Ατομική Διπλωματική Εργασία

ARM MALI T-624 GPU PERFORMANCE CHARACTERIZATION AND MODELING

Ελένη Ματσεντίδου

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2020

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Arm Mali T-624 GPU Performance characterization and modeling

Ελένη Ματσεντίδου

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 2020

Acknowledgments

At this point I would like to thank all the people that helped and supported me along the way. First of all a special thanks to my Supervisor Mr. Yanos Sazeides, without which the construction of this thesis wouldn't be possible. His constant support and feedback where catalytic for my work. Also a special thanks to my colleague Adamos Ttofari, with which I worked on this topic on our summer internship at professor's Sazeides Xi Lab, and his contribution was very helpful and much appreciated. I want to thank Andreas Proxenos, my supervisor at the Xi Lab internship and a PhD student at the University of Cyprus, who was always there willing to help me with any issue I faced. And last but certainly not list I would like to thank our Partners in Arm that provided as with the Juno platform, where I conducted my experiments, and offered their help and technical support when it was needed.

Abstract

The creation of viruses targeting a processors architectural characteristics is very important and beneficial for the device's evaluation and improvement of efficiency and stability. The task I'm focusing on is the discovery of the unknown and undocumented architectural characteristics of the Arm Mali T-624 GPU, for the efficient creation of such viruses for the device.

I do this by writing and running parallel code in the GPU with the OpenCL parallel programming framework. I analyze sometimes the power consumption of my experiments but mostly the time performance, and reveal from the results the performance parameters. Then I link each of these parameters with the architecture that causes it. After the extraction of the performance parameters I attempt to fit them into a performance model that mimics the actual device's architecture as closely as possible. I validate my model by comparing the estimated execution time of my model with the actual execution time for the code.

The goal is to automatically predict the execution time of a code in the GPU accurately and understand all the parameters that affect the performance. This deep understanding of the processors structural blocks can be used for the creation of frameworks that generate processor viruses, of which the construction requires good knowledge of the device's architecture.

Contents

Chapter 1

Introduction

1.1 Problem

There are various types of viruses that are used in terms of stress testing, to improve the efficiency and the stability of the system and at the same time detect any architectural vulnerabilities and performance bottlenecks. A couple of examples of such viruses are power and voltage noise or dI/dt viruses. These viruses target processor components like CPUs, GPUs and RAMs. In order to create such programs, manually or automatically with a framework [1], is important and very beneficial to have a deep understanding of the target's architecture first. By knowing all architectural parameters and constrains of a device you would be able to tune the virus for the specific target on the best way possible. Especially in the case of voltage noise viruses [2] where in order to succeed, the program must tune to the device's resonance frequency, in order to create high levels of impedance in the power supply network. Something that requires insides on the parameters that determine the programs performance.

1.2 Contributions

The contribution that I'm attempting to make in this problem is focused on the extraction of performance characteristics from a certain GPU model, ARM's Mali T-624, which could later be used as tools for the manual or automatic creation of processor viruses for this model. More specifically I created a parametric performance model based on observations that I have also validated through experimentation and defined some guidelines on how to predict the instruction scheduling for a given code. The goal was to locate the primitive parameters of

the GPU's performance, and verify my findings by using them to predict the performance of various benchmarks and get as close as possible to their actual performance.

1.3 Outline

First of all we are going to take a look on the architectural characteristics of a GPU, its various levels of parallelism and the ways it implements them on chapter 2. After that we are going to briefly explain the basics of the OpenCL framework for parallel programming on chapter 3, which is the framework I'm using. Then we are going to summarize all the known characteristics of the GPU model I am working with and export all the unknown performance parameters that I have to locate to create the performance model I want on chapter 4. I will give an abstract of the performance model I want to make on chapter 5. On chapter 6 I will cite the methodology for my experiments and from chapter 7 to chapter 12 I will present my findings. After that, on chapter 13 I will summarize my work by presenting you my parametric performance model. On chapter 14 I' m going to compare my work with related work.

Chapter 2

GPU Architecture

A Graphic processing unit (GPU) is a processor specifically designed for parallel processing of data. They support the creation and execution of multiple threads. Threads are sequences of instructions that can be processed in parallel. In order for the GPU to support the parallel execution of threads it has various architectural characteristics that compose multiple levels of parallelism.

2.1 Core Level Parallelism

Core Level parallelism is achieved with the usage of multiple cores in one GPU. These cores share and run the work-load. The work-load is the total amount of threads that get issued to be executed in one run and is divided in smaller groups of threads often known as thread blocks. This is to make the sharing of the work-load among the cores easier. The benefit of a multicore system is that multiple independent threads can run in the GPU simultaneously in the different cores.

2.2 Pipeline Level Parallelism

Pipeline level parallelism allows the parallel execution of multiple instructions. Pipelines are a construction of a certain number of processing stages, where in each stage a different instruction can be processed. For example if we fetch an instruction in this cycle then in the

next cycle we can go on and fetch the next instruction while the previous instruction continues to the next processing stage. The number of these stages in a pipeline is called the pipeline depth which is the number of clock cycles that it takes for an instruction to be fully executed, considering the fact that there were no stalls. Stalls are clock cycles when the pipeline is not able to move forward, often due to data dependencies among instructions or other hazards. Pipeline depth also describes the number of different instructions that the pipeline is able to process in parallel in one cycle. It's also important to note that each core can often have multiple pipelines, providing an extra level of parallelism, with instructions not only processed in parallel inside the pipeline but also simultaneously in the multiple pipelines.

Now according to the policy that a pipeline follows in terms of the order in which it fetches new instructions, we can say that it implements a certain multithreading execution. Some types of multithreading executions are the coarse-grained multithreading and the fine-grained multithreading. In coarse-grained multithreading at each cycle we fetch instructions from the same thread until the thread is finished or blocked when we continue to the next one. In fine-grained multithreading at each cycle we fetch an instruction from a different thread often following a round-robin type of scheduling (figure 2.1).

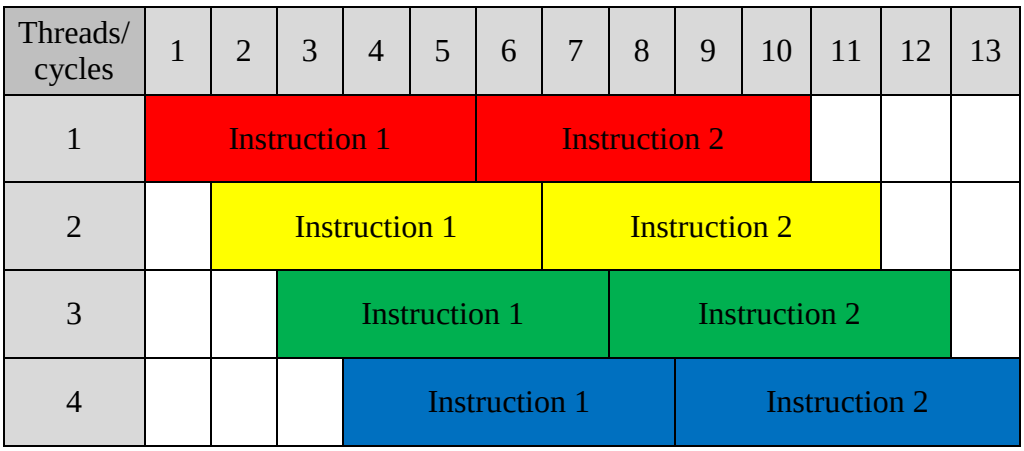

Figure 2.1 Fine-grained multithreading execution example

2.3 Instruction level parallelism

Instruction level parallelism we have when we are able to execute multiple instructions simultaneously. Each core, or even each pipeline can contain a number of different functional units. This allows the simultaneous execution of a number of different arithmetic operations. There is a technique that utilizes this ability called instruction fusion, where a small group of sequential instructions, usually a pair, can be fused in one only instruction [3]. This instructions can even have dependencies.

Another form of instruction level parallelism is the Very Long Instruction Word (VLIW) architecture. As the name suggests, in this architecture Instruction words are unusually long, often fitting 4 “normal size” instruction words. Therefore it gives the ability to execute simultaneously a set of dependent instructions instead of one, as long as available resources exist. The basic idea is that each operation in one VLIW is mapped to a different functional unit (figure 2.2). The process of grouping these independent operations is usually implemented at the compilation stage.

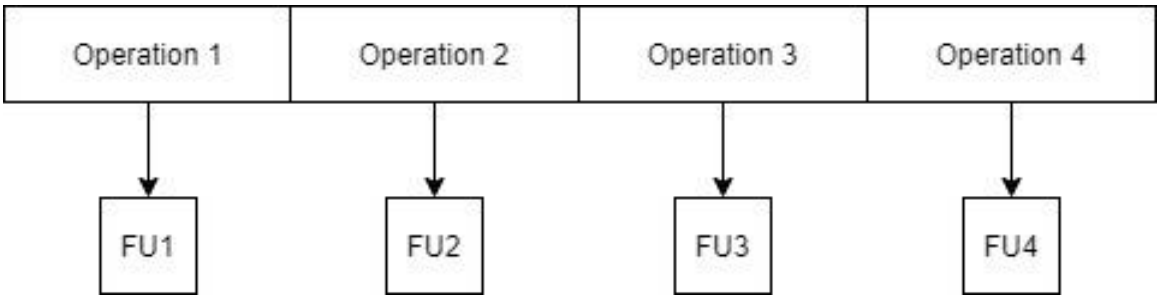

Figure 2.2 Very Long Instruction Word simple scheme

2.4 Data level parallelism

A partitioning register file architecture, is when the register file is partitioned into smaller RF. This technique can be beneficial for both VLIW and SIMD architectures, depending on the level that the file is partitioned and organized [4]. A Single Instruction Multiple Data (SIMD) architecture provides a data level parallelism. With SIMD we can group multiple data to which we want to apply the same operation and execute them all at once. This is something that usually would require from the programmer to explicitly group these data into vectors and use certain functions to apply the operations, but some processors offer the ability of auto vectorization where they are able to locate opportunities to merge multiple similar instructions into a single one that contains multiple data, at the compilation state.

