International Journal of Engineering &Technology, *7 (4) (2018) 6603-6608*

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET
doi: 10.14419/ijet.v7i4.29340
Research paper

Android multi-threading program execution on single and multi-core CPUS with matrix multiplication

Amira B. Sallow *

Department of Computer Science, Nawroz University, Duhok, Iraq
**Corresponding author E-mail: amo_bibo@yahoo.com*

Abstract

Most problems involving complex computations can be solved by implementing them using Chip Multiprocessor (CMP) approach characterized by high speed, high performance for personal computers and mobile devices. In this paper Android multi-threading Program for matrix multiplication executed on single and multi-core CPUs. the use of this technology greatly reduced the time required to execute the code of the matrix multiplication for great size loads.
The main goal of this paper is to compare the single-core technique with CMP approach to execute Android matrix multiplication Program on single and multi-core CPUs and see what limitations in single-core architecture triggered the transition to CMPs, and to know that the use of this technology greatly reduced the time required to execute code of matrix multiplication for great size loads. The results show that the parallel algorithm outperformed the sequential algorithm by an average of speedup equal to 5.2.

Keywords: *Sequential Algorithm; Parallel Algorithm; CMP; Multi-Threading; Multicore; Speedup; Android.*

1. Introduction

The rapid development of the mobile devices industry has culminated with the rise of modern operating systems, specifically optimized to use the advantages and limits of the hardware environment in order to interface with the user. While many mobile operating systems have been developed in the recent years, in today's market, the most widely adopted are Android [1], developed by Google and iOS developed by Apple. Being open-source software, Android has been extended and used by some of the major mobile device manufacturers, being advantageous from the development. [2]
The multicore processor comprises two or more cores or computational/processing units that operate in parallel to read and execute instructions. These multiple processing units or cores are fabricated on a single die. So, it's also called a Chip Multiprocessor (CMP). The key factor about the multicore processor is that it gives the same performance of a single faster processor at lower power dissipation and at a lower clock frequency by handling more tasks or instructions in parallel [3]. Multicore processors work on multiple instructions and multiple data. Multiple cores execute multiple threads (multiple processes/instructions) while using different parts of memory (multiple data). The main memory is shared by all cores. Each core is associated with its own cache and they all share the system bus. [4]
The main goal of this paper is to compare the single-core technique with CMP approach to execute Android matrix multiplication Program on single and multi-core CPUs and see what limitations in single-core architecture triggered the transition to CMPs, and to know that the use of this technology greatly reduced the time required to execute code of the matrix multiplication for great size loads. The rest of this article is organized as follows. section 2, mentions the related work. section 3, describes application execution. section 4, explains structuring applications for performance. section 5, thread basics, section 6, android application threads. section 7, the need for multiprocessing. section 8, the proposed methodology of the proposed system in details. section 4, implementation results. Section 10 illustrates the speed up. section 11 presents the conclusion.

2. Related works

(Guliani & Bagga) in 2017 [5] exhibited an investigation concentrated on multithreaded quicksort and was compared with the sequential quicksort. Each thread is assigned part of the input array after partition method is applied. Similar OS resource and address space were shared by each thread. Multithreading quicksort has illustrated a bigger efficiency over sequential quicksort and the results are validated using various performance qualifications like Maximum Frequency, Idle Time, Processor Utility, Total Execution Time and Processor Time.
The main objective of (Rinku & Asha Rani) in 2017 [6] was to explore the advantages of multi-threading on a multi-core CPU in terms of execution times. The focus was on splitting a single process into multiple code segments (threads). to demonstrate the advantage of multithreading on multi-core CPUs, they executed it on single core ARM Processor, and on quad-core ARM Cortex-A7.
(Singh et al.) in 2017 [7] developed two tools, first, using C# console. application to individually measure the cores' performance of the CPU percentage of load on each core is used as metric of performance is the measurement. While the second tool was made by using windows C# application for plotting the graph with respect to time of CPU load in percentage. The performance is measured by both tools while quicksort was running in the serial and parallel manner for a huge data elements number.

