	31956
133	7	11	11	32192
134	1	11	13	114047
135	2	11	13	72778
136	3	11	13	57925
137	4	11	13	51747
138	5	11	13	50399
139	6	11	13	49314
140	7	11	13	49991
141	1	11	15	160233
142	2	11	15	104381
143	3	11	15	82602
144	4	11	15	74817
145	5	11	15	72674
146	6	11	15	72286

2.3.2.2 Charts:





2.3.3. Program on Android Device:

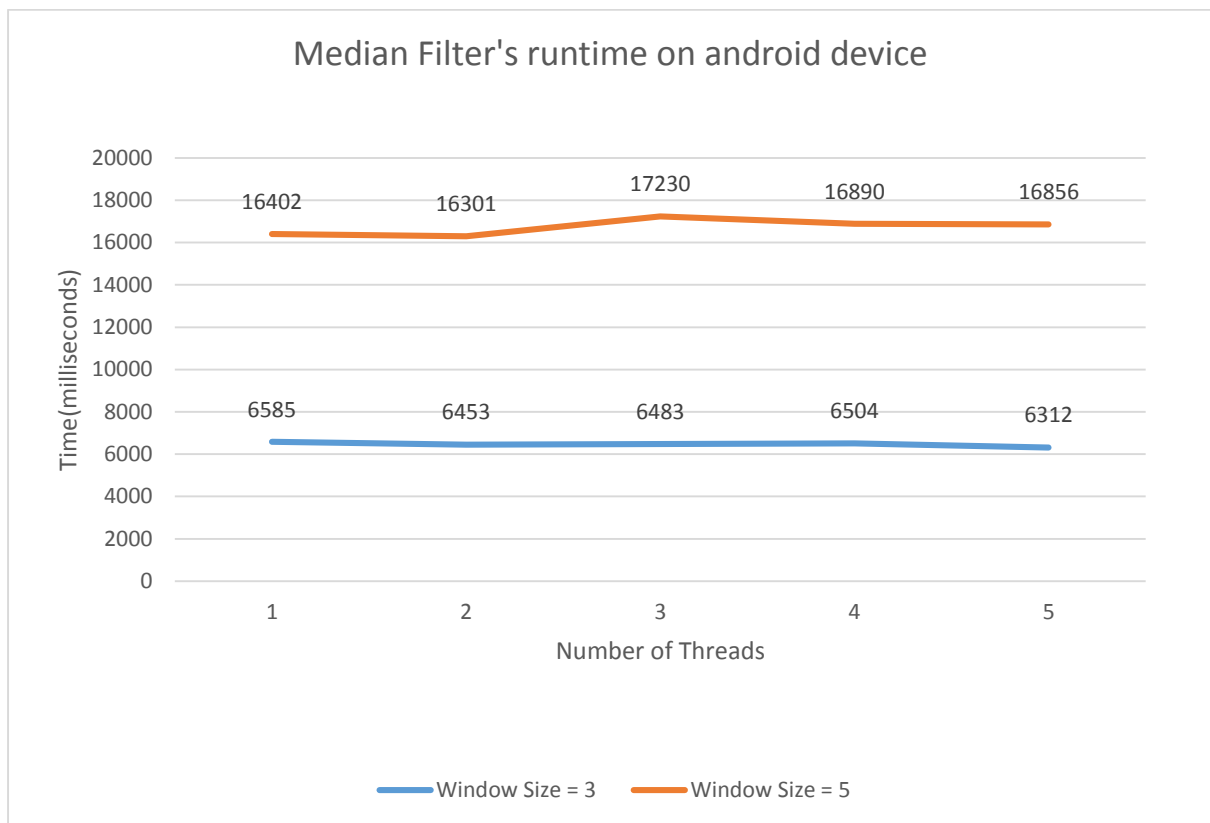
2.3.3.1. Testing System:

