Politecnico di Torino

Department of Electronic and Telecommunications

Master Degree Thesis

Study and development of a VHDL infrastructure for signals probing of GPGPU architectures

Master degree course in Electronics Engineering

Thesis Advisors:
Prof. Luca Sterpone
Prof. Matteo Reorda Sonza
Ph.D. Boyang Du

Lab Supervisor:
Ph.D. Sarah Azimi

Candidate:
Salvatore Costa
matricola *247077*

April, 2020

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

In the past century the electronic systems have become essential for the technological evolution and the improvement of the human lifetime. From the telecommunication to the aerospace field, the electronic systems practically are the heart of the modern word. In the last years, the challenge has been to design and commercialize *smart devices*, products with a limited size and nevertheless able to perform several operations with high performance and low power consumption. The main obstacle to the implementation of these devices nowadays consists in the size scaling limits imposed by the technological processes unfit to pattern to small features in the silicon wafers. Consequently, although until 90s the size scaling had permitted to improve enough the digital circuit clock frequency, in last decades the impossibility to further scaling as much as needed led to a different strategy: to increase the number of cores promoting parallel computations and increasing the number of operations executable in a single clock period. This was the strategy that permitted the development of multi-cores devices, introduced at the beginning of the 21th century. Once the interest had passed to the parallel execution instead of size scaling, multiple-thread programming has been added to multi-core systems, to further improve modularity and efficiency simply substituting the single process with a set of threads working with specific resources. These architectures are also known as SIMD: *Single Instruction Multiple Threads*.
Again the scientific challenges and the human expectations make these devices unsuitable. Nowadays, in the years of spatial exploration and robotics, in which the boundaries of human scientific domain constantly expand, the performance improvement needs to be combined with the capability to elaborate a huge amount of data. Let's think for example at the number of information that an autonomous vehicle core has to acquire and elaborate. This led to favour the usage of GPUs against CPUs, since the first implicitly consist in an array of parallel cores and are designed to treat huge set of data. Finally, in order to provide to the designer enough flexibility, the GPUs has been soon substituted by the GPGPUs (*General Purpose Graphic Processing Units*). An example of this device is the FlexGrip, analysed in detail in the chapter 3.
Beyond the necessity to provide high performance on multiple data computation, it's obvious that also accuracy is a key point in the design step. Designers must be able to implement hardware or software techniques able to harden the device, in this case the GPGPU, accord-

ing to the different type of errors that can occur. From this perspective, it's essential to be aware of the error sources in complex digital electronic systems, as a GPGPU. The errors that take place in this type of systems can be due to internal causes or external ones.
In the first category it's possible to identify, for example, the *data hazards*, errors due to an incorrect synchronization among resources trying to access to the same memory location, provoking data corruptions or mistaken operations (except for the case in which both are reading the location). Practically, the resources enter in competition and the final result depends on which one is the fastest, e.g. if one tries to write and the other to read, the read data could be the updated one or not depending on the order of operations. Another type of hazard is the so called *control hazard*, an error that occurs in pipelined systems when the branch predictor selects the wrong path so, when the actual branch condition arrives, the entire pipeline must be emptied in order to follow the correct execution path. Differently from the *data hazard*, the *control hazard* probably will not produce an output error but certainly causes a significant performance degradation. Still concerning the internal error sources, other cases are: *glitches*, spurious events due for example to cross-talk phenomena; *signal skew*, e.g. clock skew due to improper clock distribution that could produce violation of the set-up and hold times of the storage elements; *metastability*, condition occurring when a storage element samples a signal in the transition region (neither digital 0 nor 1) so that the final stored value is random and stable only after a certain time interval; *electromigration*, a phenomenon that occurs in connections in which an high DC current flows, provoking an high current density that causes creation of metal bumps (possible short circuit with neighbour connections) or voids (that collapse to open circuits).
As previously mentioned, beyond internal error sources, also the external ones are significant, where external stays for environmental. For sure the environment represents an error source and an evident proof is the *electromagnetic* disturb. Some of the most critical environmental error sources nowadays are the so called *radiation effects*, and more in details the *single event effects* (SEE).According to this, in the following sections SEEs will be analysed in details, focusing on the SEUs.

1.1 Single Event Effects

The *single event effect* (SEE) is a phenomenon that occurs when an ionizing particle is able to transfer enough charge to change state or damage the target component. These ionizing particles can be present in:

- space, e.g. in cosmic rays (GCR) or solar events (generating solar energetic particles, SEPs);
- atmosphere, due to cosmic particles that collides with the atmospheric atoms, generating cascades of protons or neutrons (these lasts cannot directly generate SEUs but the nuclear reaction by-products of neutrons and silicon can).
- device, due to radioactive impurities in component materials, such as package, emitting

alpha particles.

More in details, SEEs are generated by a single charged particle that passes through a semiconductor material and, if the *linear energy transfer* (LET, index of the energy transferred to material), overcome the critical energy of the device itself [1].

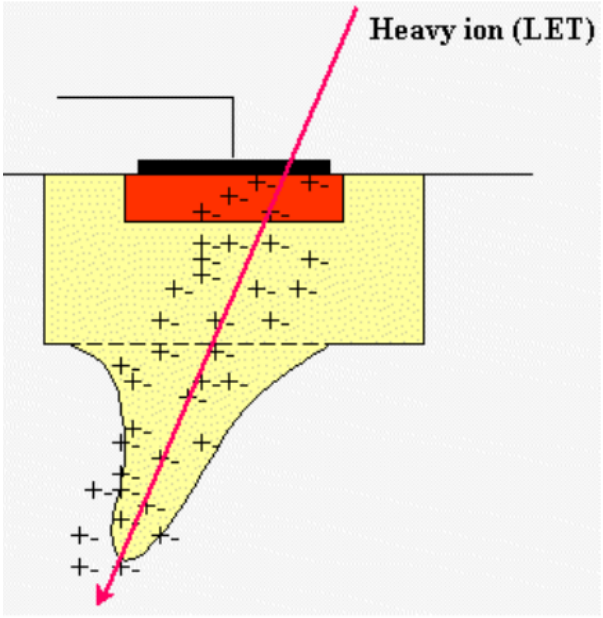


Figure 1.1: SEE generation [1].

The main problem is that the critical energy decreases with the size scaling described, as know, by the Moore's Law (in fig. 1.2), that predicts a 50% reduction of the transistor area each 3 years . This is the reason why the SEEs represent a critical problem from around 1990 on, when the technological node scaling was enough to permit to ionizing particles to easily overcome the critical energy barrier.

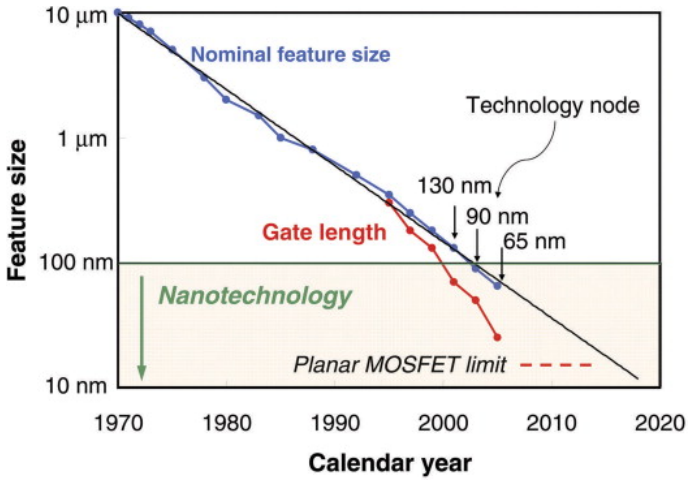


Figure 1.2: Transistor size Moore's Law.

Regarding the SEE effects, this can be divided into three main classes [2]:

- *soft errors*, typically causing bit flip in digital devices (called *single event upsets*, SEU) or a transient in an analog circuits (called *single event transient*, SET). These are the most frequent in the terrestrial environment, due to alpha particles, neutrons or protons. Different studies demonstrate that the most vulnerable components are volatile memories (e.g. SRAM, DRAM, etc.) and complex devices (e.g. FPGA, uprocessors, GPGPU, etc.). When the critical charge is very low, the probability to induce error in more than one bit increased, generating the *multiple bit upset* (MBU). These soft errors can be recovered with particular protection or mitigation strategies.
- *firm errors*, that produce temporary corruption of the device functionality due to a SEE. In this condition, the recovery is possible only by means of a reset of the entire system. An example is the *single event functional interrupt* (SEFI).
- *hard errors*, that occurs when the functionality of the device is permanently corrupted (no possible recovery). Examples are the *single event induced hard error* (SHE), transistor is locked in a permanent logic state (due to oxide damaging), or the *single event latch-up* (SEL), that cause an high current flow during latch-up state, or finally the *single event burn out* (SEB), occurring in high voltage devices. These errors can only be corrected replacing the damaged units.

1.2 Single Event Upsets

As discussed in the previous section, the SEUs are *non-destructive soft errors* that produce a bit-flip in digital circuits, and in particular in memory locations (in the aerospace applications the critical memories are the cache L1 and L2). This means that there are two way in which a SEU can be generated:

- due to collision of ionizing particles with the memory cells;
- due to a SET signal that is latched.

In both cases the consequence is the storage of a corrupted data that could create or not output errors. The key point is to understand which strategies can be adopted to eliminate or minimize the effect of SEUs. These strategies belong to the so called *radiation hardening design* and could be classified in physical strategies and logic strategies [3].
Regarding the physical strategies, some possibilities are:

- to substitute the CMOSs with BJTs, since are more robust with respect to the radiation effects;
- to embed MRAM (magneto-resistant RAM) [19];
- package shielding, to limit radiation effect (not so efficient in the aerospace environment);

- to substitute DRAM (based on capacitors) with SRAM;
- to use substrate with a wide band-gap (e.g. SiC or GaN);
- to realize circuits on insulating materials instead of semiconductors (e.g. SOS or SOI).

Vice versa the logic strategies, some possibilities are:

- modify the internal architecture of the flip-flop, even if this means to increase size or power consumption [4].
- modular redundancy (spatial or temporal) with comparison [22, 23, 24];
- EDAC, so correcting code e.g. parity bit, Hamming code [7], convolutional codes [8], etc.
- scrubbing techniques [9, 10], to restore the memory contents periodically;
- watchdog timers, to have the possibilities to exit from possible forbidden states.

Furthermore, the techniques can be divided in hardware and software techniques. Obviously, there is not a golden hardening technique since each one impacts: performance, area and power consumption. This implies that the radiation hardening design is always a trade-off between hardening and performance degradation.

CHAPTER 2

Related Works: Injections, Tests and Hardening

The development of safe-critical applications and, consequently, the necessity to guarantee a significant tolerance to faults led to develop validation and design techniques able to harden the system against SEEs. In both cases, several solutions can be taken into account, each with specific advantages and constraints. In this chapter a brief introduction of the main validation techniques and hardware- software-based design techniques have been provided. Furthermore, a description of the resilient load and store instruction optimization, developed by the Universidade Federal do Rio Grande do Sul (UFRGS), has been provided.

2.1 Fault Injection Platforms

The fault injection procedure represents the first step for the verification and validation of those systems designed to work in a radiation harsh environment, i.e. the aerospace application, or in safety-critical conditions. Validation in these cases require the analysis of the system under test when it is affected by faults. The two main strategies in this optics results to be the:

- *Simulated Fault Injection* (SFI);
- *Emulated fault injection.*

The SFI solution is based on simulations of the virtual design (VHDL) of the target system, e.g. using tool as Modelsim (used as simulation tool in this work). Consequently, a significant advantage of this verification strategy is that it is performed in the design steps, before synthesis, providing to the designers the possibility to efficiently test several solutions and hardening techniques on the virtual prototype of the final application. As discussed in [13], the SFI can be implemented using proper simulator commands, e.g. to inject bit-flips, otherwise modifying the hardware at different level of abstractions: introducing *saboteurs* (units injecting faults) or *mutants* (behavioural corrupted hardware units). This strategy is

widely used for different type of systems: high-speed LAN [11], CPUs [12], safety-critical embedded systems [13] and also quantum circuits [14]. The main drawbacks is the huge amount of simulations that must be performed in order to completely characterize the system influenced by faults. This depends on the number of fault-sensible hardware components, and consequently on the architecture complexity, but also on the complexity of the application. In fact, the wider is the execution time the higher is the number of cases to verify, since faults can occur in a larger time interval. In order to limit the number of simulations, the typical solution is to identify the software-hardware critical units [12], limiting the so called fault space and so the number of possible faulty conditions. This is the same goal of this thesis, implement a monitoring system that, providing detailed information about the performance of each module and the memory accesses, permits to identify the hardware critical modules to reduce the number of actual simulations necessary to validate the GPGPU architecture.

Vice versa, the emulated solution is based on injections on an implemented system prototype, e.g. on an reconfigurable device such as FPGA. This strategy can be implemented both with instrumentation at different levels (RTL, gate, physical) [15] or reconfiguration [16]. Obviously, considering that this solution requires a physical implementation, in this case the hardware and performance overhead must be taken into account. With respect the simulated versions, the *emulated* one permits to reduce the validation time.

2.2 Radiation Test

The radiation test is an additional validation technique that, with respect to the fault injection platforms discussed in section 2.1, permits to obtain analysis results closer to the realistic behaviour of the system working in a radiation harsh environment. This is due to the fact that the fault injection platform strategy is based on a certain number of simplifications, one of these the injection of specific faults, e.g. memory bit-flips. Vice versa, the radiation tests permit to more realistically monitor the radiated device without limiting the possible type of faults under analysis. Nevertheless, the radiation test has a non-negligible drawback: the difficulty to analyse the error propagation and cause in the system, due to the fact that the error is visible only when an output mismatch occurs or when the functionalities are corrupted. Another disadvantage is that this type of test, for sure more realistic, can be performed only on a physical device and, consequently, not in the design steps (as the simulated fault injection).

The radiation test is performed emulating the terrestrial or space radiation environment, according to the target application. In the first case a beam of alpha particles, neutrons, protons is used while in the second one an heavy ion beam. An alternative to the ion beam is the gamma rays beam [17], that represents an easier and cheaper technique. In some tests for space applications, also electron beam obtained by means of electron linear accelerator are used [18]. Typical beam energy is around tens of MeV while the typical flux is several orders of magnitude higher than the realistic one, in particular for terrestrial application, permitting to perform in few hours an experimental test equivalent to several years of operations for the

device under test. Radiations test has been used both to investigate memory elements [17], [19] and complex systems as FPGA and CPU [20], [21].
The radiation test is led properly tuning the ion or particles beam, checking also the uniformity of the beam itself, and then irradiating the entire device or only a portion of it. Depending on the type of ion beam the device must be prepared in a different way, e.g. using alpha particles it's essential to remove the package from the chip, since can act as a shield. Three different types of radiation tests, as described in [21] are:

- Static, the device is not active, e.g. in the memory radiation test the memory is not accessed during the radiation time;
- Semi-static, the device is partially active, e.g. in the memory radiation test the memory is continuously read;
- Dynamic, the device is active, e.g. in the memory radiation test the memory is accesses as in the typical usage.

During the test, in order to provide an automated monitoring of the device functionality, the system under test is typically connected to a computer as isolated as possible from the radiated space.

2.3 Hardening Design

It's evident that the fault injection platform and radiation test analysis are simply able to provide information about the effect of faults in the target application, without guaranteeing any hardening. In fact, these are investigation techniques. Obviously, there is also the necessity, as briefly discussed in section 1.2, to implement in the design step specific hardening strategies, from physical level to the logic one. Concerning complex systems and in particular GPGPU, the most common strategies can be classified in two categories: hardware and software solutions. The first ones regard modifications and optimizations of the hardware units, while the second ones typically regard optimizations of the instructions set. In the following section two radiation hardening strategies will be analysed more in details: *Triple Modular Redundancy* (TMR) and a *modified version of the Load and Store instructions* to the global memory, implemented by the Universidade Federal do Rio Grande do Sul (UFRGS) and analysed using the developed monitoring system in ch 5.

2.3.1 Triple Modular Redundancy

The TMR is based on the concept to exploit redundancy in space (physical redundancy) and in time (sampling the signal in different time instants in the same clock cycle) and then use a majority voter circuit in order to provide the correct output and, if properly implemented, to communicate to the control block an error detection. Obviously, being the majority voter a logic circuit that generates at the output the values most repeated among the inputs, this system works with two assumptions:

- only one of the three modules or samples is faulty;
- the majority voter is insensible to faults.

Obviously both assumptions are not realistic and the voter results to be the weakness of the TMR strategy [22]. A general block description of the TMR technique is the one shown in fig. 2.1.

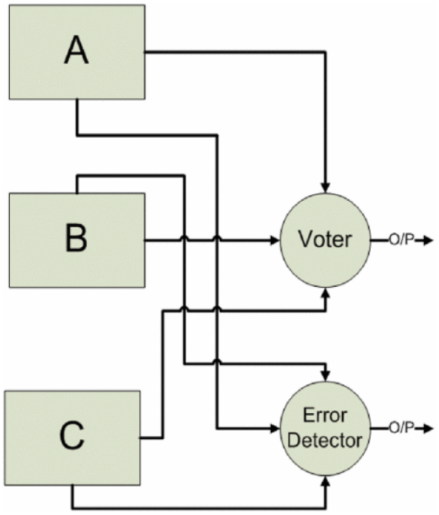


Figure 2.1: TMR block diagram [23].

It's possible to notice that the TMR block it's able to detect the error and, under the assumptions previously described, to provide the correct output, but is not able to recover the faulty module, at least without adding a proper feedback unit able to identify the faulty module and correct the error. The logic circuit of the voter and the error detector can be easily obtain using the standard digital circuit design techniques, e.g. Karnaugh map [23].
The TMR technique can be applied at different levels of abstraction, from the gate level to the module level. The main drawback of the TMR is that, as regard the spatial version, it produces a hardware overhead of 200% if applied to the entire system. Therefore, in both spatial and temporal version, the complexity and the added hardware produce a significant performance degradation. In order to limit these effects, the general solution is to use the so called *selective TMR* [24]. This solution requires the identification of the resources most sensitive to faults and the implementation of the TMR technique just for these modules. There are several methods to identify these modules, from the probabilistic analysis to a details analysis of the performance when the system is affected by faults (e.g. by means of simulated fault injection platform). In this optic, this thesis will be oriented to implement a monitoring system able to provide additional information that can be useful to identify the modules more sensitive to faults and also to analyse the performance degradation caused by the hardening techniques. Obviously, there is also the possibilities to merge different hardening techniques in order to take advantages of different strategies, e.g. TMR group coding method [25] or Scan-chain-based Multiple ERror TMR (SMERTMR) [26].

2.3.2 Software Hardening and Resilient Global Load/Store Instructions

Typical software hardening techniques are based on the *Duplication with Comparison* approach, consisting in duplicating the entire instruction set and compare the output results in proper time instant to check the convergency between the two identical operations. This obviously leads to a significant performance penalty. Analysing the single instructions, the ones introducing the highest latency are related to the accesses to the global memory, both in reading and writing modes. From this concept, it has been developed the resilient load and store instruction optimization, which aims to reduce the performance degradation introduced by the global memory accesses. In order to do this, a simple software hardening technique is not sufficient, but also hardware modifications must be taken into account. More in details, the resilient instructions implementation has required the optimization of the binary code, and as a consequence of the *decode* unit, but also of the *write* and *read* units, that accesses to the global memory. The resilient strategy is still base on duplication technique. In particular the idea is to work with a copy of the used registers. This implies that both resilient load and store work with two registers: the original and the replica. More in details:

- the resilient load substitutes the duplicated global memory load with a single load from the global memory and a move instruction to copy the loaded value in the register replica.
- the resilient store duplicates the number of store instruction checking the coherency between the original and replicated code in both source and destination locations.

Analysis on this strategy show a significant reduction of output mismatch errors (also called SDC) and execution errors (also called DUE), but with performance degradation (around 5-7%) and area overhead (around 0.15%).

CHAPTER 3

Background: FlexGrip GPGPU

The FlexGrip (FLEXible GRaphIcs Processor) is a CUDA binary-compatible open-source integer GPGPU, based on the NVIDIA G80 architecture and design by a group of researchers of the Electrical and Computer Engineering Department of the Massachusetts University [5]. More in details, this architecture has been optimized for FPGA implementation, to overcome limits of the vector processors. In fact, although both GPGPU and vector processors provide parallel processing capabilities, the first permits to increase the possible number of threads simultaneously executed, optimise memory accesses and implement the conditional control mechanism in hardware, significantly simplifying the CUDA model. In this chapter the FlexGrip architecture and functionality will be analysed deep in details in order to provide a background knowledge for the next analysis.

3.1 Instruction Hierarchy

A typical GPGPU architecture is the one in fig. 3.1. As shown, the GPGPU is made of an array of *streaming multiprocessors* (SM), each composed of a certain number of *scalar processors* (SP). Thanks to this type of hierarchical architecture, the device is able to execute the demanded task in a SIMD form (*Same Instruction Multiple Data*), with the different SPs that, inside the same SM, execute the same thread, but handling different data. Vice versa, the SM is designed as a SIMT processor (*Single Instruction Multiple Threads*) due to the fact that, inside each SM, the same instruction is parallelized among the different SPs.

In order to optimize the parallel instructions execution, an instruction hierarchy is implemented and each level is associated with a proper scheduler, that dispatches and controls the instruction flow inside the whole structure at different levels of abstraction. In this section a deep analysis on this instruction tree is provide, to clarify some figure of merits that will be significant in the next chapters.

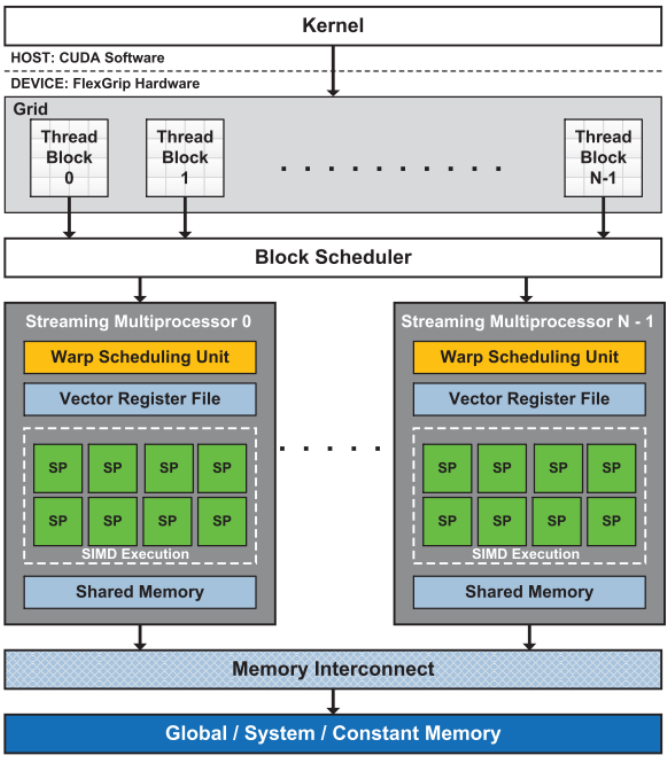


Figure 3.1: Typical GPGPU architecture [5]

Starting from the instruction set, that can be converted in a cubin code (simple CUDA binary code) or SASS code (low-level assembly, architecture specific for GPGPU NVIDIA G80), each instruction is parallelized in 32 threads. These threads are next scheduled among the scalar processors, that work with different memory segments (according to the SIMD strategy). These threads are grouped in *warps*, the smallest set of simultaneous operations with data independency. More in details, a single warp is composed of 32 threads, but this doesn't mean that a single instruction is mapped in a single warp. Vice versa, due to the fact that each thread is associated to a specific memory segment in which a location is dedicated to the *program counter*, a single warp is consisting of a group of instructions. Moreover, thanks to a dedicated *program counter*, each SP can work independently from the others, in particular in the case of branching condition. Consequently, during complex kernel executions and in particular conditional evolutions, different threads in the same warp could diverge from the normal execution flow, leading to the so call *internal divergency*.
Finally, the warps are grouped in the so called *thread blocks*, tri-dimensional array representing a set of operation that can be executed in parallel. These blocks are then organized in a bi-dimensional grid.

After the description of the instruction hierarchy, an important point now is how blocks, warps and threads are scheduled among different scalar processors. Indeed, focusing on the scalar processors, the key point is to understand how the 32 thread of each single warp are mapped in the scalar processor array. This mainly depends on the number of parallel cores, that can be set during configuration steps and influences the *thread-level parallelism* (TLP,

number of threads that system is able to parallel execute) [6]. In the FlexGrip, according to the possible TLPs that can be fixed, threads are mapped in the SPs array as follows:

- if *32 cores*, since warp is consisting of 32 threads, each core runs a thread and the whole warp is executed in once. In this case the TLP and the allocated hardware are maximum, so the system is faster. Since all threads of warp are run in parallel, the warp is composed of a single array of 32 threads. Calling *warp lane* the array of threads inside the warp that are simultaneously executed, in this case, a single warp lane is present.
- if *16 cores*, each must handle 2 different threads to completely execute the warp. According to this, in this case 2 warp lanes are present.
- if *8 cores*, each must handle 4 different threads, leading to 4 warps lanes. In this condition the TLP and allocated hardware are minimum, so the system is slower.

In the fig. 3.2 a graphical view of the instruction hierarchy is shown, while in the concept of warp lanes organization is described in the fig. 3.3.

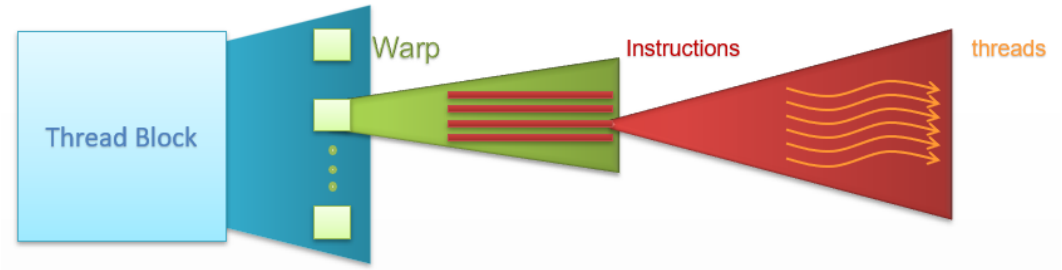


Figure 3.2: Instruction Hierarchy.

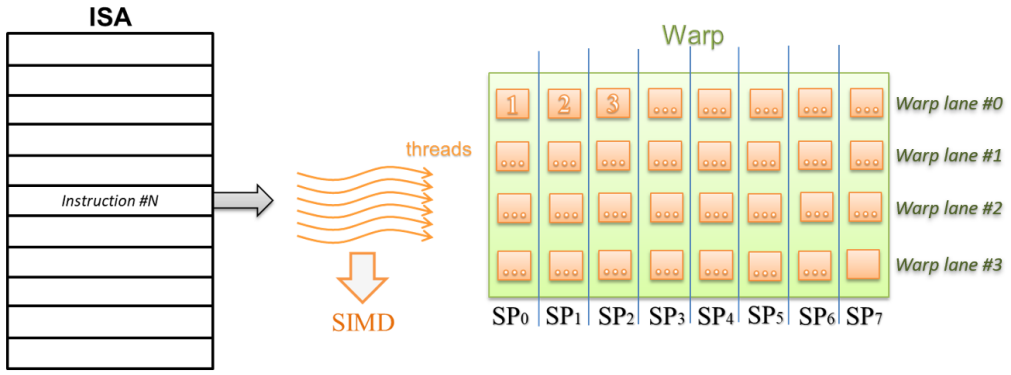


Figure 3.3: Threads arrangement inside a single warp (case of 8 cores, 4 warp lanes).

3.2 FlexGrip Architecture

In this section the internal architecture of FlexGrip will be described, focusing on the main modules. Finally, also the different memory banks present in the structure will be analysed.

3.2.1 Block Scheduler

The *block scheduler* is the top-level scheduler that, from a bi-dimensional grid of thread blocks (which dimensions are provided in the configuration stage), evaluates the total number of thread blocks and assigns the thread blocks to the *streaming multiprocessors* inside the FlexGrip GPGPU, providing to the *SM controller* the coordinates of the selected thread inside the grid (*grid_x* and *grid_y*) and inside the thread block (*block_x*, *block_y* and *block_z*). The scheduling policy employed by this scheduler is the round robin one. After the thread scheduling, the *block scheduler* passes the control to the *SM controller*, that will be analysed more in detailed in the section 3.3.

3.2.2 Warp Unit

The *Warp Unit*, which general architecture is shown in fig. 3.4, is the module that generates warps and control their execution [6].

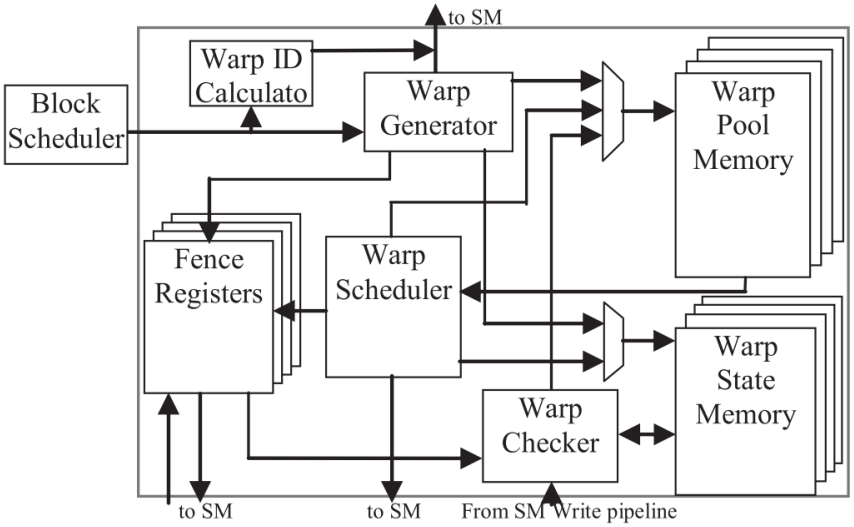


Figure 3.4: Warp Unit general architecture [6].

The main modules inside this block are: *warp generator*, *warp scheduler*, *warp checker* and a set of memory banks (*warp pool memory*, *warp state memory* and *fence registers*) described in detailed in section 3.2.4. Since this is a complex architecture, each module has its own finite state machine, in order to simplify the internal description and control.

The *warp generator* module is responsible of the creation of warps and initialization of the *warp pool memory*, *warp state memory* and *fence register*. Consequently, this module is invoked each time a new thread block is scheduled. During the creation of warp, a warp ID is created and saved into the *warp pool memory* according to the number of warp per block, the size of the *general purpose register bank* and the number of the current warp and block. Finally, before moving the control to the *warp scheduler*, the *warp generator* also creates a shared memory address for each single warp, according to the shared memory base address

and size (provided by the *block scheduler*) and the current thread block number.

The *warp scheduler* module is responsible of scheduling warps when the previous module concludes their creation. More in details, this module schedules a warp lane per time, interleaving among each warp lane a gap to wait for the *fetch* module response. This means that the *warp scheduler* module, inside the warp unit, is for all purposes a part of the pipeline (described in section 3.2.3). When the warp is completely executed the *warp scheduler* sends a signal to the *streaming multiprocessor controller*, that informs the *block scheduler* to schedule a new block.

The *warp checker* module, as the *warp scheduler*, is a part of the pipeline but, differently from the *warp scheduler* that precedes the *fetch* unit, the *warp checker* follows the *write* unit. As final part of the pipeline, this module is responsible to control the execution of the warp, checking if there is any internal divergency (described in section 3.2.5) and comparing an internal mask with the fence one to check the warp divergency completion. If the internal warp divergency is still present, the *warp checker* set the warp state as ready and a new lane is scheduled by the *warp scheduler*. Vice versa, if divergency is solved, the *warp checker* sends to the *warp scheduler* a done mask that, compared with an internal mask of the scheduler itself, permits to check the completion of the warp execution.

3.2.3 Pipeline Description

Inside each SMs in the FlexGrip GPGPU, a five-stage *pipeline* [5] is implemented according to the block diagram shown in fig. 3.5:

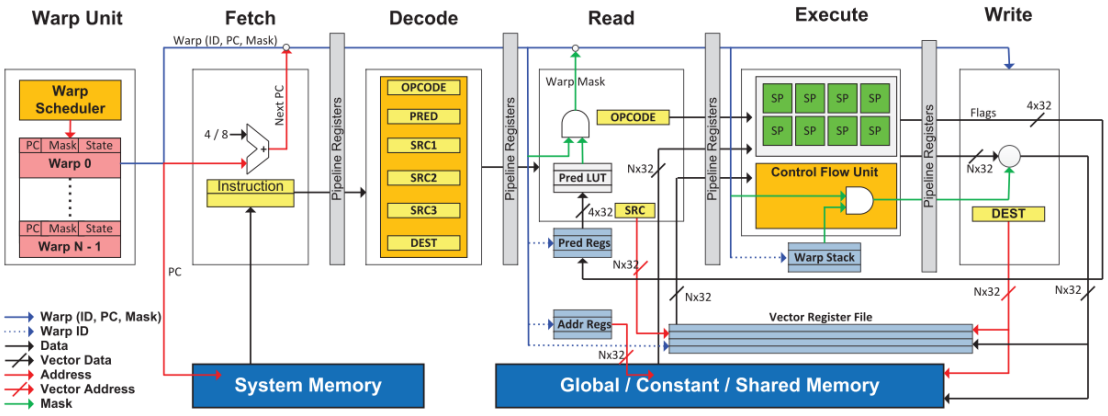


Figure 3.5: Detailed block diagram of SM [5].

From the block diagram description in fig. 3.5 is possible to notice that, during thread execution, the warp unit interacts with the pipeline. Consequently, in the five-stage pipeline also the warp unit must be considered. More in details: the actual pipeline takes care of the thread execution, vice versa the warp unit schedule warps, coordinating the instruction

execution, and check eventual internal divergency.