3. Application execution

Android is a multiuser, multitasking system that can run multiple applications at the same time and let the user switch between applications without noticing a significant delay. The Linux kernel handles the multitasking, and application execution is based on Linux processes. [Efficient Android Threading.[8]

3.1. Linux process

Linux assigns every user a unique user ID, basically a number tracked by the OS to keep the users apart. Every user has access to private resources protected by permissions, and no user except root, the superuser can access another user's private resources. Thus, sandboxes are created to isolate users. In Android, every application package has a unique user ID; for example, an application in Android corresponds to a unique user in Linux and cannot access other applications' resources. What Android adds to each process is a runtime execution environment, such as the Dalvik virtual machine, for each instance of an application. Fig (1) shows the relationship between the Linux process model, the virtual machine (VM), and the application.

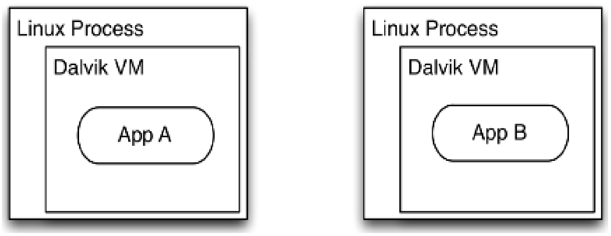


Fig. 1: Applications Execute in Different Processes and VMS.

3.2. Lifecycle

The application lifecycle is encapsulated within its Linux process, which, in Java, maps to the android.app.Application class. The Application object for each app starts when the runtime calls its onCreate() method. Ideally, the app terminates with a call by the runtime to its onTerminate(), but an application cannot rely upon this. The underlying Linux process may have been killed before the runtime had a chance to call onTerminate(). The Application object is the first component to be instantiated in a process and the last to be destroyed.

3.3. Application start

An application is started when one of its components is initiated for execution. Any component can be the entry point for the application, and once the first component is triggered to start, a Linux process is started leading to the following startup sequence:

1) Start Linux process.
2) Create runtime.
3) Create Application instance.
4) Create the entry point component for the application.

Setting up a new Linux process and the runtime is not an instantaneous operation. It can degrade performance and have a noticeable impact on the user experience. Thus, the system tries to shorten the startup time for Android applications by starting a special process called Zygote on system boot. Zygote has the entire set of core libraries preloaded. New application processes are forked from the Zygote process without copying the core libraries, which are shared across all applications.

3.4. Application termination

A process is created at the start of the application and finishes when the system wants to free up resources. Because a user may request an application at any later time, the runtime avoids destroying all its resources until the number of live applications leads to an actual shortage of resources across the system.

4. Structuring applications for performance

Android devices are multiprocessor systems that can run multiple operations simultaneously, but it is up to each application to ensure that operations can be partitioned and executed concurrently to optimize application performance. If the application doesn't enable partitioned operations but prefers to run everything as one long operation, it can exploit only one CPU, leading to suboptimal performance. Unpartitioned operations must run synchronously, whereas partitioned operations can run asynchronously. With asynchronous operations, the system can share the execution among multiple CPUs and therefore increase throughput.
An application with multiple independent tasks should be structured to utilize asynchronous execution. One approach is to split application execution into several processes because those can run concurrently. However, every process allocates memory for its own substantial resources, so the execution of an application in multiple processes will use more memory than an application in one process. Furthermore, starting and communicating between processes is slow, and not an efficient way of achieving asynchronous execution. Multiple processes may still be a valid design, but that decision should be independent of performance. To achieve higher throughput and better performance, an application should utilize multiple threads within each process. [8].

5. Thread basics

Software programming is all about instructing the hardware to perform an action. The instructions are defined by the application code that the CPU processes in an ordered sequence, which is the high-level definition of a thread. From an application perspective, a thread is an execution along a code path of Java statements that are performed sequentially. A code path that is sequentially executed on a thread is referred to as a task, a unit of work that coherently executes on one thread. A thread can either execute one or multiple tasks in sequence. [8].

