Ατομική Διπλωματική Εργασία

ARM MALI T624 GPU REVERSE ENGINEERING TO FACILITATE POWER VIRUS GENERATION

Μάριος Τσόκκος

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2022

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ARM MALI T624 GPU REVERSE ENGINEERING TO FACILITATE POWER VIRUS GENERATION

Marios Tsokkos

Επιβλέπων Καθηγητής

Γιάννος Σαζεΐδης

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2022

Acknowledgments

I want to say special thanks to Professor Mr. Yiannos Sazeidis who has suggested me this topic and has helped me to explore the graphics card in depth and we have been in touch all year to produce this final result.

It is also an omission not to mention Mrs. Georgia Antoniou who is a PhD student in the laboratory of Mr. Yiannos Sazeidis and has been helping me all this time, especially in the early stages when I was not familiar with the Junoboard r2. Furthermore she was constantly helping with all the difficulties we were facing with the performance modeling process of the GPU.

I would also like to thank again Mr. Yiannos and Mrs. Georgia because they were attending meetings with me at least one hour every week to check, to discuss the results of the experiments on the GPU.

Summary

Stress-tests that maximize the microarchitectural activity are extremely important as they affect the stability and the performance of a system. The aim of my dissertation is to build a strong knowledge on the parallelism and how the system handles the parallelism. I am doing this by reverse engineer certain features of the ARM Mali T-624 GPU in order to facilitate the creation of stress tests for the GPU in the future.

I achieve this by executing simple kernels written in OpenCL[3] on real hardware. During the execution of my kernels, I collect metrics such as execution time and power consumption. By carefully constructing the kernels and correlating their structure with their execution time and power consumption, I am able to reveal a number of different characteristics of the GPU.

The insights of this work can be used to create or to accelerate the creation of stress tests for the ARM Mali T-624 GPU. This is due to the fact that the thesis indirectly shows what are the units that can cause significant increase in the activity of GPU in terms of power consumption (power virus - stress test) and also performance (IPC virus - stress test).

There have been many attempts to generate viruses and stress-tests on real hardware such as CPUs (even ARM) and multiple for NVIDIA graphics cards but nothing like this has ever been done for ARM Mali graphics cards

Contents

Chapter 1

Introduction

1.1 Background Problem

Thermal Design Point or TDP is the power consumption of a processor under maximum load. It is consider one major design parameter for various reasons including green computing. The cooling system of processors is designed so that the system operates safely only under this “upper” limit[7].

The procedure of determining the TDP is very complex because it has to be representative[7]. The manufacturers cannot just use the sum of the maximum power consumption achieved by each component of the microarchitecture because there is no workload that can cause maximum activity to all microarchitectural components simultaneously. Also, by over-provisioning the power of a design, designers will waste resources on big heat-sinks, cooling systems and power delivery networks that could be used instead for computational reasons, like including extra cores to the design.[7]

To choose representative limits, hardware vendors use stress tests (or power stress tests in the case above). Stress tests are small programs that maximize microarchitectural activity such as power consumption, voltage noise and IPC. We try to facilitate the representative creation of stress tests for the GPU ARM Mali T-624 by revealing different undocumented characteristics of its architecture.

1.2 Contributions

Some of the main achievements that I have achieved Some of the things I managed to discover or validate by having taken into account the concept of parallelism and its management combined with the power consumption in the Mali T-624 are described below in figure 1.1.

Features Validated	Features Discovered
Number of Cores inside GPU	Addition Units
Number of arithmetic pipelines	Workload Distribution among the cores
Thread switching mechanism validation(Fine-Grained)	Arithmetic Pipeline Depth
	Register File Size – Register spilling

Figure 1.1 Validated and Discovered features

1.3 Outline

To begin with we are going through a brief look on the OPENCL framework(chapter 2) that we will use. After in the next chapter we are going to mention some important things about the Graphics cards on the architectural side and how they manage to achieve multi-way parallelism. To continue with in the chapter 4 I will then list various metrics that will be critical in the next steps such as power and time execution. Moreover next step in the chapter 5 is to list various publicly known or unknown features and which of them we are going to validate or discover. Chapter 6 will be all about how I setup the hardware and how I manage to take the measurements of power, execution time and our approach for analysis. Later on chapter 7 we are examining the available cores, also I will refer to the arithmetic pipelines and their depth and the job scheduling. In addition I am going to discover more about the addition units(vector & scalar) inside an arithmetic pipeline. Furthermore will be related work in that specific field(chapter 9) and in chapter 10 I will conclude the observations and the future work of this thesis. Last thing presented on this dissertation is the appendix A where is an examination of register file.

Chapter 2

OpenCL

2.1 OpenCL terminology

In this section we will cover some basic terms about OPENCL[3].

OpenCL: is widely used for parallel programming in both CPU and GPU processors. It is a perfect way to make portable code across different platforms.

Device: compute units that can execute OPENCL code e.g. a GPU or a CPU is a single compute unit. OpenCL devices typically correspond to a GPU.

Host: Responsible for sending data to device and important parameters for the execution to the device.

Kernel: function using OpenCL language. Can be compiled and executed on any supported devices. Kernel are always starting with the __kernel keyword.

Compute Unit: Can have multiple cores so the device can have single or multiple processing units. Work-group is something similar to a block and can be processed within one processing unit.

Work-group: group of work-items which are going to be executed on one available processing unit.

Work-item: is basically a thread equivalent. Work-items are part of a work-group. A group of work-items are executing the same kernel code but processing data for each one should be different. Also it can be identified by the global ID among every existing work-item or its local ID inside a workgroup.

2.2 OpenCL execution model

We use the OpenCL framework in order to be able to take advantage of the parallel processing of a GPU. Developers can make code that is optimized for each processing unit for any kind of platforms.

The OpenCL model is divided into two parts, the host and the devices. The host is connected and can communicate with a single or many devices. Most common scenario for parallel program is that the host is the CPU and the device is the GPU that can handle parallel code much faster and efficient. The GPU can be considered as a series of multiple compute units (CUs).

There are two things that every OpenCL program has: A kernel that is executed by the device/s and a host program that is managed by the host and is responsible for the context and the execution of the kernels. Work-items(threads) are instances of a kernel. Work–items are organized into groups called work-groups[3]. They can be identified locally by their parent workgroup or by a globally unique identifier(figure 2.1). The NDRange(figure 2.2) in the OpenCL can be 1D, 2D or 3D but in this research we are focusing on1D. The number of work-groups and their containg work-items must be declared by the programmer based on the NDRange chosen[3].

As reported by the OpenCL manual each thread or work-item can access 4 types of memory spaces[3]:

1. Global Memory: can be accessed by every work-item regardless of the work-group that belong to. Read/Write accesses may be cached.
2. Constant Memory: It is part of the global region but everything that is stored remains constant. It is host's responsibility to insert memory objects in this region.
3. Local Memory: can be accessed only inside a single workgroup. Mainly used for shared variables among the work-items. Sometimes is divided to small sections of the Global memory.
4. Private Memory: has the scope-range of a single work-item. So each work-item has its own variables that are hidden from the others.

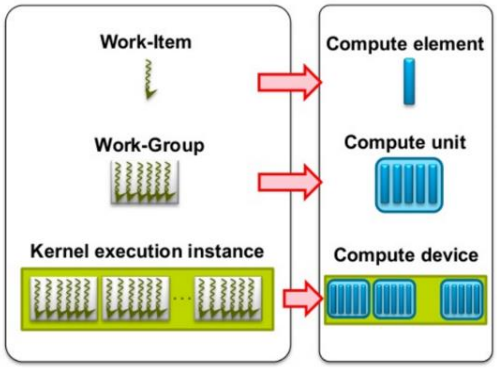
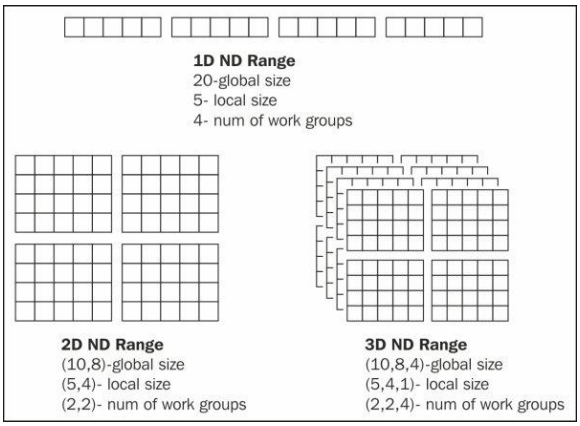


Figure (2.1) This figure visualizes the work-items and the existance of multiple work-items inside a workgroup



Figure(2.2) The figures shows the supported ND Range in OPENCL.

Depends on the programmer which one is going to use(depends on the problem)

2.3 OpenCL synchronization

The main way to achieve synchronization for the work-items that belong in the same parent work-group is a work-group barrier. If a barrier is present in the code all the work-items that are part of the same work-group must execute all the code that is above the barrier and later they will be able to continue with the execution of the rest of the code. Unfortunately OpenCL doesn't have anything to synchronize the races between work-groups[3]. Example of barrier can be found in figure 2.3

For the Command-queue there is also a certain barrier which is called Command-queue barrier. The command-queue barrier makes sure that the previously queued commands on the kernel have being executed and every single change on a memory object are being forwarded on any next enqueued commands.

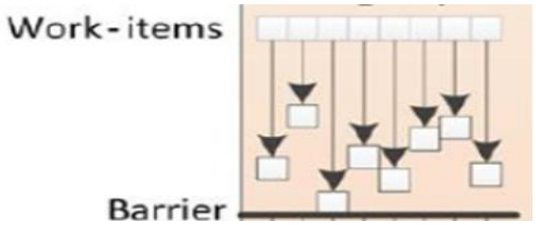


Figure2.3 Multiple work-items that are waiting until the all threads finish the execution

2.4 OpenCL Data Types

OpenCL supports scalar as well as vector types. Scalar data types include char, bool, short, int, long, float, double, half. Vector data types are the same as scalar ones except that they don't support the bool data type. Additionally, the vector types are all followed by a literal n where values of n can be: 2, 3, 4, 8, and 16 (figure 2.4) for any vector type[3]. The value of n shows the number of elements that are part of the vector. Elements inside a vector can be accessed individually with the name of the VectorName.sN where N is the position that we want to modify/access. E.g. float4 x=(float4)(1.01f, 21.0f, 31.0f, 4.10f); the programmer declares a 128 bit vector with 4x32 bit float type numbers as can be seen in figure 2.5 and performs an addition. Each iteration it adds the value of vector b in the corresponding position in vector a.

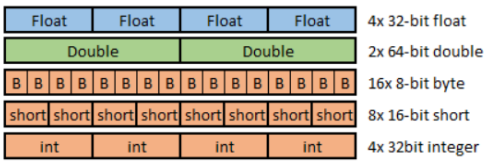


Figure 2.4 shows how different data types are existing inside a 128bit vector

```
__kernel void simple_vector_kernel(__global  int * list){

float4 a=(float4)(1.01f, 21.0f, 31.01f, 4.10f);
float4 b=(float4)(1.02f, 25.20f, 31.02f, 41.0f);

for(int j=100000000; j!=0; j--){
a=a+b;
}
```
Figure 2.5 Kernel that uses Vector data types and performs add operation between vectors

Chapter 3

GPU Architecture

3.1 GPU Definition

GPU which stands for Graphics Processing unit is an essential part of a system, and it is also known as an “accelerator”. GPUs are designed with the aim to support the parallel processing of data. Their architectural characteristics (multi-level parallelism), allows them to achieve greater parallelism than CPUs and therefore surpass their performance when processing large non-sequential problems. GPUs are multithreaded meaning that they support the creation and execution of threads (sequence of code that can be executed in parallel). In general GPUs are in need of higher memory bandwidth and larger register file size due to the fact that they execute many threads at a time and produce more result than the CPU’s do.

Today GPUs find applications in graphics, video rendering, gaming, machine learning model training etc. They can be found in any type of device from mobile devices and desktops, to servers and supercomputers.

3.2 Core Level Parallelism

GPU’s most important hardware specification is the fact that they can handle embarrassingly parallel code much more better than a modern CPU. The total amount of

work is often split to multiple threads which are part of a workgroup. Work-group is a term that refers to a group of threads sometimes 16, 32 or 64 that are going to be executed together among the available cores of the GPU. Multiple threads running at the same time sometimes share some hardware resources. To be more specific the threads can be executed in different cores in parallel and that offers extra a level of parallelism which is called core level parralelism. In order to achieve core level parallelism the programmer must find a way to distribute the problem into multiple cores and make each thread as much independent from the others as possible. This is achieved by dividing the problem into subproblems that can be solved simultaneously-indepedently using threads. Threads can be distributed more easily and fairly among cores. Generally speaking Core Level Parallelism is the parallelism offered by the GPU in the form of multiple cores.