2.5 Cache memory

GPUs of course have multiple levels of cache, with each level having a different size, inversely proportional to the distance of the cache from the functional units. What is interesting to note here is that the level 1 cache size for a GPU is typically smaller than a CPU’s but with a much higher bandwidth. Each cache can either be accessible only to a certain core or commonly shared across the whole GPU. Some architectures use the multiple level caching system to implicitly define the memory that is accessible by a thread or a group

of threads and the memory that is shared between all threads. With this technique we can ensure that the memory that a thread needs is physically closer and ready for access. Also caches can physically partition texture memory, which is used for graphics rendering, from general memory. These concepts of memory division are very common in parallel programming and I'm going to discuss more about this from a programming aspect later on.

Caches are made of blocks, also known as cache lines, and these blocks are the smaller transferable cache units. These blocks are grouped in block sets, also known as cache rows, and within those rows each block has a different tag to identify it. The number of blocks within a block set defines what we call the associativity of the cache. Addresses in cache contain the tag of the block the data belongs, the index of the cache row we can find the block if it would be in the cache and the offset within the block of the data we want (figure 2.3). When requesting a data block the cache looks in the cache row with the corresponding index and checks for the tag and the valid bit, that determines if the data in the block are valid and up to date to make a cache hit. If it fails we have a cache miss and we have to look in the next level of the cache hierarchy.

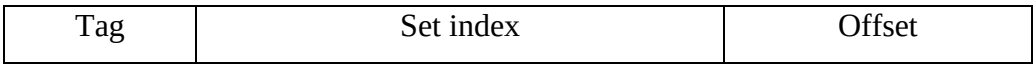

Figure 2.3 typical cache address example

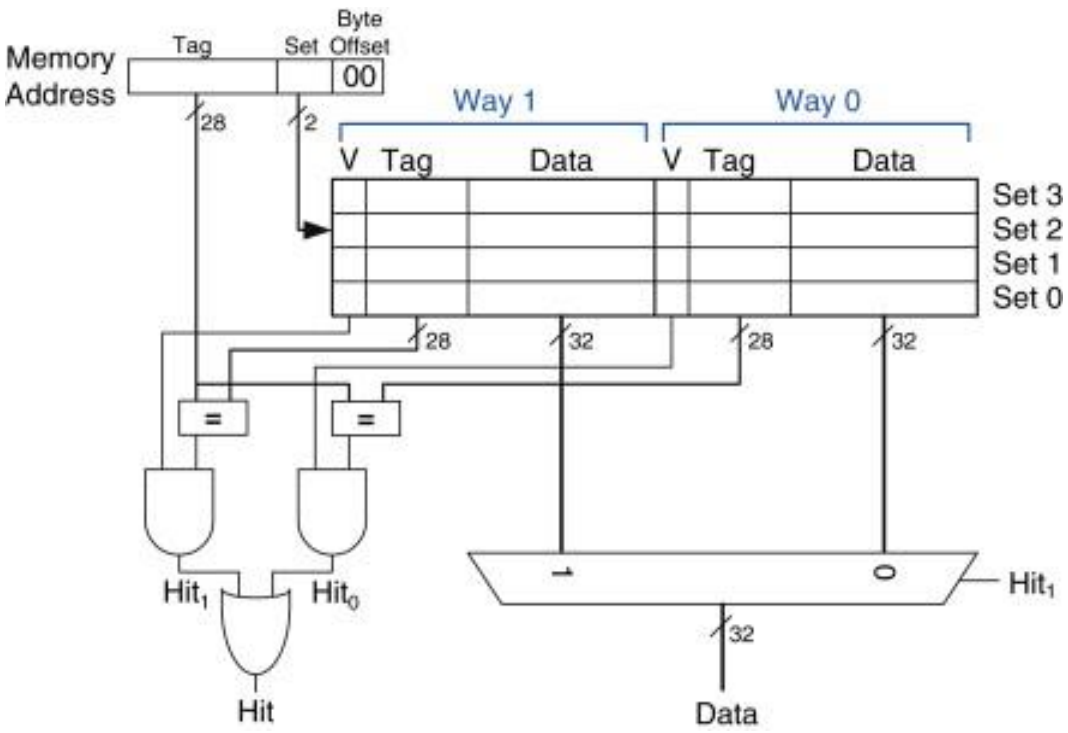

Figure 2.4 2 way association cache example from [5]

In GPUs there are some common cache issues that could arise from having multiple threads having access to the same data. One issue is cache coherence, where we would want multiple instances of the same data in different caches to be consistent. Some architectures attempt to ensure coherency and others don’t, leaving it completely up to the programmer to protect the data coherency. Another problem is the race condition issue, where two or more threads attempt to access the same data in memory at the same time, which could cause delays. In extend to the race conditions, there is also the false sharing issue where threads try to access, not the same data but instead data within the same block, but delays still arise.

Chapter 3

OpenCL

3.1 OpenCL Models

Arm Mali GPUs support the use of OpenCL and OpenGL frameworks for parallel programming. For the purposes of my research I Used the OpenCL framework, as OpenGL is actually more of a 3D graphics API. I'm using the 1.2 version of OpenCL and all my information is based on the official specification of OpenCL [6]. The Platform Model of OpenCL (figure 3.1) contains the host, where the main application runs, multiple devices where the host sends commands, compute units within the devises and processing elements within the compute units that execute the commands. In the Execution Model we have the host program that runs in the host and kernels that can run in one or multiple devices. When the host submits a kernel for execution in one or more devises, multiple instances of the kernel are created. These instances are called work-items, commonly known as threads, and each one is identified by its unique global ID. The work-items are grouped into work-groups each one having its own work-group ID and assigned in a specific compute unit. Also the work-items within a work-group have a local ID, unique to every work-item within the group. The work-group size of the groups, which is the number of work items within a work-group can be explicitly defined from the programmer or the OpenCL will make an automatic partition according to the total work-load.

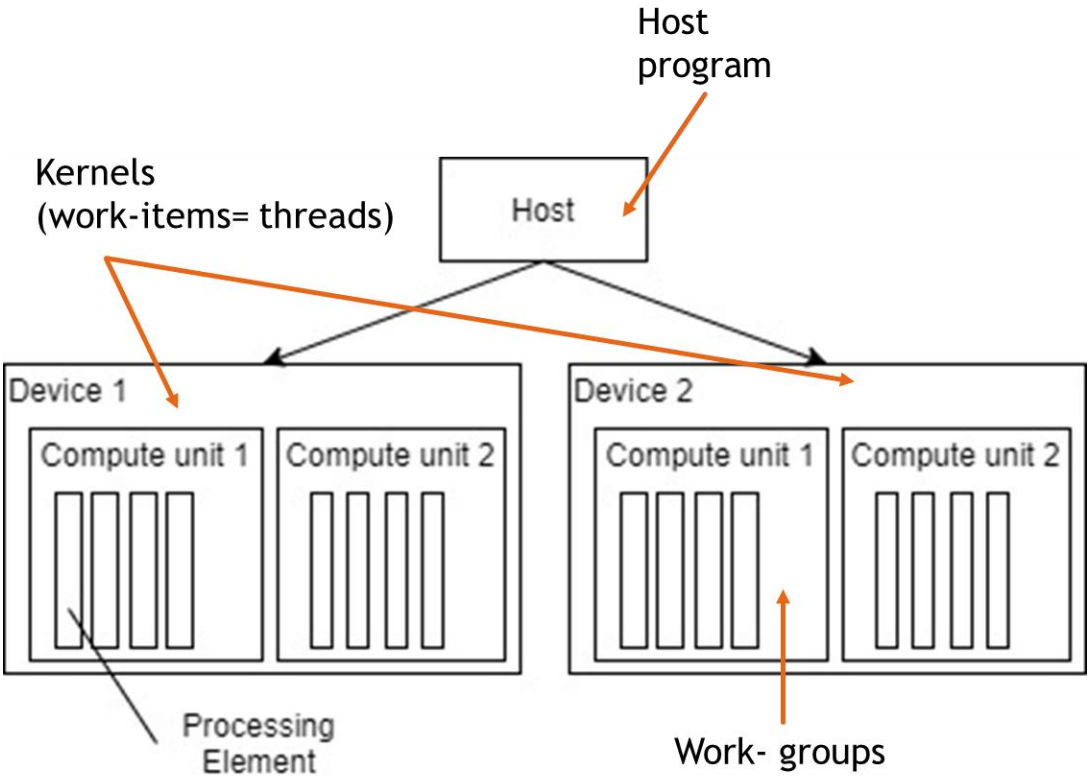

Figure 3.1 OpenCL Platform Model

The Index Space in OpenCL can be multidimensional, but I am not going to elaborate on this as for my experiments I only used one dimensional index spaces.

The OpenCL Memory Model contains the global memory which is memory accessible to all work-items, the constant memory which is global memory that remains constant throughout the execution, the local memory accessible to the work-items of a certain work-group and private memory accessible to a certain work-item. OpenCL uses memory objects that can be buffers or images, to pass data from the host to the kernel.

3.2 Data Types and Instructions

OpenCL supports a variety of both scalar and vector data types. Some basic scalar types are Booleans (bool), signed and unsigned characters (char/uchar), signed and unsigned arithmetic types like short integers (short/ushort), integers (int/uint), long integers (long/ulong), and also floats and doubles. For the corresponding vector types we simply add to the end of the type names the number n of the values we would like for this vector to have, where n takes the values of 2, 3, 4, 8 and 16. OpenCL also supports other data types that I'm not going to include as we are not going to need them. More information about OpenCL's documentation is available at the Khronos Group OpenCL Specification [6]

.

Chapter 4

ARM Mali T-624

In this chapter I will summarize all architectural characteristics that I managed to find from references of the model I'm working with. Then according to chapter 2 I will define all the unknown architectural parameters that I have to discover through experimentation, in order to compose my parametric performance model**.**

4.1 Known ARM Mali GPU Characteristics

The GPU model that we work with is Arm Mali T-624. The model belongs to the family of Midgard core GPUs. This Midgard family can have 1 to 4 active shader cores that from now on will be mentioned simply as cores. Each core contains a system of two arithmetic pipelines, one load/store pipeline and one texture pipeline. All pipelines use fine-grained multithreading execution. Each arithmetic pipeline has multiple functional units like multiplication unit and addition unit for both scalar and vector operations and a vector look up table unit. The Job Manager is a Unit that is responsible for the scheduling of the work-load in the GPU. There are two Levels of caching, with two caches in level one (L1) and one cache on level two (L2). The first L1 cache is used for general memory access and the second one for texture access, with each of size 16 KB. The L2 cache is shared through all the cores and its size varies from 32KB for each core to 64 KB (128 KB – 256 KB in total). Both cache levels are made of 64 byte cache lines. The midgrade family also has a very long instruction word (VLIW) architecture that allows the simultaneous execution of 4 different operations in different processing units.