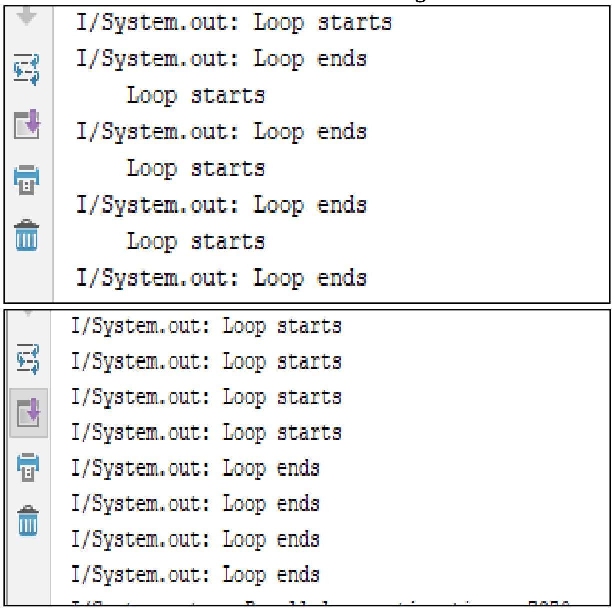
5.1. Single-threaded application

Each application has at least one thread that defines the code path of execution. If no more threads are created, all of the code will be processed along the same code path, and instruction has to wait for all preceding instructions to finish before it can be processed. The single-threaded execution is a simple programming model with deterministic execution order, but most often it is not a sufficient approach because instructions maybe postponed significantly by preceding instructions, even if the latter instruction is not depending on the preceding instructions. For example, a user who presses a button on the device should get immediate visual feedback that the button is pressed; but in a single-threaded environment, the UI event can be delayed until preceding instructions have finished execution, that degrades both performance and responsiveness. To solve this, an application needs to split the execution into multiple code paths (threads).

5.2. Multithreaded Application

With multiple threads, the application code can be split into several code paths so that operations are perceived to be executing concurrently. If the number of executing threads exceeds the number of processors, true concurrency cannot be achieved, but the scheduler switches rapidly between threads to be processed so that every code path is split into execution intervals that are processed in a sequence. Multi-threading is a popular way to improve application execution speeds through parallelism. As each thread has its own independent resource for task execution, multiple processes can be executed parallel by increasing number of threads. Parallelism is the running of threads at the same time on cores of the same CPU. Fig (2A) shows the timing diagram of sequential execution model for executing four printing operations executed each one start after the previous operation ends, Fig (2B) shows code segments (i.e. threads) running concurrently within the "context" of that process the four operations start together and end together. In multithreading environment one thread runs on one CPU core, hence a multithreaded process can be distributed over a series of processors as threads, to scale the performance. [8]

A) Sequential **B) Multi-Threading**



```
I/System.out: Loop starts
I/System.out: Loop ends
    Loop starts
I/System.out: Loop ends
    Loop starts
I/System.out: Loop ends
    Loop starts
I/System.out: Loop ends
```

```
I/System.out: Loop starts
I/System.out: Loop starts
I/System.out: Loop starts
I/System.out: Loop starts
I/System.out: Loop ends
I/System.out: Loop ends
I/System.out: Loop ends
I/System.out: Loop ends
```

Fig. 2: Execution Types

6. Android application threads

All application threads are based on the native pthreads in Linux with a Thread representation in Java, but the platform still assigns special properties to threads that make them differ. From an application perspective, the thread types are UI, binder, and background threads. [9]

6.1. UI thread

The UI thread is started when the application is started and stays alive during the lifetime of the Linux process. The UI thread is the main thread of the application, used for executing Android components and updating the UI elements on the screen. The UI thread is a sequential event handler thread that can execute events sent from any other thread in the platform. The events are handled serially and are queued if the UI thread is occupied with processing a previous event. Any event can be posted to the UI thread, but if events are sent that do not explicitly require the UI thread for execution, the UI-critical events may have to wait in the queue before being processed and before responsiveness is decreased.

6.2. Binder threads

Binder threads are used for communicating between threads in different processes. Each process maintains a set of threads, called a thread pool, that is never terminated or recreated but can run tasks at the request of another thread in the process. These threads handle incoming requests from other processes, including system services, intents, content providers, and services. When needed, a new binder thread will be created to handle the incoming request. In most cases, an application does not have to be concerned about binder threads because the platform normally transforms the requests to use the UI thread first.

6.3. Background threads

All the threads that an application explicitly creates are background threads. This means that they have no predefined purpose, but are empty execution environments waiting to execute any task. The background threads are descendants of the UI thread, so they inherit the UI thread properties, such as its priority. By default, a newly created process doesn't contain any background threads. It is always up to the application itself to create them when needed.