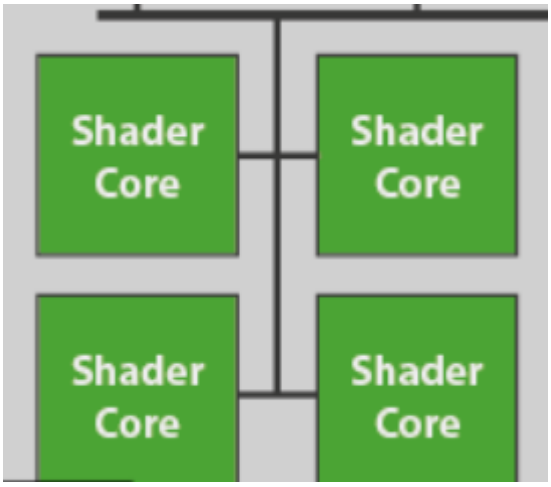


Figure 3.1 This is an example of multiple cores connected together

3.3 Pipeline Parallelism

Pipeline is a structure that organizes the execution part of the instructions. For example during the lifetime of an execution of an instruction it has to go through different stages. This stages include fetching, decoding, executing, memory and write back. In a non-pipelined system an instruction has to pass through all the stages in order for the next one to start executing. This means that an instruction has to pass throughout all the stages in order for the next one to start executing. For example if the execution time of the stages is 5s the throughput will be 0.2 instruction/second. Instead we divide the

execution lifetime into stages. Allowing based on the assumption above the execution of 5 instructions simultaneously. For example we have 5 stages, each stage has execution time of 1 second. If the pipeline is full we have throughput 1 because each cycles one instruction is being executed. Each core has some different types of pipelines such as arithmetic pipeline, instruction pipelines. Although there are some type of hazards and data dependencies like Read After Writes(RAW) that make the execution inside a pipeline much slower sometimes known as Stalls.

There are different kinds of pipelines that differ in the way the threads that are ready to be executed are treated. First of all there are different policies for fetching instructions in “in order” policy instructions are being fetched in the program order. On the other hand in the “out of order” execution the instructions are dynamically scheduled and the instructions are being fetched in the sequence that the compiler generates.

Moreover there are different kinds of pipelines policies–mechanisms that differ in the thread switching inside a pipeline .The ones that are commonly used in modern GPU’s are fine grained: which allows only a single instruction per thread inside the pipeline at a time and allows interleave execution in order to be able to hide stalls – latencies. Also inside a Pipeline there are many pipeline stages which can be used from different threads in order to increase the parallelism. To make this more specific multiple instructions from different threads can be executed together with the number of threads inside the pipeline being limited by the number of pipeline stages inside a core. Each pipeline has many stages that are used for different purposes. In fine-grained multithreading each clock cycle that passes the thread that contains the instruction is transferred to some other stage later in the hierarchy of pipeline while for each one that is being fully executed and has passed all the stages of the pipeline another new one enters the pipeline and starts from the very first stage. While at the same time those that are already in the pipeline and have not been finished are shifted on to the next stage. So if we have a thread which executes only two instructions, the pipeline depth is 5 (each stage needs only 1 cycle) and the pipeline uses fine-grained multithreading then each instruction needs exactly 5 cycles because the execution of the instructions for a certain thread can not be interleaved inside the pipeline(only one instruction per thread can be found at a time inside a pipeline) so 10 cycles in total for this thread to be executed. So

if another same thread is assigned to be executed with the same amount of instructions then both threads can have one of their instruction at the same time inside the pipeline. Then both threads need one more cycle to be executed because the second thread will finish in the next cycle right after the first thread finishes(11 cycles in total) the execution. In the scenario that we have multiple identical threads(more than the pipeline depth) with one instructions each then the total cycles needed for all of those is NumberOfThreads+pipelineDepth.

On the opposite side Coarse grained multithreading is a multithreading "policy" in which the thread switching is visible when thread that executes inside the pipeline is suddenly triggering stall cycles. So the point of pipeline is that it significantly improves pipeline utilization by taking advantage of multiple instruction and/or multiple threads.

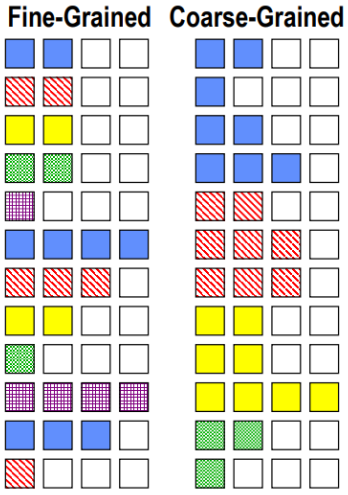


Figure 3.2 This figure shows 2 thread switching mechanisms Fine-Grained vs Coarse Grained thread with pipeline depth 4

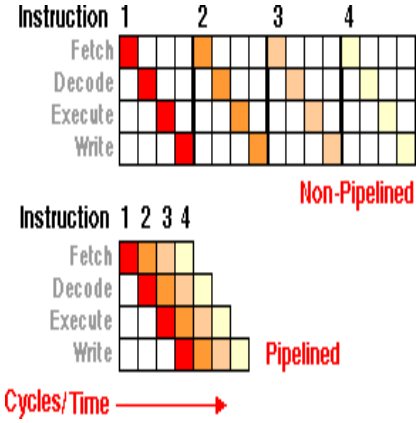


Figure 3.3 This figure shows a non-pipelined and a pipelined processor with 4 stages

3.4 Instruction Level Parallelism (VLIW instructions)

Instruction level parallelism is when we are able to execute multiple instructions at the same time. There are two techniques to achieve instruction level parallelism.

First way to achieve this is Very Long Instruction Word also known as VLIW instructions can have a wider instruction "format" in order to be able to issue many instructions and fit up to four instructions in one[8]. VLIW uses very long instructions in order to merge instructions that can be executed at the same time on different units

into one. This merging can only happen if the hardware and software dependencies allow it. Another key thing to remember is the that in this architecture instruction latencies are predetermined and there we need more registers[8]. Also VLIW architectures are in the need of high bandwidth instruction fetch mechanisms to bring the instruction words from the cache to the execution pipeline[8]. Example can be found on figure 3.4.

The other way to achieve instruction level parallelism is the Instruction fusion which is something similar but it merges smaller number of instructions and is using specialized units which can handle the operations of the instructions that are fused.

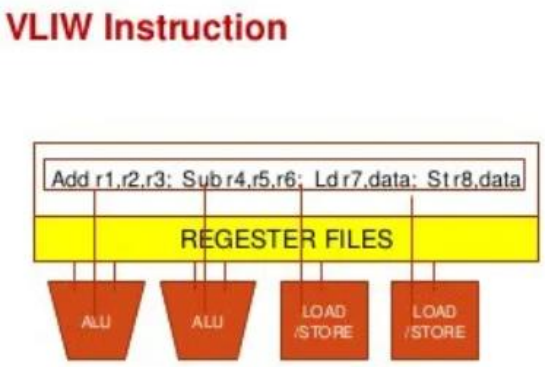


Figure 3.4 An example of VLIW instruction that contains 4 independent instructions those are using different available units

3.5 Data Level parallelism

In this subsection we are going to discuss the data level parallelism techniques that take advantage of vector units. Vector instructions are type of instructions that are performing parallel processing of data sets and store them in vector registers at the same time. There are different types of vector units such as vector addition and multiply units. Each vector can hold up many elements and every single operation inside of the vector does not depend from the others[9]. So this type of instructions are increasing the parallelism and they can be pipelined (overlapped during the execution). Vector type elements can be declared by the programmer or the compiler can make it for the programmer and this process is also called auto-vectorization[3].

3.6 Cache

Cache is widely used in every modern processing unit (e.g. GPU) in order to decrease the energy and the delay to store/load data from DRAM. Moreover cache is significantly smaller in capacity, faster than main memory as is a high-speed static random access memory (SRAM) and is placed close to the processor. Every modern processing unit has many cache levels such as Level 1 or 2 (Multi-Level Hierarchy), some of them are used to store instructions(I-caches) or data(D-cache)[10]. As we go upwards in the hierarchy the corresponding level is smaller but faster than the previous ones.

Everything that moves from DRAM to cache has a certain size and being limited to the size of cache lines. A copied cache line from DRAM that goes into the cache means that a cache entry is created[10]. The entry consists of the data as well as the tag which is the memory location. Processors that request to load or store in the memory are first checking the entries inside the caches then if the requested memory location is part of any cache lines then we have cache hit and the processor reads or writes inside the line instantly.[10] On the side if the memory location is missing from the cache then we have a cache miss and the cache handle this event by allocating a new cache entry and brings the stores the data from the DRAM and the missed request is then served.

Chapter 4

Power and Performance Metrics

4.1 Power Consumption

Power consumption in a modern system is the amount of energy per unit time and the main measurement unit is the Watts. Every processing unit has an idle state which no background processes are running and the power can be described as idle state power consumption. The idle state can change and becomes active if any processes are running. Moreover the Power consumption decreases or increases depending on how many functional units are used to execute a program in a certain time. Power consumption shoots up in a multiple cores CPU/GPU when more cores are activated. More work done (using more processors, functional units) in certain time indicates that we consume more power (Power=Energy/Time).

4.2 Dynamic Power Consumption

Dynamic Power indicates the switching activity of transistors. Dynamic Power equation[13]:

$$P_{dyn} = CV^2Af$$

- First parameter in the PDynamic equation is the Capacitance (C): which must be known from the manufacturer as it states the function of wire length and the transistor size.

- Supply voltage (V): has been improved over years with every new generation of processors .
- Activity factor (A): The activity factor states how often on average do the wires switch from 0 to 1.
- Clock frequency (f): The clock frequency of the Processor has the greatest impact on the power during the evolution of new processors. Clocking in higher frequencies requires a higher supply voltage that why the dynamic power equation has voltage parameter cubic impact.

4.3 Leakage Power Consumption

Leakage or Static Power Consumption :

$$P_{leakage} = NVe^{-et}$$

Indicates the Power that is consumed by the system due to the fact that transistors are not turnoff completely even if the system is idle[13].

Parameters inside the equations:

- N: indicates the number of transistor
- V:Voltage
- Vt: Voltage where the transistors conduct. Higher Value means faster transistor that leak more power.

4.4 Power Consumption Metrics

1. Energy: Measurement unit is joules is often considered the most basic from the metrics
2. Power: is consider to be the rate of energy dissipation in the processor. Measurement unit of the power consumption is watts(joules per second).
3. Energy-per-instruction: Indicates the energy consumed when the system is getting optimized and we want to see side by side the techniques the in the aspect of energy(microarchitecture optimizations)

4.5 Performance Metrics

Another performance metric which is consider fundamental is the time execution: is the time that passes between the start and the end of the execution of a program(depends on the type and number of the instructions that have been executed).it is an important metric because it has a direct correlation with power. Although the time execution is so important the processors are synchronizing everything using their own clock rate. Their clock cycle time is given by :

$$ClockCycleTime = \frac{1}{frequency}$$

Clock cycles for an executed program is the amount of clock cycles needed to passed in order to execute the program and they can be used to calculate the time execution:

$$\mathrm{CPU_{Time}} = \mathrm{ClockCyclesForTheProgram} * \mathrm{Clock\_Cycles\_time}$$

Cycles per instruction: Another useful metric that computes on average how many cycles each instruction needs to be executed:

$$CPI = \frac{NumberOfClockCycles}{InstructionCount}$$

Another way to compute the Execution time is :

$$CPU_{time} = \frac{InstructionCount * CPI}{Frequency}$$

Chapter 5

ARM MALI T-624 GPU (MIDGARD FAMILY)

5.1 ARM MALI T-624 GPU Specifications

Arm Mali T-624 GPU (Midgard family)	
CPU-Host	Arm Cortex A72 and A53
Number of Shader Cores	4
GPU clocked at	600MHz
Number of arithmetic pipelines inside each core	2
Cache L1 for each core	16Kb
Cache L2 shared for all cores	32-256kb
Number of store/load pipelines inside each core	1
Register Size	16B (128bits)
Supported framework	OPENCL 1.1

Figure 5.1 Mali T624 known characteristics

The GPU that we are focusing on this dissertation is the ARM Mali T-624 which is part of the Arm Midgard Family and is located on the ARM Junoboard r2. The GPU that we are discovering and validating certain features as well as the parallelism and how the GPU handles it. In this section we will emphasize on the hardware components and units that we already know as can be seen in figure 5.1 from the manufacturer of the GPU. First of all this GPU was made explicitly for the smartphone market. The reason I mention this is because mobile devices are generally consume low amounts of energy. On the hardware side we have 4 identical cores(which I am going to validate) inside which are also called shader cores by the manufacturer. The shader core clock rate for the Mali is 600 Mhz. Inside each core we can find 2 arithmetic pipelines which we are

going to validate in an upcoming section. Each arithmetic pipeline is containing Scalar and Vector units for both Multiplication and addition as well as a special function unit. To continue with a shader core has also a load/store pipeline as well as a texture pipeline that is mainly used for graphics purposes (we are not going to investigate the texture pipeline is out of the scope of this dissertation).
It is important to mention that this GPU is using the fine-grained multithreading policy to put threads to run inside a pipeline (Section 2.3). Also to increase the instruction level parallelism it uses VLIW in order to execute independent instruction in different available units. For caching purposes the GPU has a Level 1 (L1) cache that is for private usage[12] for each core and it can store data up to 16Kb. Following this there is a Level 2 (L2) cache that is 32-256Kb that is shared among the 4 shader cores. It should be noted that all the cache lines are 64 bytes. Also each register in the GPU can store data exactly up to 128 bits[14].

Some important software and microarchitectural parts shown in figure 5.2 that I have to explain:

1. JOB MANAGER: A part of the GPU which manages the connection with the Mali's driver : 1.Get in contact with the memory and then read the job descriptors 2. Tracking some job's dependencies 3. Assign jobs to the 4 cores in the GPU 4. Divides jobs to per-core tasks [4].

2. MEMORY MANAGER: There is one for all 4 shader cores. It handles the requests from memory Loads/Stores for each thread[4].

3. THREAD POOL: Is a software program that is mainly used to achieve concurrency in an parallel program. A thread pool "queues" the threads that wait for tasks to be allocated for execution[11]. This part of the GPU helps the performance and can handle short living tasks[11]. Also this unit knows the cycles needed for each thread to finish the execution.