Concerning the internal modules of the pipeline, the different stages are:

- *Fetch* stage. It's the initial stage, that fetches four or eight-bytes CUDA-binary instructions (respectively 32 or 64 bit) according to the value of the program counter associated to each warp. After the fetching operation, that is always enabled by the warp unit (more in details by the *warp scheduler*), the program counter of the warp under execution is properly updated (of 4 or 8 byte according to the type of instruction: long or short).
  Analysing more in details this module and in particular its internal finite state machine, an important fetching policy is highlighted according to the TLP set in the configuration stage. In particular, when the 8-SP or 16-SP configuration is set, and so the threads inside the warp are organized in warp lanes, what happens is that the actual fetching from the system memory is performed only for the first warp lane (enabled by the fetch_en input). Vice versa during the following warp lane fetch operations of the same instruction (enabled by the pass_en input), the *fetch* unit doesn't read again the system memory but simply maintains the output signals constant thanks to the output registers of the pipeline (that actually corresponds to the pipeline registers). This permits to significantly increase the performance and reduce the power consumption, since the reading operation of the system memory, that is large (delay) and far (delay and consumption) from the *fetch* unit, is substituted by a reading operation of a set of registers for sure smaller and closer. This feature will be analysed more in detailed in chapter 4.

- *Decode* stage. It's a single decoding stage module that interprets and decodes the binary instructions at the output of the *fetch* stage and sends the decoded information at the output to allow instruction execution.

- *Read* stage. This module, starting from the information obtained by the *decode* stage, reads the operands stored in one of the memories inside the FlexGrip architecture, from the global memory to the *vector register file*.

- *Execution* stage. This consists of multiple scalar processors, according to the configuration, and a control flow unit to handle control flow instructions (e.g. branch).

- *Write* stage. This is last stage of the pipeline, dedicated to write output data of the *execution* stage in the proper storage module: intermediate data in the *vector register file*, final results in the global memory, etc.

A key point of this module analysis is how the instruction flow is controlled among the different blocks belonging to the *pipeline*. The strategy is based on a handshake between consecutive blocks exploiting two signals:

- a *done* signal, that each unit sends to the next one in the pipeline to inform it that the output signals are steady and meaningful.

- a *stall* signal, that each unit sends to the previous one in the pipeline to inform it that the unit is busy and so not ready to sample and manage new inputs.

According to this it's possible to conclude that the *done* signal "moves" in the pipeline (actually each module generates its own *done* and *stall* signal) in the same direction of the instruction flow, vice versa the *stall* signal. A graphical description is shown in fig. 3.6:

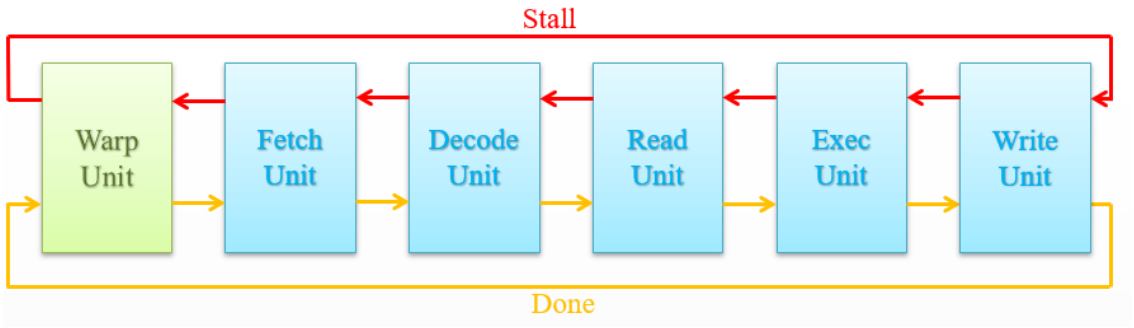


Figure 3.6: *Stall*/*Done* signals inside the pipeline.

Describing more in details the communication among two successive blocks, the second block cannot accept the output of the first one until the done signal of this block is not activated. Vice versa, the first block cannot update the output signal and handle new inputs until the next block is not free to accept new inputs. This will be crucial in the chapter 5 because the consequence of this synchronization policy is that critical module at the end of the pipeline will increase the stall time of the previous module, introducing performance degradation.

3.2.4 Memories

In this section will be provided a detailed description of the memory elements that are present in the FlexGrip GPGPU, focusing on the internal architecture and role of each one. This is significant considering that the memory locations are the target of SEUs that, as largely explained in chapter 2, cause bit-flips.

Global Memory

The *global memory* is that storage element reserved to data, e.g. initial data and final results. Consequently, the access to this memory is allowed both to the host (in the initial steps to insert starting data) and the *streaming multiprocessors* (from the *read*/*write* unit of each pipeline). The *global memory* is implemented as a dual-port RAM (random access memory) with:

- asynchronous read;
- synchronous write (with the clock rising edge).

The write/read control signals are combined in the write enable command (*we*). According to the internal architecture: when the write enable is active (we = 1), the writing operation occurs during the clock rising edge at the location pointed by the address signal; vice versa when the write enable is disabled (we = 0), the reading operation occurs and the global memory sends asynchronously the location pointed by the address signal to the output. Considering that this is a dual-port device, this is verified for both ports and each port has its own control/data signals. Moreover, the dual-port memory is able to guarantee a simultaneous read/write operation of the same input data conveniently using the control/address signals. More in details, to write the input data presents in port B to a specific location and simultaneously read this data at the output of port A, the conditions to force at the memory inputs are:

- we_b = 1;
- addr_a = addr_b.

This obviously permits to reduce the performance degradation since in a single clock cycle the memory is able to write/read the same data. In the VHDL codes, the signals related to this memory are instantiated as *gmem_(...)*. The block diagram description of the *global memory* is shown in fig 3.7:

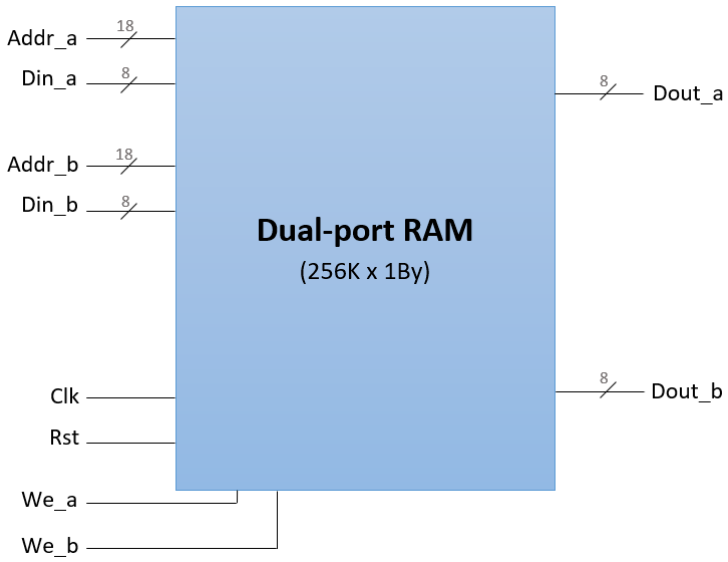


Figure 3.7: Global memory block diagram description.

Constant Memory

The *constant memory* is implemented exactly as the *global memory*: dual-port RAM with asynchronous reading operations and synchronous writing operations. So, this memory is implemented as a RAM even if it is a *constant memory* (no data overwriting). The reason is that actually the memory must be initialized by the host in the configuration phase.

Vice versa, from the *streaming multiprocessor* point of view, only reading operations are permitted. This implies that the SM accesses to this memory only by means of the *read* unit of the pipeline. In the VHDL codes, the signals related to this memory are instantiated as *cmem_(...)*. Finally, the block diagram description of the *constant memory* is shown in fig. 3.8:

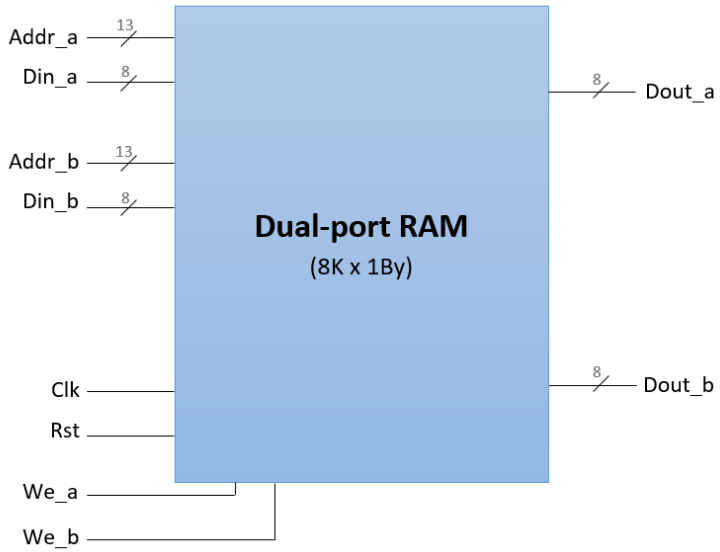


Figure 3.8: Constant memory block diagram description.

System Memory

The *system memory* is also implemented as a dual-port RAM with asynchronous reading operations and synchronous writing operations. This storage element is reserved for the instructions so:

- the writing operations is performed by the host;
- the reading operations is performed by each SM (in particular by the *fetch* units).

Differently from the memory elements previously described, the *system memory* is not directly written or read. Vice versa all the operations are performed by means of a system memory controller, that works as an interface between the memory and the elements that try to access. In VHDL codes, the signals related to this memory are instantiated as *sysmem_(...)*. The block diagram description of the *system memory* is shown in fig. 3.9:

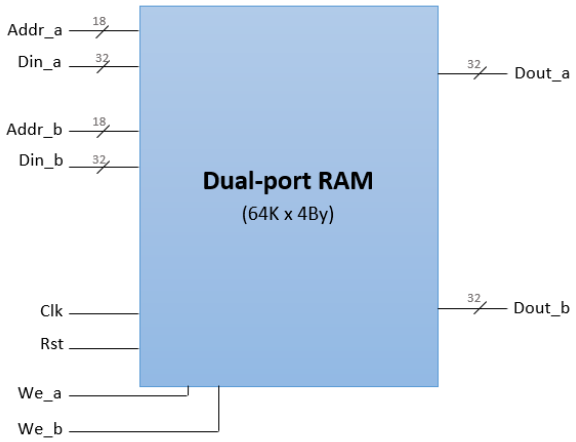


Figure 3.9: System memory block diagram description.

Shared Memory

The *shared memory* is instantiated as a dual-port register file with asynchronous reading operations and synchronous writing operations. This element is used as a communication memory between different cores of the same *streaming multiprocessor*. Moreover, also the *SM controller* access to the shared memory to write, during configuration, the *block header* (16By containing details about the block and grid dimensions and the block indexes) and the kernel data parameters (also 16By). In VHDL codes, the signals related to this memory are instantiated as *shmem_(...)*. The block diagram description of the *shared memory* is shown in fig. 3.10:

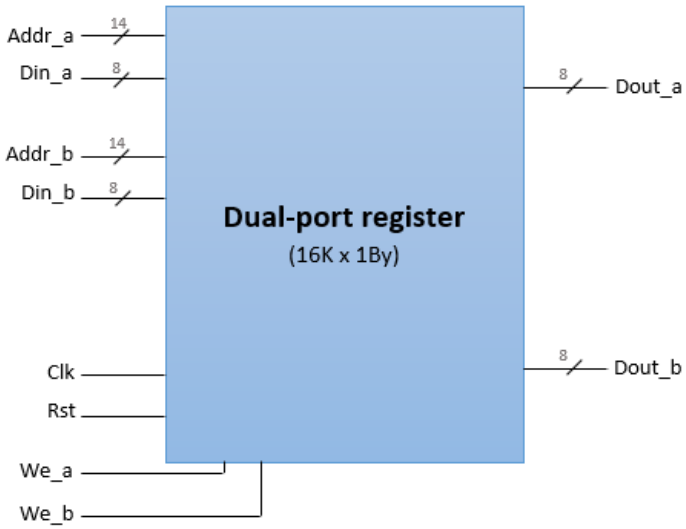


Figure 3.10: Shared memory block diagram description.

Vector Register File

The *vector register file* is the storage element reserved for the local and intermediate data generated by the cores inside each streaming multiprocessor. This register file is partitioned among the SPs of each SM and, as previously discussed, this depends on the SP-configuration selected. According to this, this module consists of an array of dual-port register (as much as the number of cores) and a module that evaluate the physical address starting from: *base address*, *register number*, *warp lane ID* and *number of cores*. In practice, depending on the SP configuration, the total address is evaluate as shown in fig 3.11:

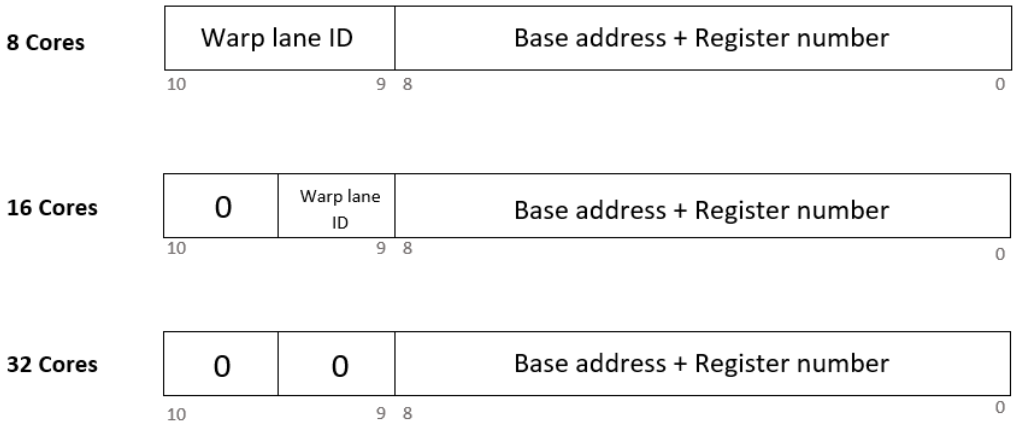


Figure 3.11: *Vector register file* address.

As regard the single dual-port register, the parallelism depends again on the number of cores. In general, since a single thread works with 512 locations in the *vector register file*, each SP is associated to a portion of *vector register file* of:

(512 locations of 32 bit) x (number of warp lanes)

As a consequence, in the worst case (8 cores) the *vector register file* contains 2048 locations (addressable with 11 bit of address). As the previous storage element, each dual-port register allows synchronous writing operations and asynchronous reading operations. In VHDL codes, the signals related to this memory are instantiated as *gprs_(...)*. The block diagram description of the *vector register file* is shown in fig. 3.12:

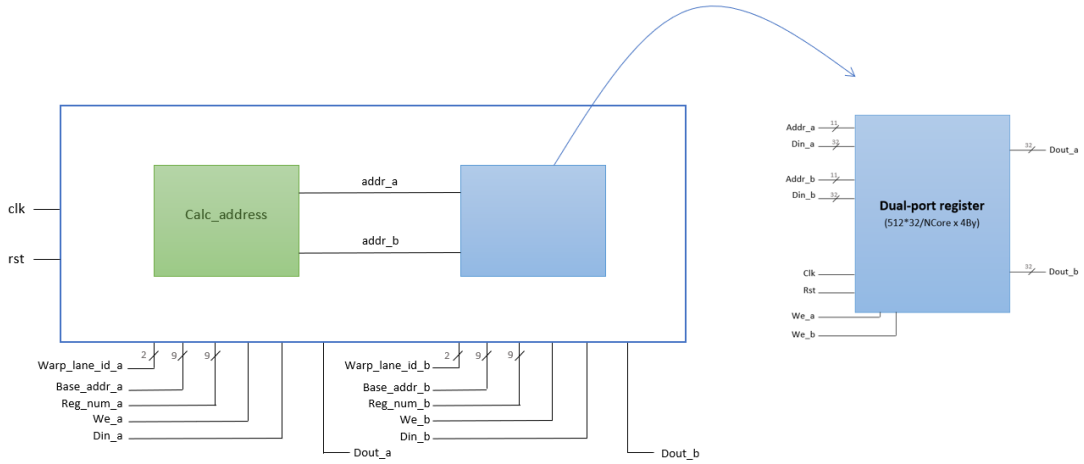


Figure 3.12: *Vector Register File* memory block diagram description.

Address Register File

The *address register file* is the memory reserved for the the storage of the address used to indirectly point to the shared memory, previously described. More in details, the complete address of the shared memory is obtained summing an offset to the address stored in the *address register file*. Since this memory contains addresses, it's used from the *read* unit of the pipeline, in which there are three read sources able to evaluate the complete address and access (among others storage elements) to the shared memory. The internal architecture of the *address register file* is similar to the *vector register file* one, except for the parallelism. So this storage module consists of an array of dual-port registers (synchronous writing and asynchronous reading) and a block able to evaluate the complete address to point to the *address register file* starting from: configuration (in terms of number of cores), warp and warp lane identifiers and register address. The complete address is shown in fig. 3.13 according to the number of cores:

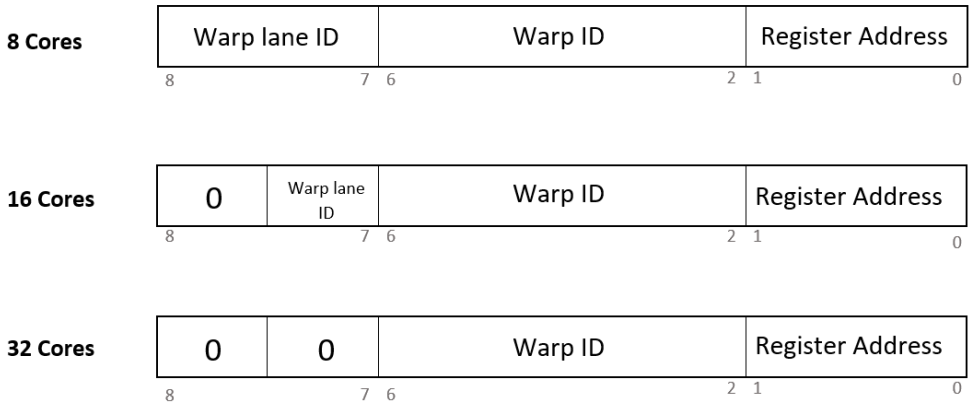


Figure 3.13: *Address Register File* address.

Also the parallelism of the single dual-port register depends on the number of cores exactly as occurs in the *vector register file*. Considering that a single thread works with 128 locations in the *address register file*, each SP is associated to a portion of *address register file* of:

(128 locations of 32 bits) x (number of warp lanes)

As a consequence, in the worst case (8 cores) the *address register file* contains 512 locations (addressable with 9 bit of address). In VHDL code, the signals related to this memory are instantiated as *addr_regs_(...)*. The block diagram description of the *address register file* is shown in fig. 3.14

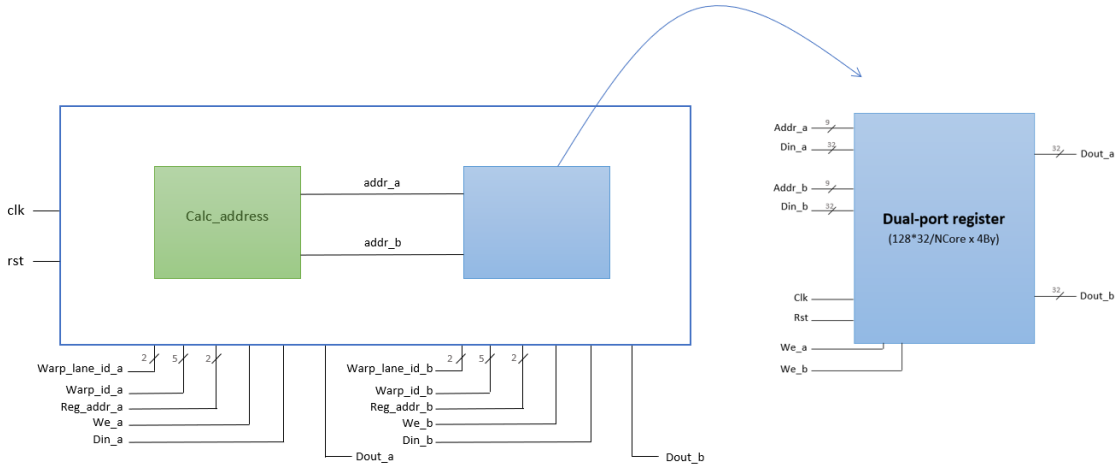


Figure 3.14: *Address Register File* memory block diagram description.

Predicate Register File

The *predicate register file* is the module that stores the predicate flags, specific bits used in the condition branches. More in details, these predicate flags are used to point to a *predicate look-up table* in which are contained the conditional instructions. The output of this look-up table is also combined to the warp mask to update the mask itself. As regard the internal architecture of the *predicate register file*, this is similar to the one of the *vector* and *address register file*, except for the parallelism. Considering that a thread works with 128 locations in the *predicate register file*, each SP is associated to a portion of *predicate register file* of:

(128 locations of 4 bits) x (number of warp lanes)

As possible to notice in the previous expression, each location consist of 4 bits and each bit corresponds to a specific flag. More in details:

- bit 0, corresponds to the "zero" flag;
- bit 1, corresponds to the "sign" flag;
- bit 2, corresponds to the "carry" flag;
- bit 3, corresponds to the "overflow" flag.

For what concern the address to point to this memory, the composition is the same discussed in the *address register file* section. The *predicate register file* is used by the read and *write* unit of the pipeline and, in VHDL codes, the signals related to this memory are instantiated as *pred_regs_(...)*. The block diagram description of the *predicate register file* is shown in fig. 3.15:

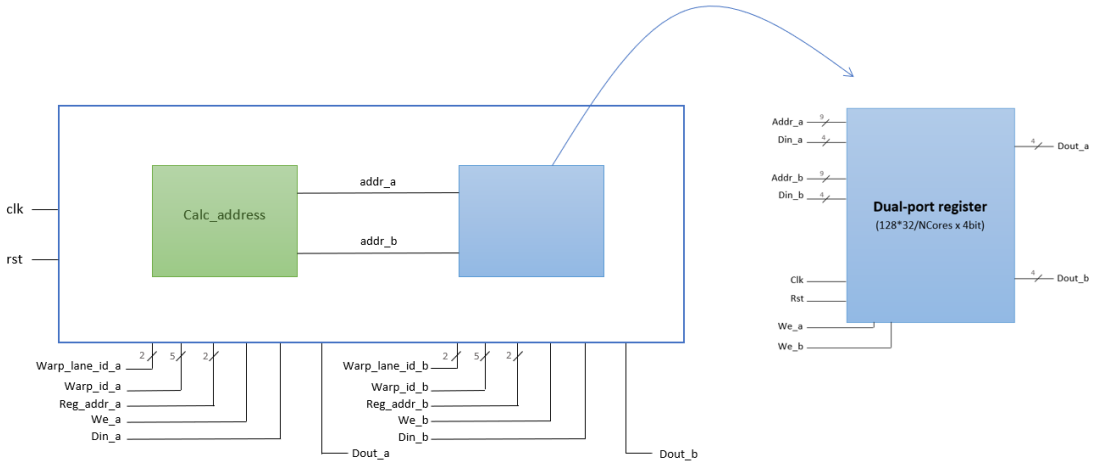


Figure 3.15: *Predicate Register File* memory block diagram description.

Warp Divergence Stack

The *warp divergency stack* is a stack used in each single warp to store information about the threads that, in case of branching execution, diverge from the correct (or usual) path. More in details, the information pushed in the stack is created starting from:

- current mask;
- instruction mask;
- next program counter;
- target address;
- some tokens properly encoded (ID_DIVERGE/ID_CALL/ID_BREAK/ID_SYNC);
- bit at 0 to fill empty bits.

The *warp divergence stack* is controlled at the interface by means of two control signals: *stack enable*, to enable the stack, and *push enable*, to perform push (if active) or pop (if not active) operations. In the pipeline this memory is used by the *read* unit and the *execution* unit (which generate push commands). According to the internal finite state of the *execution* unit, it's possible to notice that push operations in the stack occurs in the states:

- check branch, if no correct branch;
- check call, if the warp lane 0 is in the *execution* unit;
- check prebreak, if the warp lane 0 is in the *execution* unit;
- check join, if the warp lane 0 is in the *execution* unit.

Vice versa, the pop operations in the stack occurs in the states:

- check return, if stack is not empty;
- check break, if stack is not empty;
- stack pop.

In VHDL codes, the signals related to this memory are instantiated as *warp_div_(...)*. The block diagram description of the *warp divergence stack* is shown in fig 3.16:

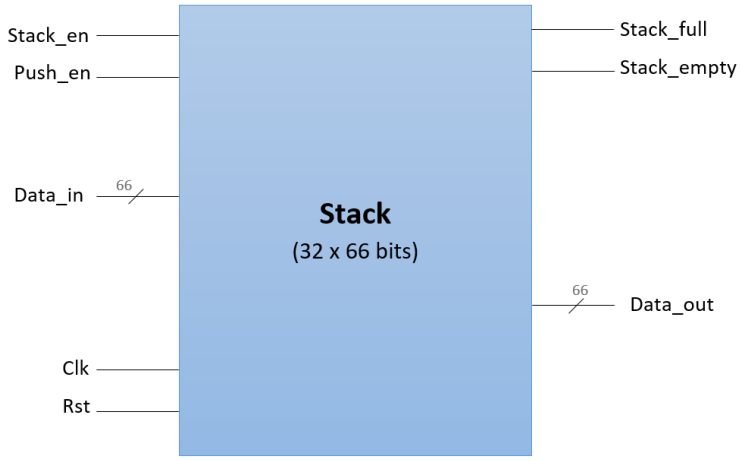


Figure 3.16: Warp divergence stack block diagram description.

Warp Pool Memory

The *warp pool memory* is a memory element present in the warp unit, reserved to store the so called *warp pool lanes*. Each warp has its own *warp pool lane*, a 128 bit-wide word containing the most significant warp information. The structure of each *warp pool lane* is the one shown in the fig. 3.17:

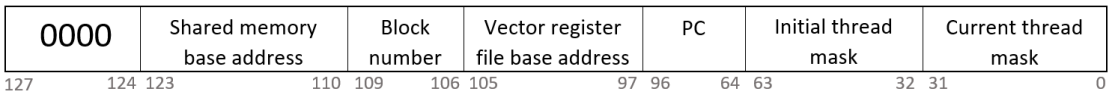


Figure 3.17: Warp pool lane.

As regards the internal architecture of the *warp pool memory*, it's based on a dual-port register with synchronous writing operations and asynchronous reading operations. More in details, this memory is filled in the configuration step by the *warp generator* module and then is read by the *warp scheduler*. In VHDL codes, the signals related to this memory are instantiated as *warp_pool_(...)*. The block diagram description of the *warp pool memory* is shown in fig. 3.18:

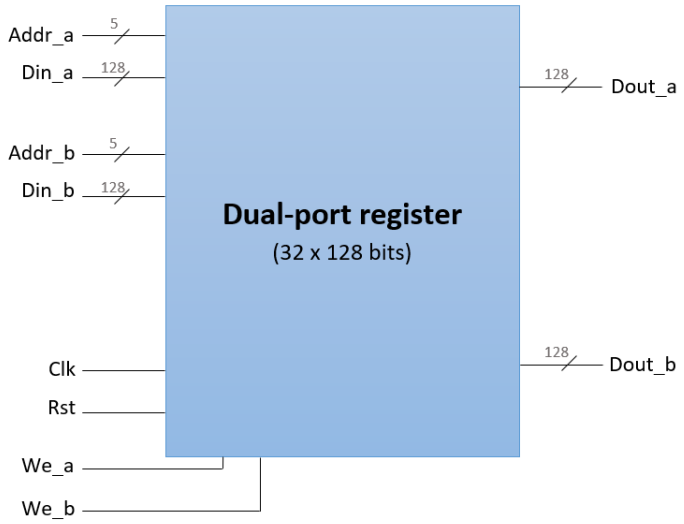


Figure 3.18: Warp pool memory block diagram description.

Warp State Memory

The *warp state memory* is another memory element present in the warp unit, reserved to store the *warp state*. The internal architecture is based on the dual-port register with synchronous writing operations and asynchronous reading operations. The possible states of a warp are four, coded on 2 bits as:

- 00 => ready (warp in idle);
- 01 => active (warp currently in the pipe);
- 10 => waiting fence (warp diverged);
- 11 => finished.

This memory is written by the warp modules (*generator*, *checker* and *scheduler*) and by the *write* unit. As regards the reading operations, these are performed by the *warp checker*, that according to the current warp state modify it or not, and by the *warp scheduler*, that decide how to handle the warp according to the warp state.
In the VHDL codes, the signals related to this memory are instantiated as *warp_state_(...)*.
The block diagram description of the *warp state memory* is shown in fig. 3.19:

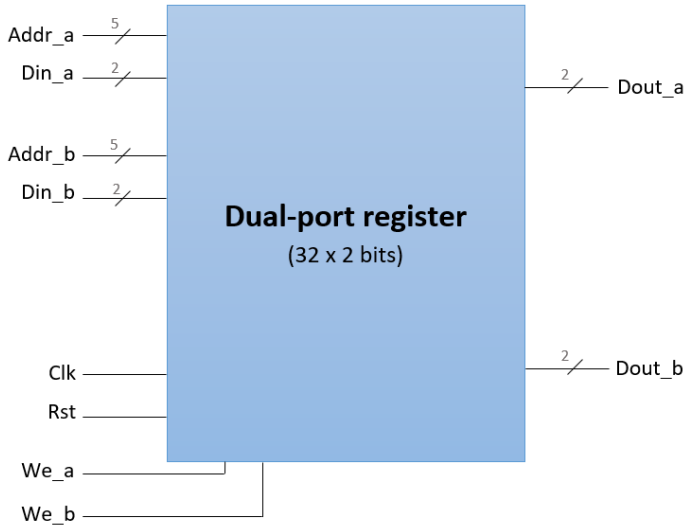


Figure 3.19: Warp state memory block diagram description.

Fence Register

The *fence register* is a memory element used to take track of all the threads blocked at a certain synchronization barrier in the case of internal divergency (explained in section 3.2.5). More in details, this store the *cta id* (corresponding to the block identifier) and the *fence enable* signals according to the corresponding load signals. Again, this works with synchronous writing operations and asynchronous reading operations. The internal architecture is based on an array of 32 fence register (since 32 corresponds to the maximum number of warps). So practically each warp has an associated fence register.
In the VHDL codes, the signals related to this memory are instantiated as *fence_regs_(...)*.
The block diagram description of the single *fence register* is shown in fig. 3.20:

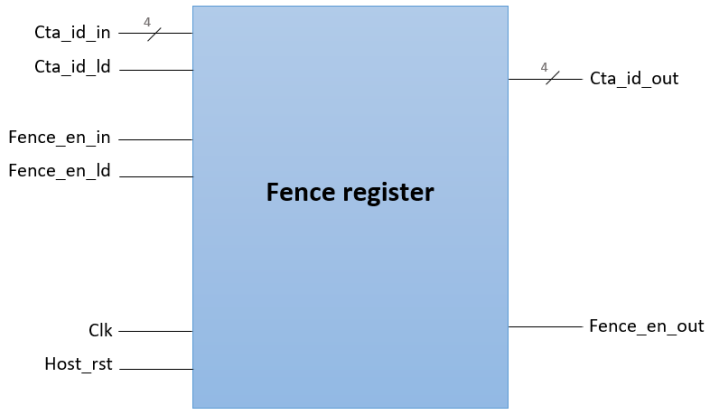


Figure 3.20: Single fence register block diagram description.

3.2.5 Internal Divergency

The FlexGrip GPGPU is able to support in hardware the thread-level branching. This implies that, during conditional branches, it's possible that different threads follow different path, generating the so called *internal divergency*. In this case is essential to reach a convergency point, also called *synchronization barrier*. Practically what the FlexGrip does is:

- to temporarily mask (so deactivate) threads that diverge from the nominal execution;
- to wait that the active threads (also called taken) reach the synchronization barrier;
- to move the control to the previously masked threads (also called not-taken) and wait that also these reach the synchronization barrier;
- when all threads reach the same synchronization barrier, the execution of the task can continue.

In particular, when the divergency occurs, the warp state is forced to *waiting fence* until all threads are synchronized. To take track of the program counter of all the threads, the *warp divergence stack* is filled by the *execution* unit of the pipeline. When the divergency occurs, there is a prebreak state, to save all the information necessary to restore the execution after divergency. Vice versa, when the stack is empty and so the threads are all synchronized, a NOP operation is executed to communicate the end of divergency.

3.3 Configuration and standard execution flow

In this section a detail description of the configuration step and the execution flow are provided, recalling the FlexGrip general architecture, shown in fig. 3.21.

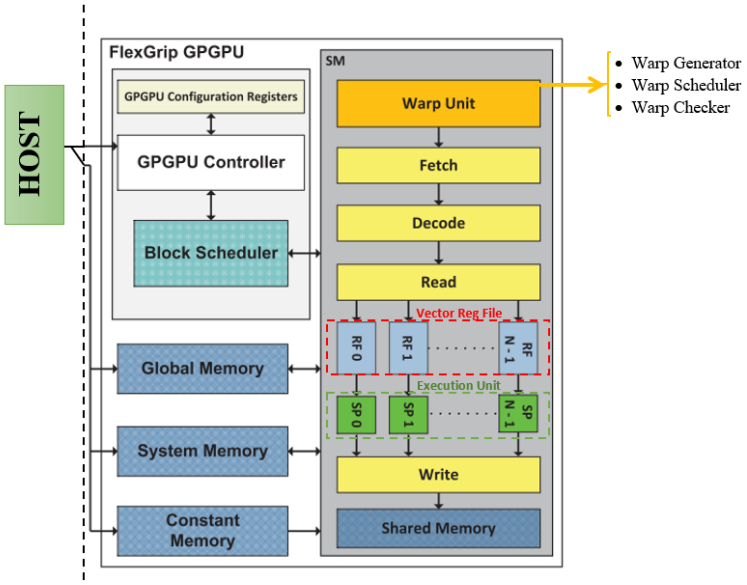


Figure 3.21: FlexGrip general block diagram description [5].