7. The need for multiprocessing

Mobile devices perform a wide variety of tasks such as Web browsing, video playback, mobile gaming, SMS text messaging, and location-based services. Due to the growth in the availability of high-speed mobile and Wi-Fi networks, mobile devices will also be used for various performance-intensive tasks that were previously handled by traditional PCs. The next generation of smartphones called "Superphones" and tablets will be used for a wide variety of tasks such as playback of high definition 1080p videos, Adobe® Flash®-based online gaming, Flash-based streaming high definition videos, visually rich gaming, video editing, simultaneous HD video downloads, encode and uploads, and real-time HD video conferencing. The quality of experience on devices based on single core CPUs rapidly degrades when users run several applications concurrently, or run performance intensive applications such as games, video conferencing, video editing, and more. In order to improve CPU performance, engineers employ several techniques, such as using faster and smaller semiconductor processes, increasing core operating frequency and voltage, using larger cores, and using larger on-die caches.
Increasing the size of the CPU core or cache delivers performance increases only up to a certain level, beyond which thermal and heat dissipation issues make any further increase in core and cache size impractical. From basic semiconductor physics, we know that increasing operating frequency and voltage can exponentially increase the power consumption of semiconductor devices. Even though engineers may be able to squeeze out higher performance by increasing frequency and voltage, the performance increase would drastically reduce battery life. In addition, processors that consume higher power would require larger cooling solutions resulting in an undesired expansion in device size. Therefore, increasing the operating frequency of the processor to meet the ever-increasing performance requirements of mobile applications is not a viable solution for the long run.[10]

8. Proposed methodology

Initially, matrix multiplication has been chosen to test the workload on different cores of CPU. matrix multiplication is executed in serial and parallel with variable size of workloads, on Dual Core, Quad Core, Octa Core of the processor. Result of execution time for each workload is stored in table1. The same implementation of computation of 10000*10000 Matrix multiplication repeated on dual core, quad core and octa core with different size of the workload with multithreading technology. Finally, Results are compared to draw the final conclusion.

8.1. Design and implementation

Many numerical algorithms to check the logic intensive execution matrix multiplication has been used. Various approaches and algorithms have been developed to make the matrix multiplication efficient. Applications of matrix multiplication in computational problems are used in various fields like scientific computing and pattern recognition and in seemingly unrelated problems such as counting the paths through a graph.

8.2. Algorithm for multiplication of matrix

The definition of matrix multiplication is that if C = AB for an x × y matrix A, and y × z matrix B, then C is an x × z matrix. From this, an algorithm can be constructed which loops over the indices i from 1 through x and j from 1 through z, computing the above using a nested loop.

```
Algorithm 1: Matrix Multiplication
Input: matrices A and B
Output: matrix C
1:      Let C be a new matrix of the appropriate size
2:      For i from 1 to x:
3:      For j from 1 to z:
4:      Let sum = 0
5:      For k from 1 to y:
6:      Set sum ← sum + Aik × Bkj
7:      Set Cij ← sum
8:      Return C
```

This is an iterative algorithm which is suitable to check the performance of multi-threading implementation.

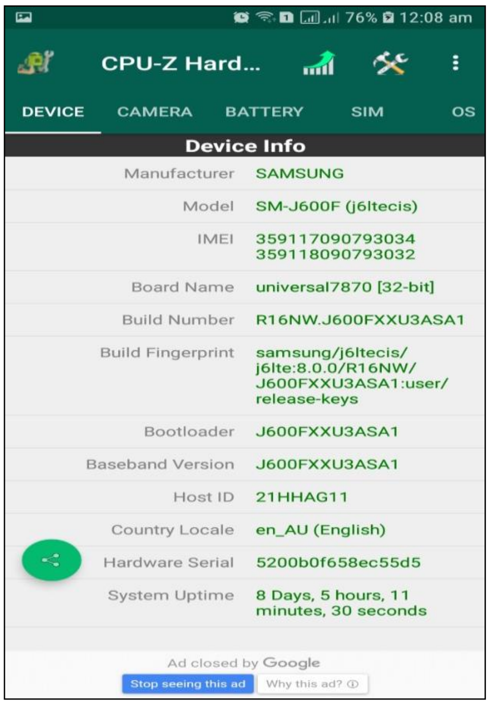
8.3. Hardware platform

The above design was implemented on SAMSUNG smartphone with the model (SM-J600F (j6ltecis)), CPU type (Octa-Core), operating system Oreo version (8.0.0), SDK (API) 26 and Dalvik VM version (2.1.0). All this device information shown in Fig (3) was captured using Android CPU-Z Hardware Info application version (1.0.7.).