4. THREAD RETIRE: This unit is responsible to keep track of the work-items/threads that finish the execution and maybe operates like a barrier that allows synchronization among the work-items.

5. MALI GPU DRIVER: Equivalent to the compiler. Compiles the code and makes optimizations such as merging multiple instructions to a more complex VLIW instructions if possible.

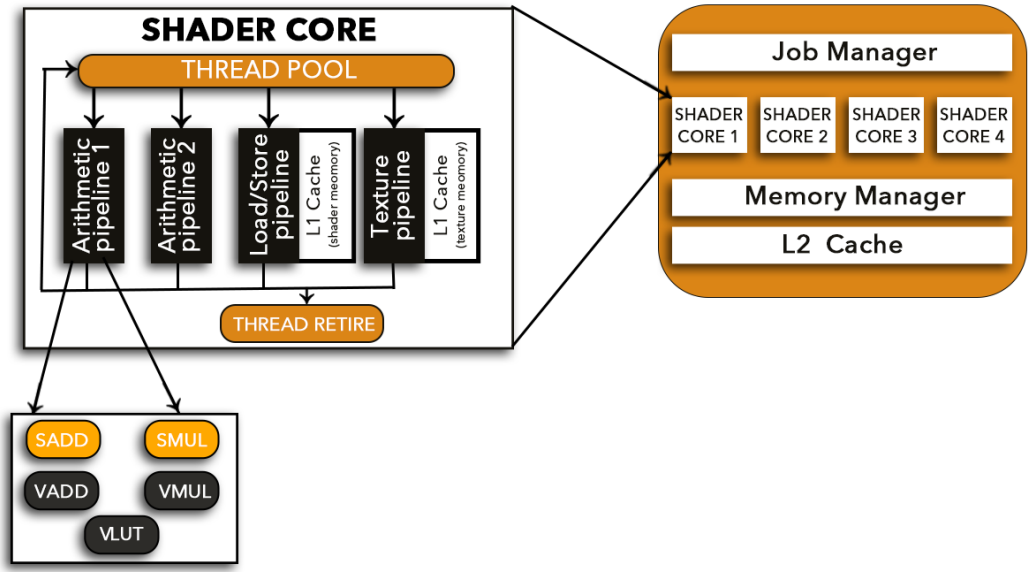


Figure 5.2 Shows the microarchitecture of the GPU Arm Mali T-624

5.2 ARM MALI T-624 GPU Specifications to reveal

The main goal of my dissertation is to reveal more about the microarchitecture of the Mali T-624. First thing to do is that I am going to discover more about the number of the pipelines that exists in this GPU and the depth of the pipeline.

Moreover we are going to discuss more about the Job Manager the way that the work-items and workgroups are going to split across the four shader cores (Work-load distribution among cores). In addition I will try to verify exactly the number of the scalar and vector functional units that exist in the arithmetic pipelines. Those units are especially made for addition. Besides this, I will try to write parallel code that is using as many work-items as possible so that I can examine the register spilling event and the register file and if it's shared in scalar and vector data types. To sum up the characteristics of the microarchitecture that I am going to validate and explore will help me to understand more about the upper limits of the Mali T-624 when executing a parallel program.

Chapter 6

ARM MALI T-624 GPU Methodology

In this chapter I explain the methodology I followed to conduct my experiments and extract the different characteristics of the Mali GPU.

6.1 Hardware Characteristics

Some hardware characteristics where already known to us (figure 5.1). First and most important is that in order to eliminate the variations we disable dynamic frequency scaling, instead we set the clock of the GPU to fixed frequency, specifically to 600 Mhz. This value is called nominal value too and we make sure that it is always locked by setting the GPU governor to performance model. There are no fluctuations of the frequency even if the system is idle and does not run any process.

6.2 GPU Kernel Setup

First I have to explain how I get the value of the metric cycles per iteration of the kernels I execute. The equation is described on figure 6.1

$$\frac{\frac{time\ execution\ in\ seconds}{number\ of\ iterations\ inside\ the\ loop}}{Clock\ Cycle\ Time} = \frac{time\ execution\ in\ seconds}{number\ of\ iterations\ inside\ the\ loop} = \frac{time\ execution\ in\ seconds}{number\ of\ iterations\ inside\ the\ loop}$$
$$\left(\frac{1}{frequency}\right) \qquad \left(\frac{1}{600000000}\right)$$

Figure 6.1 cycles per iteration

In detail, the execution time in seconds is the time it takes for the parallel program to be executed, followed by the number of repetitions of the loop and finally the clock cycle time of the system which is (1/frequency) and its constant value is approximately 1.66e-9ns. Time execution is measured by an OpenCL routine the “clGetEventProfilingInfo”.

It would be good to repeat that all my experiments are taking place on the Junoboard r2 that has an arm CPU and the GPU Arm Mali T-624. The host in the OPENCL framework is the CPU (Arm Cortex A53 & A72) and the device is the Arm Mali T-624. I am always have a 1D space of work-items and I am always passing and returning a list which at the end of the execution is going to contain the output results produced from the GPU in order to prevent dead code elimination as the compiler can easily inspect the code and execute only the code that produces data that are going to be used. Some of the parameters I pass to the device through host is a flag which deactivates and prevents the OpenCL optimizations during the compilation of the parallel code in order to make the execution easier to predict. It is also worth noting that the number of work-group and work-items changes and is not always constant in all experiments.

6.3 Power measurements Methodology

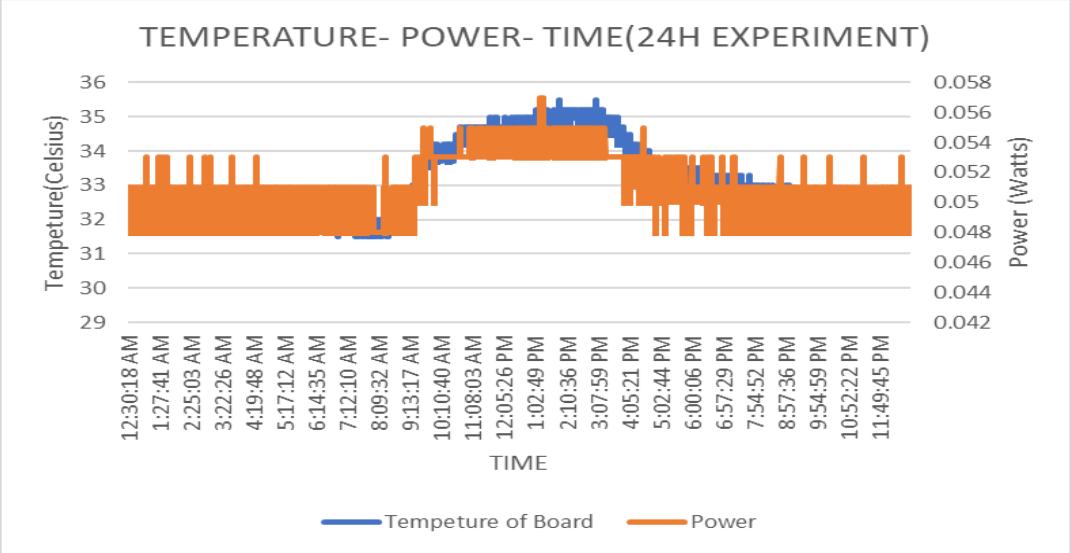


Figure 6.2 Graph for 1 day experiment measuring idle power

Furthermore is important to mention how I extract the GPU power. The value of Power in a certain time is given to the programmer using on-board power meter. Over time as I was executing different types of experiments I observed that the ambient temperature changes throughout the day see figure 6.2 and as a result the idle power of the GPU changes see equation section. We want to eliminate the effect that the idle power consumption has on the overall power consumption and so we decided to measure idle power for each experiment while there are no background processes in the system and remove it from the average power consumption of the experiment (active power). For statistical confidence and stability in my methodology I subtract the idle power from the active power value in order to avoid any mistakes due to volatility (because the experiments were executed in different ambient temperatures throughout the year) and I ran each kernel multiple times with 100 million inside the for loops.

More over to see how stable and predictable is our method for extracting the power and

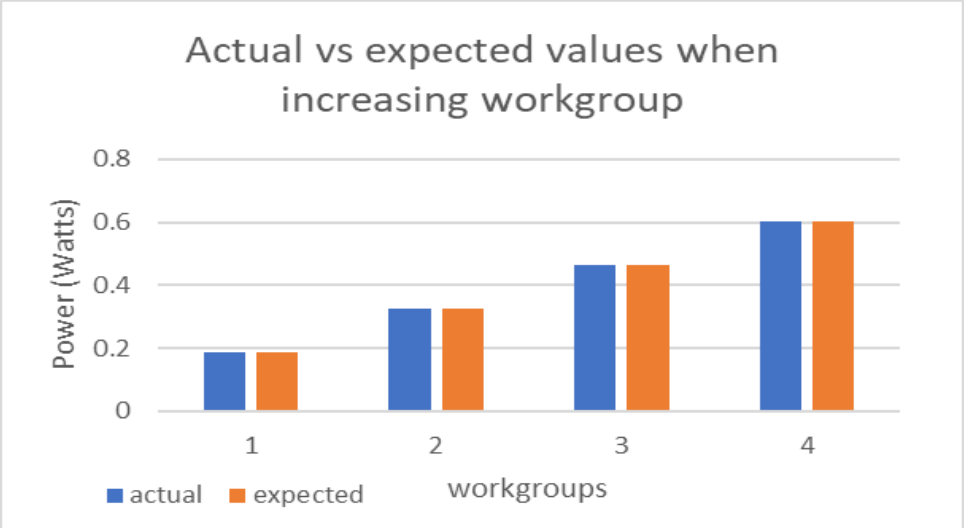


Figure 6.3 increasing workgroups and keeping the work-items to 64

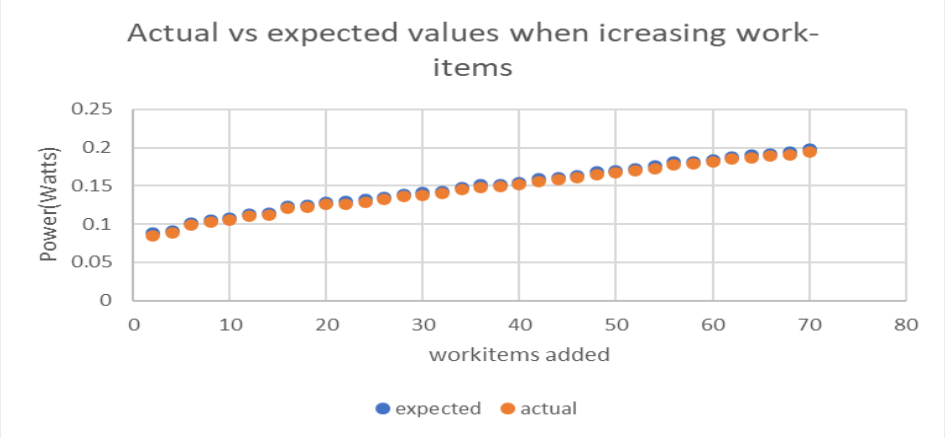


Figure 6.4 increasing work-items and keeping the work-groups to 1

find our reference-base kernel that has a for loop with 100million iterations(figure 6.5) we did the following experiments. In the figure 6.3 we have a graph that presents the actual power consumption vs the expected power consumption. We can see that the expected values are identical with the real values(expected values are found by subtracting the power consumption of work-group=2 with the values of workgroup=1) and that the power consumption when increasing the workgroups of size 64 is about 0.140 watts each time. We did the exact same experiment (figure 6.4) by increasing the number of work-items(keeping the work-groups to 1) and comparing the actual vs expected values of the power consumption. We can see that the expected values are a little higher than the actual values but the difference is negligible (expected values are found by subtracting the power consumption of work-item=2 with the values of work-item=1 and adding it each time). The values of power consumption when increasing the work-items are steady. In conclusion we can make the kernel (figure 6.5) our reference kernel as the power consumption of it when increasing work-groups or work-items follows a stable trendline.

6.4 Power measurements Validation

Objectives for this task was to make sure that the power consumption reported is as expected when changing the frequency and see that the frequency was actually set to the desirable value that I set it and its relationship with the power due to the DVFS part of the hardware that increases or decreases both proportional. Moreover I wanted to make sure that the execution time is getting increased linearly while we set the frequency each time to lower values.

Experimental setup: The frequency of the GPU was clocked each time to 24 MHz less than the previous iteration starting from 600Mhz and going down to 432 MHz. For the purposes of this Task I wrote a simple script for automation reason that can run the executable file from the COM4 connection of the junoboard keeping the work-items to 64 and work group to size 4.

Experimental results :

For each one of the work items I assign to them 100 million iterations inside a for loop and a simple scalar addition (figure 6.5)

To make sure that the frequency was set to the desirable value I check the ratio of the first and second values of power and frequency to see if they were getting decreased the same.