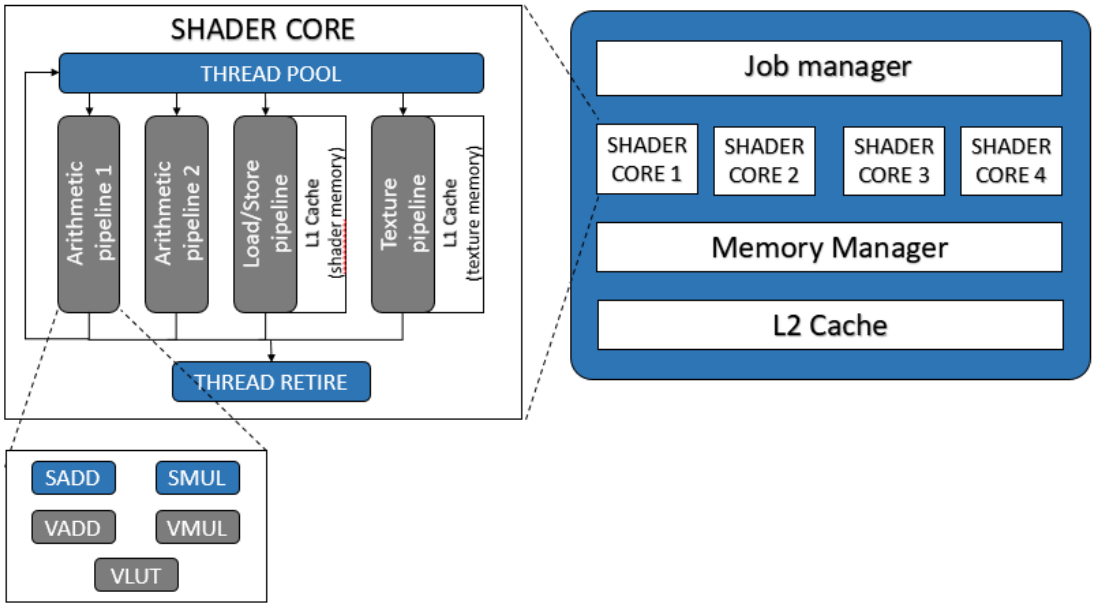

Figure 4.1 Mali midgard family Micro-Architecture

4.2 Unknown parameters to discover

In order to make complete a performance model we would have to examine and try to approach more characteristics of the Mali GPU. At the same time it would be wise to also try to verify the already knowing characteristics. One of the most important things to do is to verify the fact that it uses fine-grained multithreading and find he pipeline depth, but also verify the number of arithmetic pipelines and if both pipelines are active in all cases. Also I would have to discover the way the Job Manager schedules the work-load on the GPU. I have to determine the level of instruction parallelism that is available, more specifically find out whether there is instruction fusion and in which cases and identify the VLIW instruction level of parallelism constrains, in order to approach as closely as possible the number of VLIW instructions that result given a certain code. Also I need to find out the size of the register file to be able to predict when we are going to have a register spill. A register spill is the phenomenon where the number of registers in a program exceed the number of registers that are available and is forced to turn to the cache. Moreover I have to discover the delay of a cache load and store and check if the GPU attempts to reserve coherency and how, as well as approach the cost of race conditions and false sharing if there is any and find the delay of a cache miss in L1 and L2. With all these characteristics clarified I would be able to form an accurate performance model with parameters only defined by the programmer that doesn't require from the user any insides on the processors characteristics besides the GPU's frequency.

Chapter 5

Abstract performance model

5.1 Abstract performance model

My Goal as I have explained, is to create a performance model of the Arm Mali T-624 GPU. To achieve this I have to brake the model into 3 basic components. The first component will be responsible of the VLIW scheduling prediction. This is important, because to predict the execution time for a given kernel we have to know the number of very long instructions in which the compiler translates the code. The second component is the job manager. Given the total work-load and the size of the work-groups, which both can be set by the programmer we have to predict the partitioning of the work-items among the cores. Lastly the final and main component is the parametric performance model that based on the architectural parameters of our model, the VLIW scheduling and the job manager scheduling will estimate the execution time for the given Kernel.

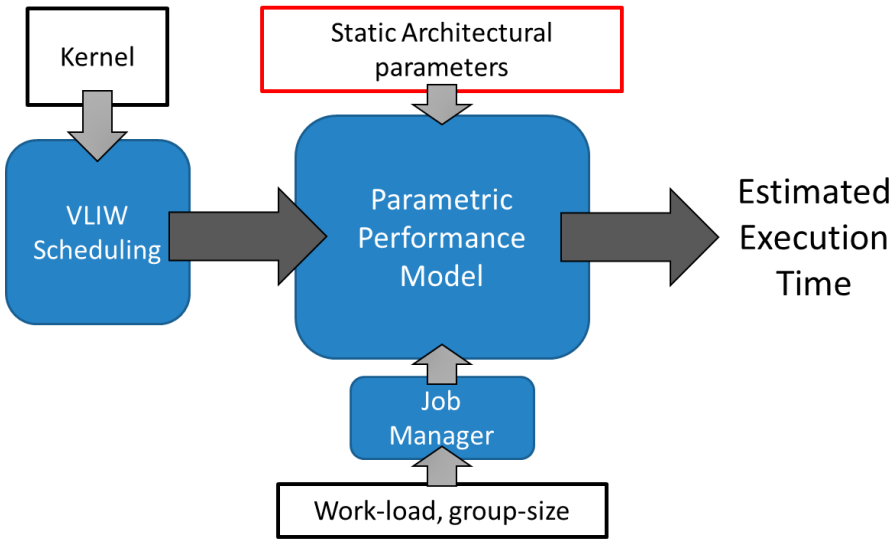

Figure 5.1 Abstract performance model

Chapter 6

Methodology

6.1 Methodology

The Basic idea around the methodology that I followed is that I'm starting with a simple performance model and as I'm moving on I add to the model the various architectural parameters that I managed to locate. This basic model is as follows$\frac{I \times CPI}{F}$, where I is the total number of instructions, CPI is the clock cycles per Instruction and F is the frequency. The way I extract all the additional parameters is through experimentation and observation of a kernel's performance and power consumption. For my experiments, as I have already said, I'm using the OpenCL Framework. I'm running my experiments on the Arm's Juno board which contains a model of Arm Mali T-624 GPU. The OpenCL Host is the CPU of the Juno board, device which I'm using only one is the GPU and the computing units are the 4 cores. I'm using a one dimensional index space. I have written a few host programs, each one with a few different parameters like for example the number of memory objects I pass to the kernels. The memory objects I use are buffers and always allocated in global memory. The reason I am doing this is that in the ARM® Mali™ GPU OpenCL Developer Guide [7] it states that local and private memory are allocated in global memory, therefore the allocation of data in different memories won't affect the performance.

Each experiment contains one or multiple kernels that I execute for different number work-groups and work-group sizes. Each experiment is designed to provide me with certain information and aims on specific parameters that I want to determine. Also when I'm building my kernels I'm using the "-cl-opt-disable" option which disables compiler optimization in order to get more predictable results and be able to analyze them easier. For every kernel I make VLIW scheduling prediction for every iteration of the kernel to estimate the number of instructions per loop.

Then I'm analyzing the experiment results of the power consumption or the performance, which is the execution time. Usually from the performance results I calculate the actual number of clock cycles that it took to the kernel to be executed with the formula $T \times F = CCY$, where T is the actual execution time, F is the GPU's frequency and CCY are the total clock cycles for the execution. Note here that the GPU's frequency is always locked at a certain value, as the DVFS (dynamic voltage and frequency scaling) is inactivated. After that I make assumptions about the architectural characteristics of the GPU and define extra parameters that I add in my performance model, or use them to provide guidelines on how to accurately predict the instruction scheduling for a code. To validate these parameters I'm comparing the estimation time that results from my new performance model with the actual execution time of the experiment I conducted.

To summarize, the steps that I follow for each experiment are the following:

1. Composition of kernel
2. Execute Kernel for a different number and sizes of work-groups and take measurements for the power consumption or the performance.
3. Calculate the number of clock cycles for each execution.
4. Observe results and extract information about the underlying architecture and update the parametric performance model, VLIW scheduling and Job Manager Scheduling methods when and if possible.
5. Check the estimation time of the new updated model with the actual performance of the kernel.

Chapter 7

Arithmetic pipeline

7.1 Number of Arithmetic Pipelines

I have observed that the second of the two arithmetic pipelines can be used only when there are work-groups with two or more work-items. When executing a kernel of a simple increment loop (figure 7.1) for different work-group sizes, and measuring the power consumption, we get the results shown in figure 7.2. There is a clear difference in the power consumption of work-groups with size one and all the rest of the work-group sizes, which seem to consume similar amounts of power. This could mean that the instructions of each work-group can only be executed in the same pipeline.

```
__kernel void simpleLoop(){
for(int i=0; i<1000000; i++){
    //do nothing
}
```

Figure 7.1 Increment loop Kernel

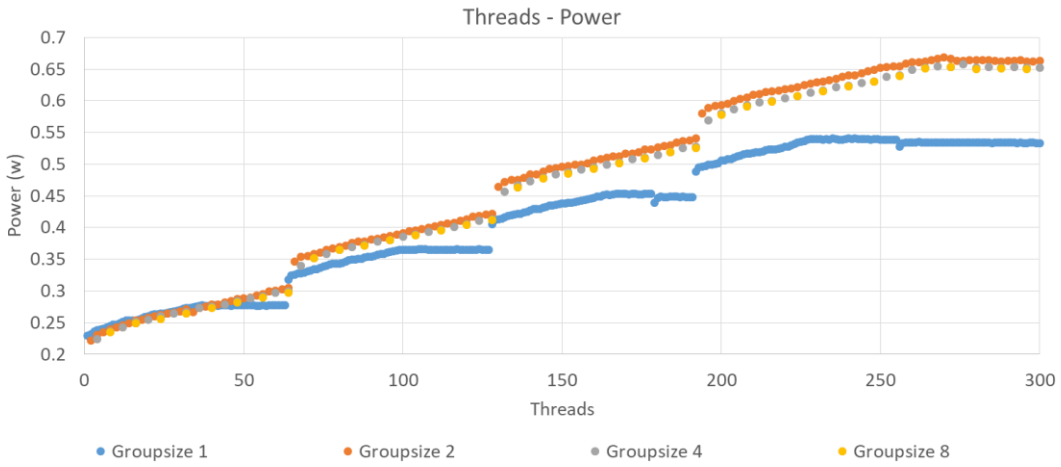

Figure 7.2 Power consumption

If we compare the execution time of work groups with size 2 and 1 we can see that in the case of group size 2 it executes the double amount of work in almost the same time. In some cases is less due to the fine-grained multithreading that we are going to examine next.