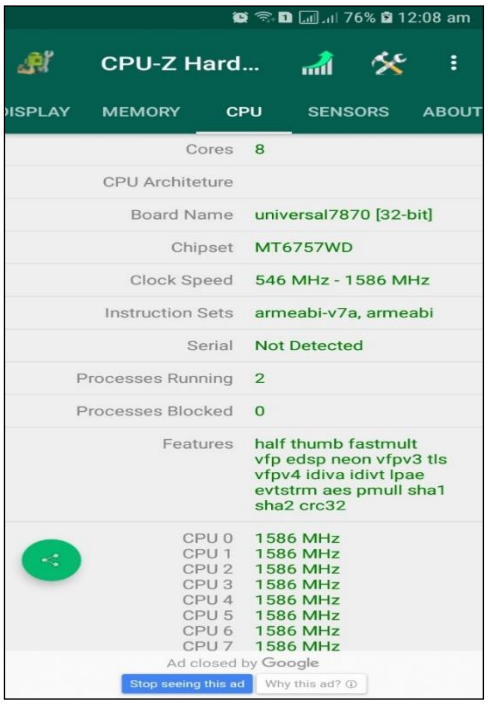
B) Device Info

B)

CPU Info



Manufacturer	SAMSUNG
Model	SM-J600F (j6ltecis)
IMEI	359117090793034 359118090793032
Board Name	universal7870 [32-bit]
Build Number	R16NW.J600FXXU3ASA1
Build Fingerprint	samsung/j6ltecis/ j6lte:8.0.0/R16NW/ J600FXXU3ASA1:user/ release-keys
Bootloader	J600FXXU3ASA1
Baseband Version	J600FXXU3ASA1
Host ID	21HHAG11
Country Locale	en_AU (English)
Hardware Serial	5200b0f658ec55d5
System Uptime	8 Days, 5 hours, 11 minutes, 30 seconds



Cores	8
CPU Architeture	
Board Name	universal7870 [32-bit]
Chipset	MT6757WD
Clock Speed	546 MHz - 1586 MHz
Instruction Sets	armeabi-v7a, armeabi
Serial	Not Detected
Processes Running	2
Processes Blocked	0
Features	half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt lpae evtstrm aes pmull sha1 sha2 crc32
CPU 0	1586 MHz
CPU 1	1586 MHz
CPU 2	1586 MHz
CPU 3	1586 MHz
CPU 4	1586 MHz
CPU 5	1586 MHz
CPU 6	1586 MHz
CPU 7	1586 MHz

Fig. 3: Samsung CPU-Z Hardware Info.

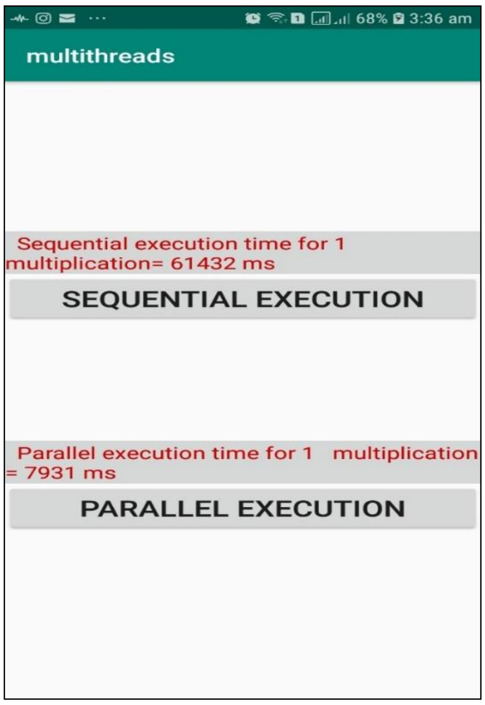
8.4. Software implementation

Software Implementation has been carried out using Android studio version (3.2.1). Fig (4) show the general view of the proposed application views which consists of two main parts. The first one is related to the multithreads main view, that the user can execute the sequential and execute the parallel proposed algorithm and show the execution time of each algorithm. The second part illustrates the CPU cores information and shows all the processor cores and their features.