Results from the figure 6.6

Power 0.853Watts/0.809Watts was equal to approx. 1.05

Frequency 600MHz/576Mhz was equal to approx. 1.04

Time 27.60sec/26.5sec was equal to 1.05

Finally I kept the iterations for the total amount of the work-items the same. This means that frequency is getting linearly decreased like the power and the time gets increased as the two other parameters got decreased.

```
__kernel void add(__global  int * list){

int x=38;

for(int i=100000000; i=0 ; i-- ){

 x=x+i;

}

 list[get_global_id(0)]=(int)x;

}
```

Figure 6.5 simple add kernel

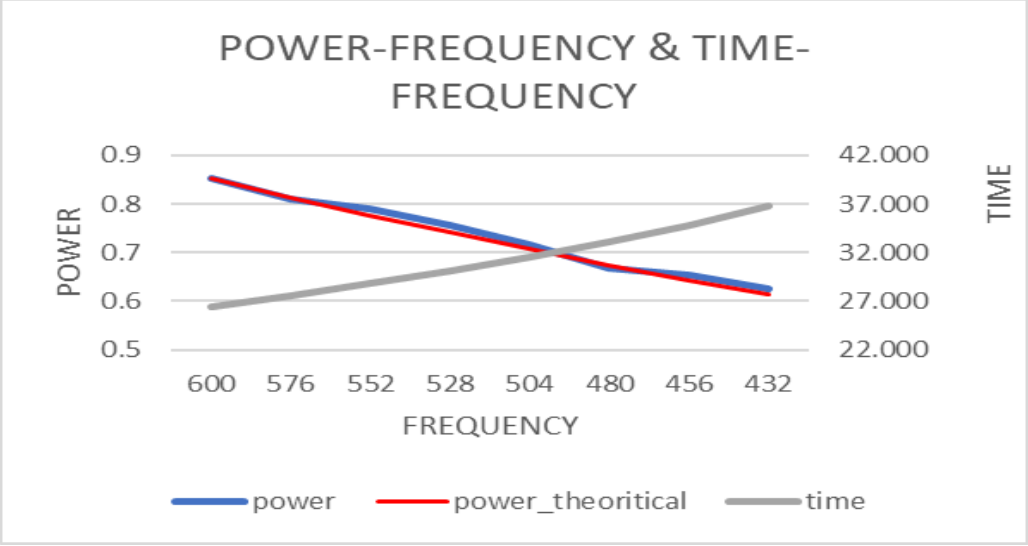


Figure 6.6 Power -Frequency-Time execution graph

To examine-validate why the power is getting reduced while we are reducing the frequency in a more theoretical model we can work with this equation

$P = C * V^2 * (a * f)$

So while we are reducing the frequency :

- Capacitance stays the same
- Voltage stays the same
- A (switching activity= on average per cycle how many transistors are changing state 0 → 1) stays the same

In conclusion I observed that in this theoretical equation only the f(frequency) can change the Power of the system.

In the graph we have the red line that represents the theoretical power values while reducing the frequency from 600 down to 432(24 MHz each time). I observed that experimental values are close to the ones that theoretical model but not exactly the same.

6.5 Approach for analysis

In this section I am going to discuss more about the approach I followed in order to decide the parallelism within the GPU.

$$Power = \frac{Energy}{Time}$$

This particular power equation will help a lot to discover parallelism. Because for example if we add another operation inside a kernel (e.g. scalar addition inside a kernel that is containing only scalar additions) and the power consumption gets increased and takes the same time to be executed that means the system can handle the extra work in parallel (and that means it has the extra units to execute it). In other words more work done in the same time means higher power consumption. In the other side from this equation we can discover the constraints on this GPU. So if we add more work than the GPU can handle in parallel we will see a time execution increment and at the same time because we add more work we need more energy. The energy that the GPU that needs to perform the extra work increases and for this reason after we overcome the maximum parallelism of the system we will see a stable value in the power consumption because the fraction of the power equation produces the same result because the extra work needs more time due to the fact that the extra work assigned have reached the parallelism limits and has to wait for the previous operations to finish (higher energy than before and higher time than before).

Another constraint that can help us decide the parallelism is the VLIW scheduling as it can fit only up to four independent instructions at the same time and not more. Also another constraint is possibly any dependences among the instructions of a kernel. For example x=x+1; and u=u+x; have dependence because u variable needs the result of x to be computed in order to add the correct value.

Chapter 7

Arm Mali GPU Shader cores

7.1 Available GPU Cores

In this chapter we are going to validate the number of cores of the GPU Arm Mali T624. As we already the GPU achieves the Core Level Parallelism using simultaneously if it needs the 4 shader cores available at the same time.

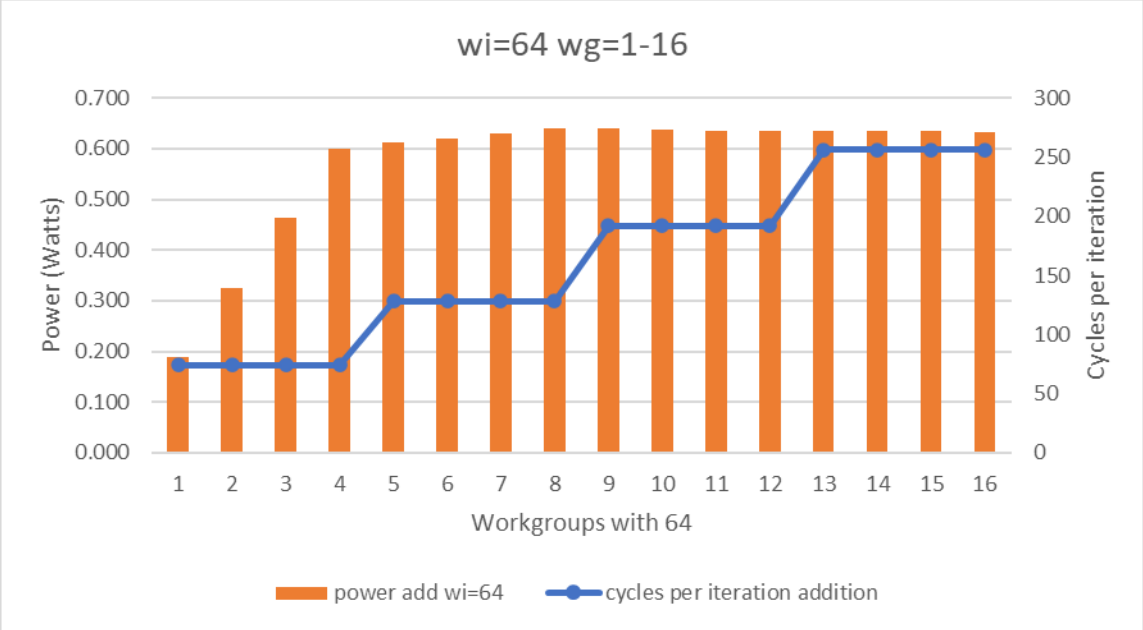


Figure7.1 Power-Cycles per iteration-Work-Groups

```
__kernel void add(__global  int * list){

int x=38;

for(int i=100000000; i=0 ; i-- ){

 x=x+i;

}

 list[get_global_id(0)]=(int)x;

}
```

Figure7.2 Base kernel

Experimental setup & Objectives: For this task I wrote two simple kernels figure 7.2 our base kernel. To make this more specific I have wrote this kernel and assign the most suitable number of workload each time to activate all the GPU Shader cores in order. I was keeping the number of work-items to 64 and I was increasing the number of workgroups each time by one as can be seen from the x-axis on the graph(figure 7.1). Also I have to say that the comparison (i!=0) depends for the value of the loop counter (i--). Moreover the addition inside the kernel also depends on the value of the loop counter as it wants to add it on the variable x so the extra addition needs to be computed later than the loop counter

Experimental results:
We can see in the bars in the graph above (figure 7.1) we can break this experimental results into two phases. First of all phase 1 is where we are increasing the workgroups from 1 to 4 and the second one is when we have 5-16 work-groups.

During the first phase the system power is getting increased approximately 0.14W each time we add another workgroup and the cycles per iteration are approximately 74 and remain stable up to 4 work-groups. So considering the fact that the power is increasing and the cycles per iteration are constant that shows that the system every 64 work-items inside a work-group a core gets activated.

In the second phase we can see a pattern which the cycles per iteration are getting increased by 37 each time we add 4 more work-groups to be executed while the power remains the same(5-8, 9-12, 13-16). This is because from the power equation can see

that adding more work needs more energy and more time if no available resources available. Energy and time factors are getting increased linearly so the fraction has the same impact on Power.

All in all Cycles per iteration and Power measurements are helping us to understand the maximum parallelism in cores since as soon as the first factor increases means that we have no other resources-cores available to execute any other workgroup at the same time. So we can verify that there are 4 cores in the GPU Mali T-624 as long as the Cycles per iterations are increasing after 4 workgroups of 64 wi the certain pattern we examined above.

7.2 Arithmetic Pipeline Depth

Objectives for this task: we are focusing on extracting the exact number of Mali's GPU Arithmetic pipeline depth (stages) which a thread could follow during the execution of a kernel