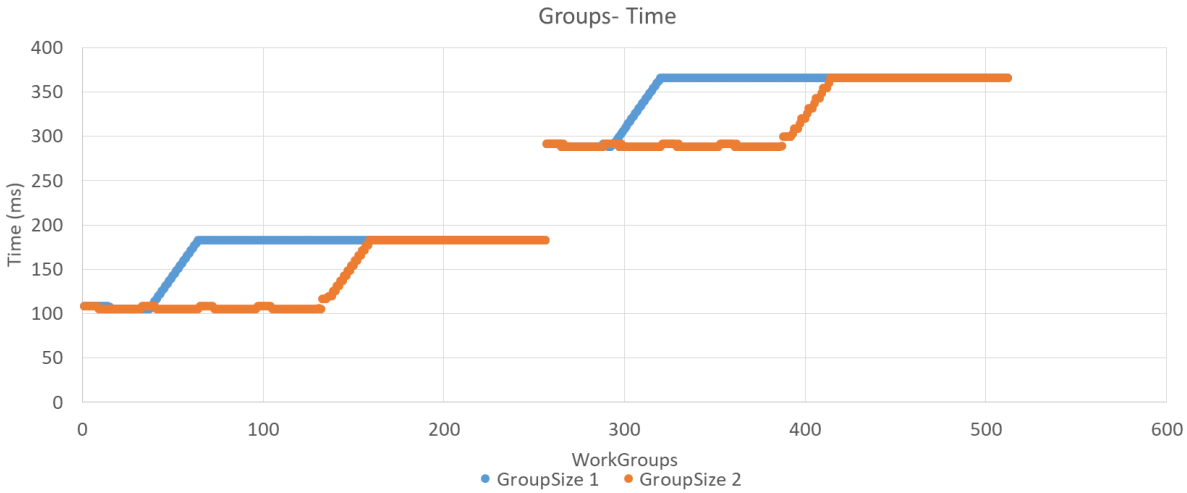

Figure 7.3 Performance comparison between size 1 and size 2

7.2 Fine Grained multithreading and pipeline depth

Our goal is to verify if our machine uses Fine Grained multithreading and what the pipeline depth is. For this purpose I used again the simple increment loop is a simple increment loop with two instructions in each iteration. One instruction incrementing the variable and another comparing it with the termination value. When running the same code for different number of work-groups of size 1 and measuring the execution time we get the following results

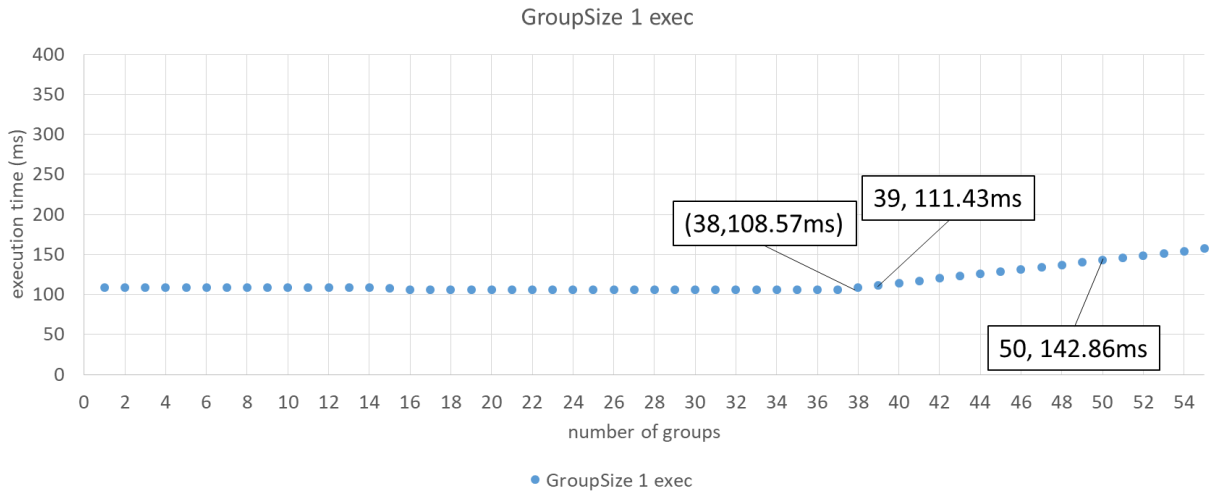

Figure 7.4 Performance of 1 work-item per work-group

We observe that the execution time remains stable when the work group number is 1 to 38. This indicates that it uses a Fine Grained multithreading execution and that the pipeline depth is 38. More specifically the pipeline in every cycle fetches instructions from different threads. But in order to fetch the next instruction of the same thread the previous instruction must fully finished. Meaning that by the time that the first instruction of the first thread finishes we can issue 38 instructions more, each of a different thread. In the following figure we can see an example of how these instructions are scheduled for 3 and 38 work-items

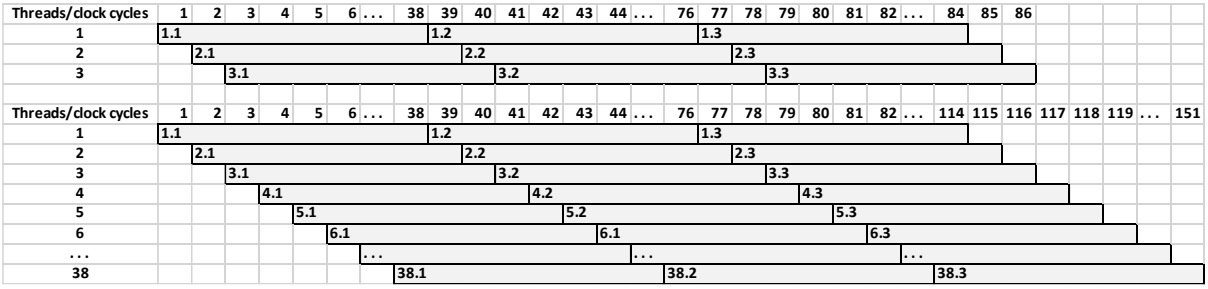

Figure 7.5 Instruction scheduling for a kernel with 3 instructions for 3 and 38 threads (work-items), where instruction 1.2 is the second instruction of the first thread etc.

When the number of threads exceeds 38 then the pipeline depth, can no longer “hold” the number of threads running in parallel. Therefore there will be a 1 cycle stall in the thread execution for each instruction with every extra work-item, like the following examples.

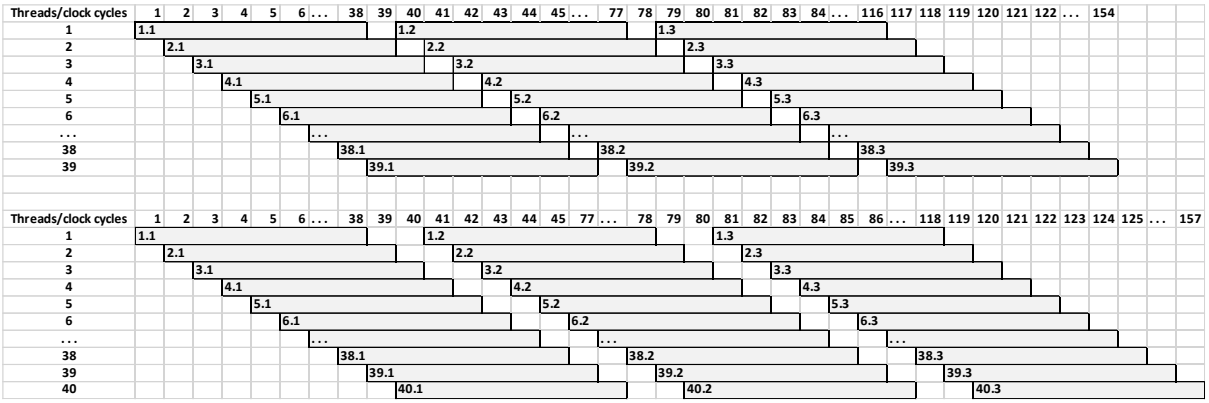

Figure 7.6 Instruction scheduling for the first 3 instructions with 39 threads and 40 threads/ work-items.

To validate my assumption, I compare the expected execution time with pipeline depth 38 and the actual time for all the observed scenarios we see that the assumption is valid. On my performance model I replace the CPI with the pipeline depth (pd) plus the number of stalls. Some examples are the following.

For 1 to 38 work-groups: $\frac{(2\times10^6)\times 38}{700\times10^6} = 0.10857s = 108.57\ ms$

For 39 work-groups: $\frac{(2\times10^6)\times 38+((39-38)\times 2\times10^6)}{700\times10^6} = 0.11143s = 111.43\ ms$

For 50 work-groups: $\frac{(2\times10^6)\times 38+((50-38)\times 2\times10^6)}{700\times10^6} = 0.14286s = 142.86\ ms$

Chapter 8

Job Manager

8.1 Work scheduling between cores

As I have mentioned before for every 64 work-items running in parallel a core gets activated. To be more specific the way that the Job manager schedules the work-load of each core is by dividing it into batches of 64, regardless of the work-group number or size. Looking back to the experiment power consumption experiment in chapter 7 (figure 7.2), for every work-item that we add to the work-load the power consumption increases linearly and for every 64 work-items there is an even bigger increment indicating the activation of a core. This happens 4 times, for each of the 4 cores and then the power consumption stabilizes.