First, the *host* sends to the GPGPU controller the information that must be stored in the configuration memory (e.g. number of blocks, grid and block dimensions, thread level parallelism, memory limits, etc). When the configuration memory is filled, the control is moved to the *block scheduler*, that creates blocks and schedules them, providing to the *SM controller* all the information related to the block (e.g. the position in the grid or the number of threads per block). Now the control is passed to the *SM controller*, which calculates the ID of blocks and threads (inside each block) and divides the shared memory among different threads, writing the block header (16By) and the kernel parameters. Next, *SM controller* divides the *Vector Register File* among all threads, writing the thread_ID in the R0 registers (first register of each portion). After that the *warp unit* takes the control and:

- generates the warp, filling the warp pool memory and warp state memory (*warp generator*);
- schedules the warps (*warp scheduler*).

Finally, the *pipeline* can start to elaborate the warp, performing all the operations needed to compute and store the results. In case of branching, the internal warp divergency could occur at the *execution* unit level and, in this case, system is handled to reach a synchronization barrier before to schedule a new instruction. At the end of *pipeline* elaboration, the *warp checker* (inside the warp unit) checks if the warp execution is completed examining specific masks and change the state of warp based on the check results. According to the warp state at the *warp checker* output, the *warp scheduler* schedules a new warp lane of the warp or a new warp. From now on the FlexGrip continues the elaboration of the instructions until the task is completely executed

CHAPTER 4

Proposed Method

As discussed in the chapter 2, several techniques can be implemented to harden the electronic systems against SEUs, but implementation of these always implies hardware overhead and, as a consequence, increase of occupied area, power consumption and performance degradation. In order to reduce these overheads still guaranteeing radiation hardening, a proper solution is the *selective hardening*: apply the hardening techniques only to the *critical resources*. An obvious question arises, which are the critical resources?
A resource can be defined critical when, if affected by SEU (or any other radiation effects), the consequence on the system or the output is significant, e.g. output errors, performance degradation, etc. This implies that the definition of *critical resource* depends on the designer choices. If the goal is to protect system against output errors, probably all the resources that could generate an output error are considered critical and so hardened. Vice versa, if the goal is to guarantee the system speed without a high accuracy (e.g. in real time systems), a critical resource will be the one that, affected by SEU, generates a performance degradation above a specific threshold.
From this perspective, it's essential to build up a *monitoring system* able to track the most meaningful internal signals of each module and so to analyse the SEU impact not only in the block in which is generated but in the entire execution chain. Combined with this monitoring system, a *fault injector* tool is used to virtually generate a bit-flip in specific memory locations and simulate the behaviour of the system on simulation environments as Modelsim.

4.1 Performance Analysis

The *monitoring systems* has been achieved identifying the most significant time intervals and, as a consequence, the signals that must be tracked. In order to be actually able to track these signals, it's essential to modify the VHDL codes, expanding the entities of the targeted components from the bottom to the top module, according to the architecture hierarchy, moving these signals up to the top module interface. This is mandatory since the testbench interacts with the device under test (DUT), so FlexGrip, declaring it as a component and consequently

has access only to the FlexGrip I/O signals. Considering the assumptions described in the section 4.1.1, the time intervals considered of interest are:

Time interval	**Initial time**	**End time**
Total Warp Execution	*fetch_en* = 1	*fetch_en* = 1
Instruction Fetching	*fetch_en* = 1	*fetch_done* = 1
Warp Lane Fetching	*pass_en* = 1	*fetch_done* = 1
Warp Lane Decoding	*fetch_done* = 1	*decode_done* = 1
Warp Lane Reading	*decode_done* = 1	*read_done* = 1
Warp Lane Execution	*read_done* = 1	*execution_done* = 1
Warp Lane Writing	*execution_done* = 1	*write_done* = 1
First Warp Lane Scheduling	*warp_scheduler_en* = 1	*pass_en* = 1
Next Warp Lane Scheduling	*pass_en* = 1	*pass_en* = 1
Total Warp Generation	*warp_generator_en* = 1	*warp_generator_done* = 1
Warp Check Latency	*write_done* = 1	*warp_checker_stall_out* = 0
Single Block Scheduling	*block_scheduler_en* = 1	*SM_controller_en* = 1
Single Block Execution	*SM_controller_en* = 1	*SM_controller_done* = 1

Table 4.1: Performance analysis: time intervals.

The intervals described in the table 4.1 will be discussed more in details, focusing also to the implementation of the monitoring systems, in section 4.1.2. These intervals permit to collect information related to the performance of each of the main module of the systems and, consequently, to identify the identify the impact of SEUs on the performance.
As discussed in chapter 1, SEUs are generated inside the memory elements as a bit-flip in the stored words. Obviously a SEU generates an impact on the system if the data corrupted by the SEU is used so, in terms of memory element, if it is read. This suggests a new possible monitoring: evaluate the time intervals in which a location is sensitive to SEUs. This corresponds to the time interval between the first writing and the last reading on the same memory location. In fact, assuming that a SEU occurs in a specific location after the last reading, this will not propagate in the structure and, at the contrary, will be overwritten by the successive writing. So, this study allows to identify two different time intervals:

- sensibility time interval, between first writing and last reading of the same location;
- indifference time interval, evaluating the time interval between to writing operations and subtracting the previous time interval. Consequently, this corresponds to the time interval between the last reading and the next writing of the same location.

To sum up, additional time intervals evaluated on the memory elements are contained in the table 4.3.

Time Intervals
First Write Last Read (VRF)
Write to Write (VRF)

Table 4.2: Performance analysis: memory elements monitoring.

Finally, to achieve a complete investigation on the memory elements and so identify the critical ones, a memory usage studied has been performed, tracking the number of access to the memories and the type of access: reading or writing. Also this information results to be essential because, according to what discussed previously, the more a location is read the higher is the probability that a SEU in that location propagates, impacting the entire system performances. Obviously, the impact on the system depends also of the type of elements stored in that memory location: a SEU in a predicate flag or in the configuration memory probably would be more dangerous that a SEU in a general purpose register. In conclusion, the memory usage information monitored are highlighted in the table 4.2.

Monitored Info
Vector Register File Usage
Vector Register File Bank Usage (Writing)
Vector Register File Bank Usage (Reading)
Vector Register File Location Usage (Writing)
Vector Register File Location Usage (Reading)
Address Register File Usage
Address Register File Bank Usage (Writing)
Address Register File Bank Usage (Reading)
Address Register File Location Usage (Writing)
Address Register File Location Usage (Reading)
Predicate Register File Usage
Predicate Register File Bank Usage (Writing)
Predicate Register File Bank Usage (Reading)
Predicate Register File Location Usage (Writing)
Predicate Register File Location Usage (Reading)

Table 4.3: Performance analysis: memory time intervals.

4.1.1 Assumptions

The monitoring system has been implemented and tested considering two significant assumptions:

- instructions mapped in a single block with a single warp of 32 threads;
- 8 SPs configuration, i.e. warp organized in four warp lanes.

This implies that, during the warp execution, in the pipeline there is a situation as the one described in fig. 4.1.

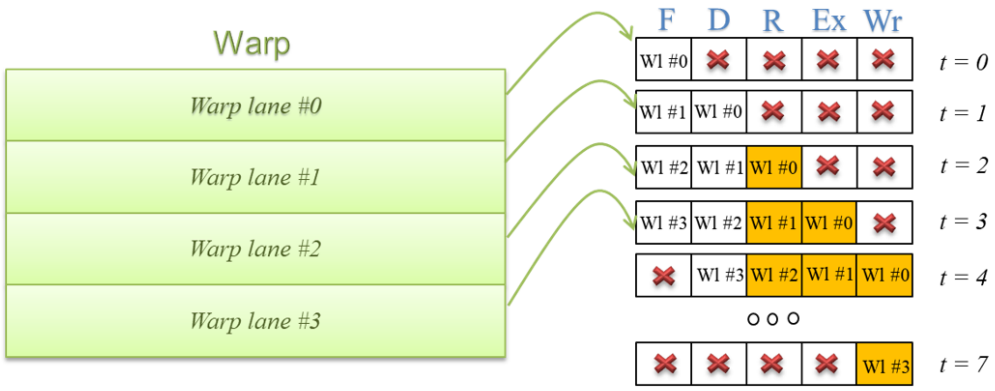


Figure 4.1: Warp execution in case of signle warp.

As shown in fig 4.1, the warp enters in the pipeline a warp lane per time. When the first warp lane (*warp lane #0*) enters in the *fetch* unit, first element of the pipe, the rest of the pipe is empty and consequently each module, except for the *fetch* one, is in an idle state. Then the next warp lane (*warp lane #1*) will enter in the pipe when the first one leaves the *fetch* unit and passes to the *decode* unit, and so on. The real consequence of the assumption to use a single warp is that, as shown in fig. 4.1, until the last warp lane doesn't leave the pipe the (*warp lane #0*) cannot enter again in the pipe. Consequently, this implies that a new instruction cannot be fetched if the previous one has not been completely executed in all the threads of the warp. This is a key point of the following analysis since allows to evaluate the Total Warp Execution as the time interval between two fetch_en (except for the last warp execution, evaluated as the interval between the fetch_en and the warp_scheduler_done signals).
Vice versa, in systems in which the instruction set is mapped in multiple warps the execution is handled as shown in fig. 4.2. The difference with respect to the case previously discussed is that now, when the last warp lane (*warp lane #3*) of the warp #0 passes from the *fetch* unit to the *decode* unit, the first warp lane (*warp lane #0*) of the warp #1 can enter in the pipe. This guarantees the maximum core usage but, obviously, complicates the monitoring systems. In this case in fact, some modifications must be added to the monitoring system described in the following sections. A possibility is to take into account also the warp_ID, associating to it the intervals shown in table 4.1. This implies that the warp_ID signal must be extended so that it passes through each monitored module and then, in the testbench, the logging processes will also consider the warp_ID related to each time interval evaluated.

Finally, as regard the fault injection study, the assumption is on the SEUs target modules. In this study the module considered sensitive to SEU are the: *vector register file*, *address register file* and *predicate register file*.

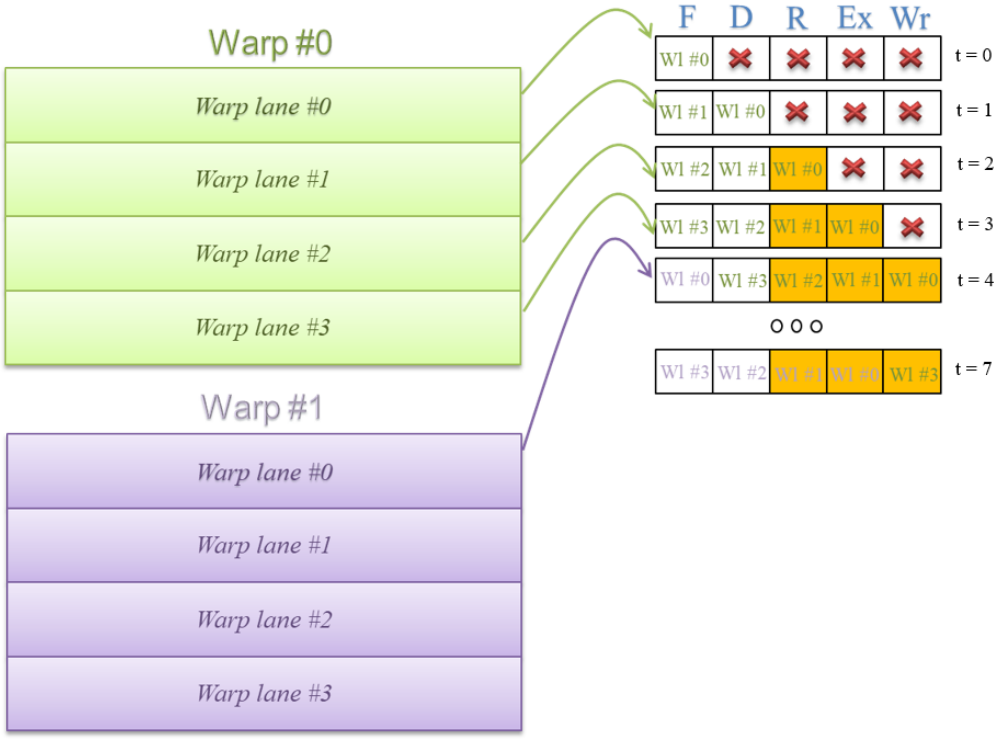


Figure 4.2: Warp execution in case of multiple warps.

4.1.2 Main module probings

In this section the monitoring system will be analysed focusing singularly on each module. In the testbench each module has an associated process dedicated for monitoring and printing the desired time intervals in specific output files. In order to guarantee an analysis as clear as possible, the description of each process is associated to a pseudo code that sums up the most significant monitoring strategies.

Fetch unit

The *fetch* unit, according to the table 4.1 and the description in section 3.2.3, can be monitored in order to achieve two time intervals:

- Total Warp Execution;
- Instruction Fetching;
- Warp Lane Decoding.

In both case the intervals have been evaluated as the time interval between the activation of the initial and done signal. For instance, the Total Warp Execution is measured between the rising edge of the fetch_en signal and the rising edge of the successive fetch_en or of the scheduler_done signals (so this interval consider also the scheduling time of the next instruction). This concept has been applied to all the measured time intervals. In order to

guarantee this condition, a proper offset has been introduced to remove the delay introduce by the monitoring system itself. As regard the Instruction Fetching, this time interval has been measured as the interval between the rising edge of the fetch_en and the rising edge of the fetch_done signal. The Warp Lane Fetching interval (interpreted as the fetching interval of the warp lanes except the first one) instead has been evaluated as the interval between the rising edge of the pass_en and the rising edge of the fetch_done signal again. The reason why the initial time of these fetching interval can be triggered by the fetch_en or the pass_en is that: the first signal is used during the first warp lane fetching to communicate to the *fetch* unit that the instruction must be fetched by the system memory; vice versa the pass_en signal is used during the successive warp lane fetching to communicate to the *fetch* unit that there is no need to fetch the instruction from the system memory, because it is already present in the output registers of the *fetch* unit itself.
In this way the monitoring system is able to track the Total Warp Execution and the Instruction Fetching time intervals, but there is the necessity to associate these intervals to the different instructions and warp lanes, in order to be able to compare the performance among these. Consequently, the monitoring system of the *fetch* unit receives also the instruction and the warp_lane_ID signals. For the same reasons, the instruction and the warp_lane_ID signals have been passed also in the other monitoring processes, implementing the entity extension described in section 4.1.
Finally, as regard the output time intervals, these have been provided in terms of clock cycles. Nevertheless, in order to provide more information and have the possibility to check the coherence of the monitoring system with respect to the Modelsim simulation, also the absolute initial and final time instants have been printed in the output files.
The pseudo code related to the *fetch* unit monitoring system is shown in fig. 4.3.

```
IF fetch enable rising edge OR scheduler enable rising edge:
        Save time instant in time
        Save time instant in init_time_1
        IF fetch enable rising edge IS NOT the first:
                Total Warp Execution = difference between two successive time

ELSIF pass enable  rising edge:
        Save time instant in init_time_2



IF fetch done rising edge:
        Save time instant in end_time
        IF warp lane rising IS the first:
                Instruction Fetching = difference between end_time and init_time_1
        ELSE:
                Warp Lane Fetching = difference between end_time and init_time_2
```

Figure 4.3: *Fetch* unit monitoring pseudo code.

Decode unit

The *decode* unit, according to the table 4.1 and the description in section 3.2.3, can be monitored in order to evaluate the Warp Lane Decoding time interval. This interval has been measured evaluating the time interval between the rising edge of the fetch_done signal, that corresponds to the enable signal of the *decode* unit, and the rising edge of the decode_done signal. As in the *fetch* unit, the decoding intervals are associated to the instruction and the warp_ID and provided at the output in terms of clock cycles and initial and final time. The pseudo code related to the *decode* unit monitoring system is shown in fig. 4.4.

```
IF fetch done rising edge:
        Save time instant in init_time

IF decode done rising edge:
        Save time instant in end_time
        Warp Lane Decoding = difference between end_time and init_time
```

Figure 4.4: *Decode* unit monitoring pseudo code.

Read unit

The *read* unit, according to the table 4.1 and the description in section 3.2.3, can be monitored in order to evaluate the Warp Lane Reading time interval. This interval has been measured as the time interval between the rising edge of the decode_done signal, that corresponds to the enable signal of the *read* unit, and the rising edge of the read_done signal. The reading intervals are associated to the instruction and the warp_ID and provided at the output in terms of clock cycles and initial and final time. The pseudo code related to the *read* unit monitoring system is shown in fig. 4.5.

```
IF decode done rising edge:
        Save time instant in init_time

IF read done rising edge:
        Save time instant in end_time
        Warp Lane Reading = difference between end_time and init_time
```

Figure 4.5: *Read* unit monitoring pseudo code.

Execution unit

The *execution* unit, according to the table 4.1 and the description in section 3.2.3, can be monitored in order to evaluate the Warp Lane Execution time interval. This interval has been measured as the time interval between the rising edge of the read_done signal, that corresponds to the enable signal of the *execution* unit, and the rising edge of the execution_done signal. The execution intervals are associated to the instruction and the warp_ID and provided at the output in terms of clock cycles and initial and final time. The pseudo code related to the *execution* unit monitoring system is shown in fig. 4.6.

```
IF read done rising edge:
        Save time instant in init_time

IF execution done rising edge:
        Save time instant in end_time
        Warp Lane Execution = difference between end_time and init_time
```

Figure 4.6: *Execution* unit monitoring pseudo code.

Write unit

The *write* unit, according to the table 4.1 and the description in section 3.2.3, can be monitored in order to evaluate the Warp Lane Writing time interval. This interval has been measured as the time interval between the rising edge of the execution_done signal, that corresponds to the enable signal of the *write* unit, and the rising edge of the write_done signal. The write intervals are associated to the instruction and the warp_ID and provided at the output in terms of clock cycles and initial and final time. The pseudo code related to the *write* unit monitoring system is shown in fig. 4.7.

```
IF execution done rising edge:
        Save time instant in init_time

IF write done rising edge:
        Save time instant in end_time
        Warp Lane Writing = difference between end_time and init_time
```

Figure 4.7: *Write* unit monitoring pseudo code.

Warp scheduler

The *warp scheduler*, according to the table 4.1 and the description in section 3.2.2, can be monitored in order to evaluate:

- First Warp Lane Scheduling;
- Next Warp Lane Scheduling.

The First Warp Lane Scheduling interval has been measured as the time interval between the rising edge of the warp_scheduler_reset signal and the rising edge of the pass_en signal. Vice versa, the Next Warp Lane Scheduling intervals have been measured as the time interval between two successive rising edge of the pass_en signal. These intervals are associated to the warp_ID but cannot be associated to the instruction, that is not fetched yet. Consequently, to have an indirect reference with the instruction, these intervals have been associated to the program counter saved in the warp lane (as described in section 3.2.4). The outputs are provided in terms of clock cycles and initial and final time. The pseudo code related to the *warp scheduler* monitoring system is shown in fig. 4.7.

```
IF warp scheduler reset rising edge:
        Save time instant in init_time

IF pass enable rising edge:
        Save time instant in end_time
        IF pass enable IS the first:
                First Warp Lane Scheduling = difference between end_time and init_time
        ELSE:
                Next Warp Lane Scheduling = difference between two successive end_time
```

Figure 4.8: *Warp scheduler* monitoring pseudo code.

Warp generator

The *warp generator*, according to the table 4.1 and the description in section 3.2.2, can be monitored in order to evaluate the Total Warp Generation time interval. This interval has been measured as the time interval between the rising edge of the warp_generator_en signal and the rising edge of the warp_generator_done signal. In the cases in which there are more than one warp a possibility could be to associate the warp_ID. The outputs are provided in terms of clock cycles and initial and final time. The pseudo code related to the *warp generator* monitoring system is shown in fig. 4.9.

IF warp generator enable *rising edge*:
Save time instant in **init_time**

IF warp generator done *rising edge*:
Save time instant in **end_time**
Total Warp Generation = *difference between* **end_time** *and* **init_time**

Figure 4.9: *Warp generator* monitoring pseudo code.

Warp checker

The *warp checker*, according to the table 4.1 and the description in section 3.2.2, can be monitored in order to evaluate the Warp Check Latency time interval. This interval has been measured as the time interval between the rising edge of the write_done signal and the falling edge of the warp_check_stall_out signal. In the cases in which there are more than one warp a possibility could be to associate the warp_ID. The outputs are provided in terms of clock cycles and initial and final time. The pseudo code related to the *warp checker* monitoring system is shown in fig. 4.10.

IF write done *rising edge*:
Save time instant in **init_time**

IF warp checker stall out *falling edge*:
Save time instant in **end_time**
Warp Check Latency = *difference between* **end_time** *and* **init_time**

Figure 4.10: *Warp checker* monitoring pseudo code.

Block scheduler

The *block scheduler*, according to the table 4.1 and the description in section 3.2.1, can be monitored in order to evaluate the Single Block Scheduling time interval. This interval has been measured as the time interval between the rising edge of the Block_scheduler_en signal and the rising edge of the SMController_en signal. The outputs are provided in terms of clock cycles and initial and final time. The pseudo code related to the *warp checker* monitoring system is shown in fig. 4.11.

IF block scheduler enable *rising edge*:
Save time instant in **init_time**

IF SM Controller enable *rising edge*:
Save time instant in **end_time**
Single Block Scheduling = *difference between* **end_time** *and* **init_time**

Figure 4.11: *Block scheduler* monitoring pseudo code.

SM controller

The *SM controller*, according to the table 4.1, can be monitored in order to evaluate the Single Block Execution time interval. This interval has been measured as the time interval between the rising edge of the SMController_en signal and the rising edge of the SMController_done signal. The outputs are provided in terms of clock cycles and initial and final time. The pseudo code related to the *warp checker* monitoring system is shown in fig. 4.12.

IF SM Controller enable *rising edge*:
Save time instant in **init_time**

IF SM Controller done *rising edge*:
Save time instant in **end_time**
Single Block Execution = *difference between* **end_time** *and* **init_time**

Figure 4.12: *SM controller* monitoring pseudo code.

4.1.3 Memory probings

As in section 4.1.2, in this section a description of the memory monitoring system has been provided and combined with a pseudo code that sum up most significant monitoring strategies. Referring to the time intervals of interest discussed in the section 4.1, the monitoring procedure used to track the memory accesses are the same considering the three memory under analysis: *vector register file*, *address register file* and *predicate register file*.

The memories under test, as discussed in section 3.2.4, consist of an array of dual-port register file and, studying the internal control and execution flow of the FlexGrip, it has been possible to notice that port A is controlled by the *read* unit, while port B by the *write* unit (except for the *vector register file*, in which port A is also used by the *SM Controller* during the thread_ID storing). In order to track the memory accesses for reading and writing operation, the monitoring system works on counters incremented under proper conditions. More in details:

- during writing operations, when the *write_en* signal of one of the two ports is activated, depending on the position of the *write_en* in the *write_en_array* (considering the assumption of 8 SPs the array consist of 8 *write_en*, one for each SP), the counter corresponding to that specific register file has been incremented. Moreover, examining the 2MSBs of the address signal (at port A or B depending on the *write_en* signal), the bank counter of a specific register file has been incremented.

- during reading operations, the problem is that the register files are asynchronous and, consequently, each time that the register is not in the write mode it is in the read mode. Nevertheless, in the analysis the goal is to track only the demanded readings. For this purpose, a signal has been added to the internal FSM of controllers of the three memories under test to inform the testbench when a reading operation is actually demanded. Again, looking at the position in the *write_en_array* in which the *write_en* signal is deactivated and looking at the 2MSBs of the address, the proper counters can be incremented, both for memory and bank usage evaluation.

The location usage has been tracked using the same procedure adopted for the memory and bank usage combined with an investigation of the address related to each single access. Consequently, to efficiently handle the location counter, a three-dimensional array has been created as shown in fig. 4.13.

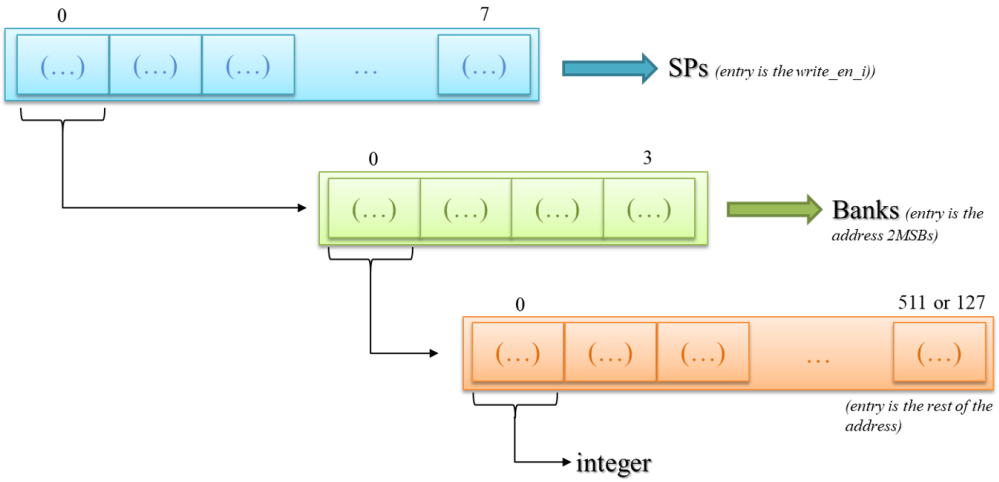


Figure 4.13: Location counters organization.

The counter array results to be organized as follow:

- the top-level array (blue one in fig. 4.13) consists of a number of elements equals to the number of cores (so depends on the configuration) and it's accessed by the *write_en* signal in the *write_en_array*.

- the medium-level array (green one in fig 4.13) consists of a number of elements equals to the number of banks (so depends on the configuration) and it's accessed by the 2MSBs of the address signal.

- the bottom-level array (orange one in fig 4.13) consists of a number of elements equals to the number of register for each bank (512 registers for the *vector register file* and 128 register for the *address* or *predicate register file*) and it's accessed by the remaining part of the address.

Finally, as regards the time intervals, these can be easily tracked exploiting the same procedures shown described for the memory, bank and location usage. In particular:

- the first write - last read interval can be tracked saving the time instant in which a writing occurs in a specific location (using as reference the three-dimensional array) and the time instant in which a reading is demanded. Consequently, overwriting at each read the reading time instant, when a new write operation is demanded, the monitoring system provide at the output the difference between the two stored time instant value and the overwrite the write time instant.

- the write to write interval can be tracked using the same strategy described for the first write - last read interval. The difference is that in this case the procedure saves two successive write instant, providing at the output the difference.

The pseudo code related to the memory monitoring system, valid for each of the three monitored memory modules, is shown in fig. 4.14.

```
IF write enable (at port j) rising edge:
        Increment the Register File write counter depending on the  write enable position in write enable array
        Increment the Bank write counter depending on the  2MSBs of address
        Increment the Location write counter depending on the remaining address bit

        IF there was a at least a reading before:
                First Write – Last Read interval = read_time – write_time

        Save time instant in write_time

        IF this IS NOT the first writing:
                Write to Write interval = difference between two successive write_time


IF read signal (related to port j) in the rising edge memory controller:
        Increment the Register File read counter depending on the  write enable (at 0) position in write enable array
        Increment the Bank read counter depending on the  2MSBs of address
        Increment the Location read counter depending on the remaining address bit
        Save the time instant in read_time

IF SM Controller done rising edge:
        Print the results
```

Figure 4.14: Memory monitoring pseudo code.

CHAPTER 5

Experimental Results

The main goal of this study is to provide, by means of the monitoring system described in chapter 4, a collection of performance and memory usage information that, combined with the fault injection method, permit to identify the critical resources and to use the selective hardening techniques. In this chapter the results of the performance analysis on specific applications will be studied in details and a description of the fault injector will be provided. Furthermore, the different fault injection strategies will be analysed and possible development and implementation of this monitoring system will be taken into account.

5.1 Performance analysis results

This section will focus on the description of the performance analysis results for different applications and versions, considering a FlexGrip configuration of 8 scalar processors (so four warp lanes). The applications under test, in collaboration with the Universidade Federal do Rio Grande do Sul (UFRGS), are:

- *Vector_sum*;
- *Vector_sum* with *resilient global store* and *load* instructions;
- *Sort*, results in appendix A;
- Sort with *resilient global store* and *load* instructions, results in appendix B;
- *FFT*, results in appendix C;
- *FFT* with *resilient global store* and *load* instructions, results in appendix D.

5.1.1 Vector_sum

The *Vector_sum* application is the simplest among the monitored ones. In this application two input vectors, stored in the global memory, are read, summed and finally stored in the global memory. According to the assumptions discussed in the section 4.1.1, in the following sections will be analysed in detail the monitoring results for each single module under test.

Fetch Unit

Starting from the *fetch* unit, first element of the pipeline, the analysis of this module permits to achieve two important information: the latency of the *fetch* unit for each warp lane but also, according to the assumptions in section 4.1.1, the time interval required to completely execute each instruction. This last time interval is one of the most significant figures of merit of this study since, simply comparing the instruction execution time the designer is able to understand if there is a performance degradation and which instruction is more influenced by the hardening technique. Moreover, the same comparison permits also to identify, if the performance degradation is due to SEUs and not to the applied techniques, which instruction is more sensitive to the SEUs effects.
The *fetch* unit monitoring results are plotted in fig 5.1 and collected in table 5.1.

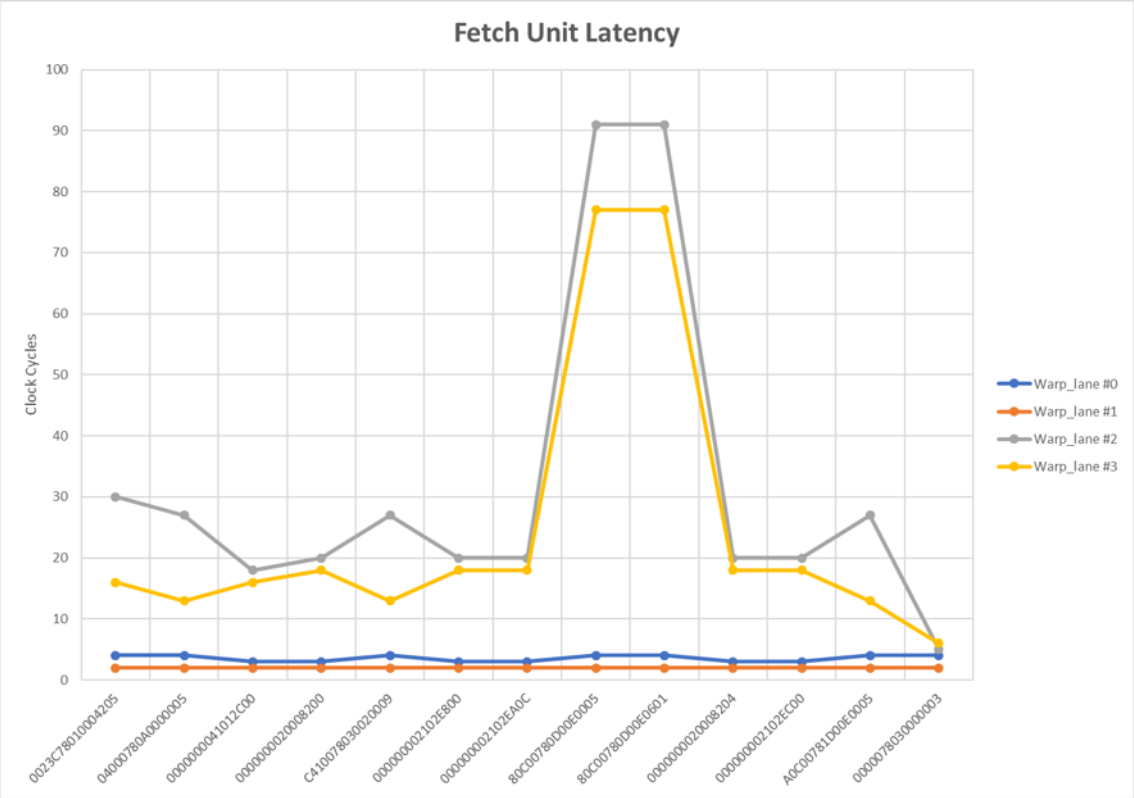


Figure 5.1: Vector_sum: *fetch* unit monitoring results.

Instruction	W.Lane #0	W.Lane #1	W.Lane #2	W.Lane #3
0023C78010004205	4	2	30	16
04000780A0000005	4	2	27	13
0000000041012C00	3	2	18	16
0000000020008200	3	2	20	18
C410078030020009	4	2	27	13
000000002102E800	3	2	20	18
000000002102EA0C	3	2	20	18
80C00780D00E0005	4	2	91	77
80C00780D00E0601	4	2	91	77
0000000020008204	3	2	20	18
000000002102EC00	3	2	20	18
A0C00781D00E0005	4	2	27	13
0000078030000003	4	2	5	6

Table 5.1: Vector_sum: fetch unit monitoring results.