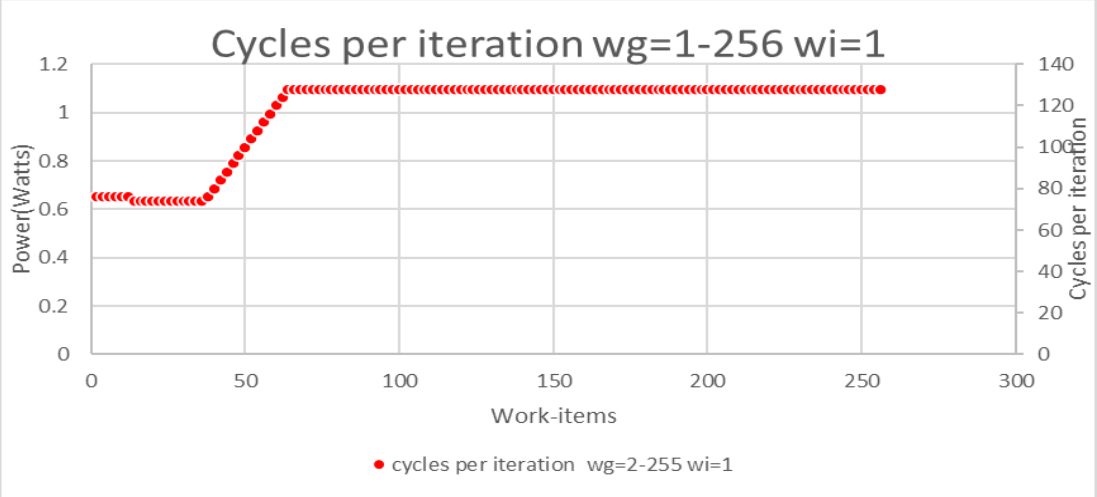


Figure 7.3 cycles per iteration – work-items graph

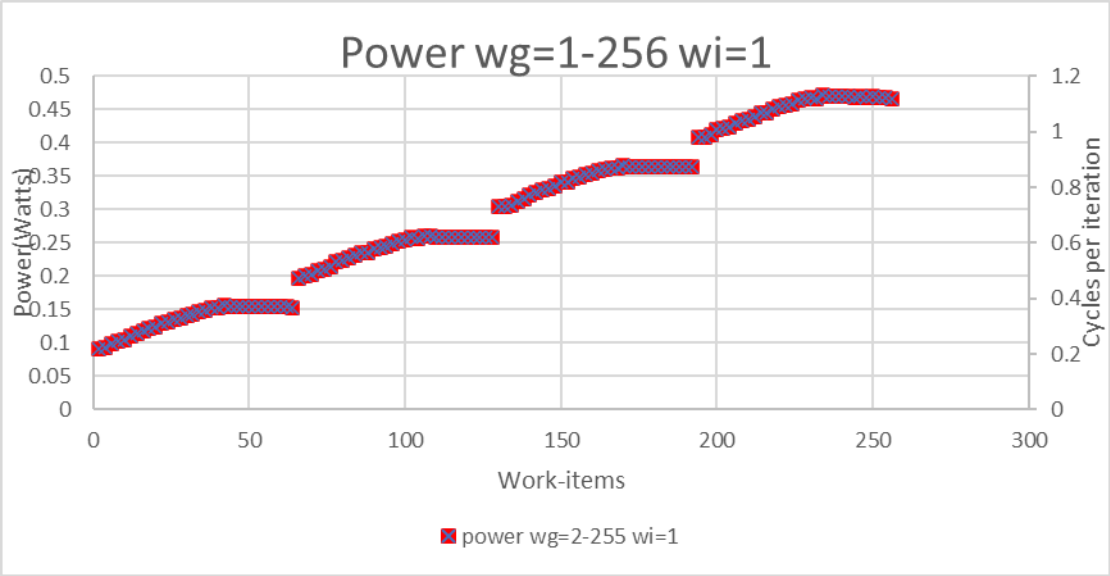


Figure 7.4 Power – work-items graph

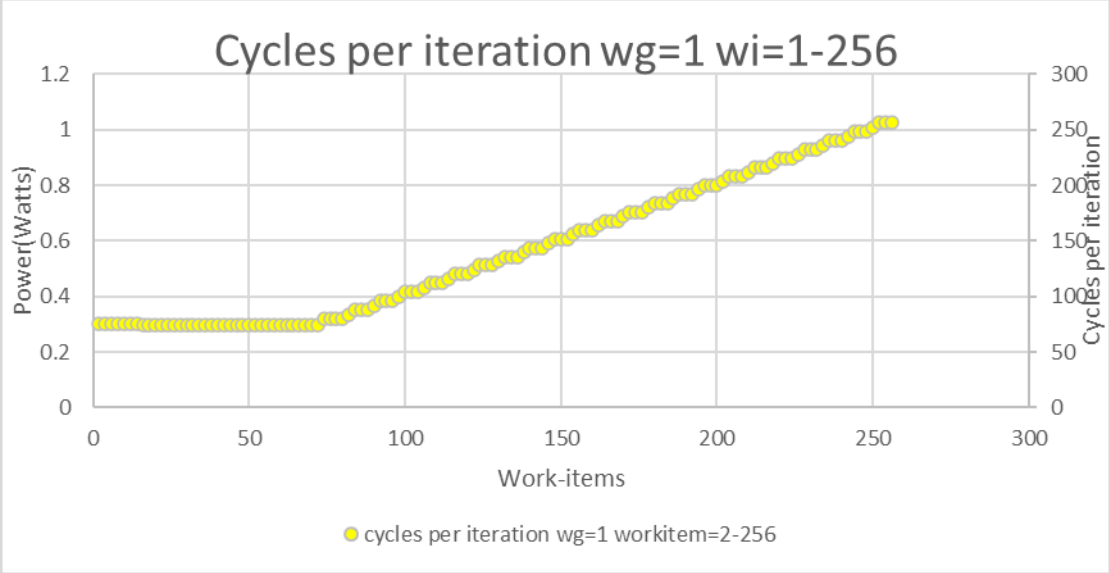


Figure 7.5 cycles per iteration – work-items graph

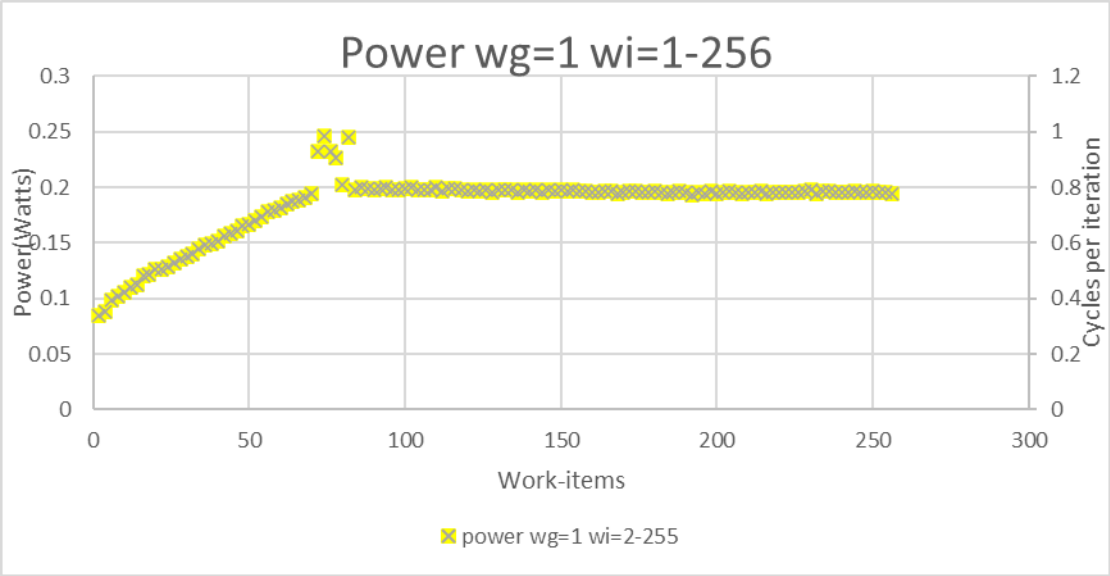


Figure 7.6 Power – work-items graph

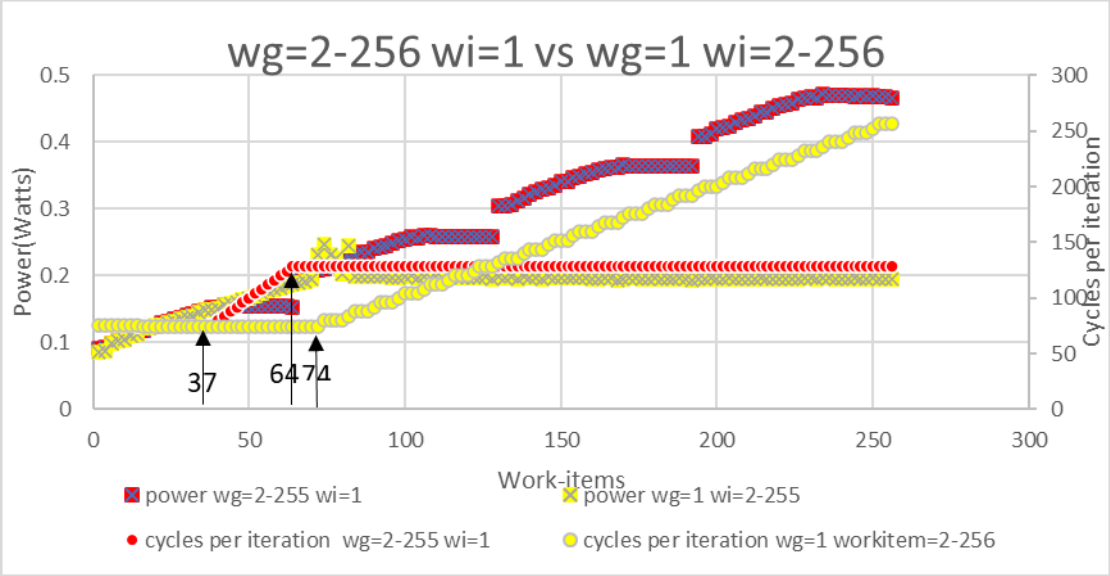


Figure 7.7 above graphs combined 7.3-7.6

Experimental setup:

In order to make this experiment possible I wrote a simple kernel (figure 7.8) which from now on we will consider it as the base kernel and contains a simple loop with 100 million iterations with a scalar add. Also I have to make clear that I used two execution Scenarios to assign threads for execution to the GPU. First scenario of execution : I kept the number of work-groups constant at 1 and I executed the experiment for many different work-items within range 1-256. The second type of execution I kept the

number of work-items inside a workgroup to 1 and I was increasing the number of workgroups within the range 1-256.

Experimental Results:

We already know that the system has two arithmetic pipelines in each core.

In the first of execution scenario on the graph (figure 7.5) we can see that the cycles per iteration are remaining stable from 1-74 work-items and the power is getting increased as expected (figure 7.6) because we have more work done in the same time so that shows at this stage we have not fully filled both pipelines with work-items. Suddenly after 74 until 256 work-items the cycles per iteration are getting increased almost linearly in relation to the work we assign to the work-group so that indicates that we have reached the maximum parallelism when we executed 74 work-items. While this was happening the power was remaining stable approximately at 0.2 W.

In the second execution scenario we can see that the cycles per iteration (figure 7.3) are remaining stable from 1-37 work-items in total and the power (figure 7.4)is getting increase so that shows at this stage that there are existing the facilities to support those work-items. Later we can see a strange behavior and the cycles per iteration are getting increased before 74 work-items like before and the power tends to stay stable from 38-64 work-items. After that within the range of 65-256 the cycles per iteration are approximately 110 and the power is following the same pattern as 1-64 thread which means that the workload is shared among other cores. If it wasn't distributed then we would see cycles per iteration increasing and the power remain constant.

The above experimental results (all combined in figure 7.7) can lead us to the conclusion that the first model is using both arithmetic pipelines inside a core that's why it can support approximately double the amount of work. Moreover the second model of execution is using only the one arithmetic pipeline due to the power stillness between 37-64 wi. So the second execution model reveals that the pipeline depth inside an arithmetic pipeline of the Mali T-624 GPU has 37 stages and with the fine grained multithreading policy combined can support up to 37 different wi at the same time. The VLIW scheduling for this experiment can be found at figure 7.9.

Another observation in this experiment is that a single workgroup can only contain up to 256 Work-items. If a programmer assigns more than that number the Driver cannot support it and cannot execute any kernel so it gives this error message "CL_INVALID_WORK_ITEM_SIZE".

From now on I can introduced safely the VLIW scheduling for each subsequent kernel in my experiments since I know now that the pipeline depth is 37. This knowledge combined with fine-grained multithreading makes us aware that each VLIW command needs 37 cycles to be executed. For example a kernel which has 2 VLIW instructions I expect it to need 74 cycles to be executed and not 38 cycles because Coarse-grained multithreading is not used in this GPU (more about this on section 3.3).The scheduling for the experiment's kernel(figure 7.8) is presented in figure 7.9

```
__kernel void add(__global  int * list){

int x=38;

for(int i=100000000; i=0 ; i-- ){

 x=x+i;

}

 list[get_global_id(0)]=(int)x;

}
```

Figure 7.8 Base kernel

cycles per iteratio	Scalar add	Scalar add	Vector add	Vector add
37	i--;			
74	x=x+i;	i!=0;		

Figure 7.9 VLIW scheduling

7.3 Work-Groups scheduling

```
__kernel void add(__global  int * list){

int x=38;

for(int i=100000000; i=0 ; i-- ){

 x=x+i;

}

 list[get_global_id(0)]=(int)x;

}
```