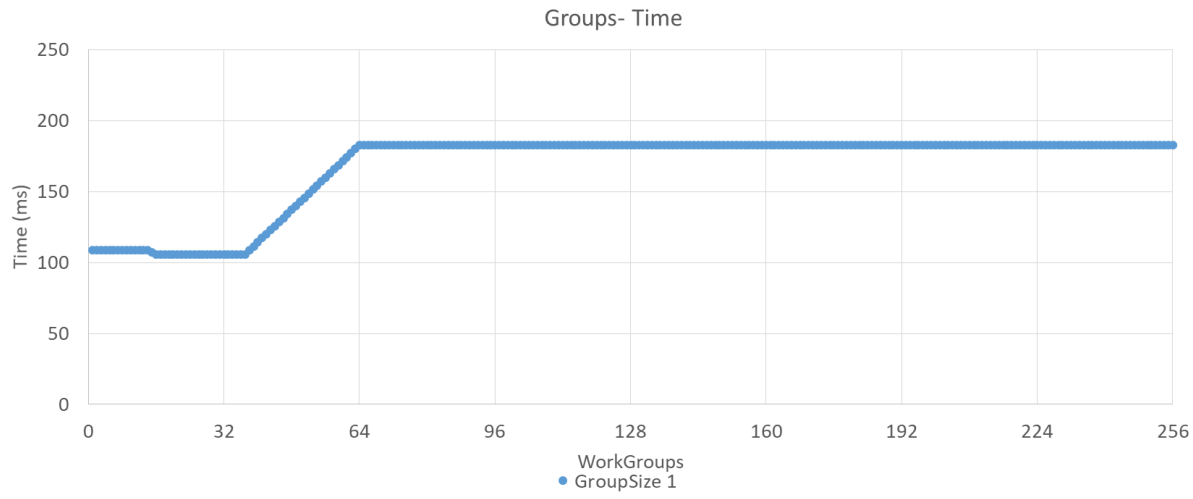

Figure 8.1 Performance for 1 work-item per work-group

This can also be seen at the time performance (figure 8.1). After the addition of the 64th work-item the execution time remains the same, as it is bound to the time that the core with

the larger amount of work needs to finish. This phenomenon can also be noticed in the execution of much more complex benchmarks, but for simplicity I cite the same simple example as we have already seen. Bellow there is a graphic representation of how the work-items get issued in the cores and how that affects the execution time, for a number of different work-group sizes

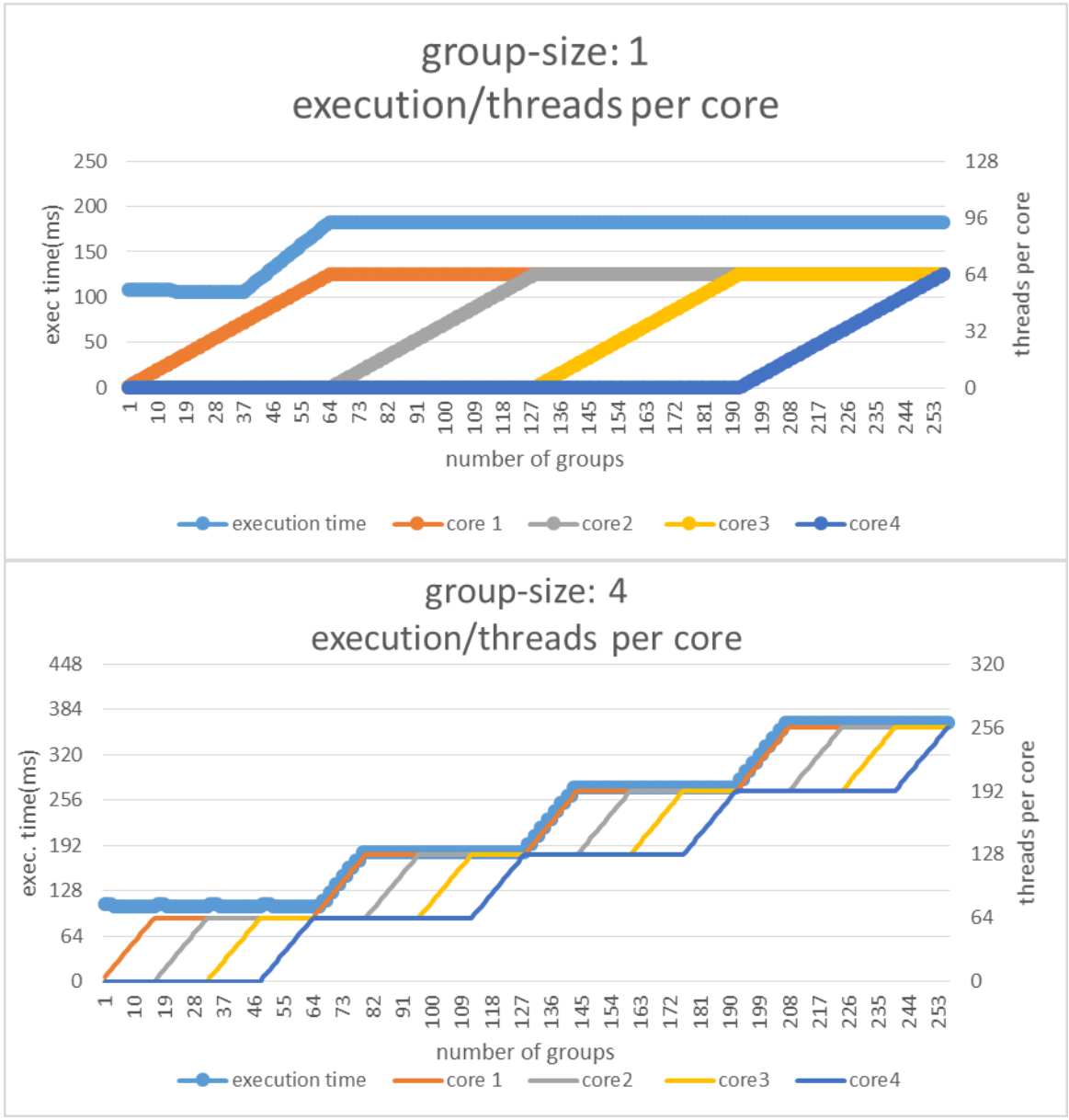

Figure 8.2 Execution along with number of threads (work-items) per core

8.2 Scheduling batches Over GPU

The GPU runs a limited amount of threads in parallel. It divides the work-load into batches, and issues the next batch only when the execution of the previous one is completely finished. These batches are consisted of 256 work-groups, unless the total number of work-items in the

256 work-groups is over 1024. Let's say that the number of work-groups in a batch is n. The time performance of the simple increment loop shows that the execution time for n+1 work-groups is equal to the execution time of n work-groups plus the execution time of 1 work-group. This is true for all the possible power of 2 work-group sizes (figure 8.3). Therefore is clear that the GPU has to complete the execution of one batch in order to continue to the next batch issued by the job manager.

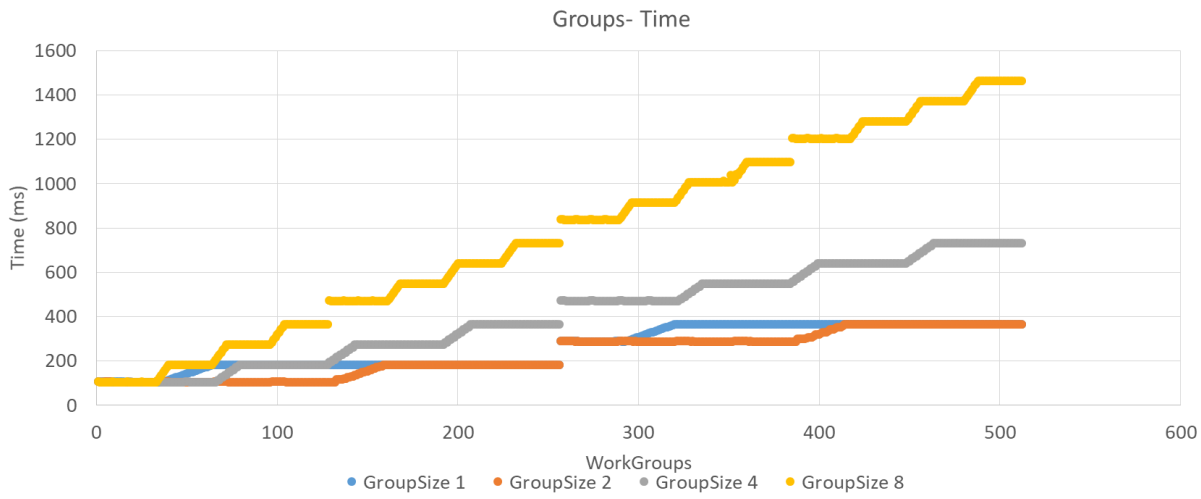

Figure 8.3 Compare Performance for different number of work-group sizes

Chapter 9

Fusion of depended operations

In most cases the processor is able to make various optimizations on the operations to improve performance. In this chapter we test the ability to fuse certain instructions that normally would have dependencies.

9.1 Logic Operation Fusion

When executing the following code (figure 9.1), we would expect the number of VLIW instructions for each iteration to be 3 as you can see in the scheduling figure that follows (figure 9.2). Yet when executing the code the execution time that we get is 126.668ms, which translates to 38 clock cycles (1 pipeline cycle) less than we would expect.

```
__kernel void func1(__global int* A){
    int x = 38;
    for(int i=0; i<1000000 && x!=0; i++){
        x++;
    }
}
```

Figure 9.1 Kernel with two comparisons

i = 0	x++		
i<1000000	x!=0		
&&			

Figure 9.2 VLIW scheduling for kernel

The only logical explanation is that the processor can fuse a couple of operations into one. If we suppose that the operations that get fused in this example are the “AND” and the “CMP” operations and run a series of examples with this two used at the same way we get the same consisted results. Therefore we can come to the conclusion that the logical processing unit has the ability to fuse small sequential operations with dependencies, in one instruction.