In fig. 5.1 it's possible to notice that the *fetch* unit latency changes among different warp lanes. More in details, during the first and second warp lanes the *fetch* unit is able to performs the fetch operation in a limited number of clock cycles. Vice versa, during the fourth and the third warp lanes the latency of the *fetch* unit increases significantly, in particular during the third warp lane. In order to understand this behaviour, it's important to recall how the instruction is executed inside the streaming multiprocessor and inside the pipeline. Considering the fig. 4.1, the warp lanes enter in the pipeline one per time as the warp lanes that are in the pipeline move toward right, leaving the *fetch* unit "empty" (so free to serve the next warp lane). Inside the pipeline, the *fetch* unit and the *decode* unit are simple both from architectural and state machine point of view. This implies that these units are able to complete their job in few clock cycles. Vice versa, the other pipeline units are more complex since *read* and *write* unit have to access to memory elements, according to specific opcodes generated at the *decode* unit output, while the *execution* unit is the real computation unit. This implies that these three final units will intrinsically spend more time to complete the demanded jobs and consequently, recalling the handshake mechanism generated in the pipe thanks to the stall and done signals (shown in fig. 3.6), they will introduce on the previous units in the pipe, e.g. the *fetch* unit, a significant delay.
Focusing now on fig. 5.1, this means that the higher *fetch* unit latency shown for the third and fourth warp lanes is due to the fact that the final units in the pipeline needs time to generate the outputs and be able to accept new inputs. More in details, when the third warp lane enter in the pipeline, the *read* unit is for the first time activated in order to serve the task demanded by the first warp lane and, as previously discussed, this introduce a delay that affects also the *fetch* unit, that is serving the third warp lane. The same situation occurs for the fourth warp lane in which the *read* unit serves the second warp lanes and the *execution* unit the first one. The reason why the latency associated to the third warp lane is higher with respect to the one associated to the fourth warp lane is related to the internal state machine

of the *read* unit. In particular, the *read* unit, when the first warp lane arrives, execute a certain number of checks that are not executed for the successive warp lane (since they are related to the same structure) and consequently the delay associate to these "check" states is generated only for the first warp lane.
Furthermore, it's possible to notice that the *fetch* unit latency associated to the first warp lane is higher than the one associate to the second warp lane. This is due to the fact that, as discussed in section 3.2.3, during the first warp lane the *fetch* unit actually access to the system memory while, during the next fetching operations related to warp lanes of the same warp, the *fetch* unit simply maintains the outputs constant, without accessing to the memory.

Concerning the total warp execution, logging the *fetch* unit the monitoring system is able to provide precise information about the number of cycles required to completely execute an instruction. For the *Vector_sum* application the results are plotted in fig. 5.2 and table 5.2.

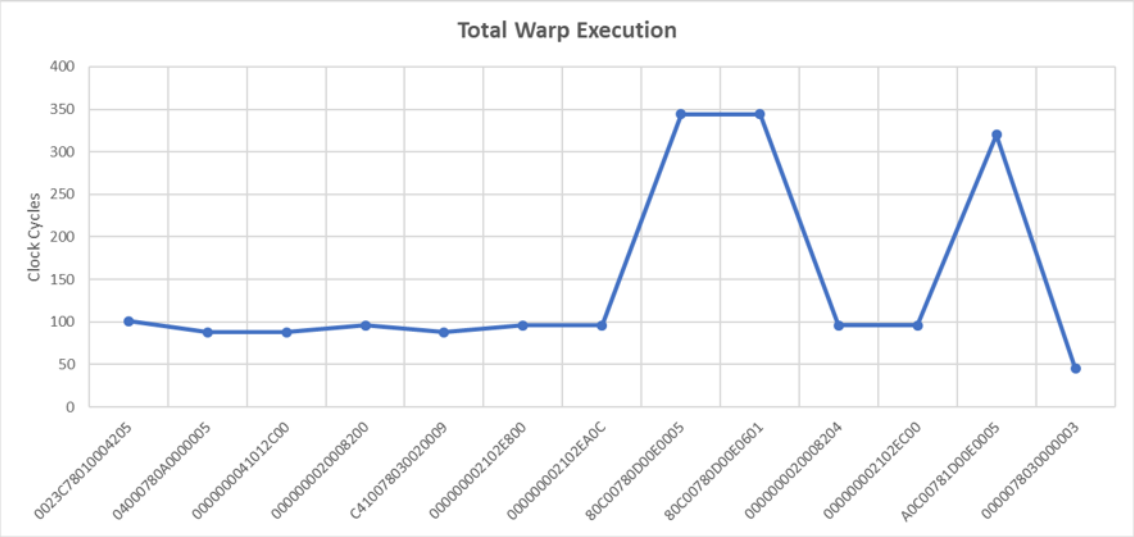


Figure 5.2: Vector_sum: total instructions execution.

From these results it's possible to notice that, for this simple application, all the instructions are executed in approximately one hundred of clock cycles, except for:

- 80C00780D00E0005 and 80C00780D00E0601. These are both load operation from the global memory, so reading from the global memory and writing of the same data into one of the local registers.
- A0C00781D00E0005. This is a store operation in the global memory, so reading from a one of the local registers and writing of the same date in the global memory.
- 0000078030000003. It's the last instruction executed and represent the return command.

In conclusion the real bottle neck for this application, and in general for most of applications, is the access to the global or external memories.

Instruction	Total warp execution
0023C78010004205	101
04000780A0000005	88
0000000041012C00	88
0000000020008200	96
C410078030020009	88
000000002102E800	96
000000002102EA0C	96
80C00780D00E0005	344
80C00780D00E0601	344
0000000020008204	96
000000002102EC00	96
A0C00781D00E0005	320
0000078030000003	45

Table 5.2: Vector_sum: total instructions execution.

Decode Unit

The second module in the pipeline is the *decode* unit. The monitoring system allows to evaluate the *decode* unit latency for each single warp lanes. The *decode* unit has a simple combinatorial architecture able to provide at the output, in few clock cycles, the *decode* input, but this considering the *decode* unit as an isolated block. Vice versa inside the pipeline, the *decode* unit is affected, as discussed for the *fetch* unit, by the delay introduce by the last three pipeline modules: *read* unit, *execution* unit and *write* unit. The *decode* unit monitoring results are plotted in fig 5.3 and collected in table 5.3.
More in details, the first warp lane will be served fast in the *decode* unit since there are no warp lanes in the next pipeline modules that can introduce a delay. The proof is given by the fact that the latency associated to the first warp lane is constant and equals to the minimum latency of the *decode* unit. Vice versa, when the second warp lane arrives to the *decode* unit, the first warp lane is in the *read* unit and block the *decode* unit until the *read* one is not ready to accept new inputs. Remembering that the *read* unit perform some checks, the results plotted in fig. 5.3 are coherent with the expectations: the latency associated to the second warp lane is the highest (since in the same time the first warp lane is in the *read* unit). This is true for all the instructions except for the storing in the main memory, in which the warp lane three is the one that has the highest associated latency. This is due to the fact that executing the store in global memory instruction, since the bottle neck is the access to the global memory and this occurs in the last unit of the pipeline, when the first warp lane arrive to the *write* unit the fourth one arrive to the *decode* and it's delayed until the store operation finishes.

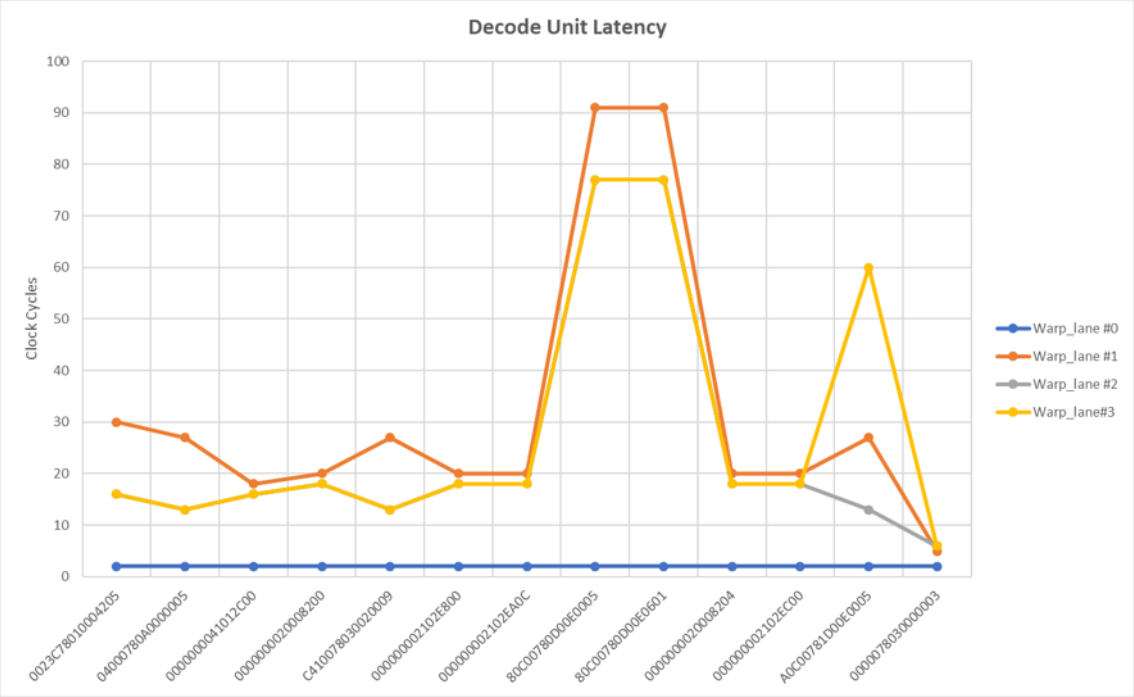


Figure 5.3: Vector_sum: *decode* unit monitoring results.

Instruction	W.Lane #0	W.Lane #1	W.Lane #2	W.Lane #3
0023C78010004205	2	30	16	16
04000780A0000005	2	27	13	13
0000000041012C00	2	18	16	16
0000000020008200	2	20	18	18
C410078030020009	2	27	13	13
000000002102E800	2	20	18	18
000000002102EA0C	2	20	18	18
80C00780D00E0005	2	91	77	77
80C00780D00E0601	2	91	77	77
0000000020008204	2	20	18	18
000000002102EC00	2	20	18	18
A0C00781D00E0005	2	27	13	60
0000078030000003	2	5	6	6

Table 5.3: Vector_sum: decode unit monitoring results.

Read Unit

The *read* unit monitoring results are plotted in fig 5.4 and collected in table 5.4. As expected, coherently to what discussed in the *fetch* and *decode* monitoring results description, the latency associated to the first warp lane results to be the highest one, except for the global

storage, in which the bottle neck of the pipeline is the *write* unit. Furthermore, except for the global storage, the latency associated to the other warp lanes in equal independently on the warp lane. This implies that for all the task execution the bottle neck of the pipeline is the *read* unit and only at the end, during the global storage, the bottle neck results to be the *write* unit.

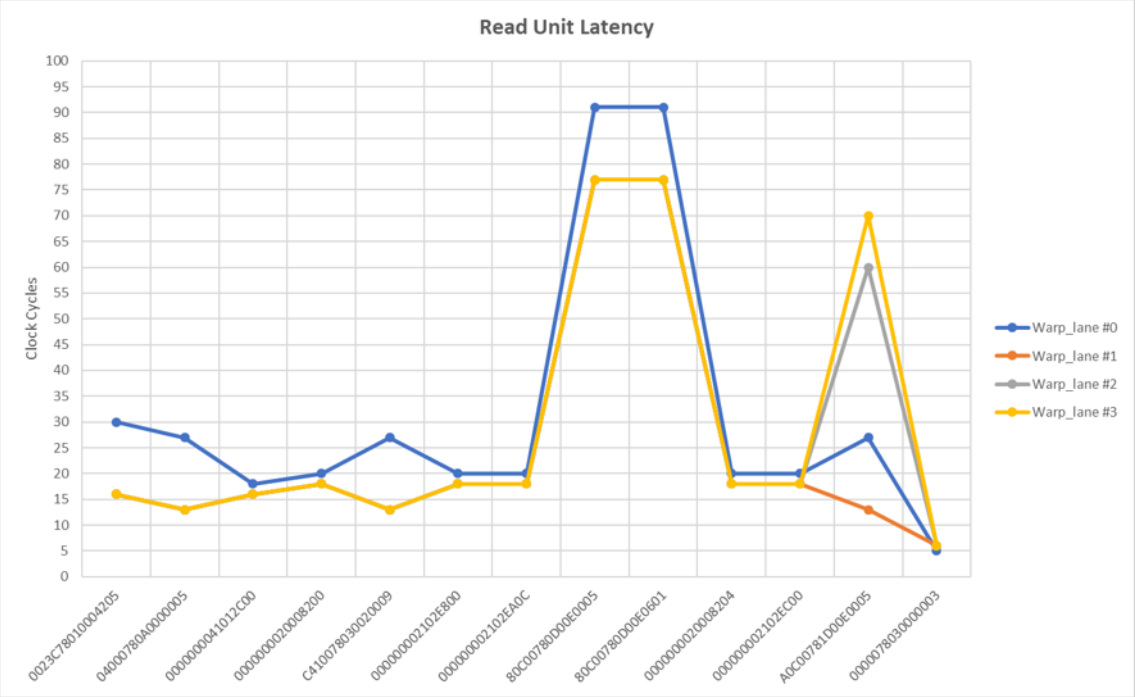


Figure 5.4: Vector_sum: *read* unit monitoring results.

Instruction	**W.Lane #0**	**W.Lane #1**	**W.Lane #2**	**W.Lane #3**
0023C78010004205	30	16	16	16
04000780A0000005	27	13	13	13
0000000041012C00	18	16	16	16
0000000020008200	20	18	18	18
C410078030020009	27	13	13	13
000000002102E800	20	18	18	18
000000002102EA0C	20	18	18	18
80C00780D00E0005	91	77	77	77
80C00780D00E0601	91	77	77	77
0000000020008204	20	18	18	18
000000002102EC00	20	18	18	18
A0C00781D00E0005	27	13	60	70
0000078030000003	5	6	6	6

Table 5.4: Vector_sum: read unit monitoring results.

Execution Unit

The *execution* unit monitoring results are plotted in fig 5.5 and collected in table 5.5. Since the bottle neck is the global memory, the latency associated to the *execution* unit is limited, except for the global store instruction, in which the *write* unit represent the bottle neck and delayed the second and next warp lanes execution (not the first warp lane one).

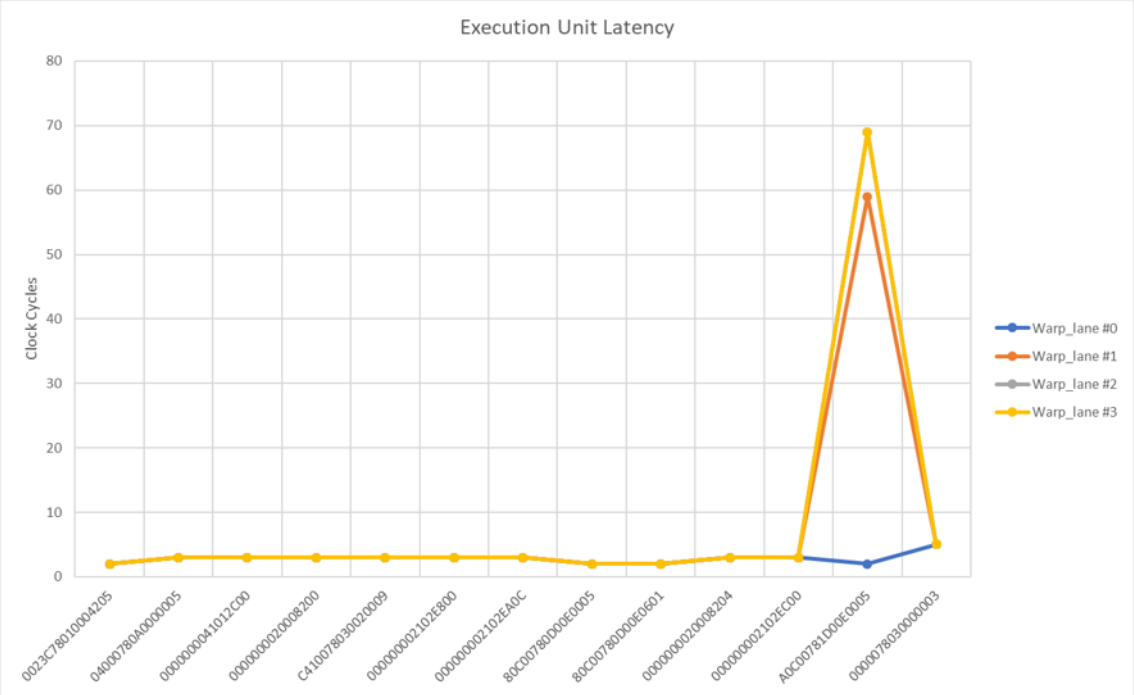


Figure 5.5: Vector_sum: *execution* unit monitoring results.

Instruction	W.Lane #0	W.Lane #1	W.Lane #2	W.Lane #3
0023C78010004205	2	2	2	2
04000780A0000005	3	3	3	3
0000000041012C00	3	3	3	3
0000000020008200	3	3	3	3
C410078030020009	3	3	3	3
000000002102E800	3	3	3	3
000000002102EA0C	3	3	3	3
80C00780D00E0005	2	2	2	2
80C00780D00E0601	2	2	2	2
0000000020008204	3	3	3	3
000000002102EC00	3	3	3	3
A0C00781D00E0005	2	59	69	69
0000078030000003	5	5	5	5

Table 5.5: Vector_sum: execution unit monitoring results.

Write Unit

The *write* unit monitoring results are plotted in fig 5.6 and collected in table 5.6. Since the *write* unit is the last in the pipeline, it's not affected by any delay and so the latency is equal among warp lanes and will be higher during the global store instruction execution.

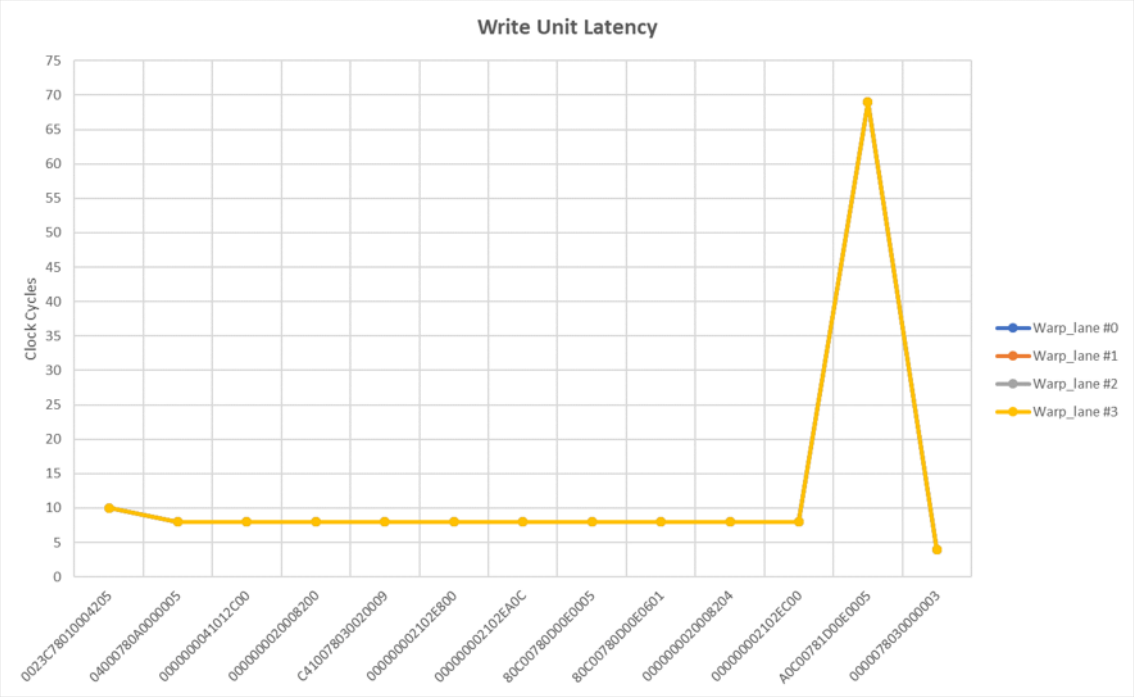


Figure 5.6: Vector_sum: *write* unit monitoring results.

Instruction	W.Lane #0	W.Lane #1	W.Lane #2	W.Lane #3
0023C78010004205	10	10	10	10
04000780A0000005	8	8	8	8
0000000041012C00	8	8	8	8
0000000020008200	8	8	8	8
C410078030020009	8	8	8	8
000000002102E800	8	8	8	8
000000002102EA0C	8	8	8	8
80C00780D00E0005	8	8	8	8
80C00780D00E0601	8	8	8	8
0000000020008204	8	8	8	8
000000002102EC00	8	8	8	8
A0C00781D00E0005	69	69	69	69
0000078030000003	4	4	4	4

Table 5.6: Vector_sum: write unit monitoring results.

Warp Scheduler

The *warp scheduler* monitoring results are plotted in fig 5.7 and collected in table 5.7. The highest latency is almost always associated to the first warp lane, e.g. due to the fact that the *warp scheduler* needs to access to the warp pool memory. The latency relate to the fourth warp lane is significant due to the delay introduce by the *read* unit, as previously discussed.

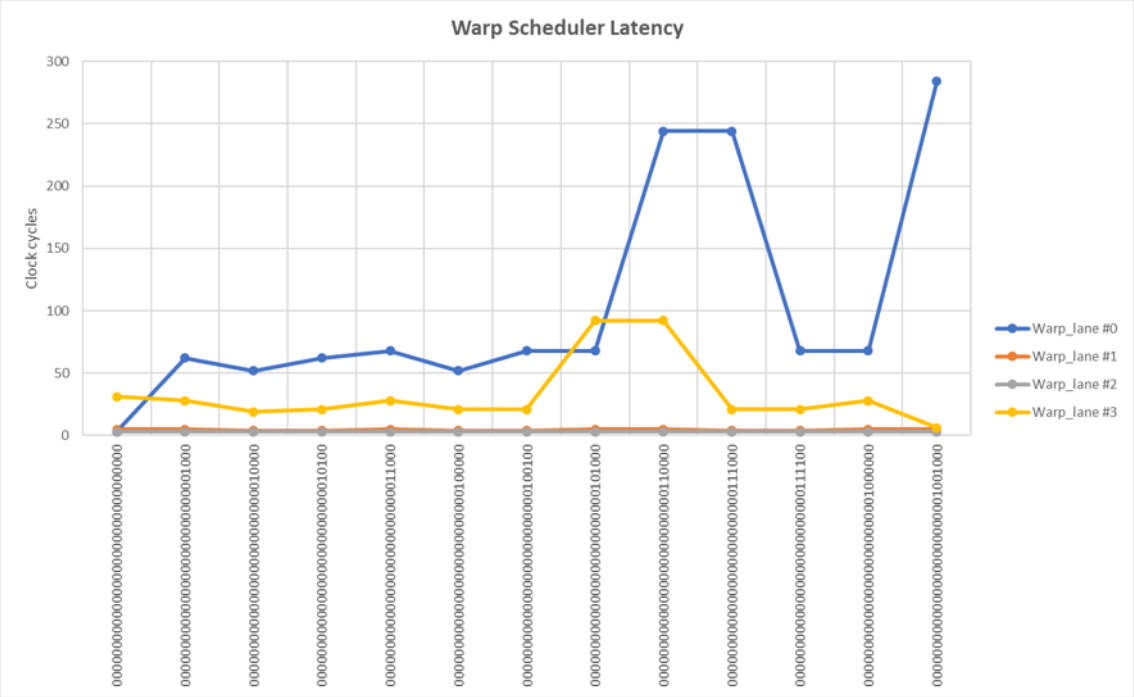


Figure 5.7: Vector_sum: *warp scheduler* monitoring results.

Instruction	W.L. #0	W.L. #1	W.L. #2	W.L. #3
00000000000000000000000000000000	3	5	3	31
00000000000000000000000000001000	62	5	3	28
00000000000000000000000000010000	52	4	3	19
00000000000000000000000000010100	62	4	3	21
00000000000000000000000000011000	68	5	3	28
00000000000000000000000000100000	52	4	3	21
00000000000000000000000000100100	68	4	3	21
00000000000000000000000000101000	68	5	3	92
00000000000000000000000000110000	244	5	3	92
00000000000000000000000000111000	244	4	3	21
00000000000000000000000000111100	68	4	3	21
00000000000000000000000001000000	68	5	3	28
00000000000000000000000001001000	284	5	3	6

Table 5.7: Vector_sum: warp scheduler monitoring results.

Warp Checker

The *warp checker* monitoring results are collected in table 5.8. The results show that the *warp checker* latency is independent on the warp lane number and the instruction executed and, more important, the latency is the minimum possible value. This is due to the fact that the *Vector_sum* application is simple and without possible conditional branching. As a consequence, the internal divergency, discussed in section 3.2.5 never occurs and the *warp checker* is practically needless, it will never check for fence.

Instruction	**W.Lane #0**	**W.Lane #1**	**W.Lane #2**	**W.Lane #3**
0023C78010004205	2	2	2	2
04000780A0000005	2	2	2	2
0000000041012C00	2	2	2	2
0000000020008200	2	2	2	2
C410078030020009	2	2	2	2
000000002102E800	2	2	2	2
000000002102EA0C	2	2	2	2
80C00780D00E0005	2	2	2	2
80C00780D00E0601	2	2	2	2
0000000020008204	2	2	2	2
000000002102EC00	2	2	2	2
A0C00781D00E0005	2	2	2	2
0000078030000003	2	2	2	2

Table 5.8: Vector_sum: warp checker monitoring results.

Warp Generator

The *warp generator* unit monitoring results are collected in table 5.9. This shown the time interval require to generate the warp and so to fill the *warp pool and state memory* and initialize the fence registers.

Time_interval	**#Cycles**
Total_warp_generation	7

Table 5.9: Vector_sum: warp generator monitoring results.

Streaming Multiprocessor Controller

The *streaming multiprocessor* unit monitoring results are collected in table 5.10 and represents the time interval the SM spends to execute the scheduled block.

Time_interval	#Cycles
SM_task_execution	2057

Table 5.10: Vector_sum: streaming multiprocessor monitoring results.

Block Scheduler

The *block scheduler* unit monitoring results are collected in table 5.11 and represents the time interval require to schedule a block.

Time_interval	#Cycles
Block_Scheduling	4

Table 5.11: Vector_sum: block scheduler monitoring results.

Vector Register File

The *vector register file* monitoring results regarding the number of accesses are shown in fig 5.8 and collected in the tables 5.12, 5.13 and 5.14.

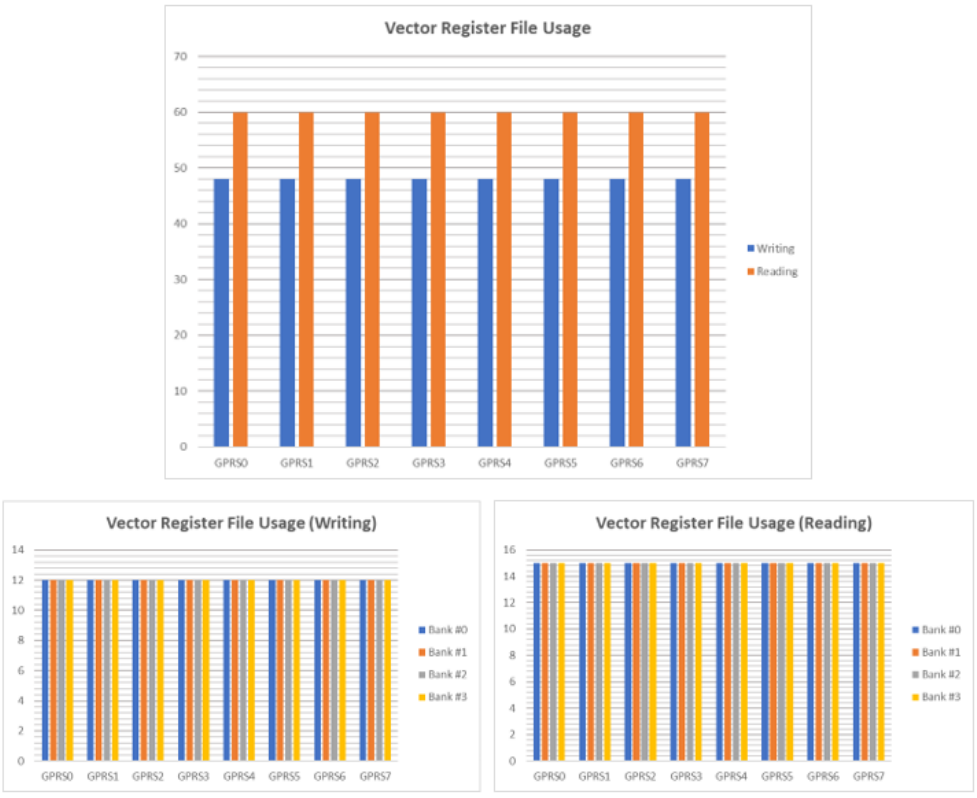


Figure 5.8: Vector_sum: *vector register file* monitoring results.

Module	# writings	# readings
GPRS0	48	60
GPRS1	48	60
GPRS2	48	60
GPRS3	48	60
GPRS4	48	60
GPRS5	48	60
GPRS6	48	60
GPRS7	48	60

Table 5.12: Vector_sum: vector register file monitoring results.

Module	Bank #0	Bank #1	Bank #2	Bank #3
GPRS0	12	12	12	12
GPRS1	12	12	12	12
GPRS2	12	12	12	12
GPRS3	12	12	12	12
GPRS4	12	12	12	12
GPRS5	12	12	12	12
GPRS6	12	12	12	12
GPRS7	12	12	12	12

Table 5.13: Vector_sum: vector register file banks monitoring results (writing).

Module	Bank #0	Bank #1	Bank #2	Bank #3
GPRS0	15	15	15	15
GPRS1	15	15	15	15
GPRS2	15	15	15	15
GPRS3	15	15	15	15
GPRS4	15	15	15	15
GPRS5	15	15	15	15
GPRS6	15	15	15	15
GPRS7	15	15	15	15

Table 5.14: Vector_sum: vector register file bank reading monitoring results (reading).

Observing the results, it's evident that the number of reading operations is higher than the writing one. Additionally, the memory access result to be balanced among different register files, banks and consequently among the 32 threads.
Regarding the time interval between first writing and last reading and between two successive writing operations, in table 5.15 and 5.16 the results have been collected.

GPRS0			
Bank	1st writing (ns)	last reading (ns)	interval
0	2085	3585	1500
1	2325	3755	1430
2	2565	3925	1360
3	2805	4095	1290
0	3595	5205	1610
1	3765	5375	1610
2	3935	5545	1610
3	4105	5715	1610
0	5345	6135	790
1	5515	6325	810
2	5685	6515	830
3	5855	6705	850
0	6245	7175	930
1	6435	7315	880
2	6625	7455	830
3	6815	7595	780
0	8085	10015	1930
1	8275	10795	2520
2	8465	11575	3110
3	8655	12355	3700
0	14155	16775	2620
1	14935	16965	2030
2	15715	17155	1440
3	16495	17345	850
0	17845	18895	1050
1	18035	19605	1570
2	18225	20315	2090
3	18415	21025	2610

Table 5.15: Vector_sum: *vector register file* write-to-read monitoring results for location #0 of the register file associated to core #0.

GPRS0			
Bank	1st writing (ns)	2nd writing (ns)	interval
0	2085	3595	1510
1	2325	3765	1440
2	2565	3935	1370
3	2805	4105	1300
0	3595	5345	1750
1	3765	5515	1750
2	3935	5685	1750
3	4105	5855	1750
0	5345	6245	900
1	5515	6435	920
2	5685	6625	940
3	5855	6815	960
0	6245	8085	1840
1	6435	8275	1840
2	6625	8465	1840
3	6815	8655	1840
0	8085	14155	6070
1	8275	14935	6660
2	8465	15715	7250
3	8655	16495	7840
0	14155	17845	3690
1	14935	18035	3100
2	15715	18225	2510
3	16495	18415	1920
0	17845	22145	4300
1	18035	22145	4110
2	18225	22145	3920
3	18415	22145	3730

Table 5.16: Vector_sum: *vector register file* write-to-write monitoring results for location #0 of the register file associated to core #0.

In the time interval tables only the results related to the location #0 of the register file #0

have been reported, since the behaviour is similar also for the other register file locations. More in detail, in the table is represented, using colours, which are the longer time intervals. This is a significant figure of merits considering that, as discussed in section 4.1, this is proportional to the sensibility of the location to the SEUs or better to the probability that the specific location is affected by SEUs. In fact, the larger is the time interval between first writing and last reading, the higher is the probability that a SEUs occurs.
Finally, considering the actual number of *vector register file* locations used, each thread will access to 4 locations over 128 and the total *vector register file* utilization is the one shown in fig 5.9.

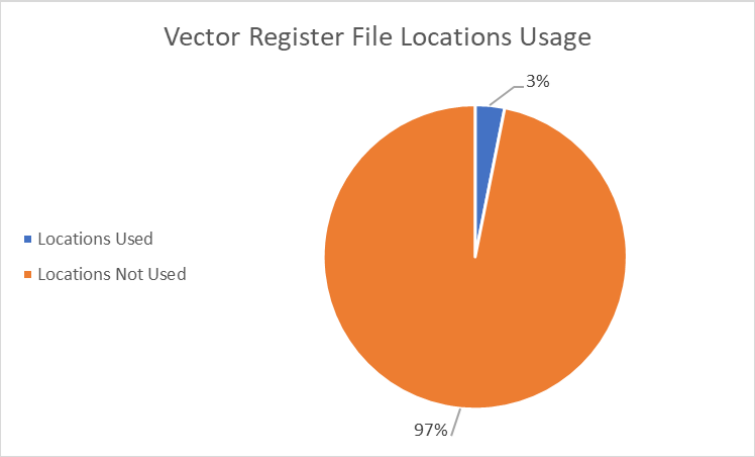


Figure 5.9: Vector_sum: *vector register file* location usage.

Address Register File

The *vector register file* monitoring results regarding the number of accesses are collected in the tables 5.17. As it's possible to notice, the *address register file* is never used. consequently, considering that this memory stores the addresses to point to the shared memory (that allows the communication among different SP of the same SM), this implies that the cores work completely in parallel.