Figure 7.10 Base kernel

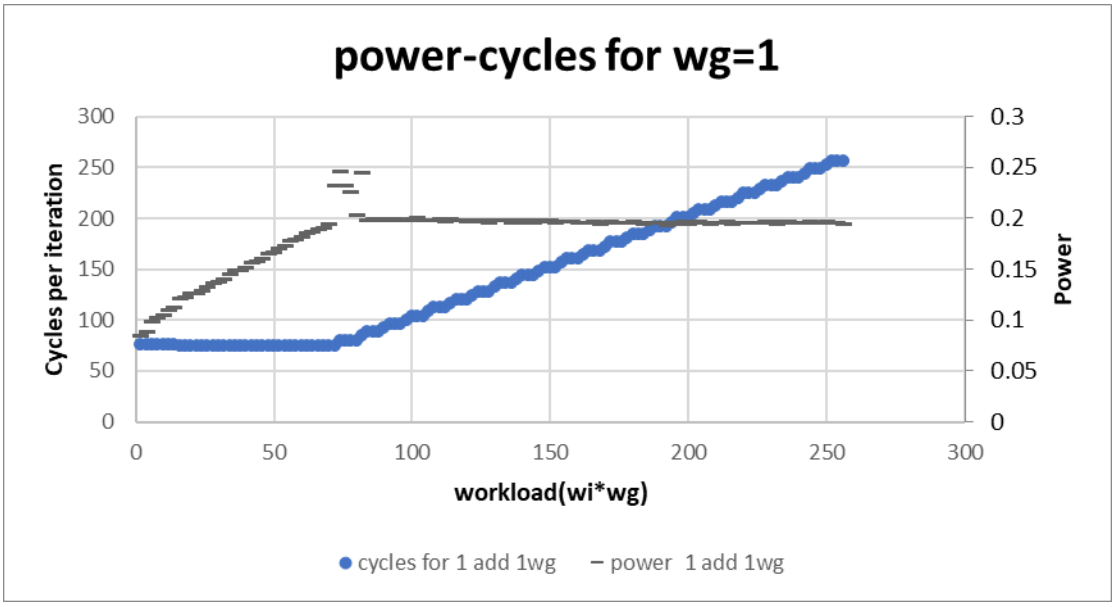


Figure 7.11 Cycles-Power-Work Load

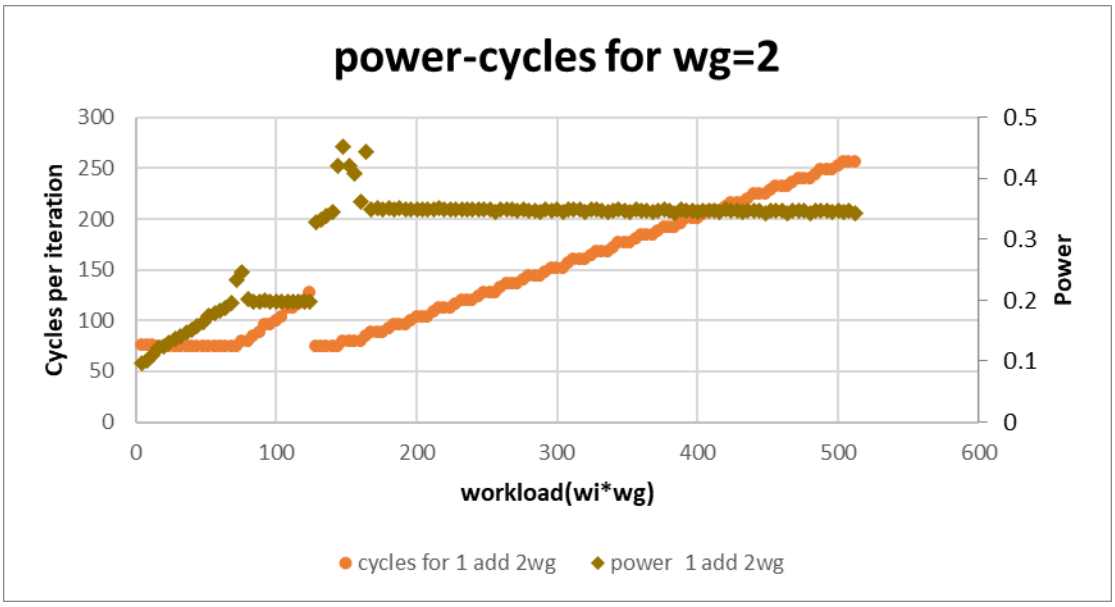


Figure 7.12 Cycles-Power-Work Load

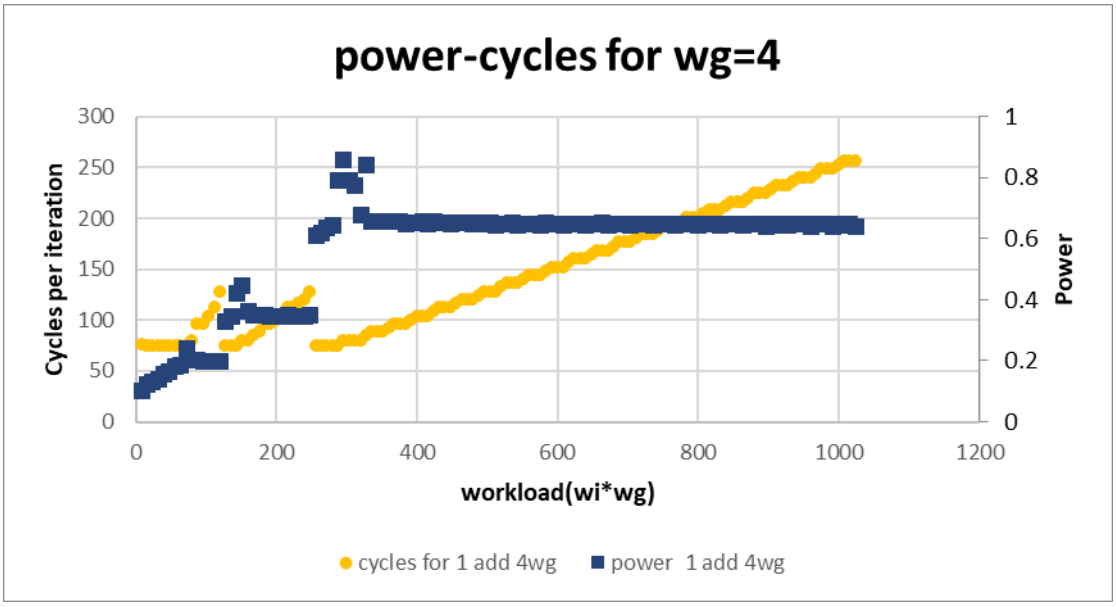


Figure 7.13 Cycles-Power-Work Load

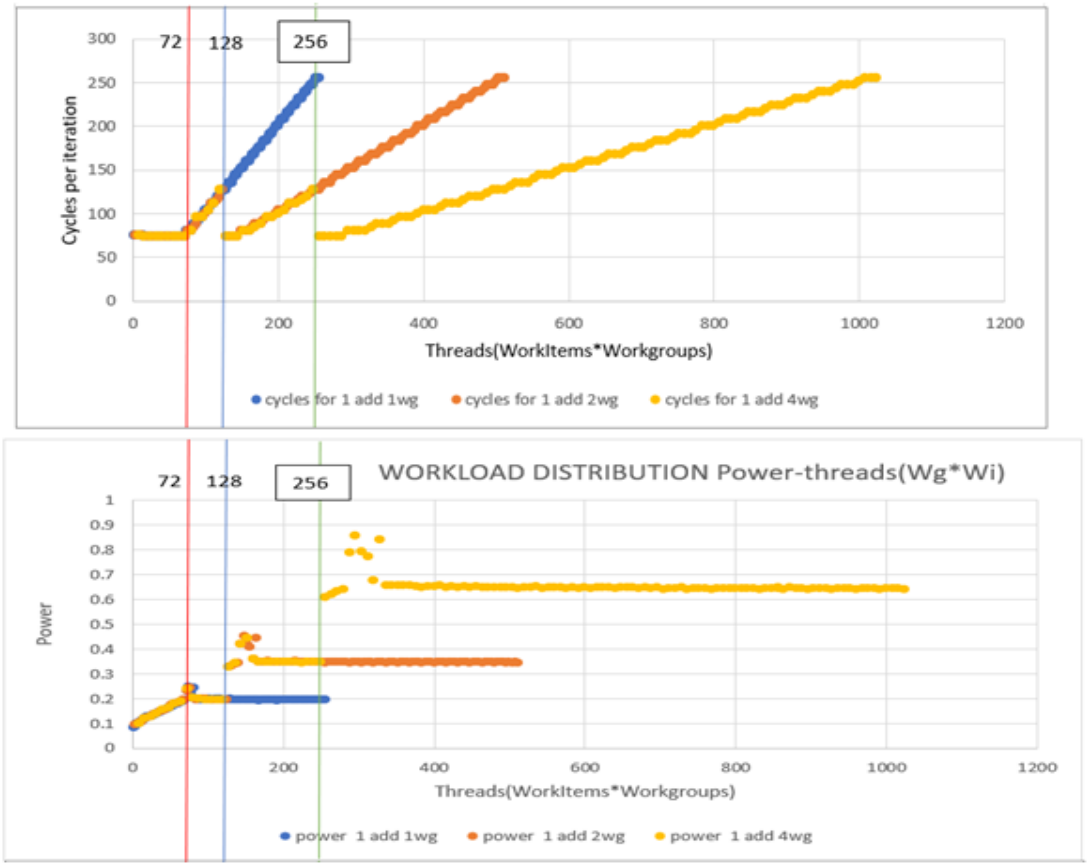


Figure 7.14 All the above figures combined (Figures 7.11 - 7.13)

Objectives for this task: In the previous section I managed to validate the number of Shader Cores inside the Mali T-624 GPU. Now we are taking a further step and in this section I am focusing to extract a model which the GPU tends to follow in order to distribute the workload among the cores. The life cycle of a thread begins with queuing in the thread pool, then passes to the GPU units and then retires after the execution of all its commands.

Experimental Setup: In order to make this experiment possible we have used again the base kernel (figure 7.10) and executed it with the following combinations of: 1,2 and 4 workgroups and wi=1-256 in order to extract some results for the distribution of work among the cores. Also I have to mention that the graphs x-axis is containing the total workload of threads that means:

$$workload = workgroups * workitems$$

Also I have to say that the comparison (i!=0) depends for the value of the loop counter (i--). Moreover the addition inside the kernel also depends on the value of the loop counter as it wants to add it on the variable x.

Experimental results:

Phase 1(Results from figure 7.11):

For example if we take the blue line that corresponds to the cycles per iteration from 1-72 work-items (72=1*72) those work-items are executed in the same processor because the power consumption is getting increased almost negligible decimal digits between 0.1 - 0.2 and the cycles per iteration are stable to 74 (also can be seen from the power values of the shader core experiment that they are inside 1 core). The same thing can be applied on all three lines in the graph above (4*18 or 1*72) if the workload is less than 72(on figures 7.12 and 7.13). Suddenly after we pass 72 work-items the cycles per iteration are getting increased because in that point we have hit the maximum parallelism possible inside a single processor so each additional wi that is arriving needs few more cycles to be executed so the power stays the same because the GPU has more work to do in more time. (Power = Energy / time execution). I am referring to the workload 73-127 threads.

Phase 2 (Results from figure 7.12)

For 2 work-groups we have the same distribution of the work as it was on the figure 7.2 if the workload was less than 128 work-items in total (e.g. 2*60=120 so it is executed like I discussed above on the phase 1 (inside a single processor). But then when the work-items inside a work-group are more than 64. For instance here we had 2 workgroups and we assign 64 or more work-items inside those 2 workgroups so that leaves us with 128 work-items in total. Then we can see from the cycles per iteration that there is existing processing units to execute those work-items faster because a second core gets activated (according to the higher power consumption in that moment). Cycles per iteration got decreased from 124 to 74 when the new core got activated and the power increased ≈ 0.140W. Finally after the core gets activated all the rest work-items added were executed inside those two cores(no more than 2 cores got activated).

Phase 3 (Results from figure 7.13):

Same thing can be applied to this figure 7.4 until the driver found out about the assignment of 64 work-items inside four workgroups(4 * 64 = 256). Then at that particular moment the Job Manager split the work equally to all four cores. The reason that our assumption is validated is because the power consumption got increased and the cycles per iteration got decreased so the Job Manager took advantage of all the four shader cores of the Mali T-624. The rest work-items added after 288 have extra cycles per iteration penalty as there is no more cores to execute the extra work assigned so the power remains the same.

Also in figure 7.15 I will show the scheduling inside a VLIW instruction for the kernel in 7.10

cycles per iteratio	Scalar add	Scalar add	Vector add	Vector add
37	i--;			
74	x=x+i;	i!=0;		

Figure 7.15 VLIW Scheduling

Models extracted from this experiment:

$$Number\ of\ activated\ cores = \min\left(\frac{wi}{64}, workgroups, number\ of\ cores\right)$$

7.4 Work-Items Scheduling

Objectives for this task: In the previous section I managed to show the scheduling in Arm Mali T-624 for a given number of work-groups. Now we are taking a further step and in this section I am focusing to extract a model which the GPU tends to follow in order to distribute the Work-Items among the cores.

Experimental setup: For this task I wrote a simple kernel same as figure 7.10 (our base kernel). To make this more specific I have wrote this kernel and assign the most suitable number of workload each time to activate all the GPU Shader cores in order. The execution scenario I followed was used before when we discovered the pipeline depth. I was keeping the number of work-items to 1 and I was increasing work-groups in the range 1 – 256.

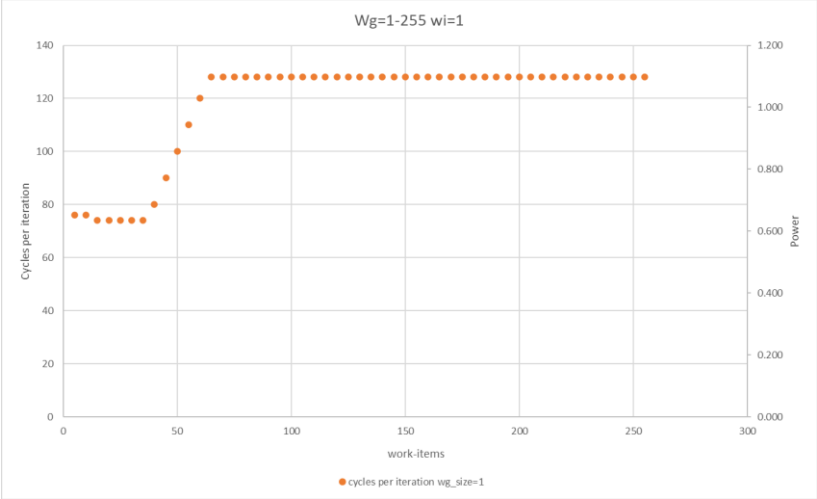


Figure 7.16 Cycles per iteration

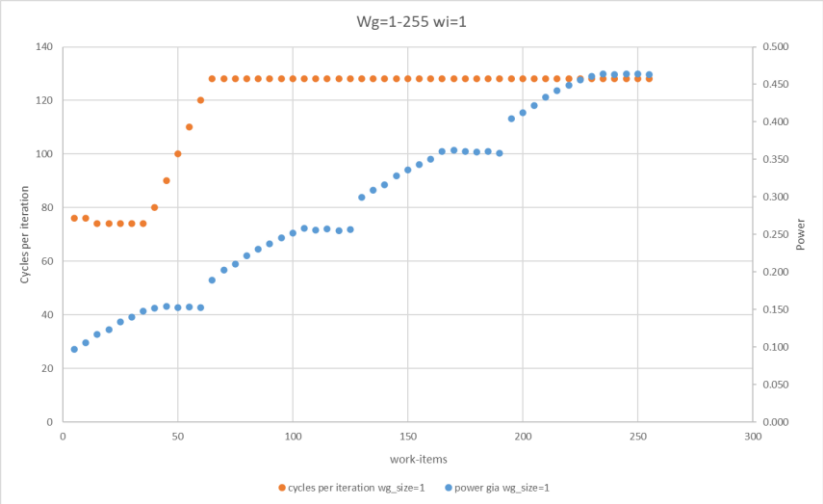


Figure 7.17 Cycles per Iteration and Power

Experimental results: Like discussed before in the experiment that I discovered the pipeline depth this execution model is taking advantage of only the one of the two arithmetic pipelines that exist inside each core. That is validated because as can be seen from the figure 7.16 the Cycles are stable to 74 because he have only two VLIW instructions and each one takes 37 cycles per iteration and the power gets increased when we get closer to 37 (more work done at the same time). But after 37 until 64 there is no space inside the single arithmetic pipeline inside a core to execute more VLIW instructions (figure 7.15) as we increasing the number of work-items so each additional work-item added has to wait for an old one to finish execution(fine-grained multithreading) and the power consumption fraction remains constant due to the energy and time linear increment.

When I had assigned more than 64 work-items a new core got activated because cycles per iteration dropped significantly and the power got increased in the same pattern like 1-64 work-items and the cycles per iteration remained constant because the work-items after 64 got assigned to a new processor. As can be seen the power is significantly lower than the other experiments(figure 7.11) where both arithmetic pipelines were utilized with work-items.

In conclusion with this execution scenario a core gets activated every 64 new work-items that got assigned.

Chapter 8

Addition Units

8.1 Scalar data type addition units

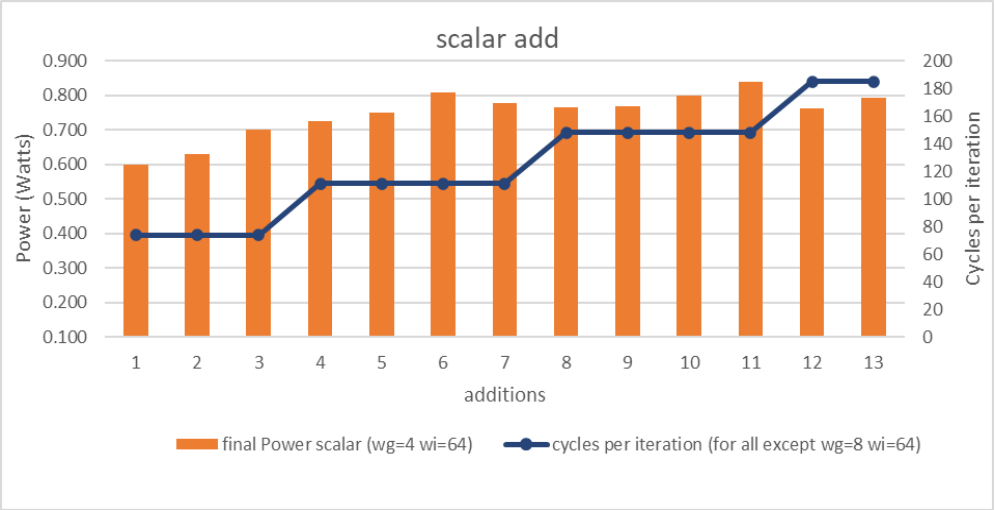


Figure 8.1 Power - Cycles – Scalar add operations

Objectives for this task: Discover how many scalar additions can the GPU handle in parallel. Moreover I wanted to make sure that the execution time is getting increased linearly while I add more additions for each work-item if were no more available units inside a core.