9.2 Multiply and Add Operation Fusion

In a similar way with the logical operations fusion we also have a fusion between operations of the multiplication and addition unit. In the following example (figure 9.3), we have a couple of depended multiply and add operations. The VLIW instruction scheduling of one iteration for this benchmark is as shown on figure 9.4.

```
__kernel void mult_add_2(__global int* A){
    unsigned int x = 38;
    unsigned int y = 28;
    for(int i=0; i<1000000 && x!=0; i++){
        y = x * y;
        x = x + y;
        y = x * y;
        x = x + y;
     }
}
```

Figure 9.3 kernel with multiply and add operations

i = 0	y = x * y_ x = x + y		
i<1000000	y = x * y_ x = x + y		
&&_ x!=0			

Figure 9.4 VLIW scheduling for kernel

The actual execution time is 190.002ms, for frequency set to 600MHz and running one work-group with one work-item. The estimation time from my performance model is 190ms as well, $\frac{(3\times10^6)\times 38}{600\times10^6} = 0.19s$

Chapter 10

Instruction Level Parallelism

On this chapter we will explore and verify the opportunities of instruction level parallelism the specific model allows with the VLIW architecture.

10.1 Scalar units

With the exceptions of the operation fusions that we have already covered, instructions with dependencies cannot coexist in the same VLIW instruction. For this experiment we are going to use the kernel of the previous chapter (figure 9.3) and we will constantly add multiply-add operations and measure the performance. For every couple of multiply and add operations we add the performance for 1 work-item gets charged with an additional 38 clock cycles, with exception the kernel with 6 multiply add couples, where we have a register spill. More about the register spill is on the next chapter

Number of multiply-adds	Execution time	Cycles per iteration
2	190.002	114
3	253.338	152
4	316.670	190
5	380.001	228
6	536.670	322

Figure 10.1 Execution time graph for 1 work-item of size 1

If now we examine a similar scenario, but this time with couples of instructions independent from one another (figure 10.2), the instructions can feet to the same VLIW word (figure 10.3). This causes the phenomenon of auto vectorization, where scalar instructions can be executed as vectors. The estimation of execution time is $\frac{(3 \times 10^6) \times 38}{600 \times 10^6} = 0.19s$ and the actual time is 190.002ms.

```
__kernel void func2(__global int* A){
    unsigned int x = 38;
    unsigned int y = 28;
    unsigned int z = 58;
    unsigned int k = 48;
    for(int i=0; i<1000000 && y!=0 && k!=0; i++){
        x = y * x;
        y = y + x;
        z = k * z;
        k = k + z;

        x = y * x;
        y = y + x;
        z = k * z;
        k = k + z;
    }
}
```

Figure 10.2 Kernel with independent couples of multiply add instructions

i = 0	y = x * y_ x = x + y	z = k * z_ k = k + z	
i<1000000	y = x * y_ x = x + y	z = k * z_ k = k + z	
&&_ y!=0			
&&_ k!=0			

Figure 10.3 VLIW instruction Scheduling

But the VLIW word has a size constrain. It is large enough to fit 4 different instructions. There for in the following example (figure 10.5), although the processor has enough resources to execute the increment and the comparison of variable i at the same time with the 4 multiply and add operations, it won't execute them all as they don't fit in the VLIW word (figure 10.4). The estimation of execution time is $\frac{(5 \times 10^6) \times 38}{600 \times 10^6} = 0.3167s$ and the actual time is 316.67ms.

i = 0	y = x * y_ x = x + y	z = k * z_ k = k + z	f = g * f_ g = g + f
i<1000000	h = j * h_ j = j + h	y = x * y_ x = x + y	z = k * z_ k = k + z
f = g * f_ g = g + f	h = j * h_ j = j + h		
&&_ y!=0			
&&_ k!=0			
&&_ g!=0			
&&_ j!=0			

Figure 10.4 VLIW instruction scheduling

```
__kernel void func4(__global int* A){
    unsigned int x = 38;
    unsigned int y = 28;
    unsigned int z = 58;
    unsigned int k = 48;
    unsigned int f = 78;
    unsigned int g = 68;
    unsigned int h = 98;
    unsigned int j = 88;
    for(int i=0; i<1000000 && y!=0 && k!=0
 && g!=0 && j!=0; i++){
        x = y * x;
        y = y + x;
        z = k * z;
        k = k + z;
        f = g * f;
        g = g + f;
        h = j * h;
        j = j + h;

        x = y * x;
        y = y + x;
        z = k * z;
        k = k + z;
        f = g * f;
        g = g + f;
        h = j * h;
        j = j + h;
    }
```

Figure 10.5 Kernel with 4 multiply-add instruction couples

10.2 Vector Units

In contrast to the scalar data, each core cannot process more than one SIMD instruction at the same time. The following examples show the execution of multiple multiply-add operations independent to one another (figure 10.7). The VLIW word is as follows (figure 10.6). The estimated execution time of this scheduling is the same with the actual execution time.

i = 0	y = x * y_ x = x + y		
i<1000000	z = k * z_ k = k + z		
f = g * f_ g = g + f			
&&_ y!=0			
&&_ k!=0			
&&_ g!=0			

Figure 10.6 VLIW scheduling

```
__kernel void vec_func3(__global int* A){
    int4 x = (int4)38;
    int4 y = (int4)28;
    int4 z = (int4)58;
    int4 k = (int4)48;
    int4 f = (int4)78;
    int4 g = (int4)68;
    for(int i=0; i<1000000 && (all(y!=(int)(0))) &&
 (all(k!=(int)(0))) && (all(g!=(int)(0))); i++){
        x = y * x;
        y = y + x;
        z = k * z;
        k = k + z;
        f = g * f;
        g = g + f;
    }
}
```

Figure 10.7 Kernel with independent SIMD multiply-adds

Chapter 11

Register Files

On this chapter I would like to explore and identify the register file size, and how scalar and vector variables get registered

---

11.1 Register File Size

To check the size of the Register file I need to create a kernel that could causes a register spill. To be more specific I'm writing kernels with multiple independent operations, each utilizing a different register. At each kernel I'm increasing the number of instructions and as a result the number of Registers the kernel requires and try to find out when a register spill will occur. We have the same kernel as in chapter 9 (figure 9.3) as a base and to that I'm adding a multiply-add operation for each run, which as we said is a fused operation. For each multiply-add operation the kernel requires two extra registers, therefore according to the analysis of the registers for the base code iterations(figure 11.1), a kernel with n multiply-add operations would require 5+2n registers.

```
(x,y & i from previous loop)
R1 = i+1
R2 = R3 < 1000000
R4 = x!=0
R5=R3&&R4
R6=x*y
R7=x+R6
R8=R7*R6
R9=R7+R8
```

Figure 11.1 Register Analysis

From the performance results in comparison with the expected time from my performance model (figure 11.2) we find that the register spill is happening for the first time at the 6th multiply-add operation where the kernel requires 17 registers. So in conclusion the register file has 16 registers for each work-item and 64x16=1024 registers per core. To verify we conduct the same experiment for deferent data types and the results (figure 11.2) show that in all cases the register spilling happens when we reach the 17th register.

Number of multiply-adds	integer	float	double	estimated time
2	190.002	190.002	190.002	190.000
3	253.338	253.337	253.336	253.333
4	316.670	316.670	316.670	316.667
5	380.001	380.001	380.001	380.000
6	536.670	536.670	536.670	443.333
7	631.670	631.670	631.670	506.667
8	711.670	711.670	711.670	570.000
9	790.001	790.001	790.001	633.333
10	855.001	855.001	853.339	696.667

Figure 11.2 Performance results for different data types

11.2 Scalar and vector variables in register file

To understand how vectors are stored in the register file in comparison to scalar variables and if they there is a separate register file for vectors or not we are going to make the following experiments. Similarly to the previous chapter we have a base code to which we add a multiply-add operation at every run. But this time I'm using vector type variables instead (figure 11.3). The results show that the register spill is happening at the exact same point as for the scalar types (figure 11.4). Therefore the vector and scalar data share the same file, and the registers are big enough to fit 128bit vectors. So in conclusion the register file for each core has a size of 1024x128bit.

```
__kernel void mult_add_2(__global int* A){
    int4 x = (int4)(38);
    int4 y = (int4)(28);
    for(int i=0; i<1000000 && (all(x!=(0)));
 i++){
        y = x * y;
        x = x + y;
        y = x * y;
        x = x + y;
    }
}
```

Figure 11.3 Kernel with multiply-add operations on vector type data

Number of multiply-adds	integer	float	expected time
2	190.001	190.001	190.000
3	253.337	253.338	253.333
4	316.670	316.670	316.667
5	380.001	380.001	380.000
6	536.670	536.670	443.333
7	631.670	631.670	506.667
8	711.670	711.670	570.000
9	790.002	790.001	633.333
10	855.001	855.001	696.667

Figure 11.4 Performance results for different data types

Chapter 12

Cache Analysis

On this chapter I'm going to analyze some characteristics of the two caching levels, like caching delay, coherence, race conditions and false sharing.

12.1 Cache Store delay

In order to figure out the load and store delay, I implemented an experiment that examines each one separately. For the store delay I used a kernel where each thread stores a variable in a global table, given from the Host, at the position that their global id indicates (figure 12.1). The data flow graph below (figure 12.2), shows that the critical instruction path is 2

```
__kernel void simple_write(__global int* A){
    int x = 38;
    int id = get_global_id(0);
    for(int i=0; i<1000000; i++){
        A[id]=x;
    }
}
```

Figure 12.1 Kernel that stores variable in a global object

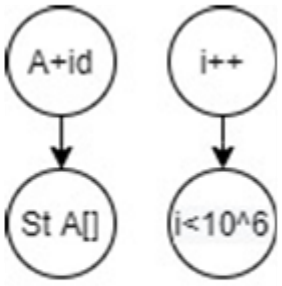

Figure 12.2 Dataflow graph

When measuring the performance of this kernel and calculating the clock cycles that one work-group of one work-item needs we find that it takes 154 cycles, which is close to assume that it needs around 4 pipeline cycles (4x38=152). But the clock cycles that 64 work-groups need are 192 which is exactly 3 pipeline cycles per iteration.

instructions x pipeline_depth + stalls = 3 x 38 + (64-38)*3 = 192

Therefore I come to the conclusion that there is a form of cache parallelism that is bigger than the pipeline's parallelism and when the pipeline begins to have stalls the cache delay (figure 12.3), which is equal around to 40 cycles for one store instruction (154-(3*38)=40), starts to decrease and eventually hides completely behind the pipeline stalls.