Module	**# writings**	**# readings**
ADDREG0	0	0
ADDREG1	0	0
ADDREG2	0	0
ADDREG3	0	0
ADDREG4	0	0
ADDREG5	0	0
ADDREG6	0	0
ADDREG7	0	0

Table 5.17: Vector_sum: vector register file monitoring results.

Predicate Register File

The *predicate register file* monitoring results regarding the number of accesses are shown in fig 5.10 and collected in the tables 5.18, 5.19. Observing the results, it's possible to notice that the *predicate register file* is never written, due to the absence of conditional branching in the application. Additionally, the memory access result to be balanced among different register files, banks and consequently among the 32 threads.

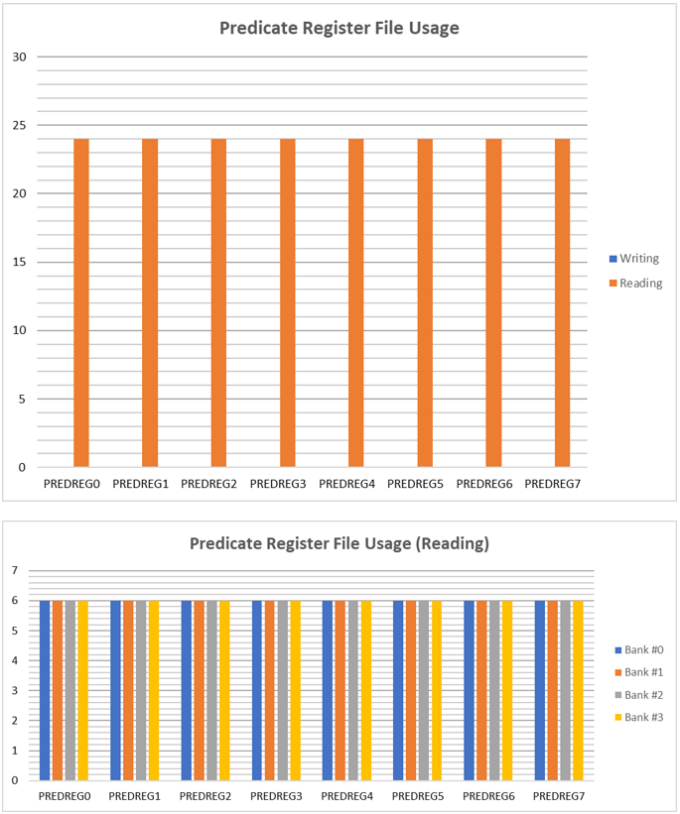


Figure 5.10: Vector_sum: *predicate register file* monitoring results.

Module	# writings	# readings
PREDREG0	0	24
PREDREG1	0	24
PREDREG2	0	24
PREDREG3	0	24
PREDREG4	0	24
PREDREG5	0	24
PREDREG6	0	24
PREDREG7	0	24

Table 5.18: Vector_sum: predicate register file monitoring results.

Module	Bank #0	Bank #1	Bank #2	Bank #3
PREDREG0	6	6	6	6
PREDREG1	6	6	6	6
PREDREG2	6	6	6	6
PREDREG3	6	6	6	6
PREDREG4	6	6	6	6
PREDREG5	6	6	6	6
PREDREG6	6	6	6	6
PREDREG7	6	6	6	6

Table 5.19: Vector_sum: predicate register file bank reading monitoring results (reading).

5.1.2 Vector_sum with Resilient LOAD/STORE

Analysed the performance analysis results of the *Vector_sum* application, a hardened version of the same application has been studied. This permits to quantify the performance degradation and the overhead in terms of memory access due to the resilient LOAD/STORE instruction.

Fetch Unit

The *fetch* unit monitoring results are shown in fig 5.11 and collected in the table 5.20.

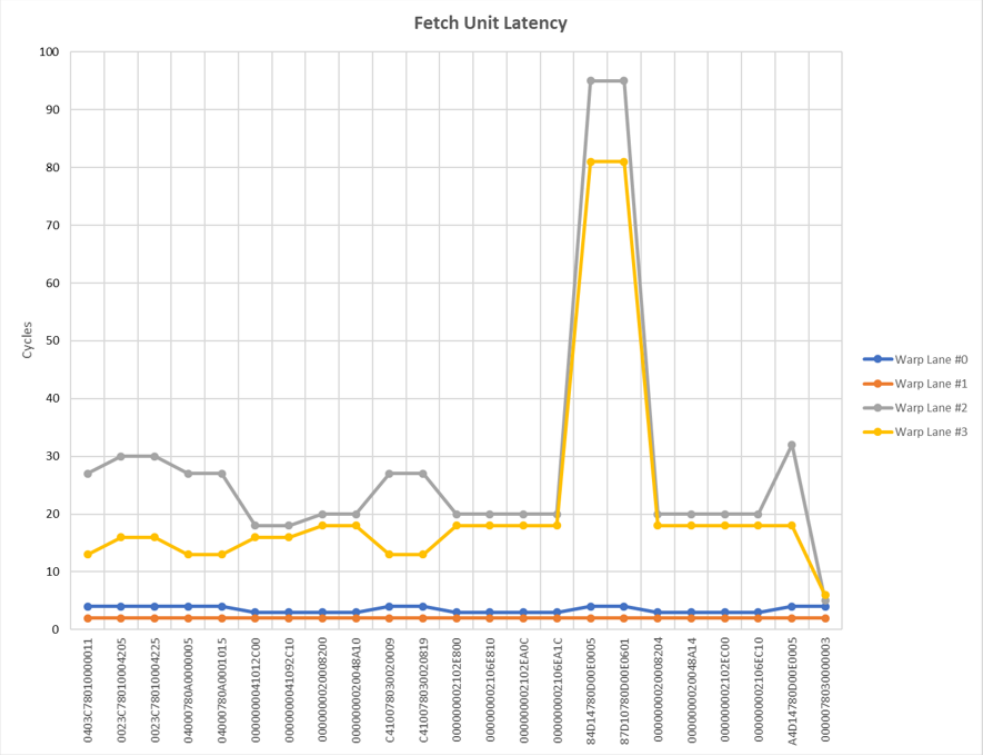


Figure 5.11: Vector_sum with resilient global access: *fetch* unit monitoring results.

Instruction	W.Lane #0	W.Lane #1	W.Lane #2	W.Lane #3
0403C78010000011	4	2	27	13
0023C78010004205	4	2	30	16
0023C78010004225	4	2	30	16
04000780A0000005	4	2	27	13
04000780A0001015	4	2	27	13
0000000041012C00	3	2	18	16
0000000041092C10	3	2	18	16
0000000020008200	3	2	20	18
0000000020048A10	3	2	20	18
C410078030020009	4	2	27	13
C410078030020819	4	2	27	13
000000002102E800	3	2	20	18
000000002106E810	3	2	20	18
000000002102EA0C	3	2	20	18
000000002106EA1C	3	2	20	18
84D14780D00E0005	4	2	95	81
87D10780D00E0601	4	2	95	81
0000000020008204	3	2	20	18
0000000020048A14	3	2	20	18
000000002102EC00	3	2	20	18
000000002106EC10	3	2	20	18
A4D14780D00E0005	4	2	32	18
0000078030000003	4	2	5	6

Table 5.20: Vector_sum with resilient global access: fetch unit monitoring results.

Comparing these results with the one obtained and discussed for the standard *Vector_sum* application, it's possible to notice that the latency associated to the fourth warp lane has the same trend in both the application. Nevertheless, comparing the data shown in tables 5.1 and 5.20, it's evident that the latency associated to the global store and load instructions, related to the third and fourth warp lanes, is higher in the application with the resilient LOAD/STORE.

Considering the total time to complete the application, the expectation is that the resilient version will spend a higher amount of time, in fact the number of executed instruction is larger. The total warp execution results are shown in fig 5.12 and collected in the table 5.21. Again, comparing tables 5.2 and 5.21, it's possible to that not only the *fetch* unit latency but the entire warp execution of the global LOAD/STORE instruction increases in the resilient version. More in details, the performance overhead for the LOAD instruction results to be around 5,8 % while for the STORE instruction around 5,3 %.

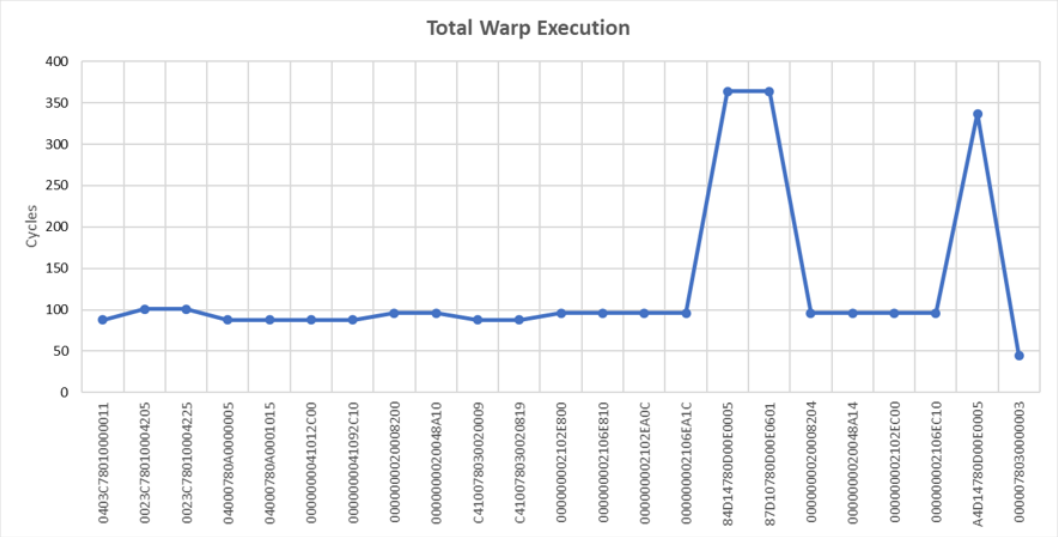


Figure 5.12: Vector_sum with resilient global access: total instructions execution.

Instruction	**Total warp execution**
0403C78010000011	88
0023C78010004205	101
0023C78010004225	101
04000780A0000005	88
04000780A0001015	88
0000000041012C00	88
0000000041092C10	88
0000000020008200	96
0000000020048A10	96
C410078030020009	88
C410078030020819	88
000000002102E800	96
000000002106E810	96
000000002102EA0C	96
000000002106EA1C	96
84D14780D00E0005	364
87D10780D00E0601	364
0000000020008204	96
0000000020048A14	96
000000002102EC00	96
000000002106EC10	96
A4D14780D00E0005	337
0000078030000003	45

Table 5.21: Vector_sum with resilient global access: total instructions execution.

Decode Unit

The *decode* unit monitoring results are shown in fig 5.13 and collected in the table 5.22. Again, comparing with the results shown in table 5.3, it's possible to notice a latency overhead associated to the resilient LOAD/STORE operation. Vice versa the latency trend is similar to the standard *Vector_add* one.

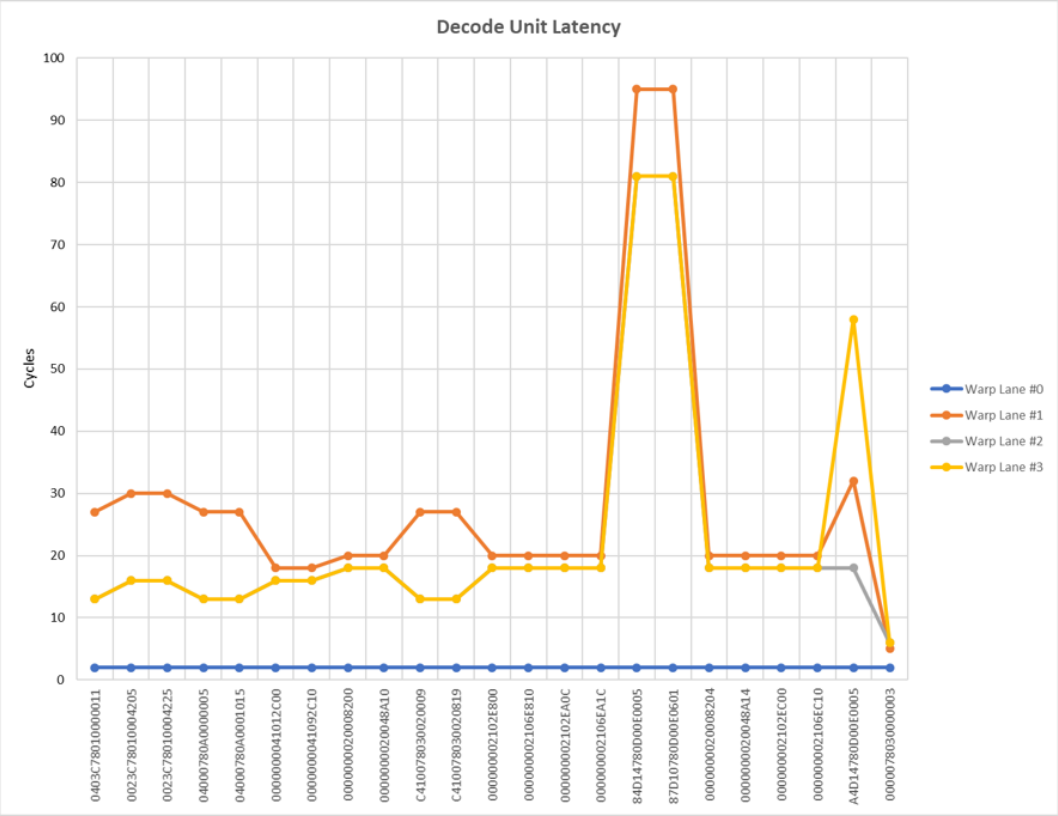


Figure 5.13: Vector_sum with resilient global access: *decode* unit monitoring results.

Instruction	W.Lane #0	W.Lane #1	W.Lane #2	W.Lane #3
0403C78010000011	2	27	13	13
0023C78010004205	2	30	16	16
0023C78010004225	2	30	16	16
04000780A0000005	2	27	13	13
04000780A0001015	2	27	13	13
0000000041012C00	2	18	16	16
0000000041092C10	2	18	16	16
0000000020008200	2	20	18	18
0000000020048A10	2	20	18	18
C410078030020009	2	27	13	13

C410078030020819	2	27	13	13
000000002102E800	2	20	18	18
000000002106E810	2	20	18	18
000000002102EA0C	2	20	18	18
000000002106EA1C	2	20	18	18
84D14780D00E0005	2	95	81	81
87D10780D00E0601	2	95	81	81
0000000020008204	2	20	18	18
0000000020048A14	2	20	18	18
000000002102EC00	2	20	18	18
000000002106EC10	2	20	18	18
A4D14780D00E0005	2	32	18	58
0000078030000003	2	5	6	6

Table 5.22: Vector_sum with resilient global access: decode unit monitoring results.

Read Unit

The *read* unit monitoring results are shown in fig 5.14 and collected in the table 5.23.

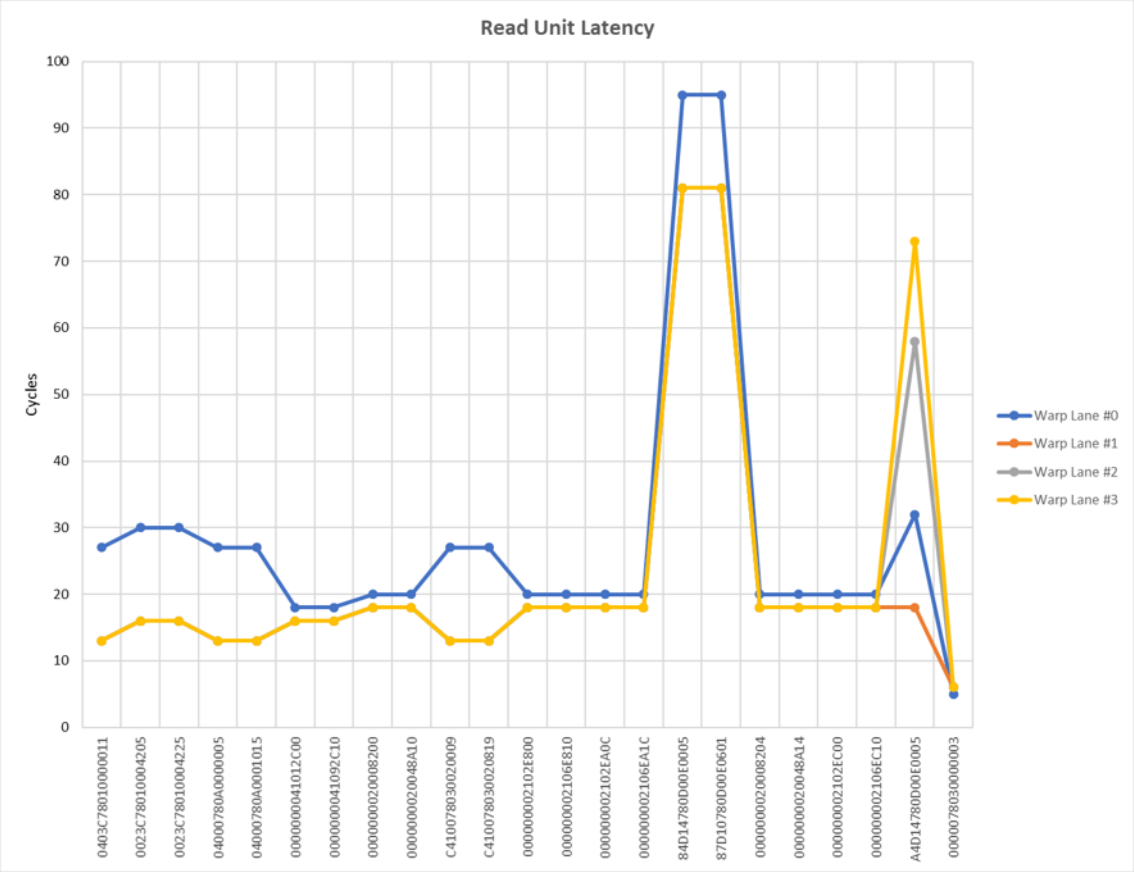


Figure 5.14: Vector_sum with resilient global access: *read* unit monitoring results.

Instruction	W.Lane #0	W.Lane #1	W.Lane #2	W.Lane #3
0403C78010000011	27	13	13	13
0023C78010004205	30	16	16	16
0023C78010004225	30	16	16	16
04000780A0000005	27	13	13	13
04000780A0001015	27	13	13	13
0000000041012C00	18	16	16	16
0000000041092C10	18	16	16	16
0000000020008200	20	18	18	18
0000000020048A10	20	18	18	18
C410078030020009	27	13	13	13
C410078030020819	27	13	13	13
000000002102E800	20	18	18	18
000000002106E810	20	18	18	18
000000002102EA0C	20	18	18	18
000000002106EA1C	20	18	18	18
84D14780D00E0005	95	81	81	81
87D10780D00E0601	95	81	81	81
0000000020008204	20	18	18	18
0000000020048A14	20	18	18	18
000000002102EC00	20	18	18	18
000000002106EC10	20	18	18	18
A4D14780D00E0005	32	18	58	73
0000078030000003	5	6	6	6

Table 5.23: Vector_sum with resilient global access: read unit monitoring results.

Again, comparing the results shown in table 5.4 and 5.23, it's possible to observe a latency overhead associated to the resilient LOAD/STORE operation. As regard the latency trend, it results to be similar to the standard version of the *Vector_add*.

Execution Unit

The *execution* unit monitoring results are shown in fig 5.15 and collected in the table 5.24. Comparing the results with the ones obtained for the standard *Vector_sum*, collected in table 5.5, it's possible to notice a latency overhead for the STORE instruction but not for the LOAD one. This it's an obvious result considering that, since the LOAD instruction execution produces a delay in the *read* unit of the pipe, since the *execution* unit is after the *read* one in the pipeline, it's not affected by the *read* unit delay.

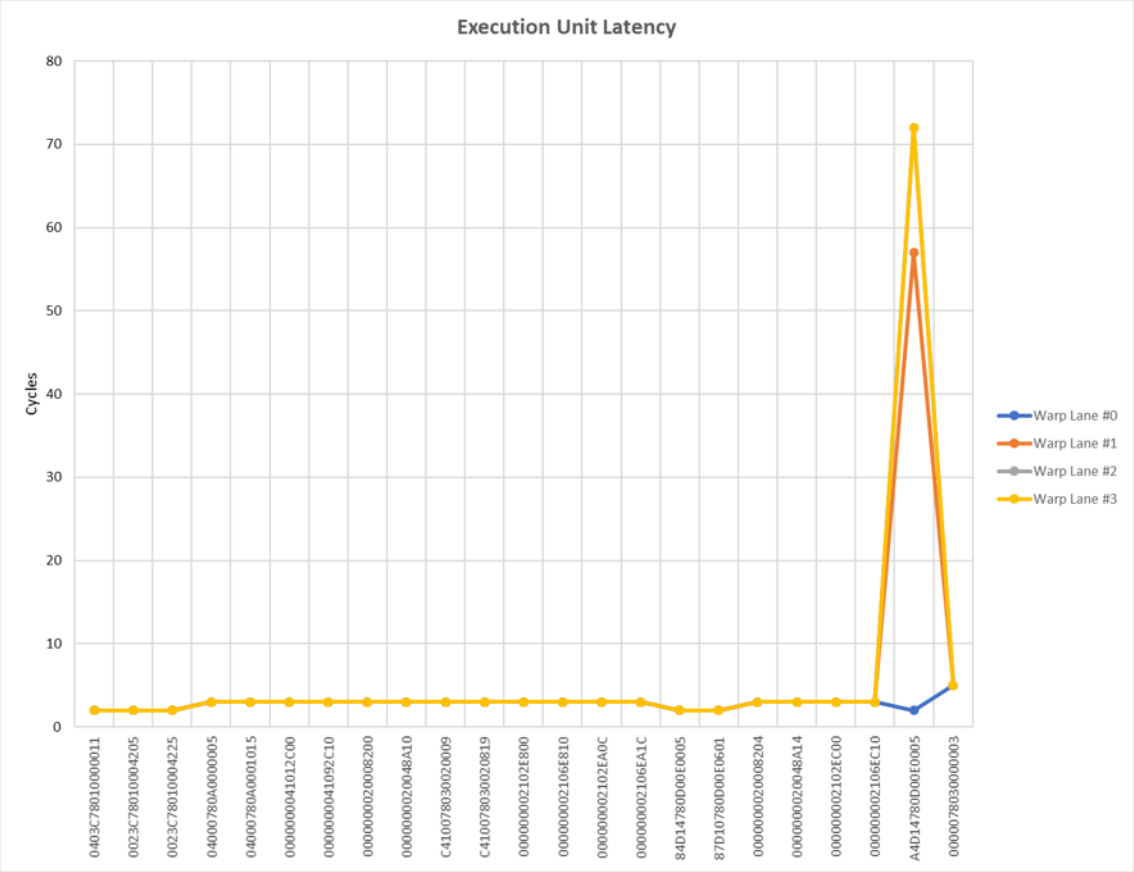


Figure 5.15: Vector_sum with resilient global access: *execution* unit monitoring results.

Instruction	W.Lane #0	W.Lane #1	W.Lane #2	W.Lane #3
0403C78010000011	2	2	2	2
0023C78010004205	2	2	2	2
0023C78010004225	2	2	2	2
04000780A0000005	3	3	3	3
04000780A0001015	3	3	3	3
0000000041012C00	3	3	3	3
0000000041092C10	3	3	3	3
0000000020008200	3	3	3	3
0000000020048A10	3	3	3	3
C410078030020009	3	3	3	3
C410078030020819	3	3	3	3
000000002102E800	3	3	3	3
000000002106E810	3	3	3	3
000000002102EA0C	3	3	3	3
000000002106EA1C	3	3	3	3
84D14780D00E0005	2	2	2	2
87D10780D00E0601	2	2	2	2
0000000020008204	3	3	3	3

0000000020048A14	3	3	3	3
000000002102EC00	3	3	3	3
000000002106EC10	3	3	3	3
A4D14780D00E0005	2	57	72	72
0000078030000003	5	5	5	5

Table 5.24: Vector_sum with resilient global access: execution unit monitoring results.

Write Unit

The *write* unit monitoring results are shown in fig 5.16 and collected in the table 5.25. Again, the results are similar to the one obtained for the standard *Vector_sum*, except for the LOAD-/STORE instructions.

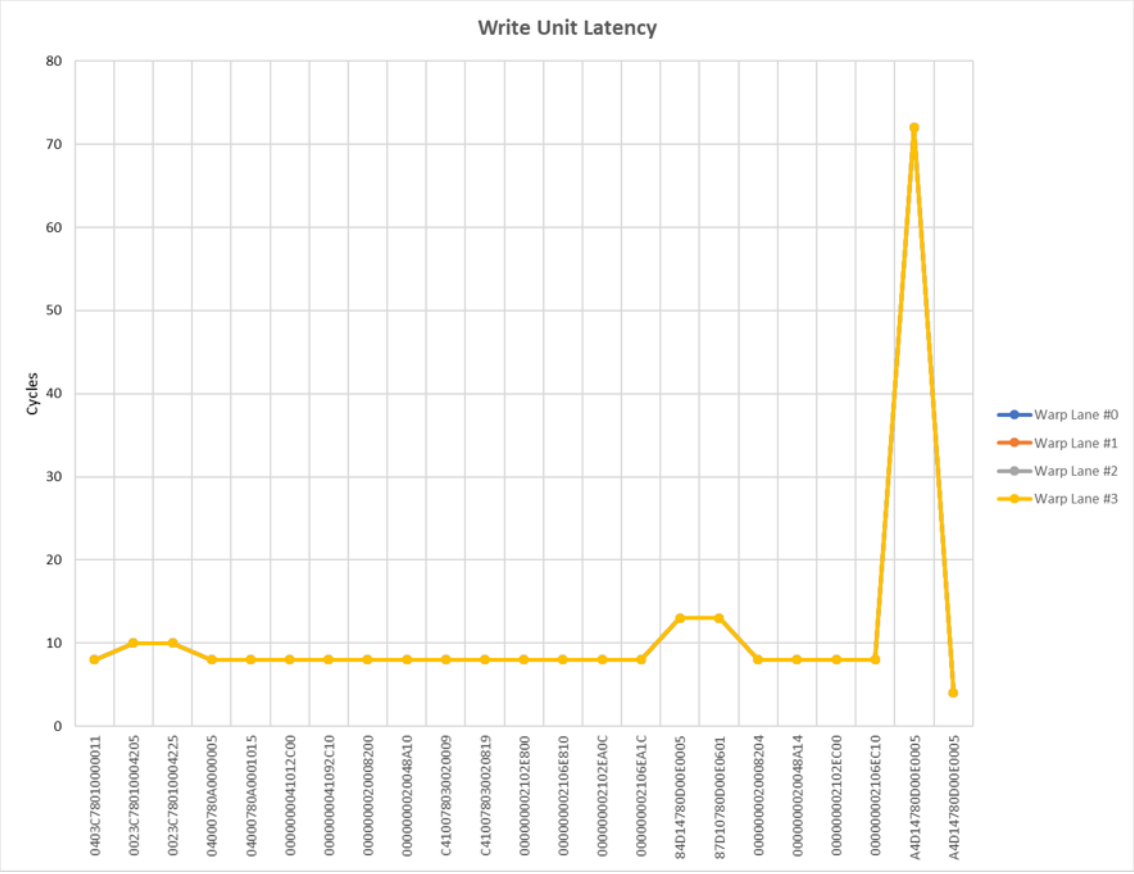


Figure 5.16: Vector_sum with resilient global access: *write* unit monitoring results.

Instruction	W.Lane #0	W.Lane #1	W.Lane #2	W.Lane #3
0403C78010000011	8	8	8	8
0023C78010004205	10	10	10	10
0023C78010004225	10	10	10	10
04000780A0000005	8	8	8	8
04000780A0001015	8	8	8	8
0000000041012C00	8	8	8	8
0000000041092C10	8	8	8	8
0000000020008200	8	8	8	8
0000000020048A10	8	8	8	8
C410078030020009	8	8	8	8
C410078030020819	8	8	8	8
000000002102E800	8	8	8	8
000000002106E810	8	8	8	8
000000002102EA0C	8	8	8	8
000000002106EA1C	8	8	8	8
84D14780D00E0005	13	13	13	13
87D10780D00E0601	13	13	13	13
0000000020008204	8	8	8	8
0000000020048A14	8	8	8	8
000000002102EC00	8	8	8	8
000000002106EC10	8	8	8	8
A4D14780D00E0005	72	72	72	72
A4D14780D00E0005	4	4	4	4

Table 5.25: Vector_sum with resilient global access: write unit monitoring results.

Warp Scheduler

The *warp scheduler* monitoring results are shown in fig 5.17 and collected in the table 5.26. Comparing the results shown in table 5.26 and 5.26, it's possible to notice a latency overhead associated to the resilient LOAD/STORE operation. The latency trend associated to each warp lane is similar to the standard *Vector_add* one.

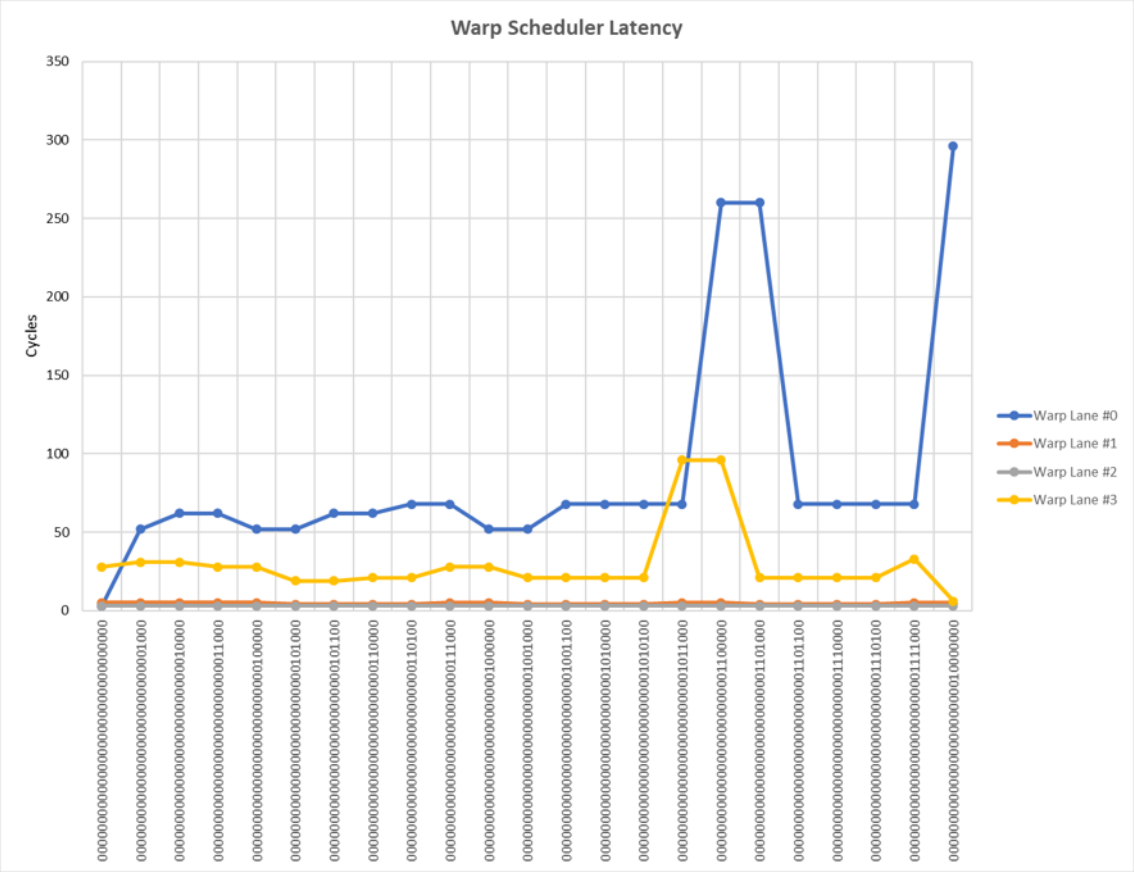


Figure 5.17: Vector_sum with resilient global access: *warp scheduler* monitoring results.

Program Counter	**W.L. #0**	**W.L. #1**	**W.L. #2**	**W.L. #3**
00000000000000000000000000000000	3	5	3	28
00000000000000000000000000001000	52	5	3	31
00000000000000000000000000010000	62	5	3	31
00000000000000000000000000011000	62	5	3	28
00000000000000000000000000100000	52	5	3	28
00000000000000000000000000101000	52	4	3	19
00000000000000000000000000101100	62	4	3	19
00000000000000000000000000110000	62	4	3	21
00000000000000000000000000110100	68	4	3	21
00000000000000000000000000111000	68	5	3	28
00000000000000000000000001000000	52	5	3	28
00000000000000000000000001001000	52	4	3	21
00000000000000000000000001001100	68	4	3	21
00000000000000000000000001010000	68	4	3	21
00000000000000000000000001010100	68	4	3	21
00000000000000000000000001011000	68	5	3	96
00000000000000000000000001100000	260	5	3	96
00000000000000000000000001101000	260	4	3	21

00000000000000000000000001101100	68	4	3	21
00000000000000000000000001110000	68	4	3	21
00000000000000000000000001110100	68	4	3	21
00000000000000000000000001111000	68	5	3	33
00000000000000000000000010000000	296	5	3	6

Table 5.26: Vector_sum with resilient global access: warp scheduler monitoring results.

Warp Checker

The *warp checker* monitoring results are collected in the table 5.27. For this module the behaviour results to be exactly equal to the standard *Vector_sum* application. This is due to the fact that, even if the version is different, the application is the same. Consequently, considering that the *warp checker* is used to handle the internal warp divergency, if in the original application the conditional flow is absent, the same occurs also for the resilient version. So, as in the standard version, also in the resilient one the reach the maximum efficiency.