A) Main View

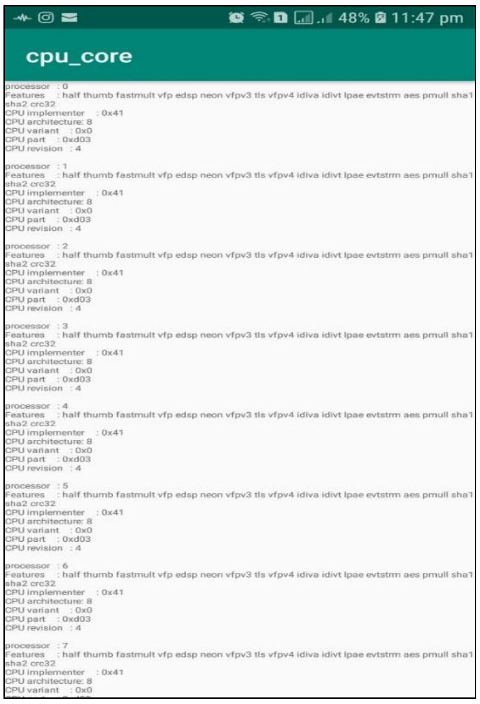
B)

CPU Core View



Sequential execution time for 1 multiplication= 61432 ms

Parallel execution time for 1 multiplication = 7931 ms



```
processor   : 0
Features    : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt lpae evtstrm aes pmull sha1
sha2 crc32
CPU implementer    : 0x41
CPU architecture: 8
CPU variant    : 0x0
CPU part    : 0xd03
CPU revision    : 4

processor   : 1
Features    : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt lpae evtstrm aes pmull sha1
sha2 crc32
CPU implementer    : 0x41
CPU architecture: 8
CPU variant    : 0x0
CPU part    : 0xd03
CPU revision    : 4

processor   : 2
Features    : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt lpae evtstrm aes pmull sha1
sha2 crc32
CPU implementer    : 0x41
CPU architecture: 8
CPU variant    : 0x0
CPU part    : 0xd03
CPU revision    : 4

processor   : 3
Features    : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt lpae evtstrm aes pmull sha1
sha2 crc32
CPU implementer    : 0x41
CPU architecture: 8
CPU variant    : 0x0
CPU part    : 0xd03
CPU revision    : 4

processor   : 4
Features    : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt lpae evtstrm aes pmull sha1
sha2 crc32
CPU implementer    : 0x41
CPU architecture: 8
CPU variant    : 0x0
CPU part    : 0xd03
CPU revision    : 4

processor   : 5
Features    : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt lpae evtstrm aes pmull sha1
sha2 crc32
CPU implementer    : 0x41
CPU architecture: 8
CPU variant    : 0x0
CPU part    : 0xd03
CPU revision    : 4

processor   : 6
Features    : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt lpae evtstrm aes pmull sha1
sha2 crc32
CPU implementer    : 0x41
CPU architecture: 8
CPU variant    : 0x0
CPU part    : 0xd03
CPU revision    : 4

processor   : 7
Features    : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt lpae evtstrm aes pmull sha1
sha2 crc32
CPU implementer    : 0x41
CPU architecture: 8
CPU variant    : 0x0
```

Fig. 4: Samsung CPU-Z Hardware Info.

8.5. Implementation results

This section performs the analysis of the proposed system on one core(sequential), dual core, quad core and octa core CPU. Here, two types of analysis are performed; execution time and speedup. matrix multiplication serial and parallel version are used for the analysis. All the experiment is done using Android smartphone.
The results delineated in TABLE (1) show that the parallel algorithm outperformed the sequential algorithm by an average of speedup equal to 5.2. In the beginning, when operands were respectively sequential matrix multiplication of size (10000x10000) and parallel matrix multiplication of size (10000x10000), the difference was not that evident. However, when numbers became larger, the gap increased and the execution time was speeded up by around 7.7. Fig (5) shows that Sequential execution takes time more than Parallel matrix multiplication.