Device name	LG - F160L (d1lu)
Android version	4.1.2
API Level	16
Java VM	Dalvik 1.6.0
RAM	1809
SOC	Qualcomm Snapdragon S4 1.51 GHz
Model	LGE MSM8960 D1LKR
Architecture	Krait
Core	2
Process	28 nm
Image	800 * 600

2.3.3.2. Test Cases:

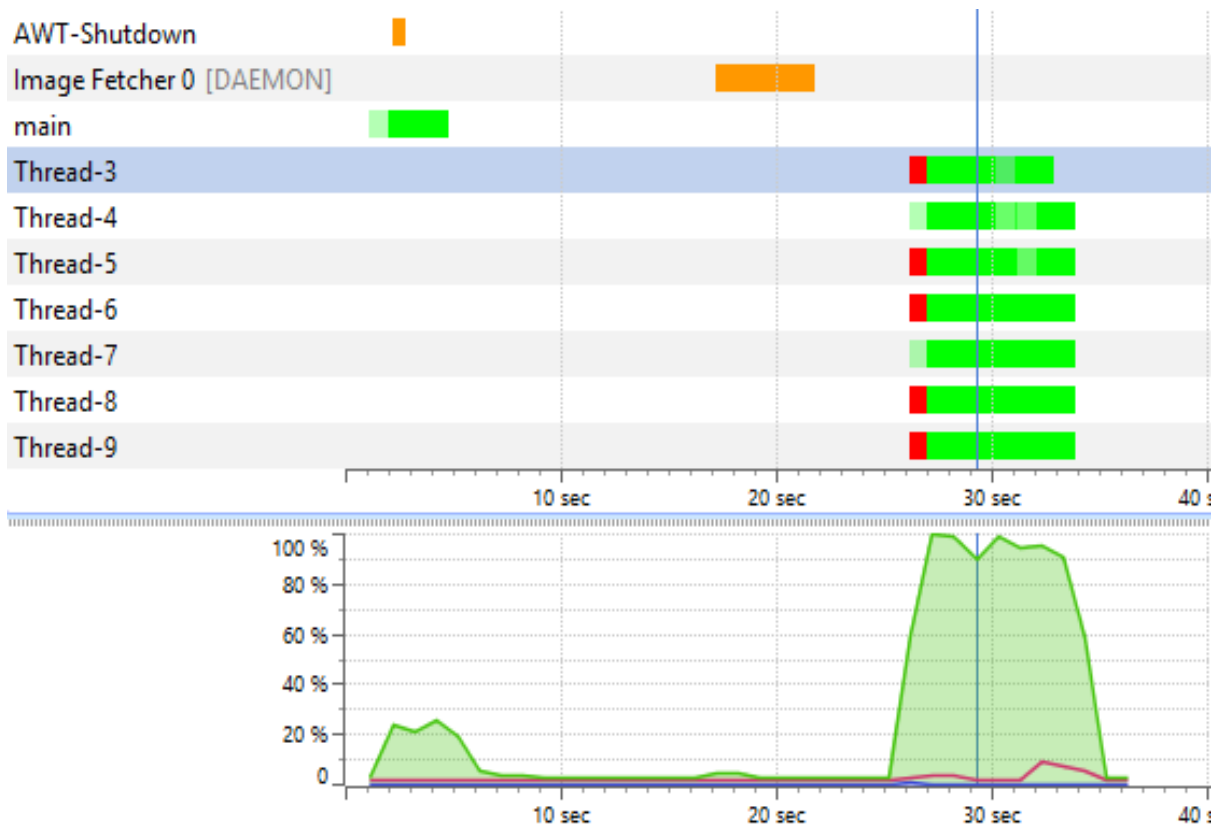
No.	Threads	Runs	Window Size	Runtime
1	1	1	3	6585
2	2	1	3	6453
3	3	1	3	6483
4	4	1	3	6504
5	5	1	3	6312
6	1	1	5	16402
7	2	1	5	16301
8	3	1	5	17230
9	4	1	5	16890
10	5	1	5	16856