Instruction	**W.Lane #0**	**W.Lane #1**	**W.Lane #2**	**W.Lane #3**
0403C78010000011	2	2	2	2
0023C78010004205	2	2	2	2
0023C78010004225	2	2	2	2
04000780A0000005	2	2	2	2
04000780A0001015	2	2	2	2
0000000041012C00	2	2	2	2
0000000041092C10	2	2	2	2
0000000020008200	2	2	2	2
0000000020048A10	2	2	2	2
C410078030020009	2	2	2	2
C410078030020819	2	2	2	2
000000002102E800	2	2	2	2
000000002106E810	2	2	2	2
000000002102EA0C	2	2	2	2
000000002106EA1C	2	2	2	2
84D14780D00E0005	2	2	2	2
87D10780D00E0601	2	2	2	2
0000000020008204	2	2	2	2
0000000020048A14	2	2	2	2
000000002102EC00	2	2	2	2
000000002106EC10	2	2	2	2
A4D14780D00E0005	2	2	2	2
A4D14780D00E0005	2	2	2	2

Table 5.27: Vector_sum with resilient global access: warp checker monitoring results.

Warp Generator

The *warp generator* monitoring results are collected in the table 5.27. Comparing with the table 5.9, in both application versions the *warp generator* spends the same time to completely generate the warp.

Time_interval	#Cycles
Total_warp_generation	7

Table 5.28: Vector_sum with resilient global access: warp generator monitoring results.

Streaming Multiprocessor Controller

The *streaming multiprocessor controller* monitoring results are collected in the table 5.29. Comparing with the table 5.10, it's possible to observe that the resilient version produces around 48 % of performance degradation.

Time_interval	#Cycles
SM_Task_execution	3047

Table 5.29: Vector_sum with resilient global access: streaming multiprocessor monitoring results.

Block Scheduler

The *block scheduler* monitoring results are collected in the table 5.30. Comparing with the table 5.11, in both application versions the *warp generator* spends the same time to completely schedule the thread block.

Time_interval	#Cycles
Block_scheduling	4

Table 5.30: Vector_sum with resilient global access: block scheduler monitoring results.

Vector Register File

The *vector register file* monitoring results regarding the number of accesses are shown in fig 5.18 and collected in the tables 5.31, 5.32 and 5.33. Comparing with the results obtained for the standard *Vector_sum* (fig 5.8 and tables 5.12, 5.13 and 5.14), it's possible to observe that the number of writing operation are duplicate but the number of reading operations is more than doubled. This is due to the implementation of the resilient global LOAD and STORE instructions. Since the LOAD operation is translated in a LOAD and a MOVE, the first operation corresponds to a writing operation on the VRF, vice versa the second operation

corresponds to a reading operation on the VRF and then a writing operation. This means that each global LOAD introduces an additional read operation, producing more than the double of reading operations.

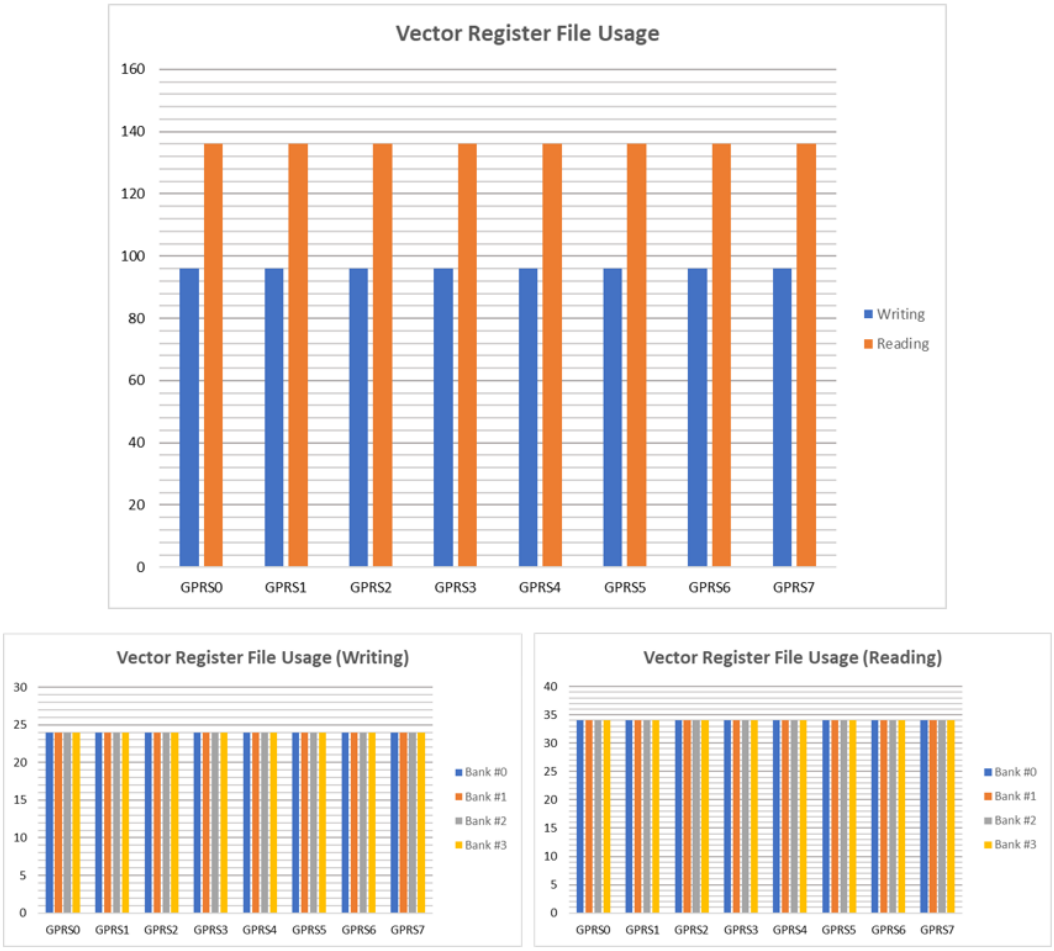


Figure 5.18: Vector_sum with resilient global access: *vector register file* monitoring results.

Module	**# writings**	**# readings**
GPRS0	96	136
GPRS1	96	136
GPRS2	96	136
GPRS3	96	136
GPRS4	96	136
GPRS5	96	136
GPRS6	96	136
GPRS7	96	136

Table 5.31: Vector_sum with resilient global access: vector register file monitoring results.

Module	Bank #0	Bank #1	Bank #2	Bank #3
GPRS0	24	24	24	24
GPRS1	24	24	24	24
GPRS2	24	24	24	24
GPRS3	24	24	24	24
GPRS4	24	24	24	24
GPRS5	24	24	24	24
GPRS6	24	24	24	24
GPRS7	24	24	24	24

Table 5.32: Vector_sum with resilient global access: vector register file banks monitoring results (writing).

Module	Bank #0	Bank #1	Bank #2	Bank #3
GPRS0	34	34	34	34
GPRS1	34	34	34	34
GPRS2	34	34	34	34
GPRS3	34	34	34	34
GPRS4	34	34	34	34
GPRS5	34	34	34	34
GPRS6	34	34	34	34
GPRS7	34	34	34	34

Table 5.33: Vector_sum with resilient global access: vector register file banks monitoring results (reading).

Considering now the vector register locations usage shown in fig 5.19, as expected considering that the resilient instruction are based on duplication, the total amount of used register is doubled with respect to the standard *Vector_sum*, fig 5.9.

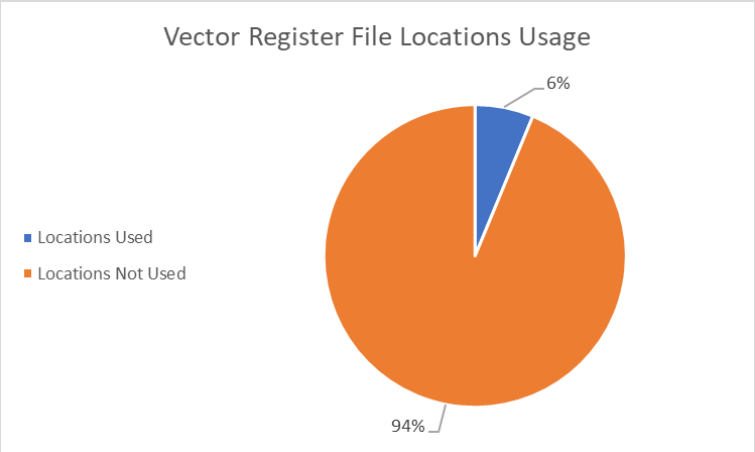


Figure 5.19: Vector_sum: *vector register file* location usage.

Address Register File

The *address register file* monitoring results regarding the number of accesses are collected in the table 5.34. As in the standard version, the *address register file* is not used.

Module	# writings	# readings
ADDREG0	0	0
ADDREG1	0	0
ADDREG2	0	0
ADDREG3	0	0
ADDREG4	0	0
ADDREG5	0	0
ADDREG6	0	0
ADDREG7	0	0

Table 5.34: Vector_sum with resilient global access: address register file monitoring results.

Predicate Register File

The *predicate register file* monitoring results regarding the number of accesses are shown in fig 5.20 and collected in the tables 5.35 and 5.36. With respect to the standard version (fig 5.10 and tables 5.18 and 5.19) the number of reading operations increases, but the writing operations still remain absent.

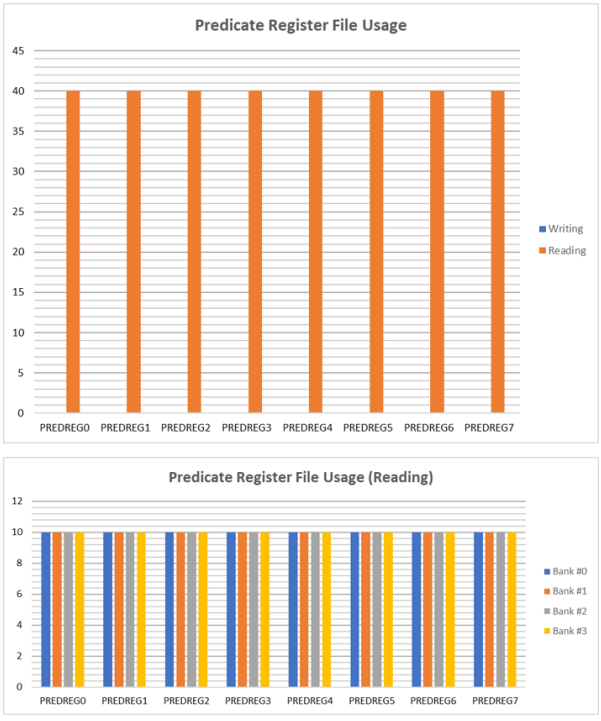


Figure 5.20: Vector_sum with resilient global access: *predicate register file* monitoring results.

Module	# writings	# readings
PREDREG0	0	40
PREDREG1	0	40
PREDREG2	0	40
PREDREG3	0	40
PREDREG4	0	40
PREDREG5	0	40
PREDREG6	0	40
PREDREG7	0	40

Table 5.35: Vector_sum with resilient global access: predicate register file monitoring results.

Module	Bank #0	Bank #1	Bank #2	Bank #3
PREDREG0	10	10	10	10
PREDREG1	10	10	10	10
PREDREG2	10	10	10	10
PREDREG3	10	10	10	10
PREDREG4	10	10	10	10
PREDREG5	10	10	10	10
PREDREG6	10	10	10	10
PREDREG7	10	10	10	10

Table 5.36: Vector_sum with resilient global access: predicate register file banks monitoring results (reading).

5.2 Fault Injection Platform

Considering that the performance analysis monitoring system has been implemented in order to provide detailed information regarding the internal behaviour of the FlexGrip, the goal is to analysed this information when a SEE effects occurs, in order to be able to identify and harden only the critical resources. For this purpose, it's essential to have a tool able to inject fault in the architecture under test.

5.2.1 Fault Injector

The *fault injector* considered in this study, focused on the effects of SEUs, it's able to generate a bit-flip in one of the memory locations defined in an input file called signal list (*SL.txt*) in a random time instant. Once the fault injector read the signal list input file, it launches a *golden simulation* of the application on the FlexGrip, so a simulation without faults, storing the output memory and all the performance analysis results. During the golden simulation, the fault injector save also the *kernel start* and *done* instant, defining the time interval in

which the FlexGrip is actually execute the application, in order to inject faults only during the application execution and not during the FlexGrip configuration.
When the golden simulation is completed the fault injector, according to the number of faults to inject (defined at the input), run a certain amount of fault simulation injecting a bit flip randomly in the time interval between *kernel start* and *done* and randomly in one of the memory locations defined in the signal list file. Again, for each simulation, the injector saves the performance analysis results and the output memory. Finally, the fault injector classified the fault effects in four categories:

- *Detected Unrecoverable Error* (DUE), also known as *hang*, if the simulation doesn't terminate correctly (e.g. the *kernel done* is not detected);
- *Masked* (or *silent*), if the output memories of the golden and fault simulations are equal;
- *Silent Data Corruption* (SDC), if the output memories of the golden and fault simulations are not equal (in this case the fault injector save the differences in a proper file).
- *Time-Out*, if the fault simulation requires more time to be completed with respect to the golden one, producing a performance degradation.

Moreover, the fault injector classified the SEU effects in *detected* and *not detected*. Finally, once the fault simulations are completed, the fault injector provide a summary specifying the total execution time and the number of fault effects.

5.2.2 Fault Injection Strategy

In order to complete analysed the fault effects, the fault injector should inject a number of fault proportional to the memory locations and clock cycles. In particular it should launch a simulation injecting a fault in each memory location at each clock cycle. Consequently, this implies to run millions of simulations. Vice versa reducing the number of simulations, for sure the simulation time decreases but also the accuracy fault injection analysis. Therefore, it's important to find a trade-off among number of targeted locations and number of injected faults. Assuming to have a fixed number of injected faults (e.g. 2000), and so of fault simulations, in the following sections will be analysed different fault injection strategies, considering as targeted module the *vector register file* and as tested application the standard version of the *Vector_sum*. From the performance analysis the number of register locations of the *vector register file* used by the standard *Vector_add*, and as a consequence the ones in which faults must be injected, are four locations in each bank of each register file in the *vector register file* (assuming, as in 5.1, 8 cores). This means that standard *Vector_sum* accesses to 128 locations of 32 bit each. Instead, considering the *predicate register file*, the number of locations used is 32 locations of 4 bit (1 location for each thread).

1° **Strategy: inject all used registers with same number in a single register file**

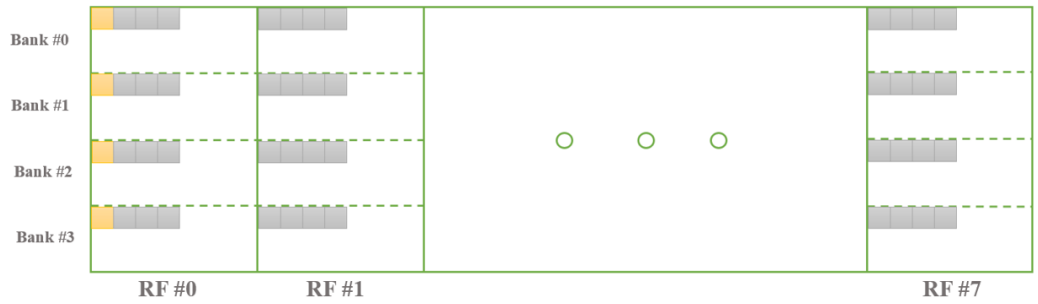


Figure 5.21: Fault injection 1° strategy.

Considering the number of signals in the signal list, this results to be equal to the product between the number of register under test in the single fault injection study (in this case four locations) and the number of bit of each register:

#signals in signal list = #registers under test x #bit of parallelism

In the standard *Vector_sum* application the total number of signals in the signal list is 128 so, considering 2000 injected faults, an average of 15 faults per each signal in the entire execution interval. Moreover, to inject in all the used registers, it's necessary to run 32 fault injections, since there are four used locations in each bank and eight register files.

2° **Strategy: inject all used registers with same number and bank**

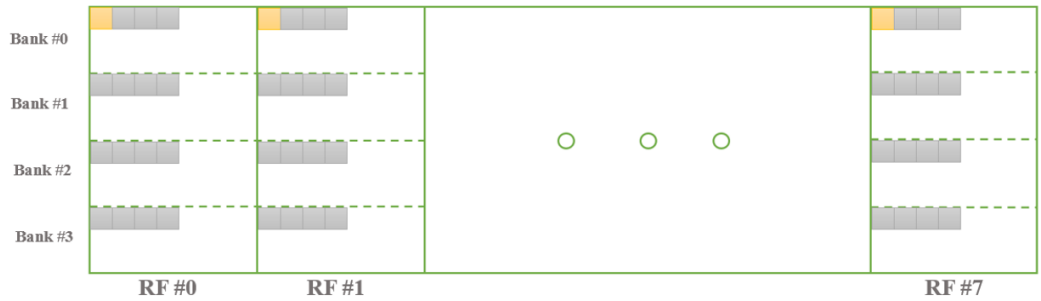


Figure 5.22: Fault injection 2° strategy.

With this strategy the number of signals in the signal list corresponds to 256 (eight registers of 32 bits under test) so, considering 2000 injected faults, an average of 7 faults per each signal in the entire execution interval. Moreover, to inject in all the used registers, it's necessary to run 16 fault injections, since there are four used locations in each bank and four banks.

3° **Strategy: inject all used registers with same number in all banks**

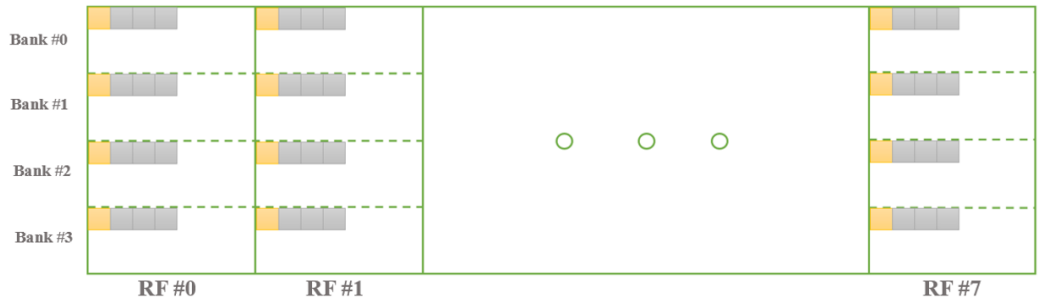


Figure 5.23: Fault injection 3° strategy.

With this strategy the number of signals in the signal list corresponds to 1024 (one register of 32 bits under test per each bank and register file) so, considering 2000 injected faults, an average of 1 fault per each signal in the entire execution interval. Moreover, to inject in all the used registers, it's necessary to run 4 fault injections, since there are four used locations in each bank.

4° **Strategy: inject all used registers of single bank**

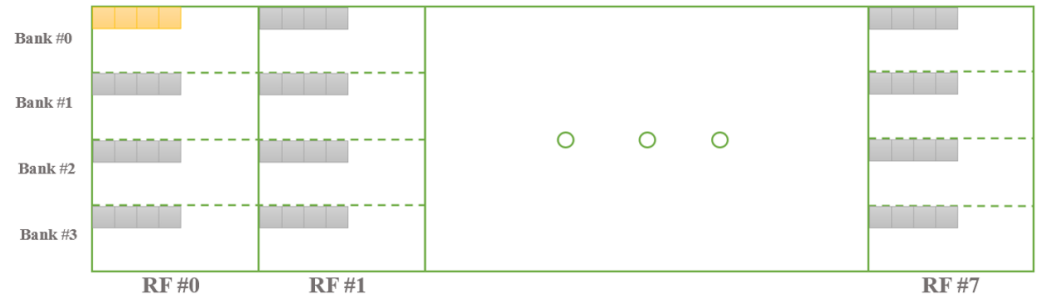


Figure 5.24: Fault injection 4° strategy.

With this strategy the number of signals in the signal list corresponds to 128 (four register of 32 bits under test) so, considering 2000 injected faults, an average of 15 faults per each signal in the entire execution interval. Moreover, to inject in all the used registers, it's necessary to run 32 fault injections, since there are four banks and eight register files.

5° **Strategy: inject all used registers of single register file**

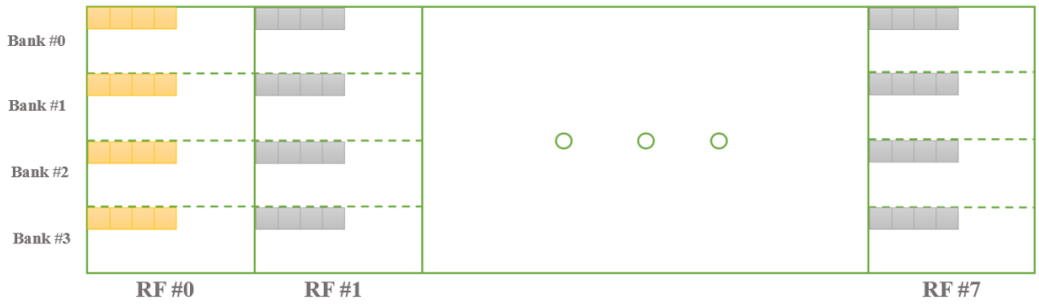


Figure 5.25: Fault injection 5° strategy.

With this strategy the number of signals in the signal list corresponds to 512 (four register of 32 bits under test per each bank) so, considering 2000 injected faults, an average of 4 faults per each signal in the entire execution interval. Moreover, to inject in all the used registers, it's necessary to run 8 fault injections, since there are eight register files.

6° **Strategy: inject all used registers of all banks and register files**

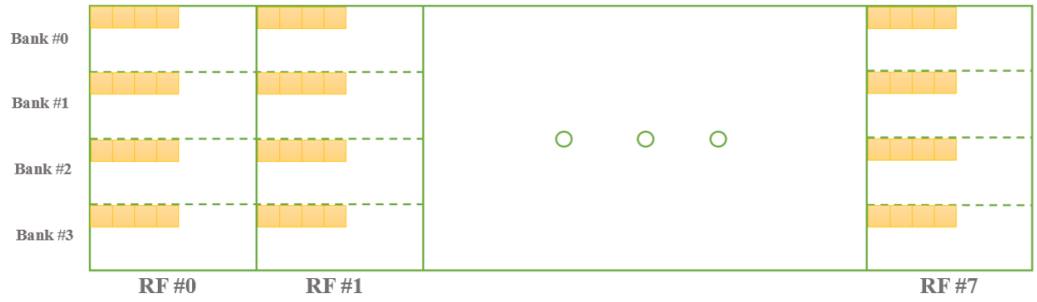


Figure 5.26: Fault injection 6° strategy.

With this strategy the number of signals in the signal list corresponds to 4096 (four register of 32 bits under test per each bank and register file) so, considering 2000 injected faults, an average of 1 fault each two signal in the entire execution interval. Moreover, to inject in all the used registers, it's necessary to run a single fault injection.

Injection strategies summary

From the strategies previously described it's possible to observe that the higher is the number of locations under test, the lower is the number of fault injection simulations that must be run to completely analysed the *vector register file*. Nevertheless, this implies also a reduction

of the number of faults that are injected in each signal and, consequently, a reduction of the accuracy. So, the unique solutions in this case is to increase the number of injected faults or reduce the number of registers injected in a single fault injection. In table 5.37 have been reported the information described in the strategies description for two different applications and memory elements. This could help in the selection of the correct strategy.

VECTOR_SUM				
Memory	**Strategy**	**Number of signals**	**Faults per signal**	**Number of injections**
Vector Register File	1	128	15	32
	2	256	7	16
	3	1024	1	4
	4	128	15	32
	5	512	4	8
	6	4096	0,5	1
Predicate Register File	1	16	125	8
	2	32	62	4
	3	128	15	1
	4	/	/	/
	5	/	/	/
	6	/	/	/

FFT				
Memory	**Strategy**	**Number of signals**	**Faults per signal**	**Number of injections**
Vector Register File	1	128	15	136
	2	256	7	68
	3	1024	1	17
	4	544	3	32
	5	2176	<1	8
	6	17408	0,1	1
Predicate Register File	1	16	125	16
	2	32	62	8
	3	128	15	2
	4	8	250	32
	5	32	62	8
	6	64	1	1

Table 5.37: Strategy parameters for *Vector_sum* and *FFT*.

The tables correspond to an injection of 2000 faults. Obviously the higher is the number of injected faults, the higher will be the number of faults per signal. This table has been filled considering that:

- *Vector_sum* uses 4 locations of the *vector register file* and 1 location of the predicate

one (for this reason the 4°, 5° and 6° strategies have no sense for the *predicate register file*).

- *FFT* uses 17 locations of the *vector register file* and 2 locations of the predicate one.

5.2.3 Fault Injection Results for *Vetcor_sum*

A fault injection simulation has been run for the *Vector_sum* in order to check the correctness of the fault injector and analysed the effects of SEUs injected in the vector and *predicate register file* (in the used locations). The number of faults injected randomly in the execution time has been fixed to 2000 and the strategy adopted the third one. Consequently, the tool injects faults in all the location with the same register number, independently on the bank and the register file. The results of this fault injections are shown in tables 5.38 and 5.39.

Location #0 Vector Register File		
Type	*Total #*	*Percentage*
Evaluated	2000	
Masked	1397	69%
Detected	0	0%
SDC	603	30%
Hang	0	0%
Time_out	0	0%

Location #1 Vector Register File		
Type	*Total #*	*Percentage*
Evaluated	2000	
Masked	1187	59%
Detected	0	0%
SDC	813	40%
Hang	0	0%
Time_out	0	0%

Location #2 Vector Register File		
Type	*Total #*	*Percentage*
Evaluated	2000	
Masked	1462	73%
Detected	0	0%
SDC	537	26%
Hang	1	0%
Time_out	0	0%

Location #3 Vector Register File		
Type	*Total #*	*Percentage*
Evaluated	2000	
Masked	1687	84%
Detected	0	0%
SDC	312	15%
Hang	1	0%
Time_out	0	0%

Table 5.38: Vector register file fault injection results for the *Vector_sum*

Location #0 Predicate Register File		
Type	*Total #*	*Percentage*
Evaluated	2000	
Masked	0	0%
Detected	0	0%
SDC	0	0%
Hang	2000	100%
Time_out	0	0%

Table 5.39: Predicate register file fault injection results for the *Vector_sum*

The injection results show that, as regard the *vector register file*, the most critical locations is the location #1, that causes an higher number of SDCs and so of output errors. Vice versa the location with lowest sensitivity to SEUs results to be the location #3, that generates only 15% of SDCs.
For what concern the *predicate register file*, it's possible to observe that each single injected fault generates an hang. Consequently, in the optics of a selective hardening, this location must be hardened to guarantee the correct execution of the *Vector_sum* application.

5.3 Conclusions and Future Developments

This thesis work has been focused to the development of a monitoring system on a GPGPU architecture (i.e. FlexGrip based on NVIDIA G80) in order to monitor the execution of different application and provide at the output specific log files referred to the main internal modules of the FLexGrip itself. This allows to analyse in details the kernel execution of several application but also provides additional information that, in a fault injection study for radiation hardening design, permit to very precisely identify the critical modules and, consequently, to adopt selective hardening techniques. These methods, combined with the performance analysis and the fault injection study, allow to harden the system against radiation effects, avoiding an excessive area or performance overhead. Therefore, the implemented monitoring system can be also used to prove the efficiency of an hardening technique in terms of introduced performance degradation. As a proof, this work has been used in collaboration with the Universidade Federal do Rio Grande do Sul (UFRGS) to quantitatively identify the performance overhead of several applications modified with the resilient global LOAD and STORE instruction with respect to the original version of the same applications. The monitoring system has been developed under specific assumption that, for instance, does not include the multiple warp configuration. In this case the monitoring system must be modified in order to track also the warp ID and, consequently, to associate each of the results obtained and described in this thesis work to the correct warp.
Furthermore, the monitoring system and fault injector has been developed considering as goal to study the effect of SEUs. Nevertheless, with proper modification of the fault injector, using the same monitoring system, also the effect of other radiation effects can be studied, e.g. the SET.
In conclusion, this thesis proposed a monitoring system able to provide additional performance information that certainly are significant both in the design and validation steps of a radiation hardening technique.

References

[1] N. Bidokhti, "*SEU concept to reality (allocation, prediction, mitigation),*" 2010 Proceedings - Annual Reliability and Maintainability Symposium (RAMS), San Jose, CA, 2010, pp. 1-5.

[2] R. Edwards, C. Dyer and E. Normand, "*Technical standard for atmospheric radiation single event effects, (SEE) on avionics electronics*," 2004 IEEE Radiation Effects Data Workshop (IEEE Cat. No.04TH8774), Atlanta, GA, USA, 2004, pp. 1-5.

[3] H. Aboutaleb and B. Monsuez, "*Trade-off in logical radiation hardening: Approach, mechanisms, and reliability impacts*," 2016 Annual Reliability and Maintainability Symposium (RAMS), Tucson, AZ, 2016, pp. 1-6.

[4] J. E. Knudsen and L. T. Clark, "*An Area and Power Efficient Radiation Hardened by Design Flip-Flop*," in IEEE Transactions on Nuclear Science, vol. 53, no. 6, pp. 3392-3399, Dec. 2006.

[5] K. Andryc, M. Merchant and R. Tessier, "*FlexGrip: A soft GPGPU for FPGAs*," 2013 International Conference on Field-Programmable Technology (FPT), Kyoto, 2013, pp. 230-237.

[6] B. Du, J. E. R. Condia, M. S. Reorda and L. Sterpone, "*About the functional test of the GPGPU scheduler*," 2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS), Platja d'Aro, 2018, pp. 85-90.

[7] X. Wang and J. Xu, "*Second Hamming Code Used in Radiation Hardened Communication Environment*," 2016 Sixth International Conference on Instrumentation & Measurement, Computer, Communication and Control (IMCCC), Harbin, 2016, pp. 593-596.

[8] L. Frigerio, M. A. Radaelli and F. Salice, "*Convolutional Coding for SEU mitigation*," 2008 13th European Test Symposium, Verbania, 2008, pp. 191-196.

[9] A. Sari and M. Psarakis, "*Scrubbing-based SEU mitigation approach for Systems-on-Programmable-Chips*," 2011 International Conference on Field-Programmable Technology, New Delhi, 2011, pp. 1-8.

[10] S. Zheng et al., "*A Rapid Scrubbing Technique for SEU Mitigation on SRAM-Based FPGAs*," 2019 IEEE International Symposium on Circuits and Systems (ISCAS), Sapporo, Japan, 2019, pp. 1-5.

[11] D. T. Stott, G. Ries, Mei-Chen Hsueh and R. K. Iyer, *"Dependability analysis of a high-speed network using software-implemented fault injection and simulated fault injection,"* in IEEE Transactions on Computers, vol. 47, no. 1, pp. 108-119, Jan. 1998.

[12] E. Cioroaica et al., "*Accelerated Simulated Fault Injection Testing*," 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Toulouse, 2017, pp. 228-233.

[13] J. Perez, M. Azkarate-askasua and A. Perez, "*Codesign and Simulated Fault Injection of Safety-Critical Embedded Systems Using SystemC*," 2010 European Dependable Computing Conference, Valencia, 2010, pp. 221-229.

[14] O. Boncalo, M. Udrescu, L. Prodan, M. Vladutiu and A. Amaricai, "*Using Simulated Fault Injection for Fault Tolerance Assessment of Quantum Circuits*," 40th Annual Simulation Symposium (ANSS'07), Norfolk, VA, 2007, pp. 213-220.

[15] R. Leveugle and A. Prost-Boucle, "*A new automated instrumentation for emulation-based fault injection*," 2010 First IEEE Latin American Symposium on Circuits and Systems (LASCAS), Foz do Iguacu, 2010, pp. 200-203.

[16] C. Lopez-Ongil, M. Garcia-Valderas, M. Portela-Garcia and L. Entrena, "*Autonomous Fault Emulation: A New FPGA-Based Acceleration System for Hardness Evaluation*," in IEEE Transactions on Nuclear Science, vol. 54, no. 1, pp. 252-261, Feb. 2007.

[17] Kuei-Shu Chang-Liao and Kuang-Hsien Feng, "*Radiation hardness of static random-access-memory tested using dose-to-failure and gamma-ray exposure*," in IEEE Transactions on Reliability, vol. 47, no. 2, pp. 155-158, June 1998.

[18] A. J. Williams, M. R. McEwen and A. R. DuSautoy, "*Radiation testing for space applications at the National Physical Laboratory*," 1998 IEEE Radiation Effects Data Workshop. NSREC 98. Workshop Record. Held in conjunction with IEEE Nuclear and Space Radiation Effects Conference (Cat. No.98TH8385), Newport Beach, CA, USA, 1998, pp. 148-151.

[19] G. Tsiligiannis et al., "*Testing a Commercial MRAM Under Neutron and Alpha Radiation in Dynamic Mode*," in IEEE Transactions on Nuclear Science, vol. 60, no. 4, pp. 2617-2622, Aug. 2013.

[20] Chatzidimitriou, Athanasios Bodmann, Pablo Papadimitriou, George Gizopoulos, Dimitris & Rech, Paolo. (2019). "*Demystifying Soft Error Assessment Strategies on ARM CPUs: Microarchitectural Fault Injection vs. Neutron Beam Experiments*". 10.1109/DSN.2019.00018.

[21] Quinn, Heather M.. "*Challenges in Testing Complex Systems.*" IEEE Transactions on Nuclear Science 61 (2014): 766-786.

[22] G. Foucard, P. Peronnard and R. Velazco, "*Reliability limits of TMR implemented in a SRAM-based FPGA: Heavy ion measures vs. fault injection predictions*," 2010 11th Latin American Test Workshop, Pule del Este, 2010, pp. 1-5.