Table 1: Time Results of Experimental Environment

Load	CPU Time		Speed up
	Sequential	Parallel	
One multiplication	9100	7494	1.2
Two multiplication	16090	7125	2.3
four multiplication	31841	7188	4.4
Eight multiplication	61432	7931	7.7
Ten multiplication	70892	13107	5.4
Twelve multiplication	87601	11547	7.6
Fourteen multiplication	103585	15745	6.6
Sixteen multiplication	115115	17674	6.5

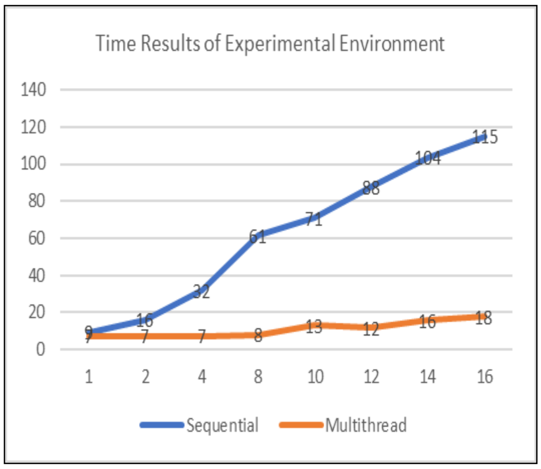
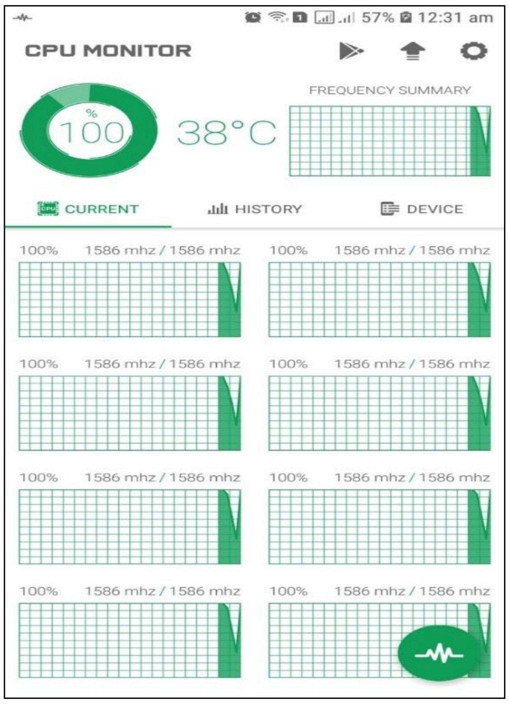


Fig. 5: Comparison Between Sequential and Parallel Matrix Multiplication.

The CPU usage charts are shown in Fig (7) when Sequential and Parallel computation were running were captured using Android System Monitor-CPU-Ram Booster, Battery Saver application version (6.7.5).

A) Sequential CPU Performance **B) Parallel CPU Performance**



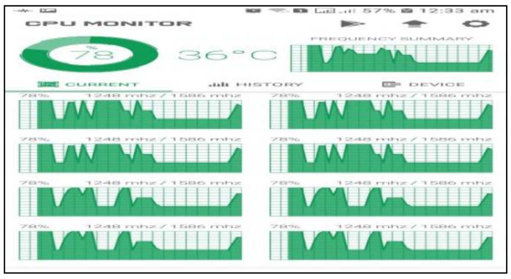


Fig. 6: Android System Monitor-CPU Application.

8.6. Speedup

The speedup of code explains how much performance gain is achieved by running our program in parallel on multiple processors. The meaning is that the time the program takes to run on a single processor, divided by the time the program takes to run on multiple processors. The value of speedup is between 0 and p, where p is the number of processors. The speedup is defined by the following formula from [11]

$$Sp=Ts/Tp \qquad (1)$$

Where:

- T_s is the execution time of the sequential Program.
- T_p is the execution time of the parallel program with p processors

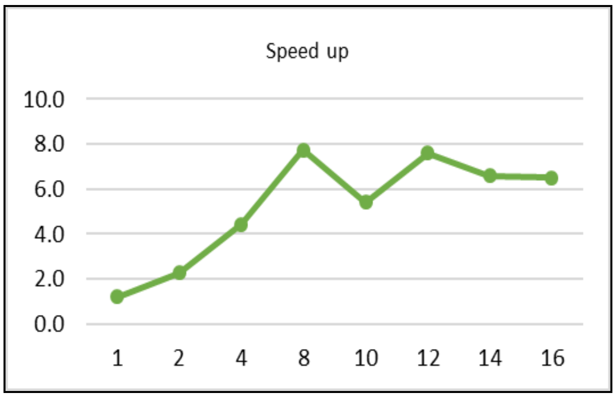


Fig. 7: Speedup Ratio by Using Parallel Matrix Multiplication.