2.3.3.3. Charts:

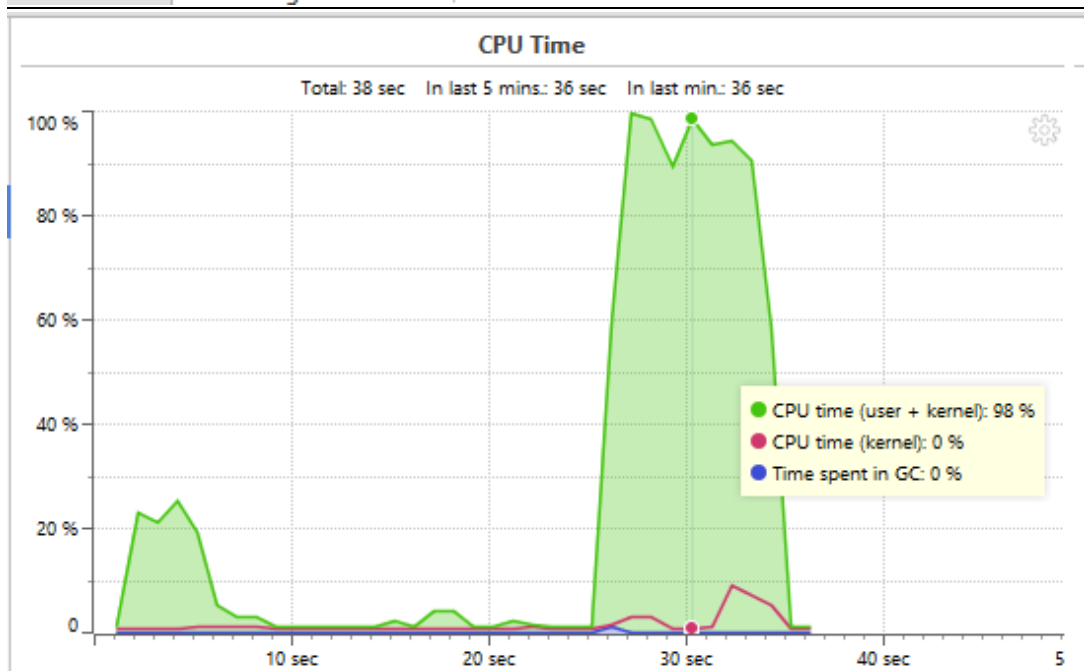


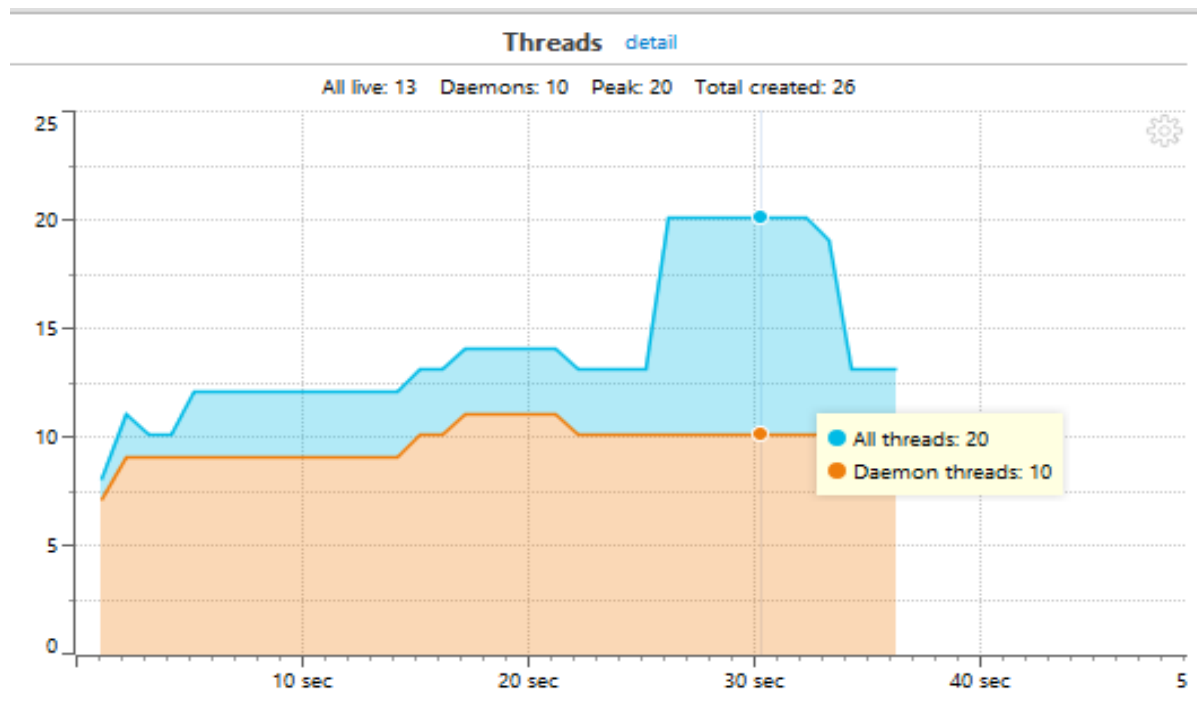
2.3.4. Profiling Information:

- All the profiling Information detail is included in the snapshot file, which can be viewed by the profiling tool: **YourKit Java Profiler 2015 build 15050**.
- Some profiling Information
 - 7 threads with window size is 15

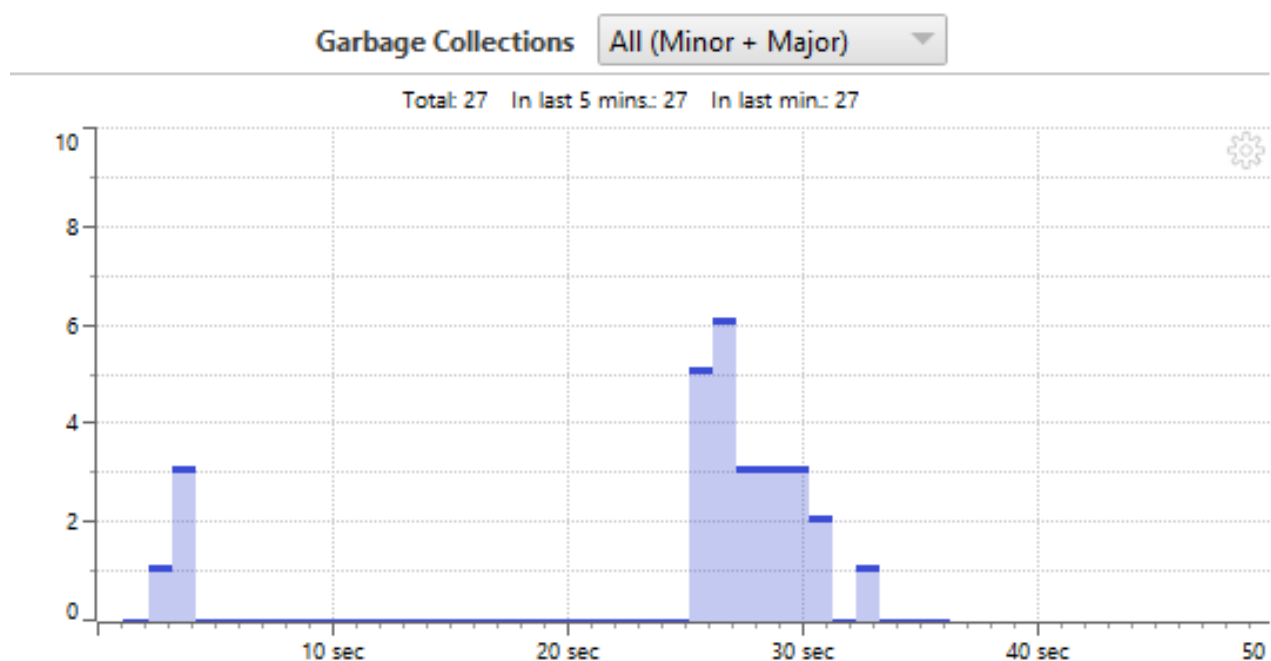
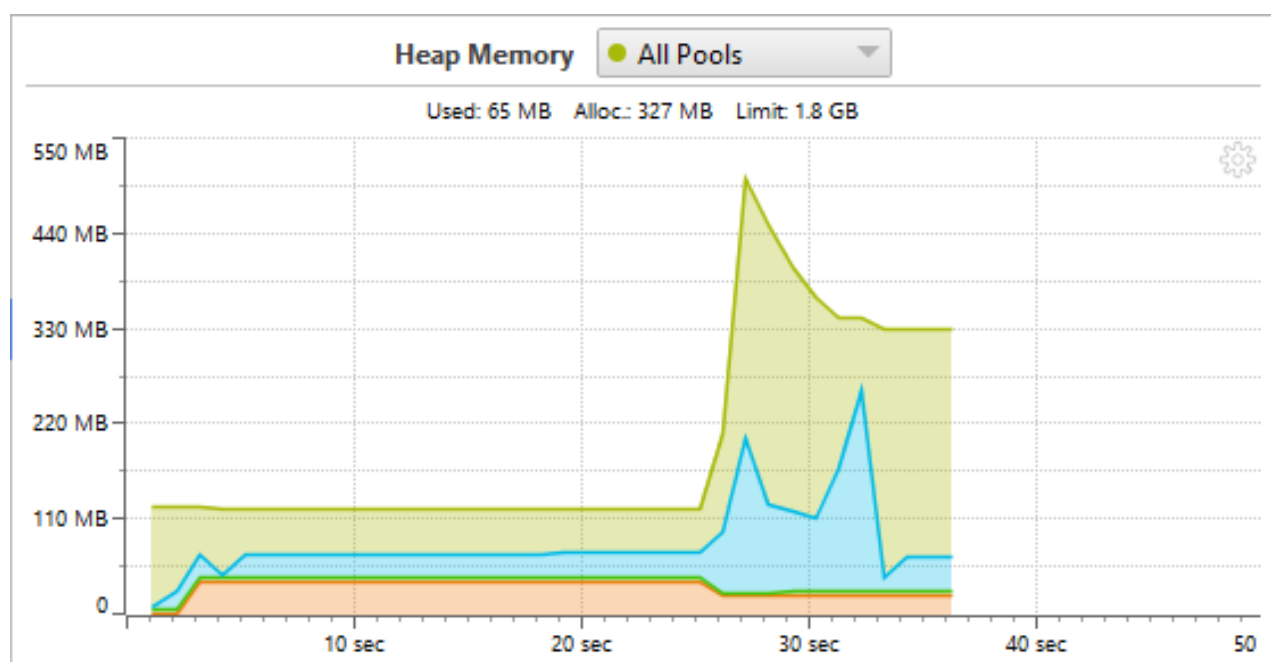