[23] K. S. Siddiqui and M. A. Baig, "*FRAM based TMR (triple modular redundancy) for fault tolerance implementation*," 2011 International Conference on Information and Communication Technologies, Karachi, 2011, pp. 1-5.

[24] She, Xiaoxuan Samudrala, P.K.. (2009). "*Selective Triple Modular Redundancy for Single Event Upset (SEU) Mitigation*". Proceedings - 2009 NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2009. 344 - 350. 10.1109/AHS.2009.9.

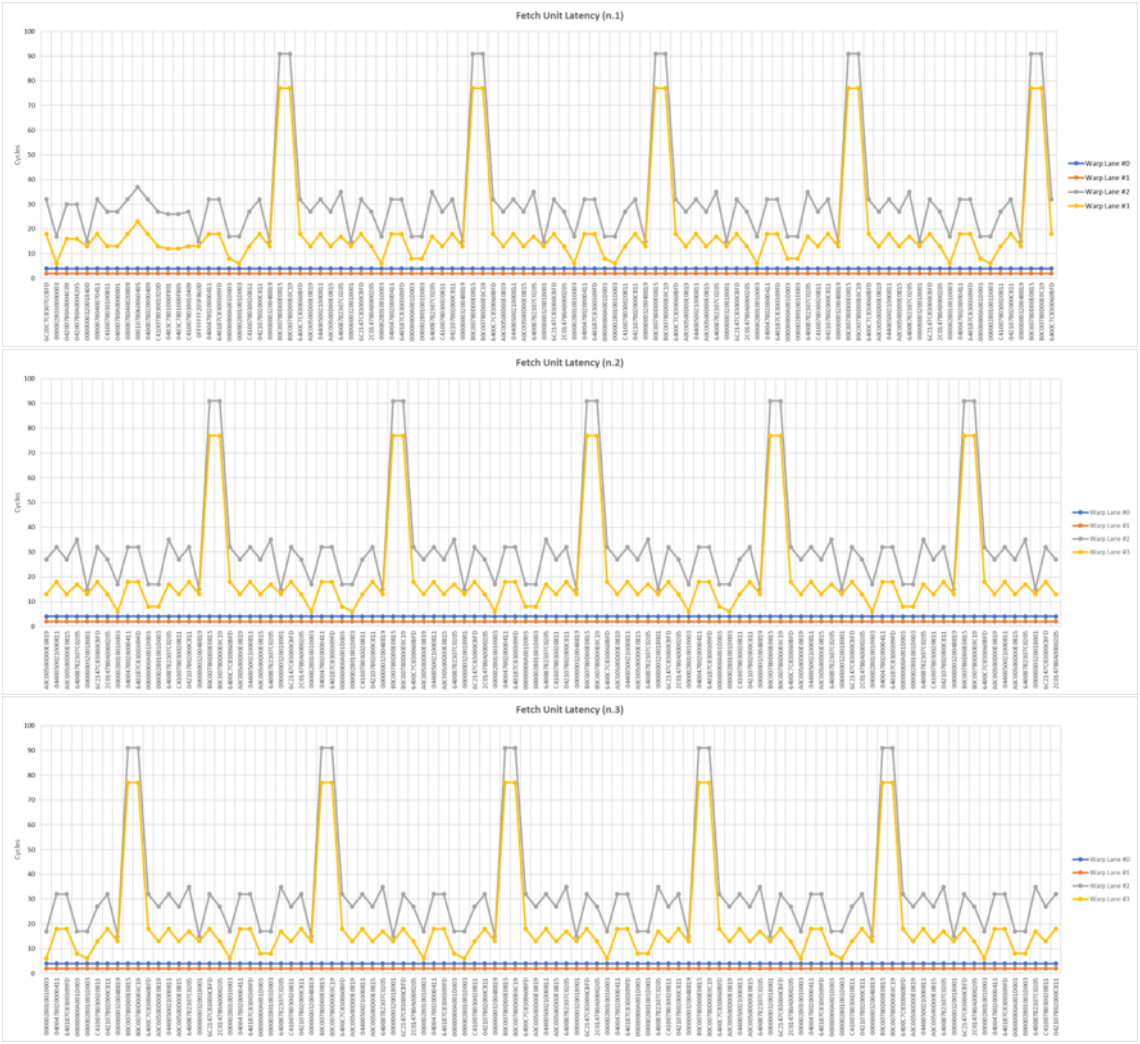
[25] Y. Jin, Y. Huan, H. Chu, Z. Zou and L. Zheng, "*TMR Group Coding Method for Optimized SEU and MBU Tolerant Memory Design*," 2018 IEEE International Symposium on Circuits and Systems (ISCAS), Florence, 2018, pp. 1-5.

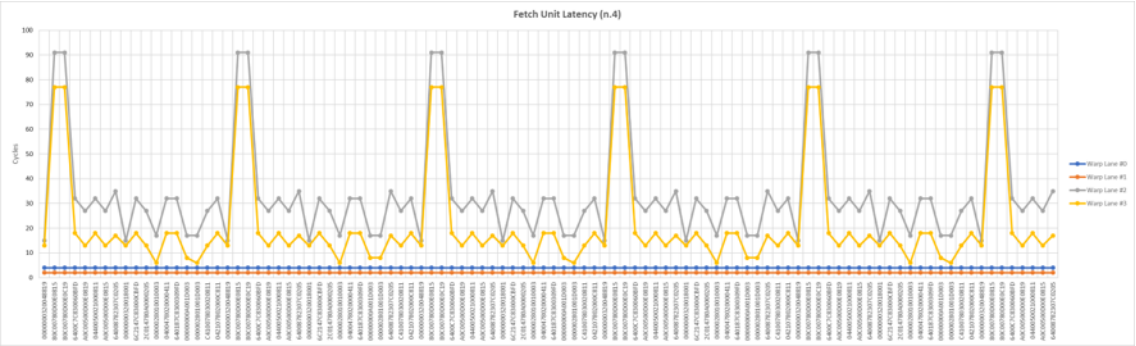
[26] F. A. Siddiqui and P. Gour, "*Scan-chain-based multiple error recovery in TMR systems (SMERTMR)*," 2014 Innovative Applications of Computational Intelligence on Power, Energy and Controls with their impact on Humanity (CIPECH), Ghaziabad, 2014, pp. 374-378.

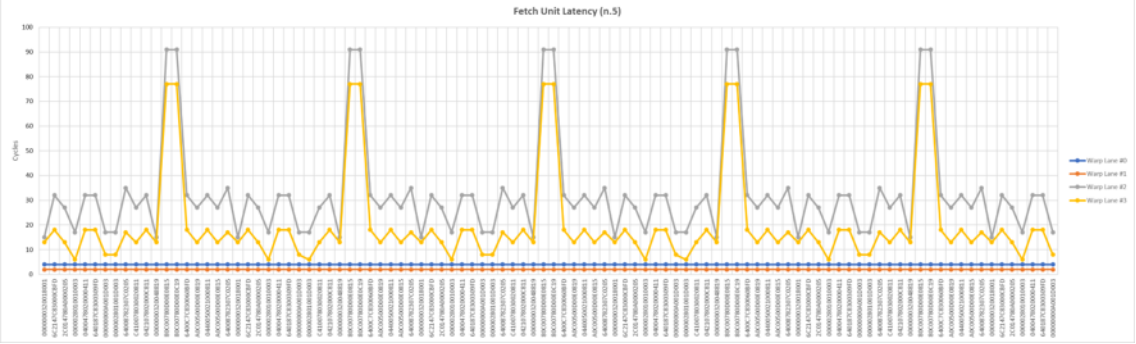
APPENDIX A

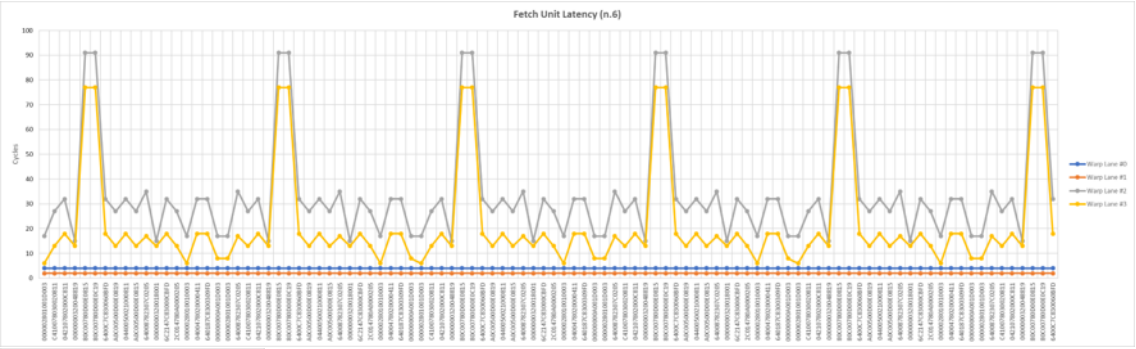
Sort

A.1 Fetch Unit

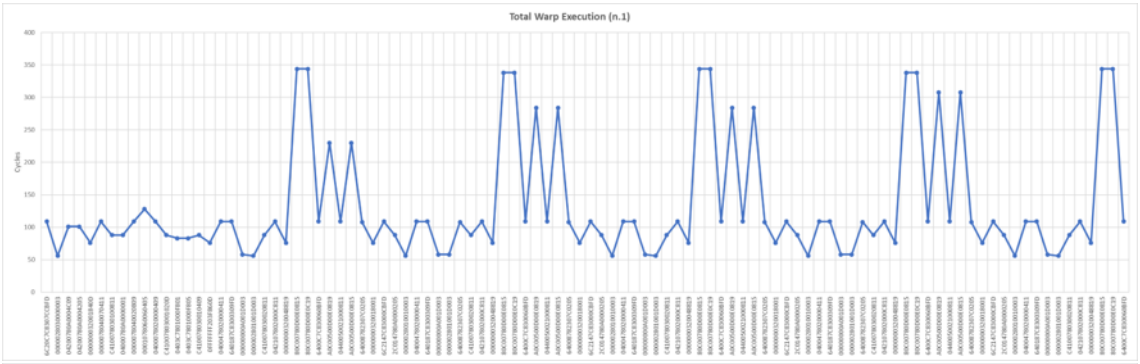


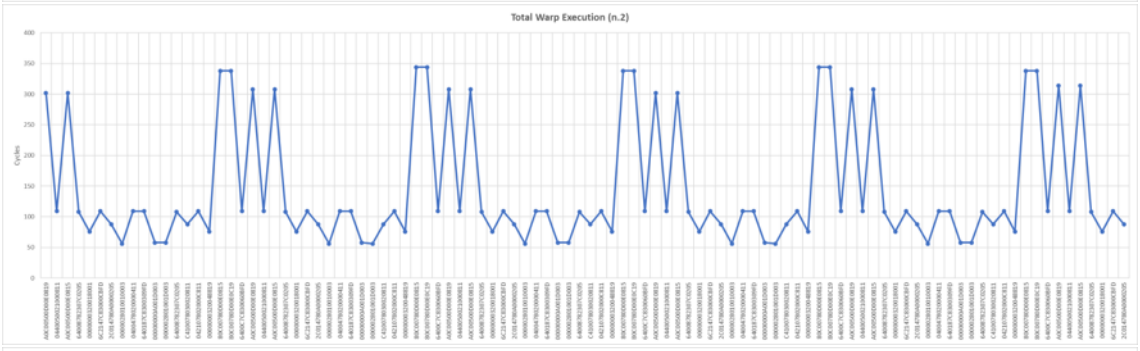

Fetch Unit Latency (n.4)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

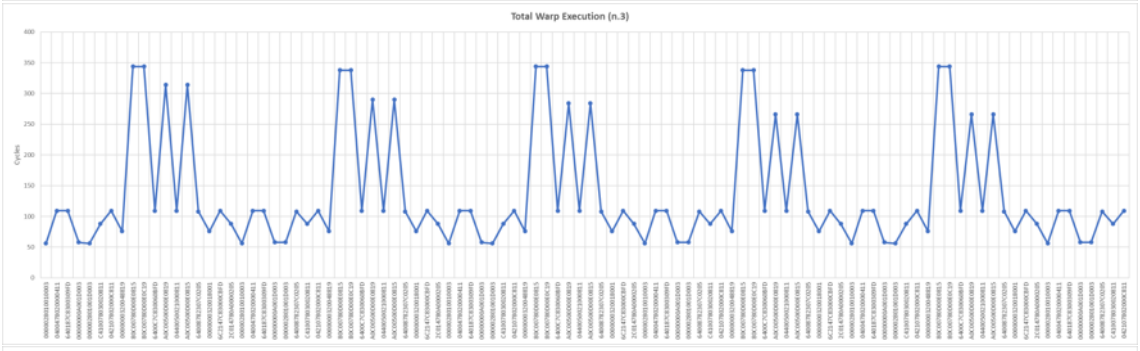

Fetch Unit Latency (n.5)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

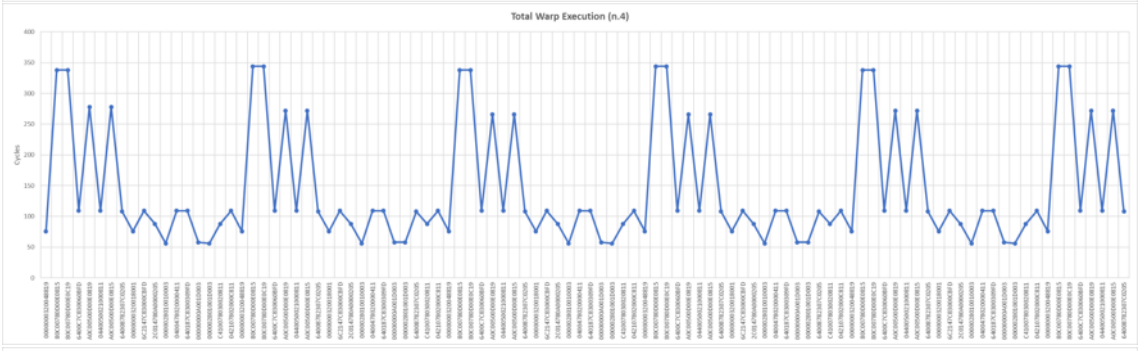

Fetch Unit Latency (n.6)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

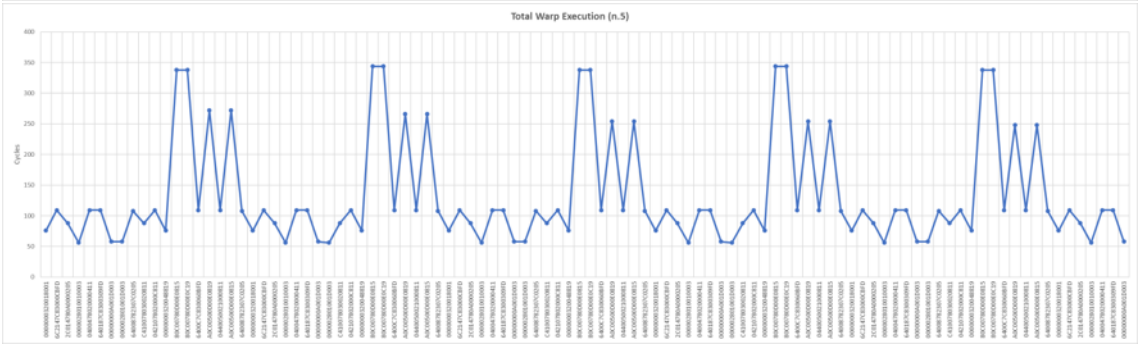

Fetch Unit Latency (n.7)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

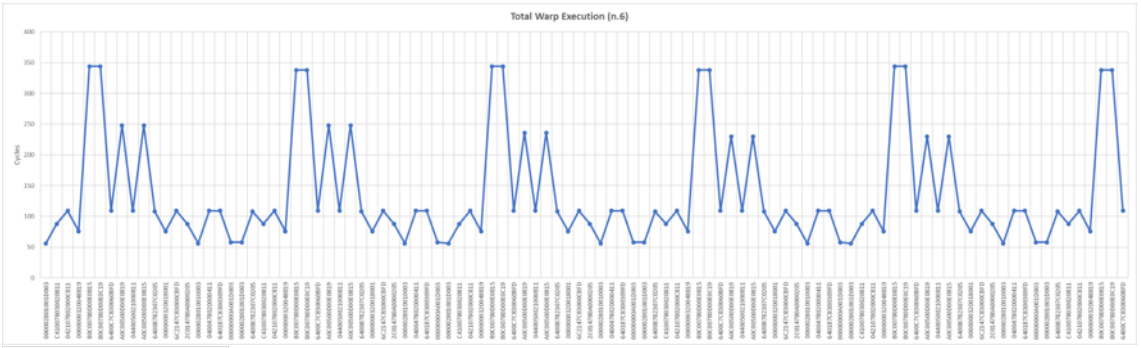

Total Warp Execution (n.1)
Cycles

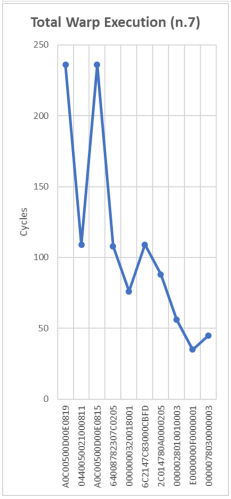

Total Warp Execution (n.2)
Cycles


Total Warp Execution (n.3)
Cycles

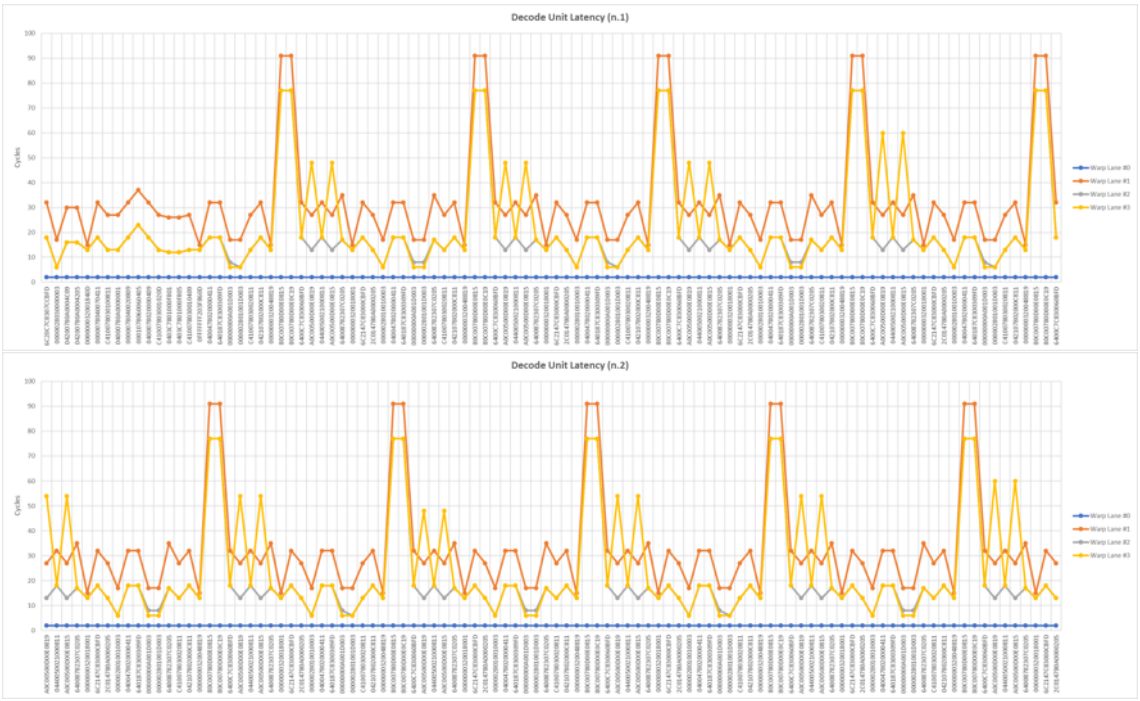

Total Warp Execution (n.4)
Cycles

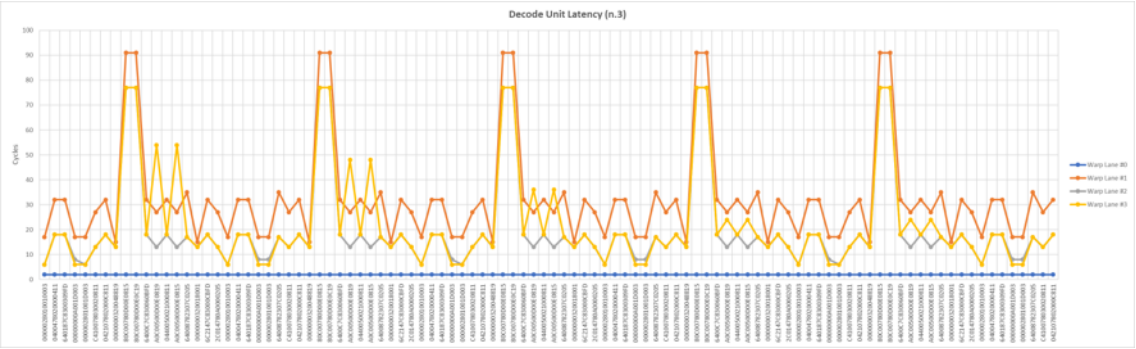

Total Warp Execution (n.5)
Cycles

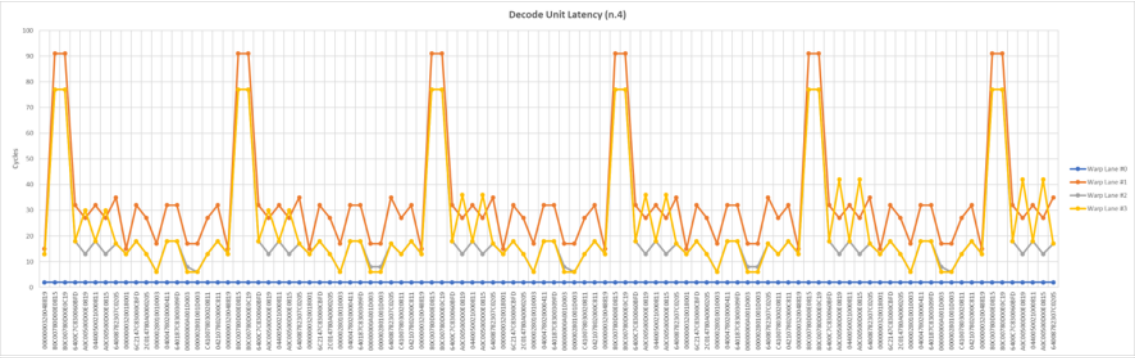


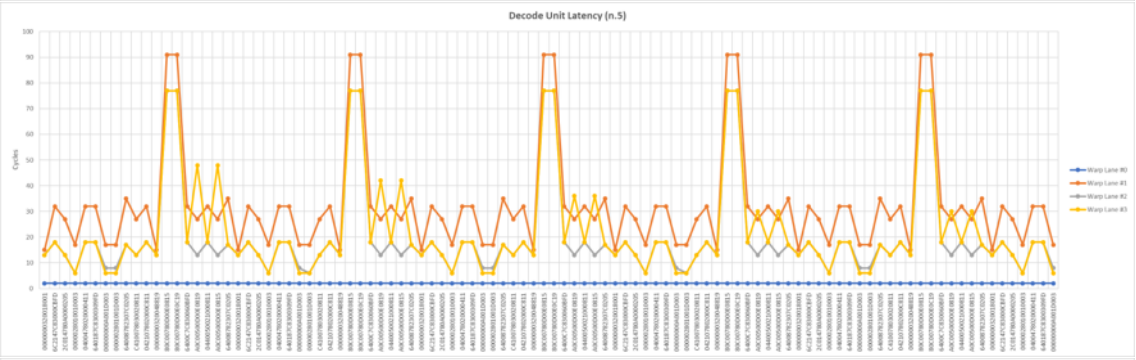


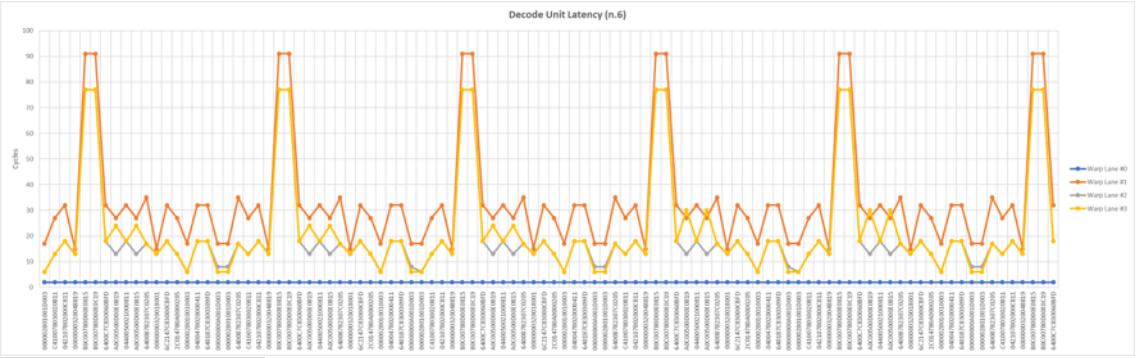
A.2 Decode Unit

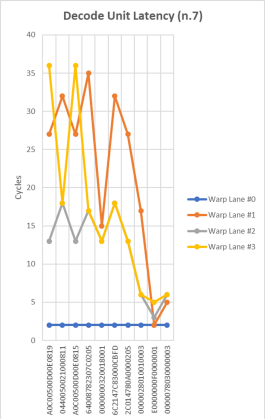



Decode Unit Latency (n.3)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

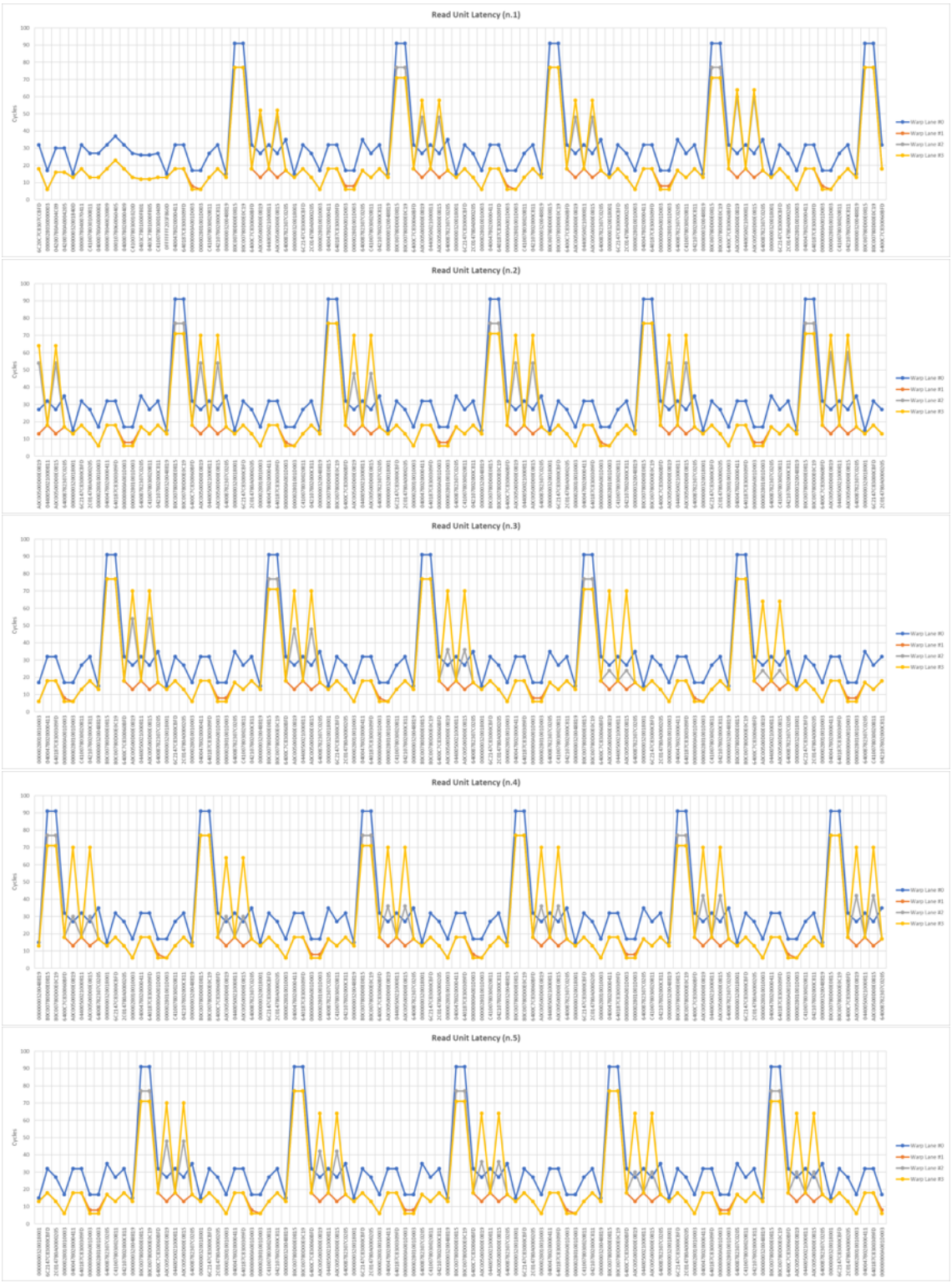

Decode Unit Latency (n.4)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

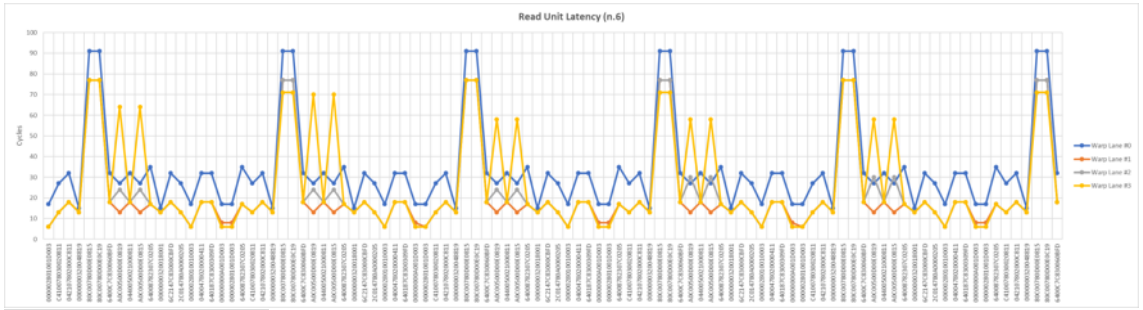

Decode Unit Latency (n.5)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

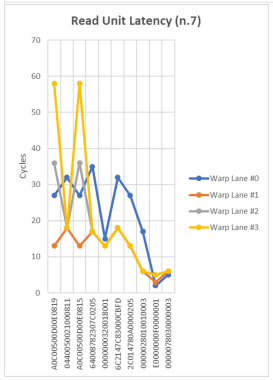

Decode Unit Latency (n.6)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3


Decode Unit Latency (n.7)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

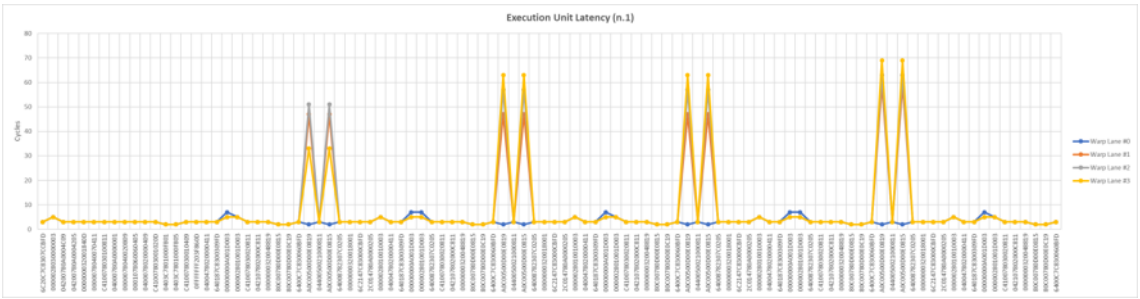
A.3 Read Unit

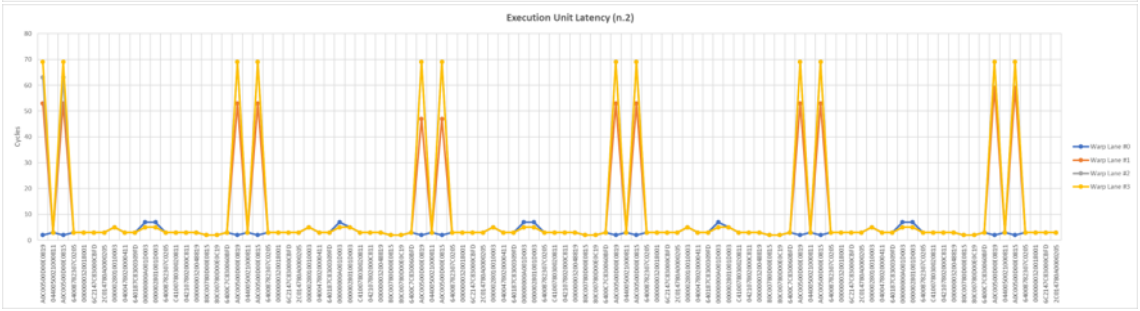

Read Unit Latency (n.1)
Read Unit Latency (n.2)
Read Unit Latency (n.3)
Read Unit Latency (n.4)
Read Unit Latency (n.5)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