Experimental setup: For the purposes of this Task I wrote a simple kernel(figure 8.3) which consists of a for loop with 100m iterations and inside of it I was adding one more independent addition each time starting from 1 up to 13 scalar additions (additions

added are equal to the number on the x-axis on the graph). Each work-item was executing A * N additions + N additions for the loop (A=additions N=number of iterations). There is dependence between the comparison (i!=0) and the loop decrement (i--) so the are not executed in the same VLIW instruction. Moreover the first addition inside the kernel also depends on the value of the loop counter as it wants to add it on the variable x.

Observations for this experiment: A simple thing of the utmost importance to mention is the fact that the loop is using an addition unit to accomplish the counter increment. According to the graph (figure 8.1)we can see that the GPU can execute 4 scalar additions in parallel. The fact that confirms this is again the cycles per iteration and the power in the graph. The cycles can show that the addition parallelism inside the GPU is 4 because each additional scalar add after a group of 4 costs 37 more cycles(VLIW scheduling figure 8.2). The first addition added depends on the value of the counter(i) that is why is executed in the next VLIW instruction and not the same as that.
Also is essential to mention that although it has 10 addition slots(=2*1 scalar + 2*4 vector) the GPU tends to:

- use both vector unit and scalar units to execute multiple scalar additions only 1 addition inside(more later on the next section). When a scalar operation is assigned to a wi is executed by scalar or vector unit inside the core.

Theoretical models extracted from this experiment:

$$Cycles\ per\ iteration = (Number\ of\ scalar\ additions + 1) Mod\ 4 * 37$$

cycles per iteratio	Scalar add	Scalar add	Vector add	Vector add
37	i--;			
74	x+=i;	i!=0;	d+=9;	b=b+2;
111	z=z+4;	c=c+6;	e=e+2;	r=r+10;
148	t=t+12;	a1=a1+3;	a2=a2+20;	a3=a3+9;
185	a4=a4+8;	a5=a5+3;		

Figure 8.2 VLIW Scheduling

```
__kernel void scalar_add(__global int *list){
int x=38;
int c=55;
int z=0;
int b=399;
int d=9;
int e=41;
int r=16;
int t=99;
int a1;
int a2;
int a3=999;
int a4=88;
int a5=99;
int4 y=(int4)8;
int4 f=(int4)9;
for (int i=100000000; i!=0 ; i--){
x=x+i;
d=d+9;

b=b+2;
z=z+4;

c=c+6;
e=e+2;

r=r+10;
t=t+12;

a1=a1+3;
a2=a2+20;

a3=a3+9;
a4=a4+8;

a5=a5+3;
}
list[get_global_id(0)*13]=z;
list[get_global_id(0)*13+1]=d;
list[get_global_id(0)*13+2]=b;
```

Figure 8.3 kernel

8.2 Vector data type addition units

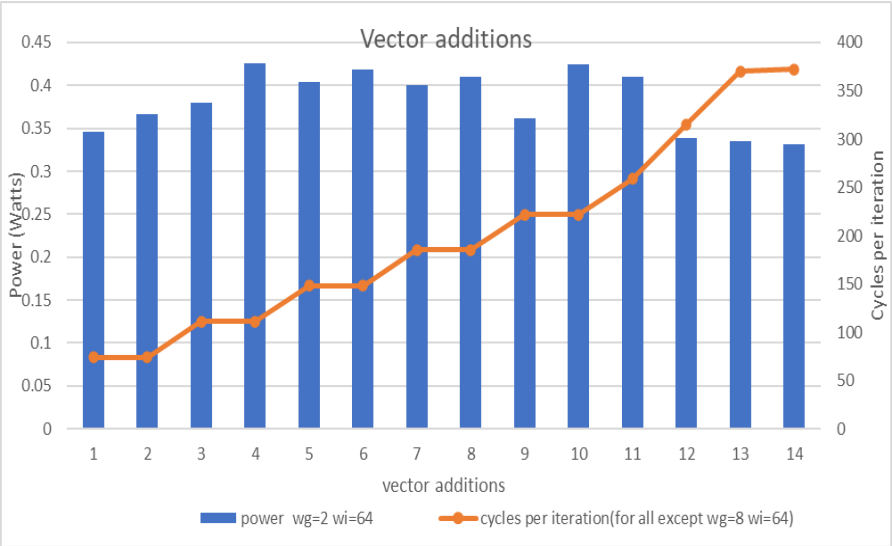


Figure 8.4 Power - Cycles – Vector add operations

Objectives for this task: Discover how many Vector additions can the GPU handle in parallel and what is the cost in cycles when we exceed the amount of parallel vector additions the GPU can handle.

Experimental setup: For the purposes of this Task I wrote a simple kernel (figure 8.5) which consists of a for loop with 100 million iterations for statistical confidence when measuring power and inside of it I was adding one more vector type addition each time starting from 1 up to 14 vector additions (additions added are equal to the number on the x-axis on the graph). Each vector was independent from the others added inside the "for" loop. As I mentioned above in the OPENCL data types section the int4 in the kernel code stands for 128 bit vector that has 4x32 bit integers. Also I have to say that the comparison (i!=0) depends for the value of the loop counter (i--). Moreover the addition inside the kernel also depends on the value of the loop counter as it wants to add it on the variable y.

Experimental results: According to the graph on figure 8.4 the GPU tends to execute 2 vector additions in parallel as we can see from the cycles per iteration(VLIW scheduling figure 8.6). The first addition added depends on the value of the counter(i) that is why is executed in the next VLIW instruction and not the same as that.

```
__kernel void vecadd(__global int *list){
int4 y=(int4)8;
int4 x=(int4)38;
int4 c=(int4)55;
int4 z=(int4)0;
int4 f=(int4)41;
int4 b=(int4)399;
int4 d=(int4)9;
int4 e=(int4)41;
int4 r=(int4)16;
int4 t=(int4)99;
int4 a1=(int4)5;
int4 a2=(int4)66;
int4 a3=(int4)999;
int4 a4=(int4)88;
int4 a5=(int4)99;
int4 a6=(int4)95;
int4 a7=(int4)1;
int4 a8=(int4)66;
int4 a9=(int4)9999;
int4 a10=(int4)898;
int4 a11=(int4)909;
for(int i=100000000; i!=0 ; i--){
y=y+i;
x=x+4;

c=c+6;
z=z+9;

f=f+2;
b=b+7;

d=d+10;
e=e+12;

r=r+2;
t=t+44;

a1=a1+7;
a2=a2+7;

a3=a3+4;
a4=a4+10;
}
```

Figure 8.5 kernel

The reason that we can support our hypothesis is simply because the cycles per iteration are getting increased by 37 cycles every N/2 (where N the vector additions). Also the power consumption is getting increased enough in the first two operations. After the first 2 additions the power keeps an almost constant pattern in its consumption. Combining those together with the power equation our hypothesis is getting confirmed because as we assign more operation and need more time to execute them because of the fact that there are no more than 2 vector units. All in all power remain stable after the first 2 additions.

We can also say with confidence that both pipelines have the same number of addition vector units (specifically 2) since the experiment was performed by filling both pipelines completely with wi, however the time remained constant depending on the operations of additions that I put in the kernel (the corresponding values on the x-axis)

Theoretical models extracted from this experiment:

$$Cycles\ per\ iteration = (Number\ of\ add\ vector\ operations\ MOD\ 2\ ) * 37$$

cycles per iteratio	Scalar add	Scalar add	Vector add	Vector add
37	i--;			
74		i!=0;	y=y+l;	x=x+4;
111			c=c+6;	z=z+9;
148			f=f+2;	b=b+7;
185			d=d+10;	e=e+12;
222			r=r+2;	t=t+44;
259			a1=a1+7;	a2=a2+7;
296			a3=a3+4;	a4=a4+10;
333				
370				

Figure 8.6 VLIW Scheduling

8.3 Scalar and Vector data type addition units

Figure 8.7 Power - Cycles – Scalar and Vector add operations

```
__kernel void add_mixed(__global int *list){
int4 x=(int4)38;
int4 c=(int4)55;
int z=0;
int b=399;
int d=9;
int e=41;
int r=16;
int t=99;
int4 a1=(int4)33;
int4 a2=(int4)3;
int a3=999;
int a4=88;
int a5=99;
int4 y=(int4)8;
int4 f=(int4)9;
for (int i=100000000; i!=0 ; i--){
d=d+i;
y=y+8;
f=f+2;
b=b+2;
x=x+4;
c=c+6;
e=e+2;
t=t+7;
a1=a1+2;
a2=a2+7;
a3=a3+1;
a4=a4+2;

}
```

Figure 8.8 kernel

Objectives for this task: validate the assumptions we made before on the 2 previous sections of this chapter for the vector and scalar units. Previous we assumed that there are two units for vector and scalar additions respectively. So in this experiment we are trying to validate that assumption by adding both operations in the same kernel.
Experimental setup: I made for the purposes of this task a kernel (figure 8.8) only with add operations which combines both scalar and vector data types. Important note: the loop is using an addition unit to accomplish the counter increment. Also I have to mention that each addition is independent from the others added inside the kernel and the counter comparison with zero depends on the loop counter subtraction(i--).

Experimental results: For the convenience of the reader I should mention that the x-axis states how many scalar (sc) and vector (vec) instructions I insert in the kernel at any time.
This kernel validates:

1. The number of Vector units are 2 in each pipeline
2. The number of Scalar units are 2 in each pipeline and during the execution it also uses vector units to achieve 4 additions in parallel.

In the graph figure 8.7 we see that the cycles per iteration are getting increase 37 cycles each time we add 2sc and 2vec instruction combined. Moreover in some point in the graph and the VLIW scheduling(figure 8.9) we can see that if we add 2 scalar and 2 vector and trying to add another scalar there is a cycles penalty of 37 so it is a proof that my second scenario is true. Also we managed to validate and the first scenario because when we add a third vector to be executed the we will observe the same penalty as before while we where using both pipelines inside each one of the two cores. The first addition added depends on the value of the counter (i) that is why is executed in the next VLIW instruction and not the same as the counter.

Theoretical models extracted from this experiment:

$$cycles\ per\ iteration = \big((2 * scalarAdds + 2 * vectorAdds)\ MOD\ 4\big) * 37$$

abs(scalarAdds-vectorAdds)=0-2 in order to make the formula work properly

cycles per iteratio	Scalar add	Scalar add	Vector add	Vector add
37	i--;			
74	d=d+l;	i!=0;	y=y+8;	f=f+2;
111	b=b+2;	e=e+2;	x=x+4;	c=c+6;
148	t=t+7;	a3=a3+1;	a1=a1+2;	a2=a2+7;
185	a4=a4+2;			

Figure 8.9 VLIW Scheduling

Chapter 9

Related Work

9.1 Related Work

As I have mentioned before there are some related works in the creation of Power viruses in the modern GPU's and CPU's but nothing the same have ever been done in Arm GPU's. I found some other studies that have be done on the GPU NVIDIA GeForce GTX 2080 "Power and Performance Characterization of Computational Kernels on the GPU"[15]. In that paper the authors are trying to do a power modelling and performance characterization for various types of kernels such as computationally or memory intensive ones. Also they mentioned that they were executing the kernels and changing the clock speed of the memory and processor to see the affects in power and performance. Secondly I have managed to find another one study "Highly Configurable Power Virus for GPGPUs"[16] that focuses on Power viruses development with a tool that the authors made. During the experiments they were measuring the power consumption on different NVIDIA GeForce series GPU(eg.GTX480) for various viruses made using genetic algorithms. Finally I found a related work that was about power virus development for Arm Cortex CPU's also using generic algorithms "GeST: An Automatic Framework For Generating CPU Stress-Tests"[1]

Chapter 10

Conclusion

10.1 Conclusion

This work presents the reverse engineering analysis of ARM Mali GPU T-624. The aim of this work is to reverse engineer characteristics of the GPU in order to facilitate future power virus generation targeting the specific GPU architecture. The analysis is focused on both validating publicly available features of the GPU like number of cores and number of arithmetic pipelines and also discovering new ones like pipeline depth, scheduling etc. This is achieved by monitoring the cycles per iteration and the power consumption of simple kernels. As far as we know there is no other publicly available work apart from [17], that characterizes the specific model of GPU with the aim to use the analysis for stress test generation. We extend the previous work by investigating additional microarchitectural features of the GPU. Finally, the thesis main contribution is that it provides insights regarding the maximum parallelism achieved and how is managed by the GPU.

10.2 Future Work

There are many things that can be done in the future to improve my work. First of all it will be great to see an attempt that manages to extract more about the parallelism of the Arm Mali T-624 for units I haven’t discovered such as Multiply unit, Dot product(VLUT) unit parallelism. Also it will be great to see the affect that have in the power consumption more computationally and memory expensive programs that take full advantage of all functional units such as Image processing or various benchmarks.