Stack Traces CPU Usage Estimation





- **Thread-3 [RUNNABLE]**
 - ↑ [java.awt.image.BufferedImage.getRGB\(int, int\)](#)
 - ↑ [runnable.FilterRunnable.medianFilter\(BufferedImage, int, int, int\) FilterRunnable.java:108](#)
 - ↑ [runnable.FilterRunnable.run\(\) FilterRunnable.java:65](#)
 - ↑ [java.lang.Thread.run\(\)](#)
- **Thread-4 [RUNNABLE]**
 - ↑ [runnable.FilterRunnable.medianFilter\(BufferedImage, int, int, int\) FilterRunnable.java:121](#)
 - ↑ [runnable.FilterRunnable.run\(\) FilterRunnable.java:65](#)
 - ↑ [java.lang.Thread.run\(\)](#)
- **Thread-5 [RUNNABLE]**
 - ↑ [java.awt.image.BufferedImage.getRGB\(int, int\)](#)
 - ↑ [runnable.FilterRunnable.medianFilter\(BufferedImage, int, int, int\) FilterRunnable.java:108](#)
 - ↑ [runnable.FilterRunnable.run\(\) FilterRunnable.java:65](#)
 - ↑ [java.lang.Thread.run\(\)](#)
- **Thread-6 [RUNNABLE]**
 - ↑ [java.awt.image.BufferedImage.getRGB\(int, int\)](#)
 - ↑ [runnable.FilterRunnable.medianFilter\(BufferedImage, int, int, int\) FilterRunnable.java:108](#)
 - ↑ [runnable.FilterRunnable.run\(\) FilterRunnable.java:65](#)
 - ↑ [java.lang.Thread.run\(\)](#)
- **Thread-7 [RUNNABLE]**
 - ↑ [java.awt.image.BufferedImage.getRGB\(int, int\)](#)
 - ↑ [runnable.FilterRunnable.medianFilter\(BufferedImage, int, int, int\) FilterRunnable.java:108](#)
 - ↑ [runnable.FilterRunnable.run\(\) FilterRunnable.java:65](#)
 - ↑ [java.lang.Thread.run\(\)](#)
- **Thread-8 [RUNNABLE]**
 - ↑ [java.awt.Color.<init>\(int, int, int\)](#)
 - ↑ [runnable.FilterRunnable.medianFilter\(BufferedImage, int, int, int\) FilterRunnable.java:130](#)
 - ↑ [runnable.FilterRunnable.run\(\) FilterRunnable.java:65](#)
 - ↑ [java.lang.Thread.run\(\)](#)
- **Thread-9 [RUNNABLE]**
 - ↑ [java.util.Arrays.sort\(int\[\]\)](#)
 - ↑ [runnable.FilterRunnable.medianFilter\(BufferedImage, int, int, int\) FilterRunnable.java:127](#)
 - ↑ [runnable.FilterRunnable.run\(\) FilterRunnable.java:65](#)
 - ↑ [java.lang.Thread.run\(\)](#)



2.3.5. Summary:



2.3.5.1. Expectation:

- The program run on android will slower than on PC due to:
 - The performance of ARM architect have lower performance than x86 but consume the smaller amount of energy.
 - The CPU on PC testing device is stronger and have more core than in my android testing device.
- The runtime decrease respectively with the number of threads for the program with the number of threads smaller than the number of processors of the system (in this experiment on PC is 4 while on android is 2)
 - Eg: The runtime of program with 1 thread will double the runtime of program with 2 threads. And the runtime will not decreased if the number of threads exceeds the number of core in system.

2.3.5.2. Result:

- **On Android:**
 - The Runtime is the same with all number of threads.
 - The Runtime is much higher than PC
 - There are a few bugs in program on android
- **On PC**
 - The Runtime with 4 threads is lowest.
 - The Runtime with 1 thread is highest.
 - The Runtime decreases sharply from running at 1 thread to running at 2 threads, then it decreases slightly to the lowest point at 4 threads.
 - After that, the Runtime rises slowly when runs at 5 threads.
 - Then, the Runtime fluctuates for the rest.
 - The Runtime increases linearly with the number of runs.
 - The Runtime increases as the window size increases

2.3.5.3. Result Images:

Original Image	
Window size 3x3	

5x5



7x7



9x9



11x11



13x13



15x15



17x17



19x19



21x21



2.3.5.4. Explanation:

- The Program runs fastest with 4 threads because the System has the CPU with 4 cores, so there are maximum 4 threads can run simultaneously. The program run with the higher number of threads is slower because with the number of threads higher than 4, the program has to spend time to create threads, manage and switch between them while with 4 threads, the program does not need to switch between threads. In addition, the program with many threads will consume much amount of memory
- However, I cannot explain the result on android device.