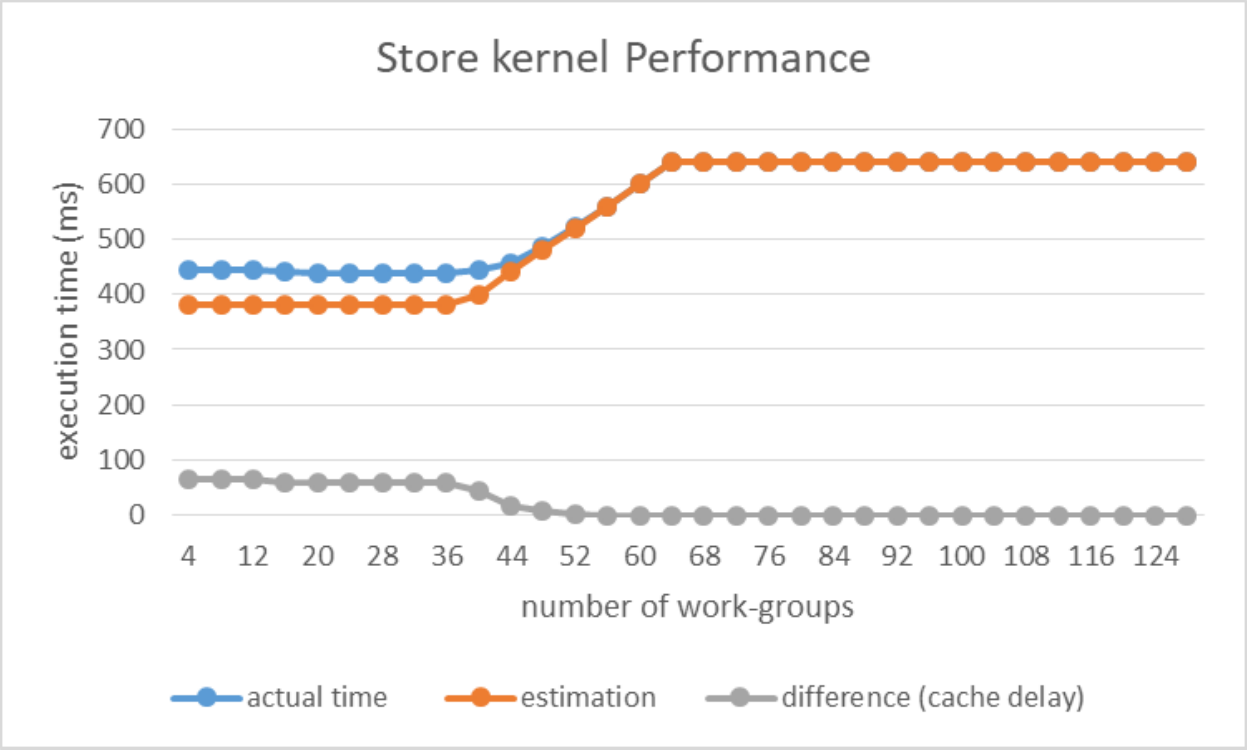

Figure 12.3 Performance analysis for a simple store kernel

So assuming that the kernel needs 3 pipeline cycles per iteration without any cache delays and knowing that the first instruction takes 1 pipeline cycle (simple addition instruction), we come to the conclusion that a Store instruction takes 2 pipeline cycles

12.2 Cache Load delay

To examine the load delay we would have to use a slightly more complex kernel, in order to avoid any compiler optimizations that could skip the load or alter the performance in any way. The code that I'm using is the following (figure 12.4). The data flow graph (figure 12.5), shows that the critical instruction path has length of 6 instructions. The performance

for one work-group with one work-item is 466.671ms, which is 280 cycles. The performance for 64 work-groups is 640ms (384 clock cycles), which is exactly 6 instructions per iteration. Therefore a Load instruction takes one pipeline cycle. This time the cache delay for one load instruction that eventually hides behind the pipeline stalls, is 52 clock cycles (280-(6*38)=52)(figure 12.8).

instructions x pipeline_depth + stalls = 6 x 38 + (64-38)*6 = 384

```
__kernel void read(__global int* A){
    int x = 38;
    int id = get_global_id(0) * 32;
    for(int i=0; i<1000000 && x!=0; i++){
        x=A[id+(i%32)];
        x += x * x;
    }
}
```

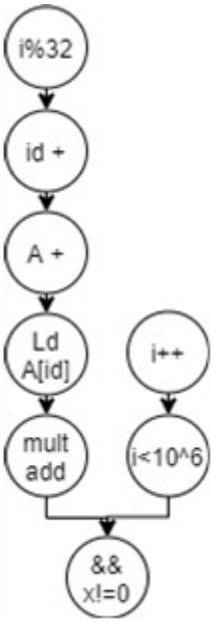

Figure 12.4 Kernel for Load data from global object

Figure 12.5 Dataflow graph\

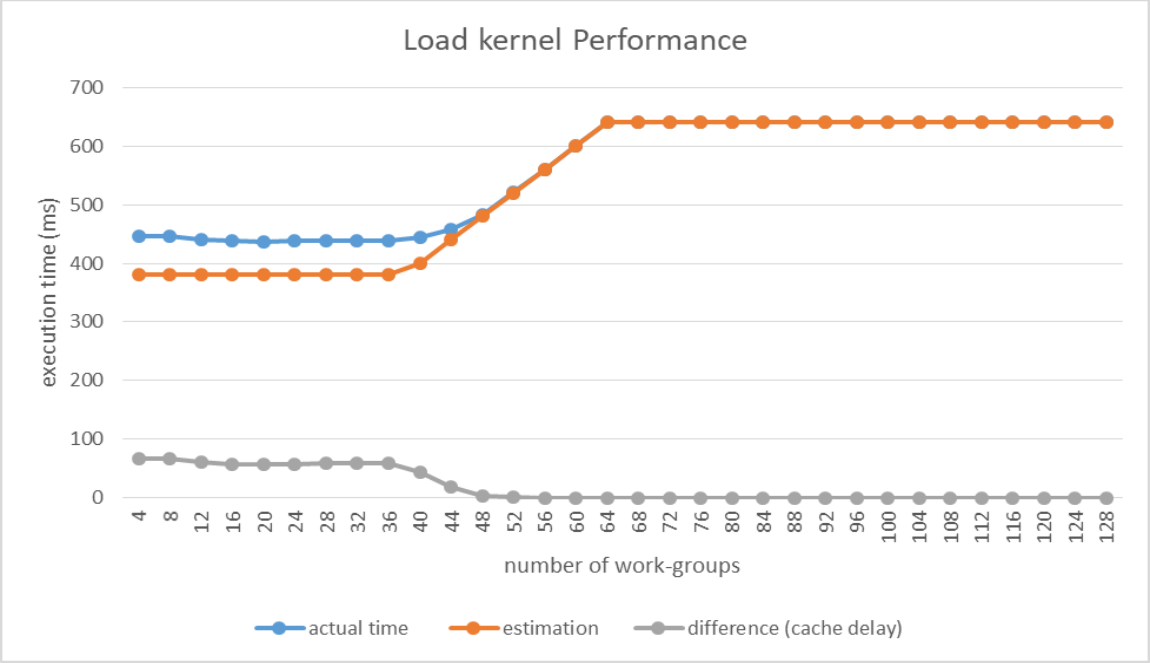

Figure 12.6 Performance analysis for Load Kernel

12.3 Cache coherence

A very important factor that effects the performance in cases of cache access, is the potential effort of the GPU to keep some cache coherence implementation of some coherence.
The Mali GPU has, no coherence between work items in core. We are going to compare two kernels that attempt to increase a value in a global memory address. Al the work items of the first kernel try to access the same address while the work-items of the second kernel access different addresses.

```
__kernel void coh_check1(__global int* A){
    for(int i=0; i<1000000; i++){
        A[0]++;
    }
}
__kernel void coh_check2(__global int* A){
    uint id = get_global_id(0);
    for(int i=0; i<1000000; i++){
        A[id]++;
    }
}
```

Figure 12.7 Kernel with race conditions (above) and without (bellow)

When measuring the performance of these two kernels for number of work-groups from 1 to 64 and work-group size 1, we ensure that only one core is utilized as I explained earlier. By comparing the performances, it is clear that they are identical. This indicates that in the second kernel there was no attempt to keep any form of coherency. The luck of coherence is also visible to the results of these kernels. The first one increases all the values successfully to 10^6 and the second one increases the value only to 10^6 instead of the work-items number times 10^6 that it supposed to.

12.4 Race conditions and False sharing through cores

There might be no coherence on the inside of one core and as result no race condition delays, but the same does not apply when we have race conditions between 2 different cores. When measuring the performance of the previous two kernels for work-group numbers larger than 64 there is a big delay in the second kernel where race conditions occur

```
__kernel void coh_check4(__global int* A){
    int id = get_global_id(0);
    int x = id%64;
    for(int i=0; i<1000000; i++){
        A[x]++;
    }
}
__kernel void coh_check5(__global int* A){
    int id = get_global_id(0);
    int x = id%64*64+(int)(id/64);
    for(int i=0; i<1000000; i++){
        A[x]++;
    }
}
```

Figure 12.8 Kernel with race conditions and kernel with false sharing

The results of the second kernel are still incorrect, but this time the value number is not consistently 10^6, it varies. There might be no coherence but there are definitely race condition delays when work-items from different cores attempt to access the same address in cache. Also this delay increases in proportion to the number of cores that try to access the same memory address, as for every core that gets activated, every 64 additional work-items the delay increases accordingly (figure 12.9).

What is interesting to note is that not only race conditions exist but also false sharing. The following two kernels have work-items that attempt to increase 64 values in global memory, in a loop. The values for each work-item in the first kernel are interleaved with a 64 step which results to groups of work-items hitting the same address values, and the values in the second kernel are divided among the work-items in a way that each of the 64 work-items of a core hits the same cache block as the corresponding work-items of the rest 3 cores, but not the same address. Note that this is possible only for a total number of work-groups smaller than 4096. When comparing the performance of each kernel for numbers of work-groups from 1 to 256 with work-group size 1, their performances are identical (figure 12.9). This validates that false sharing also causes delays.