Further more it will be ideal to see another attempt that tries to reveal more about the power consumption and performance impact when trying to load/store on L1 and L2.
Most importantly the findings of my dissertation can be also used to create more representative stress tests(Power Viruses) for the GPU either manually or by incorporating the findings to an existing framework. An example of such framework is GeST[1][2]. GeST is an automatic framework for generating CPU stress tests. It uses genetic algorithms search and can be used to maximize different metrics. GeST, is publicly available and it's software architecture provide the flexibility to incorporate easily extra features to the framework such as support for GPU stress-tests.

10.3 Lessons Learned

One of many things I learned in the context of dissertation is that the Mali GPU Driver is intelligent enough to understand that if the code has constant input and produces no output can easily eliminate most of the code also known as Dead Code elimination.
Also even though the GPU was targeting the smartphone market and it was a low powered device and clocked in a low frequencies it can handle heavy load very well by achieving multi-way parallelism. In conclusion smartphone targeting GPU's by Arm are promising.

References

[1] Zacharias Hadjilambrou et al. “GeST: An Automatic Framework For Generating CPU Stress-Tests”, IEEE, 25 April 2019, https://ieeexplore.ieee.org/document/8695639

[2] Zacharias Hadjilambrou et al. “ Sensing CPU Voltage Noise Through Electromagnetic Emanations”, IEEE, October 2017, from https://ieeexplore.ieee.org/document/8082515

[3] The OpenCL Specification, Version: 1.2, Document Revision: 19, Khronos OpenCL Working Group, https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

[4] Ian Bratt “The ARM® Mali-T880 Mobile GPU”, IEEE, 07 July 2016, https://ieeexplore.ieee.org/document/7477462

[5] ARM® Mali™ GPU OpenCL Developer Guide, https://armkeil.blob.core.windows.net/developer/Files/pdf/graphics-and-multimedia/Guides/Arm%20Guide%20to%20OpenCL%20Programming.pdf

[6] Wikipedia Foundation. (2022, April 20). *Mali (GPU)*. Wikipedia. Retrieved May 11, 2022, from https://en.wikipedia.org/wiki/Mali_(GPU)

[7] Ganesan K, Ganesan K, Austin Uof Tat, John LK, John LK, Chicago Uof, et al. Maximum Multicore Power (mampo): Proceedings of 2011 International Conference for High Performance Computing, networking, storage and analysis [Internet]. ACM Conferences.2011https://dl.acm.org/doi/abs/10.1145/2063384.2063455?casa_token=zUjO-3s1SGMAAAAA%3AURBS-E6bGXUjOJZM6SqHbQNEzUmYpo9daD3Y3P_0PERJBOk0ECjPp3MxITYE8x3PQWSoEV7TENYt

[8] Banerjia S. Instruction fetch mechanisms for VLIW architectures with compressed encodings [Internet]. IEEE Xplore. 2002 https://ieeexplore.ieee.org/abstract/document/566462

[9] Espasa R. Exploiting instruction- and data-level parallelism. Ieeexplore.ieee.org, https://ieeexplore.ieee.org/abstract/document/621210

[10] Wikipedia Contributors. CPU cache [Internet]. Wikipedia. Wikimedia Foundation; 2019. https://en.wikipedia.org/wiki/CPU_cache

[11] Thread pool [Internet]. Wikipedia. Wikimedia Foundation; 2021 https://en.wikipedia.org/wiki/Thread_pool#:~:text=In%20computer%20programming%2C%20a%20thread,execution%20by%20the%20supervising%20program

[12] Harris P. Arm Mali GPUs Best Practices Developer Guide 2019 https://armkeil.blob.core.windows.net/developer/Arm%20Developer%20Community/PDF/Arm%20Mali%20GPU%20Best%20Practices.pdf

[13] Kaxiras S. Computer Architecture Techniques for Power-Efficiency [Internet]. Ieeexplore.ieee.org, https://ieeexplore.ieee.org/document/6812802?arnumber=6812802

[14] Learn the basics: The Midgard Shader Core [Internet]. Documentation – arm developer,https://developer.arm.com/documentation/102560/0100/Midgard-Tripipe-Execution-Core

[15] Jiao, Y., 2022. Power and Performance Characterization of Computational Kernels on the GPU, https://ieeexplore.ieee.org/abstract/document/5724833

[16] Verlinden, M., n.d. Highly Configurable Power Virus for GPGPUs. Available at: http://www.nickkelly.io/projects/papers/perf.pdf

[17] Matsentidou, E. *Ατομική Διπλωματική Εργασία* - *dms.cs.ucy.ac.cy*. ARM MALI T-624 GPU PERFORMANCE CHARACTERIZATION AND MODELING https://dms.cs.ucy.ac.cy/op/op.Download.php?documentid=16636&version=1

Appendix A - Register File Size

Scalar data type register spilling

To start with I want to I want to mention and explain the term register spilling: when the Mali Driver(Compiler) does the register assignment for each variable in a kernel when producing the machine code if the kernel has more variables than the GPU register file can handle-support then the GPU has to transfer some registers to the cache.

The main purpose was to examine the register file in the Arm Mali T-624 GPU for scalar data type and to see the penalty in cycles when the register spilling phenomenon is happening.

For the purposes of this experiment I wrote a kernel with multiply and addition pair. In this kernel every addition dependents on the result of the multiplication above it. Also the counter comparison with zero(i!=0) dependents from the result of counter subtraction. For the sake of brevity I did not put the whole kernel in the figure A.3 above, however for each subsequent operation(corresponding to the x-axis at the graph in figure A.1) I used different variables.

Experimental Results: We can see two major phases in the graph according to the cycles per iteration (orange line). The VLIW scheduling contains is presented on figure A.4

1st phase of execution(1-5 mult-adds): First pair of mult-add take 74 cycles and each subsequent mult-add takes an additional 37 cycles in order to be executed.

2st phase of execution(6-9 mult-adds): At the previous phase we saw that the penalty for each subsequent mult-add was 37 cycles. Example of register assignment for the kernel I use can be found at figure A.2. Also cycles penalty as can be seen in the figure A.5 below.

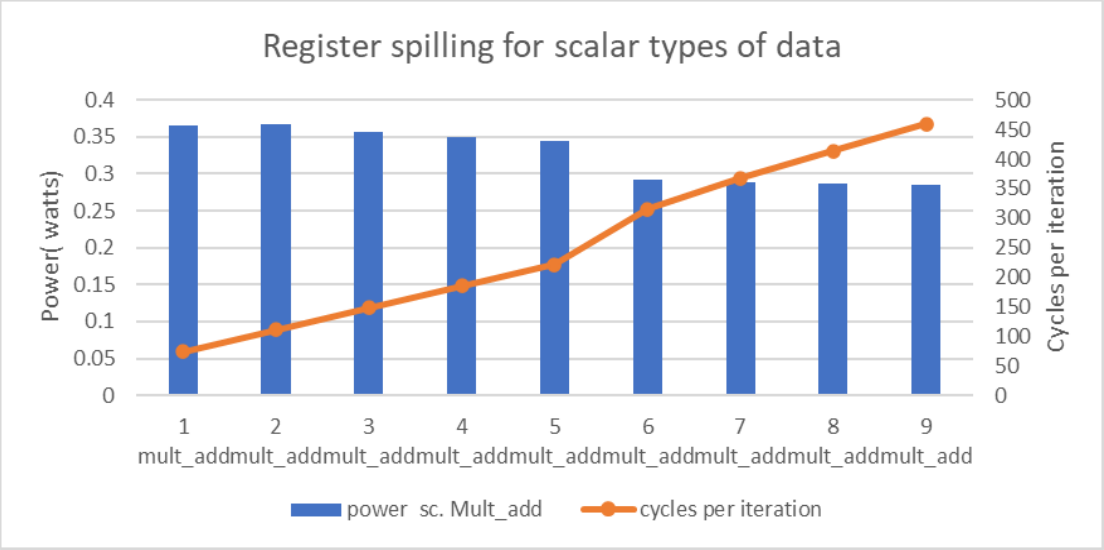


Figure A.1 Register Spilling Experiment

program variables assignment to register	
register 1	i
register 2	i!=0
register 3	y1=x1*y1;
register 4	x1=x1+y1;
register 5	y2=x2*y2;
register 6	x2=x2+y2;
register 7	y3=x3*y3;
register 8	x3=x3+y3;
The same pattern of assignment for each subsquential mult-add added in the code	

Figure A.2 Register Assignment

```
__kernel void register_spilling_scalar(__global int *list){
unsigned int x=38;
unsigned int y=38;

for (int i=100000000; i!=0; i--){
y=x*y;
x=x+y;


}
```

Figure A.3 kernel

cycles per iteration				
74	i--;			
111		i!=0;	y=y*x; x=x+y;	
148			y1=y1*x1; x1=x1+y1;	
185			y2=y2*x2; x2=x2+y2;	
222			y3=y3*x3; x3=x3+y3;	
315			y4=y4*x4; x4=x4+y4;	
368			y5=y5*x5; x5=x5+y5;	
414			y6=y6*x6; x6=x6+y6;	
460			y7=y7*x7; x7=x7+y7;	

Figure A.4 VLIW Scheduling

scalar data types		
cycles per iteration	expected cycles per iteration	register spilling penalty (real - expected)
74	74	0
111	111	0
148	148	0
185	185	0
222	222	0
315	259	56
368	296	72
414	333	81
460	370	90

Figure A.5 Cycles per iteration actual vs expected

Vector data type register spilling

Main objectives for this task was to examine the register file in the Arm Mali T-624 GPU for Vector data type and to see the penalty in cycles when the register spilling phenomenon is happening.

For the purposes of this experiment I wrote a kernel with multiply and addition pair for vector data types (figureA.7). In this kernel every addition dependents on the result of the multiplication above it. For the sake of brevity I did not put the whole kernel in the figure A.7 above, however for each subsequent operation(corresponding to the x-axis at the graph in figure A.6) I used different variables. In figure A.9 there is a VLIW scheduling for the code in figure A.7

Experimental Results: We can see two major phases in the graph on figure A.6 according to the cycles per iteration (orange line).

1st phase of execution(1 - 5 mult-adds): First pair of mult-add take 74 cycles (same as scalar) and each subsequent mult-add takes an additional 37 cycles in order to be executed

2st phase of execution(6 - 9 mult-adds): At the previous phase we saw that the penalty for each subsequent vector mult-add was 37 cycles. Example of register assignment for the kernel I use can be found at figure A.2 as it can get applied in the same way for vectors). Vector register spilling has a few more cycles penalty as can be seen in the figure A.8 (penalty cycles highlighted with red color) below.

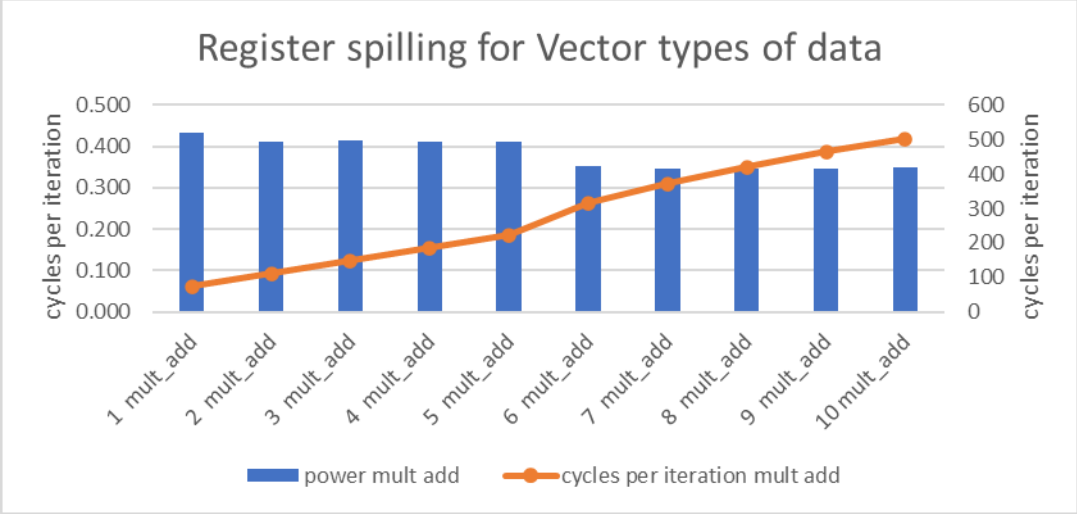


Figure A.6 Register Spilling vector data types

```
__kernel void register_spilling_vector(__global int *list){

 int4 x=(int4)38;
 int4 y=(int4)38;
for (int i=100000000; i!=0; i--){
y=x*y;
x=x+y;

}}
```

Figure A.7 kernel

vector data types		
cycles per iteration	expected cycles per iteration	register spilling penalty (real - expected)
74	74	0
111	111	0
148	148	0
185	185	0
222	222	0
315	259	56
372	296	76
419	333	86
465	370	95

Figure A.8 Cycles actual vs expected

cycles per iteration				
74	i--;			
111		i!=0;	y=y*x; x=x+y;	
148			y1=y1*x1; x1=x1+y1;	
185			y2=y2*x2; x2=x2+y2;	
222			y3=y3*x3; x3=x3+y3;	
315			y4=y4*x4; x4=x4+y4;	
372			y5=y5*x5; x5=x5+y5;	
419			y6=y6*x6; x6=x6+y6;	
465			y7=y7*x7; x7=x7+y7;	

Figure A.9 VLIW Scheduling