Fig (7) showing the speed up ratio for different workload size, as can be seen here, at low workload size no speedup is achieved. After the workload size increases enough, the overall speed and speed gain increase as well by about 7.7.

9. Conclusion

Through this paper, it has been demonstrated that the use of multi-core processor with multithreading can improve the execution time of compute intensive processes. Using the example of matrix multiplication shown that parallel matrix multiplication utilizes CPU cores compared to its sequential version. On dual-core processor speedup achieved is 2.3. In case of quad-core CPU, Cores speedup achieved is 4.4, while on octa-core speedup achieved is 7.7, the average speed up was 5.2. The parallel version of matrix multiplication better utilizes the CPU cores in all the cases on a dual-core processor, quad-core processor, and on an octa-core processor. The CPU utilization is not directly proportional to the number of cores in parallel programming, because of the following factors parallelism overhead, thread creation time, time spent at synchronization, the granularity of task decomposition, etc. The result also shows that the octa-core CPU provides better result to dual, quad-core CPU on different workload size of inputs.

References

[1] Omar Ahmed; and Amira Sallow, “Android Security: A Review,” Acad. J. Nawroz Univ., vol. 6, no. 3, pp. 135–140, 2017. https://doi.org/10.25007/ajnu.v6n3a99.

[2] R. Gyorödi, D. Zmaranda, V. Georgian, and C. Gyorödi, “A Comparative Study between Applications Developed for Android and iOS,” Int. J. Adv. Comput. Sci. Appl., vol. 8, no. 11, pp. 176–182, 2017. https://doi.org/10.14569/IJACSA.2017.081123.

[3] Anil Sethi; Himanshu Kushwah, “Multicore Processor Technology-Advantages and Challenges,” Int. J. Res. Eng. Technol., vol. 04, no. 09, pp. 87–89, 2015. https://doi.org/10.15623/ijret.2015.0409015.

[4] B. Ahsan; O. Fatma; and Z. Mohamed, “Chip Multiprocessor: Challenges and Opportunities Bushra Ahsan ElectricalEngineering Department of Computer Science City University of New York Department of Computer Science School of Computers and Information,” pp. 54–65, 2008.

[5] G. S. Guliani and R. Bagga, “Time sharing based multithreading approach to Quicksort,” 3rd IEEE Int. Conf., pp. 3–10, 2017. https://doi.org/10.1109/CIACT.2017.7977314.

[6] D. R. Rinku and M. Asha Rani, “Analysis of multi-threading time metric on single and multi-core CPUs with Matrix Multiplication,” Proc. 3rd IEEE Int. Conf. Adv. Electr. Electron. Information, Commun. Bioinformatics, AEEICB 2017, pp. 152–155, 2017. https://doi.org/10.1109/AEEICB.2017.7972402.

[7] T. Singh, D. K. Srivastava, and A. Aggarwal, “A novel approach for CPU utilization on a multicore paradigm using parallel quick-sort,” 3rd IEEE Int. Conf., pp. 1–6, 2017. https://doi.org/10.1109/CIACT.2017.7977382.

[8] A. Goransson, Efficient Android Threading: Asynchronous Processing Techniques for Android Applications, vol. 6, no. 2. 2014.

[9] Hawkar Shaikha; and Amira Sallow, “Mobile Cloud Computing: A Review,” Acad. J. Nawroz Univ., vol. 6, no. 3, pp. 129–134, 2017. https://doi.org/10.25007/ajnu.v6n3a96.

[10] Nvidia, “The Benefits of Multiple CPU Cores in Mobile Devices,” Nvidia White Pap., pp. 1–23, 2010.

[11] S. R. M. Zeebaree, “Design and simulation of High-Speed Parallel / Sequential Simplified DES code breaking based on FPGA,” 2019. https://doi.org/10.1109/ICOASE.2019.8723792.