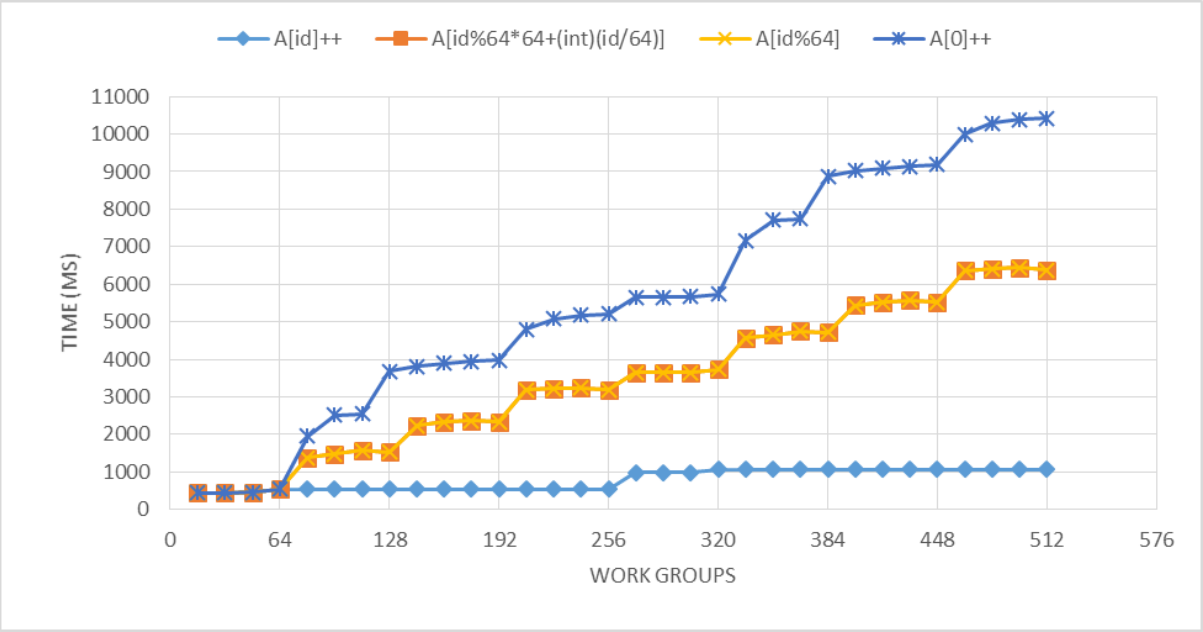

Figure 12.9 Performance of kernels that access memory on a different offset

Chapter 13

Performance model

On this chapter I'm going to analyze some characteristics of the two caching levels, like caching delay, coherence, race conditions and false sharing.

13.1 My Performance model

To summarize, based on the architectural characteristic that I discovered I created a performance model that takes as parameters the work-group size (gs), the total work-load (wl) and the number of VLIW instruction words for a kernel instance or work-item (I) and returns an estimation for the kernels performance. It has one constant parameter for the pipeline depth (pd) that equals 38, and many other variables that can be derived from the three parameters in the input of the function. Those variables the normal batch-size (maxB), the number of batches over the GPU (n), the size of each batch (bi), the maximum number of threads in a core for a certain batch (maxCi), the number of utilized or "active" pipelines (ap) and the stalls that will be expected in the execution of a thread for the given code (stalls).
The function f is the main function of the model and it calculates the execution time. The exact formula is shown in figure 13.1

$$f(wl, gs, I) = \sum_{i=1}^{n} \frac{I \times (pd + stalls(wl, gs, pd))}{F}$$

- $maxB(gs) = \min(1024\,,\ 256 \times gs)$
- $n = \left\lceil \frac{wl}{maxB} \right\rceil$
- $b_i(wl, gs) = \begin{cases} wl - (n-1) \times maxB(gs), & i = n \\ maxB(gs), & else \end{cases}$
- $maxC_i(wl, gs) = 64 \left\lfloor \frac{maxB(gs)}{256} \right\rfloor + \min(64,\ wl\%256)$
- $ap = \min(gs,\ 2)$
- $stalls(wl, gs,\ pd) = \max(maxC_i(wl, gs)/ap - pd, 0)$

Figure 13.1 Parametric performance model formula

The VLIW scheduling of the kernel has to be made manually in order to find out the Instruction number parameter (I) of the performance model. To make the scheduling you would have to consider the parameters of instruction fusion, the available resources of functional units that allow you to operate multiplication, addition, and vector look up table operations to a maximum of 128bit data per cycle, and finally the length of the instruction word which equals to 4 conventional instruction words. Note that the work scheduling of the job manager is included in the formula in figure 13.1

13.2 Cases that the model applies and potential of expansion

This model so far covers and is able to predict the performance under some circumstances. The cases in which is not able yet to estimate the execution time correctly is when the program's number of registers exceeds the number of 16, for work group sizes from 1-64(powers of two) and where the last executed batch has only 1 to 64 work-items, where we have the phenomenon of register spill, when we have code that accesses cache again when the last executed batch has 1 to 64 work-items and finally when we have cache issues like race conditions, false sharing and misses. Also another part of the performance model that requires more exploration is the part of the VLIW scheduling, as more cases need to be checked.

13.3 Application Example

We are going to see step by step an example on how to apply the performance model on a Kernel. For the example we are going to use the kernel of section 10.1, in figure 10.2. According to the VLIW scheduling on figure 10.3 the number of instructions per iteration for the kernel is 5 and the number of iterations is 10^6, so the total number of instructions for a work-item is 5 x 10^6 (I). We are going to estimate the execution of a work-load of 257 work-items (wl) and work-group size 2 (gs). When passing these parameters to the performance model we get the following results

- $maxB(2) = \min(1024\ ,\ 512) = 512$
- $ap = \min(2,\ 2) = 2$
- $n = 1$
- $b_1(256,2) = 256$
- $maxC_i(256,2) = 64 \left\lfloor \frac{512)}{256} \right\rfloor + min(64,\ 0) = 128$
- $stalls(256,2,\ 38) = \max(26,0) = 0$

$$f(256,2,5) = \frac{5 \times 10^6 \times (38 + 26)}{600 \times 10^6} = 0.533 = 533ms$$

So the estimated execution time is 533ms and the actual execution time when running the kernel is 533.33ms.

Chapter 14

Related Work

14.1 Related Work

Some related work that I was able to find is the papers “Power and Performance Characterization and Modeling of GPU-Accelerated Systems” [8] and “Demystifying GPU Microarchitecture through Microbenchmarking” [9]. The most basic difference from my work and those two papers is the fact that they are both focusing on NVIDIA GPU architectures, when my model is an ARM Mali GPU. Also in the first paper [8] they create the performance models for both performance and power, by running benchmarks and statistically analyzing some performance counters. They are not aiming on the discovery and better understanding of the architectural parameters of their model, which is our main goal in order to create a framework to create viruses. On the second paper [9], where they were a little bit closer to what I was trying to achieve, except of the fact that they were working on a different model to which they had more background information about its architecture they also didn’t try to apply their findings to create a parametric performance model like I did.

Chapter 15

Future Work

15.1 Future Work

As I have mentioned before, there are certain improvements and additions that need to be made to the performance model in order to have a more complete model that covers all possible cases of kernels. The most important issues that need to be examined at this point are the cache misses latency, and the VLIW scheduling constrains.

I ‘m not focusing on the cache delay that we observe when the pipeline is not busy enough, because for the creation of processor viruses we usually want a full and busy pipeline. On the other hand, to be able to predict the cache miss latency would be very important, as we typically search for scenarios that utilize as many units as possible at the same time and a cache miss can definitely help us to achieve that.

And lastly a better understanding of the VLIW scheduling is crucial in order to come up with kernels that can cause the simultaneous execution of the maximum amount of operations. We need such kernels to use them for stress tests and as parts of other viruses that target the processors power or aim to cause voltage drops.

Chapter 16

Conclusion

15.1 Conclusion

To summarize the main goal of my work was to locate and reveal the architectural characteristics of Arm Mali T-624 GPU, and use them as parameters to create a parametric performance model. The purpose of this characterization and modeling is to be used in frameworks that create processor viruses.

My work in comparison to other related work, aims to the performance characterization and modeling of an Arm Mali GPU. Something that as far as it comes to my knowledge was not attempt before with real hardware and a specific model to this extend. The related work I found was mostly on NVIDIA GPU architectures. Also the focus of my work was different from other similar work on other platforms, as my focus was to use my findings as tools for the creation of processor viruses.

The performance model that I managed to create applies under some circumstances and further additions to the model would be necessary to cover all possible cases and kernel scenarios. I cited some cases that I found needing farther experimentation and analysis, but I’m certain that as you explore even more such cases will come to the surface. Nevertheless, for the cases that my model covers it can estimate the execution time successfully and accurately.

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, 25 October 2017, https://ieeexplore.ieee.org/document/8082515

[3] Michael and Semerath, “Instruction Fusion”, Patent Application Publication for Arm Limited and Cambridge, 04 May 2017, http://www.freepatentsonline.com/y2017/0123808.html

[4] Scott Rixner et al. “Register Organization for Media Processing”, IEEE, 06 August 2002, https://ieeexplore.ieee.org/document/824366

[5] https://www.sciencedirect.com/topics/computer-science/set-associative-cache

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

[7] ARM® Mali™ GPU OpenCL Developer Guide, file:///C:/Users/User/Downloads/arm_mali_gpu_opencl_developer_guide_100614_0302_00_en.pdf

[8] Yuki Abe et al. “Power and Performance Characterization and Modeling of GPU-Accelerated Systems”, IEEE, 14 August 2014, https://ieeexplore.ieee.org/document/6877247

[9] Henry Wong et al. “Demystifying GPU Microarchitecture through Microbenchmarking”, IEEE, 19 April 2010, https://ieeexplore.ieee.org/abstract/document/5452013

[10] Renji Thomas et al., “EmerGPU: Understanding and Mitigating Resonance-Induced Voltage Noise in GPU Architectures”, IEEE, 02 June 2016, https://ieeexplore.ieee.org/document/7482076

[11] Arm® Mali™ Offline Compiler, User Guide, https://static.docs.arm.com/101863/0700/mali_offline_compiler_user_guide_101863_0700_00_en.pdf

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

[13] Ivan Grasso et al. “Energy Efficient HPC on Embedded SoCs: Optimization Techniques for Mali GPU”, IEEE, 14 August 2014, https://ieeexplore.ieee.org/abstract/document/6877248

[14] Michel Steuwer et al. “Matrix Multiplication Beyond Auto-Tuning:Rewrite-based GPU Code Generation”, IEEE, 17 November 2016, https://ieeexplore.ieee.org/document/7745278