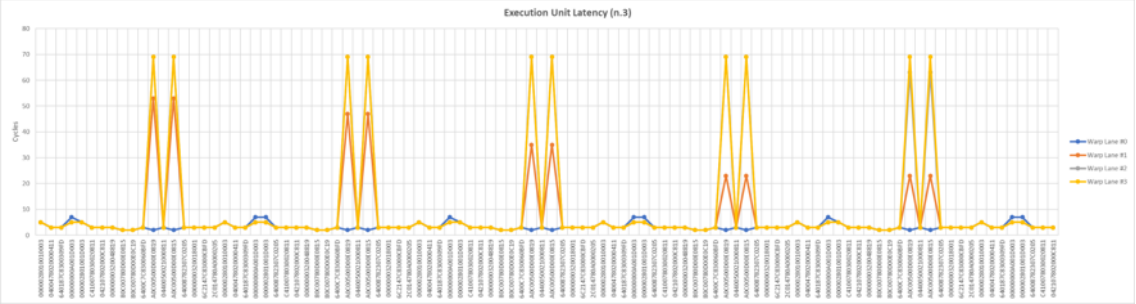


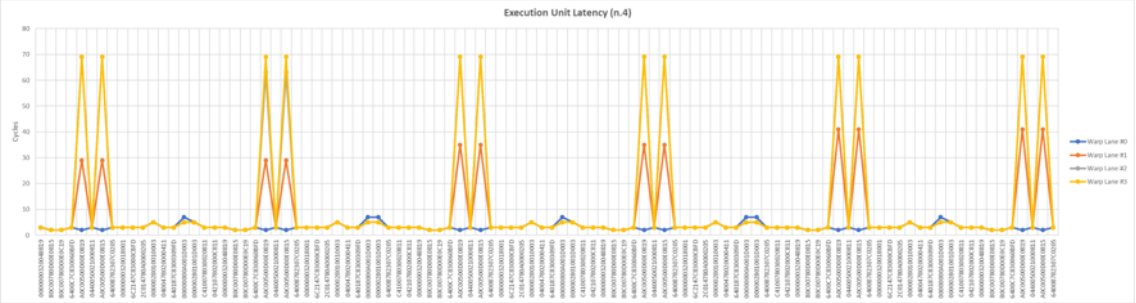


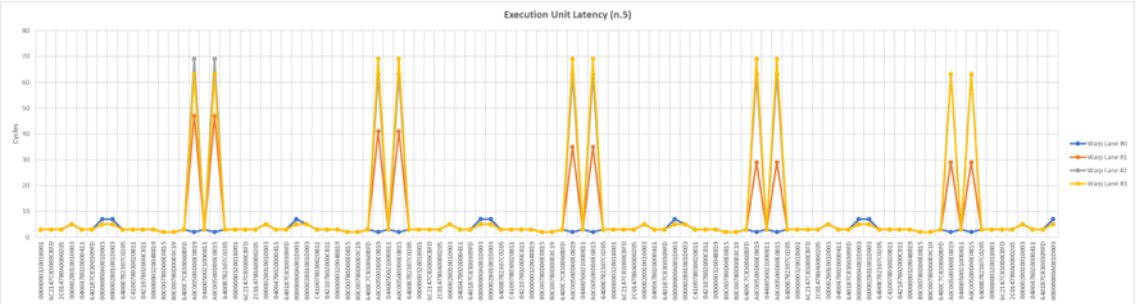
A.4 Execution Unit

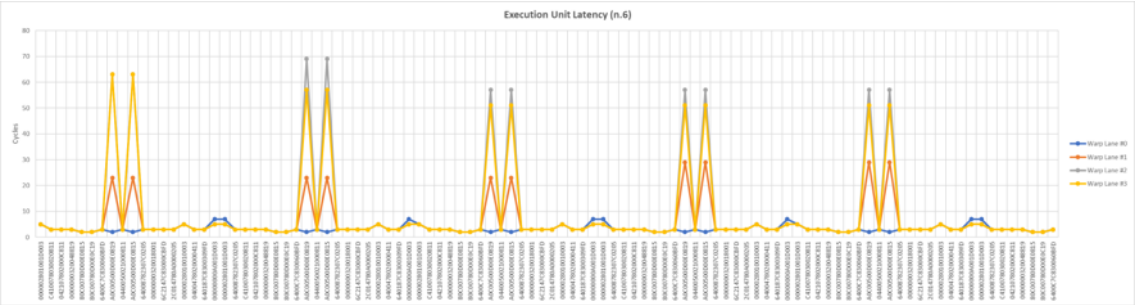


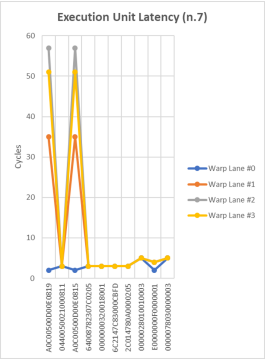



Execution Unit Latency (n.3)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

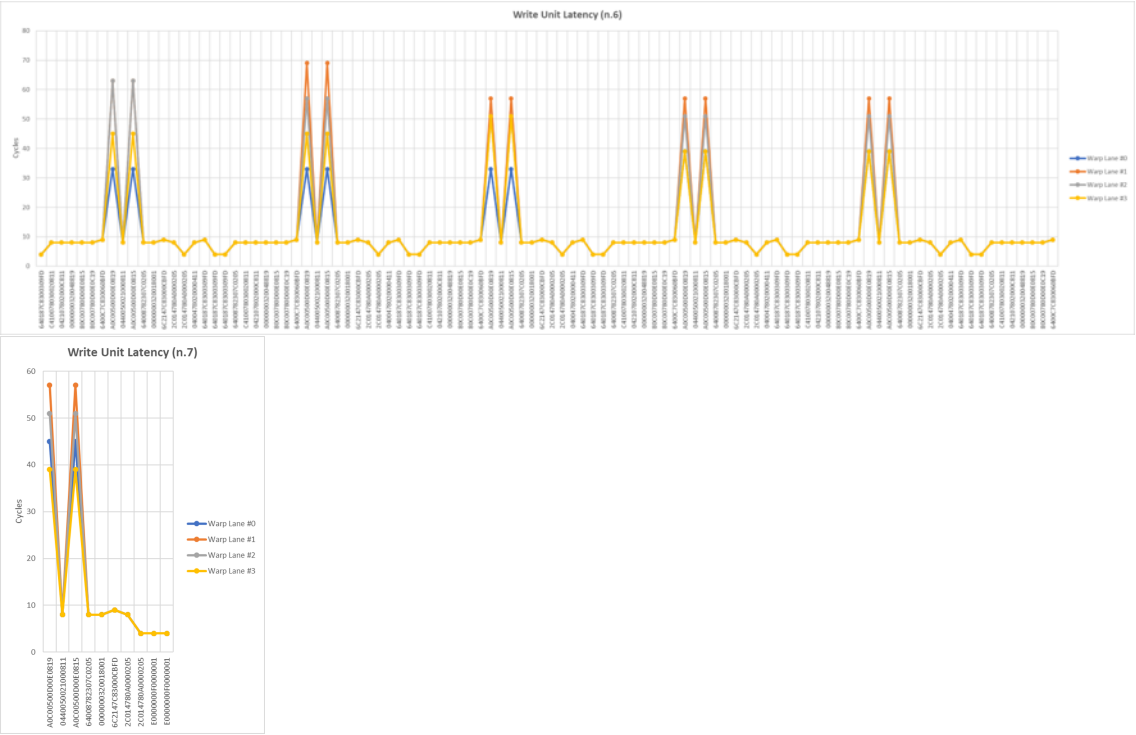

Execution Unit Latency (n.4)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3


Execution Unit Latency (n.5)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

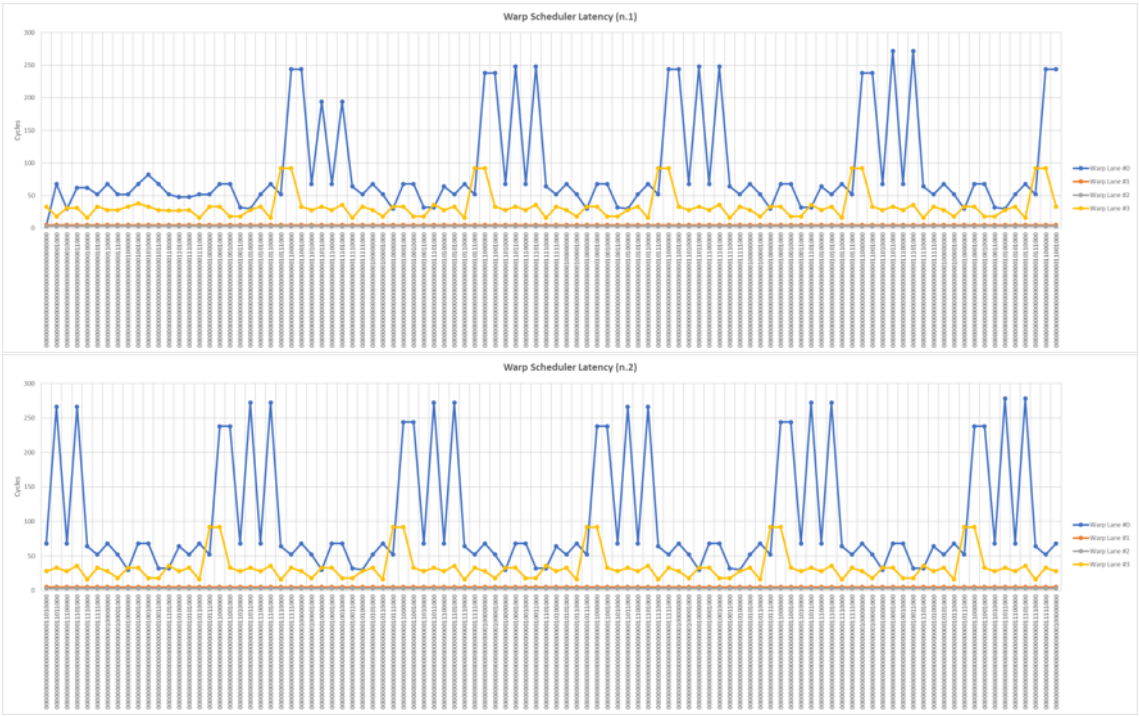

Execution Unit Latency (n.6)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

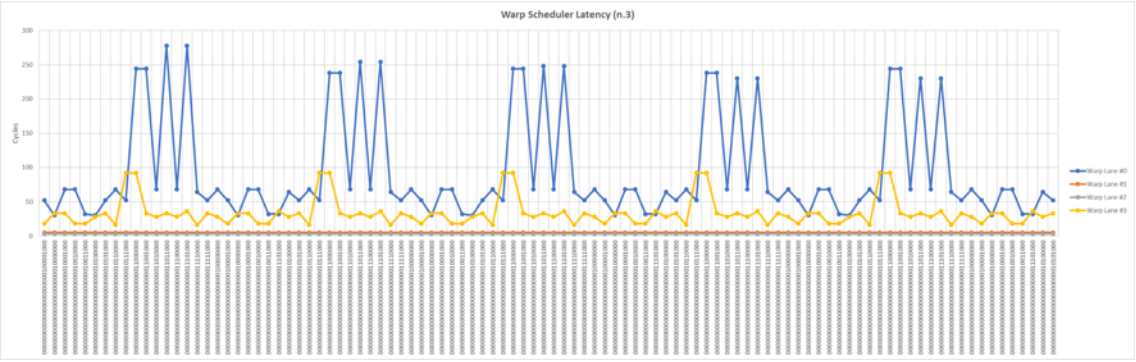

Execution Unit Latency (n.7)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

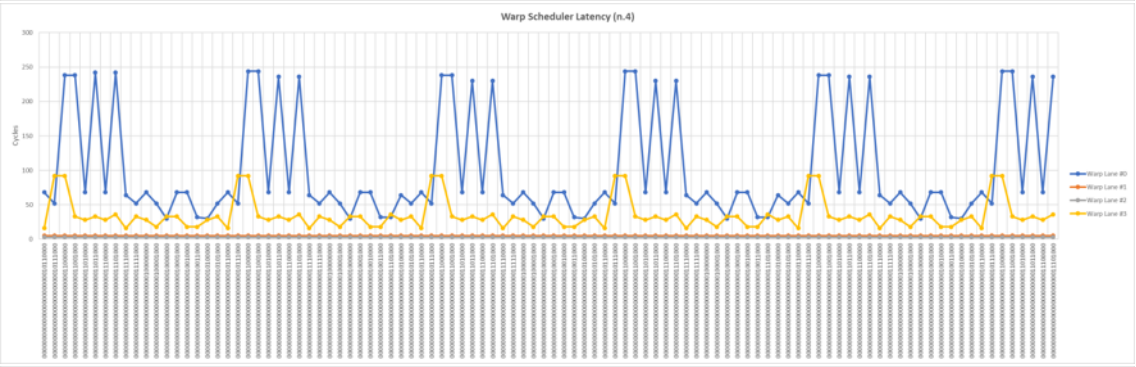
A.5 Write Unit

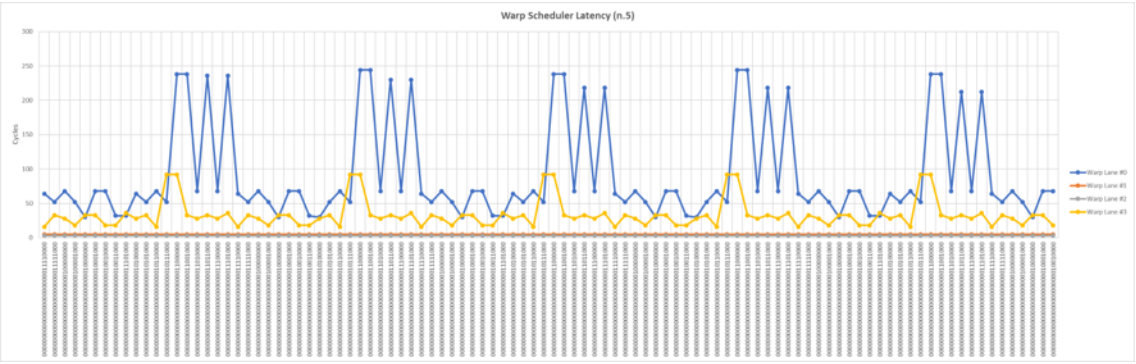


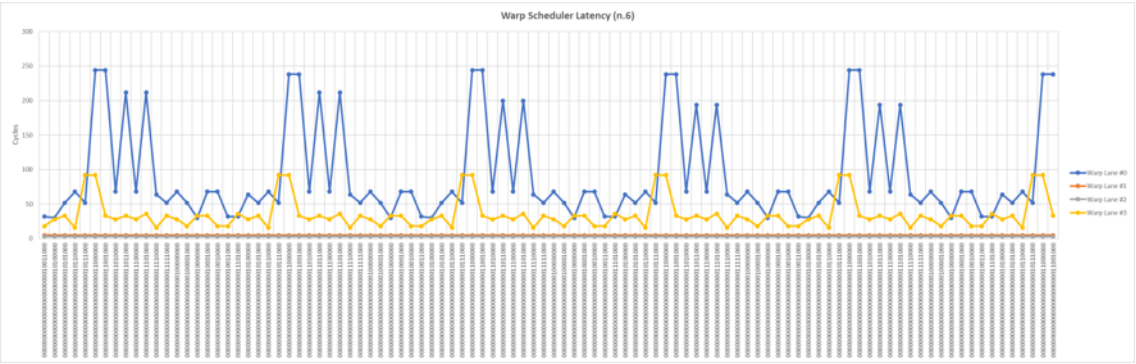
A.6 Warp Scheduler

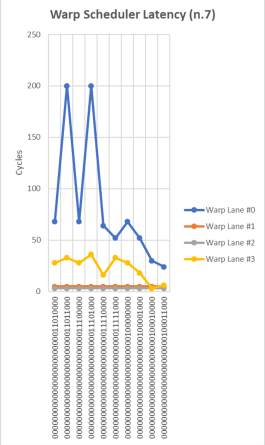



Warp Scheduler Latency (n.3)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3


Warp Scheduler Latency (n.4)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3


Warp Scheduler Latency (n.5)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3


Warp Scheduler Latency (n.6)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3


Warp Scheduler Latency (n.7)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

A.7 Warp Checker

Independently on the instruction and the warp lane, the warp checker always spends 2 clock cycles to complete the check operation.

A.8 Warp Generator

Time_interval	#Cycles
Total_warp_generation	7

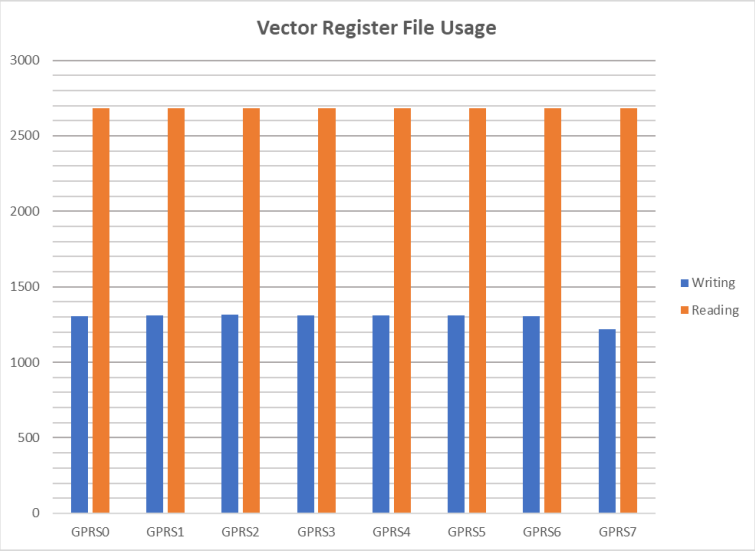
A.9 Streaming Multiprocessor Controller

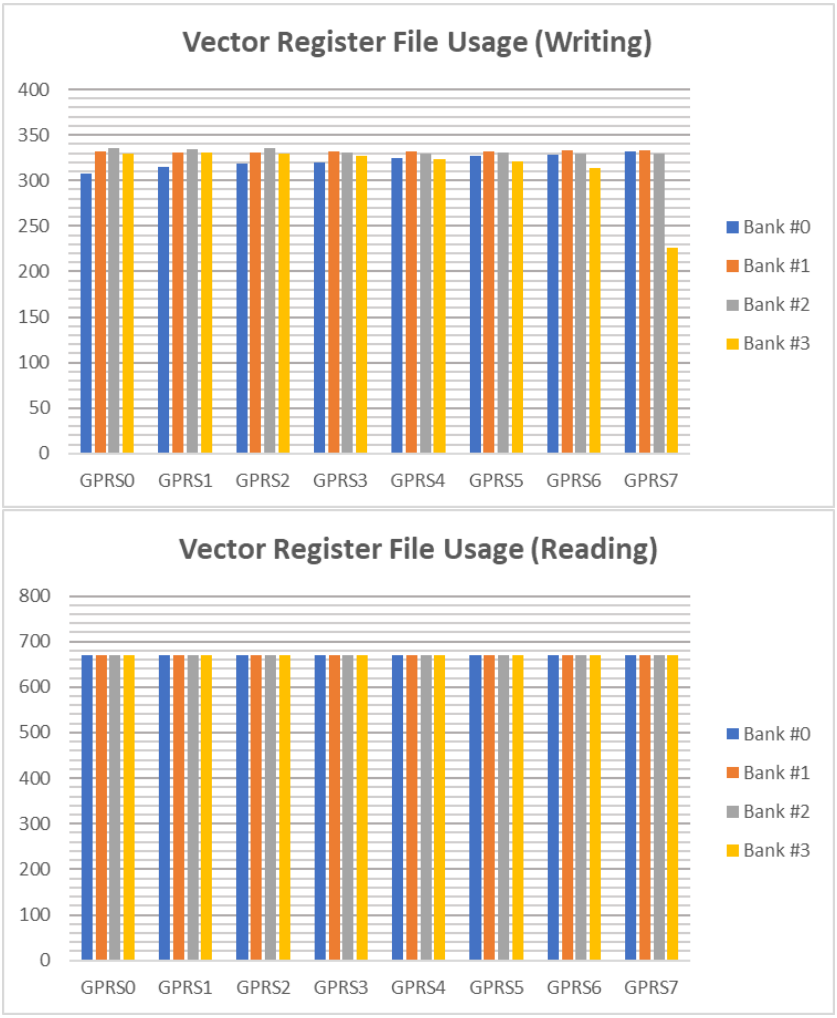
Time_interval	#Cycles
SM_task_execution	83091

A.10 Block Scheduler

Time_interval	#Cycles
Block_Scheduling	4

A.11 Vector Register File

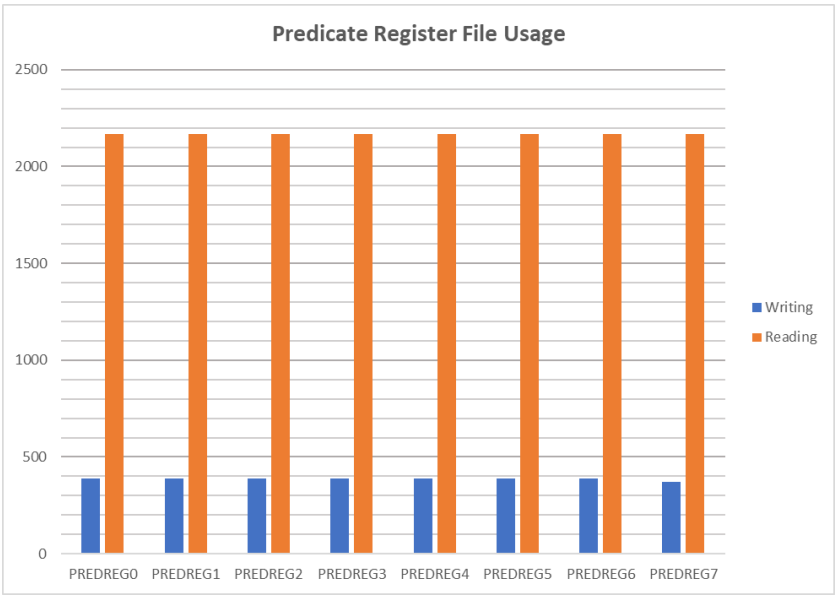


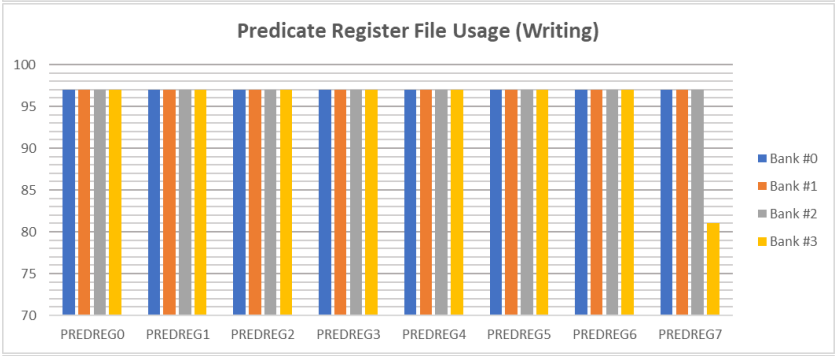


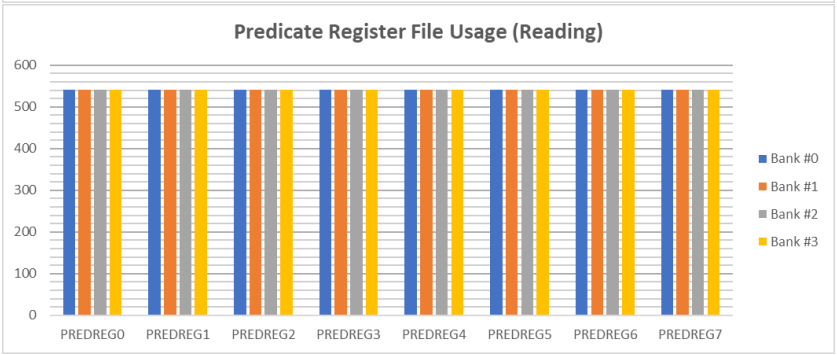
A.12 Address Register File

Module	# writings	# readings
ADDREG0	0	0
ADDREG1	0	0
ADDREG2	0	0
ADDREG3	0	0
ADDREG4	0	0
ADDREG5	0	0
ADDREG6	0	0
ADDREG7	0	0

A.13 Predicate Register File



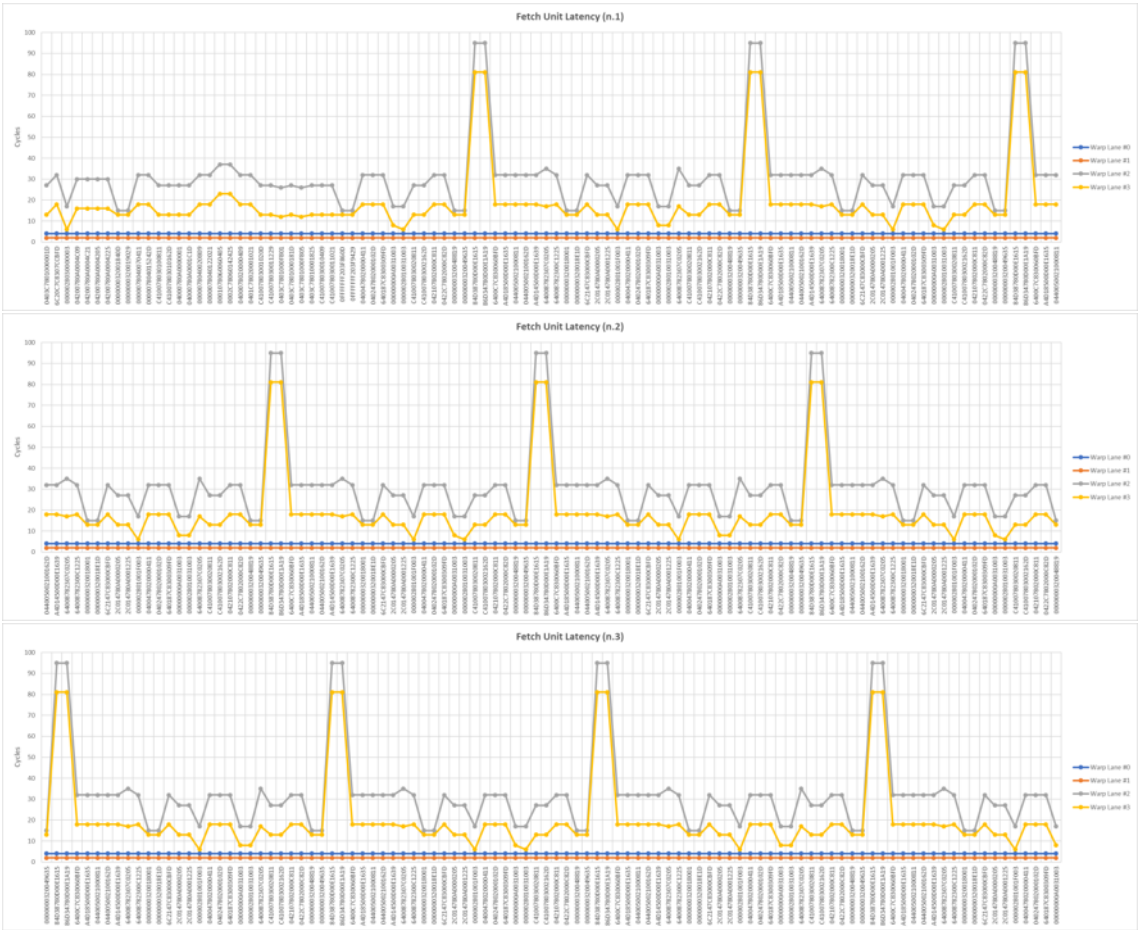


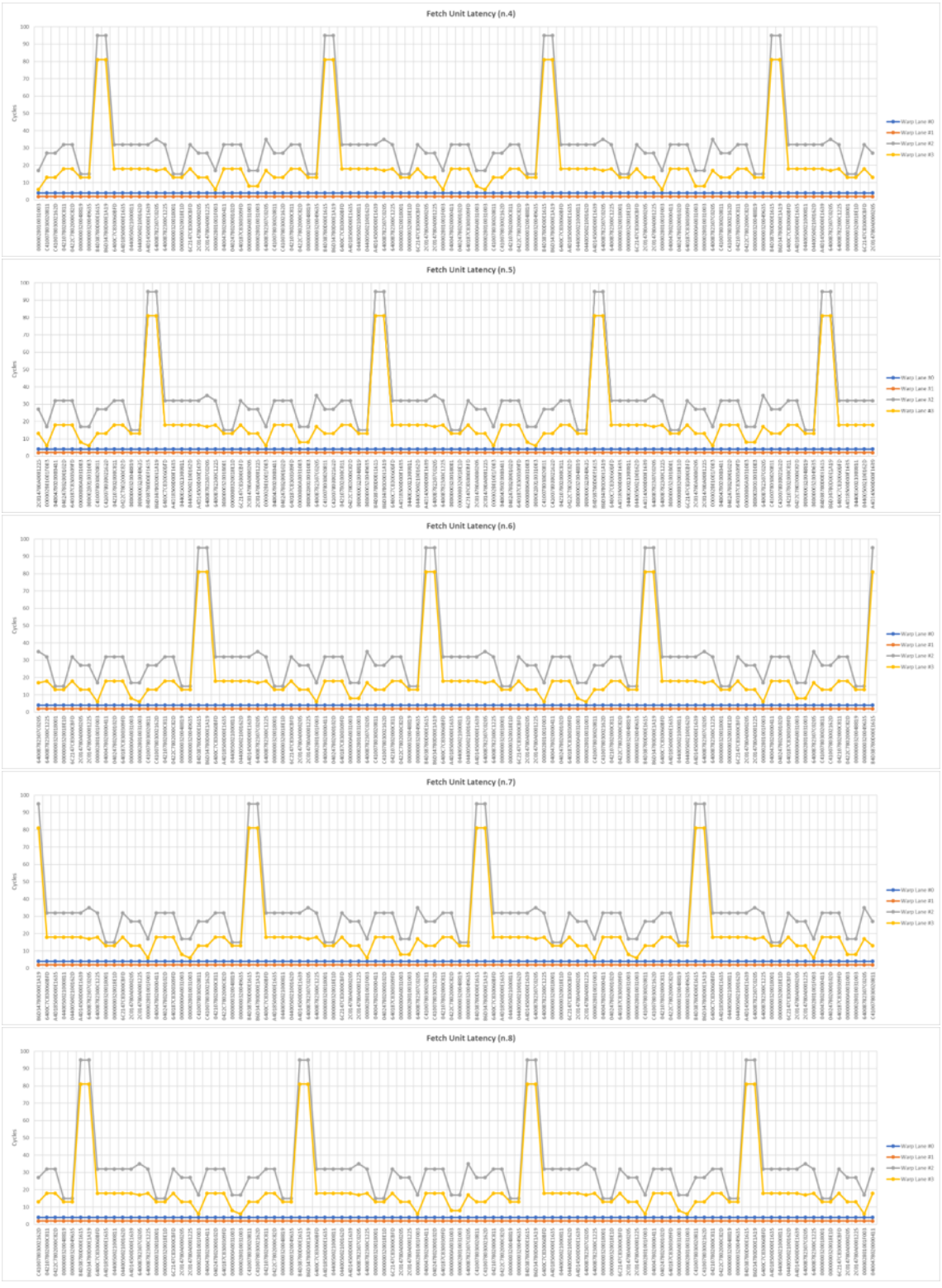


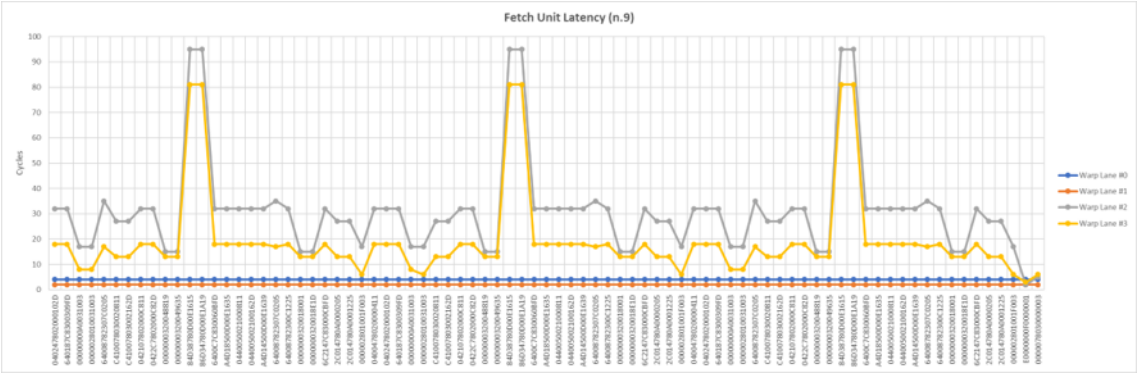
APPENDIX B

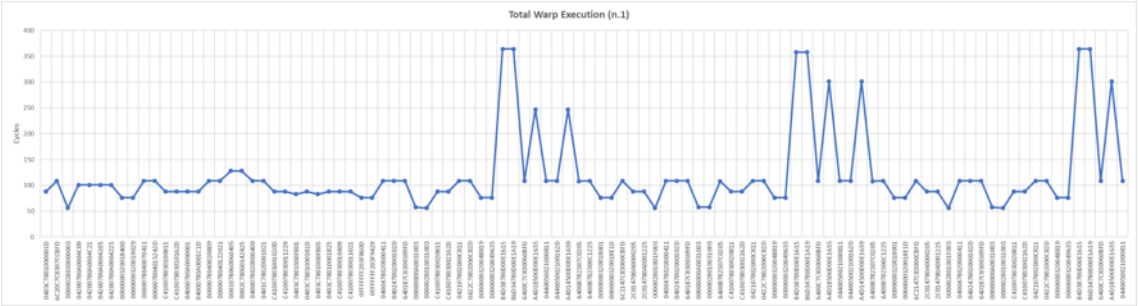
Sort with Resilient LOAD/STORE Instruction

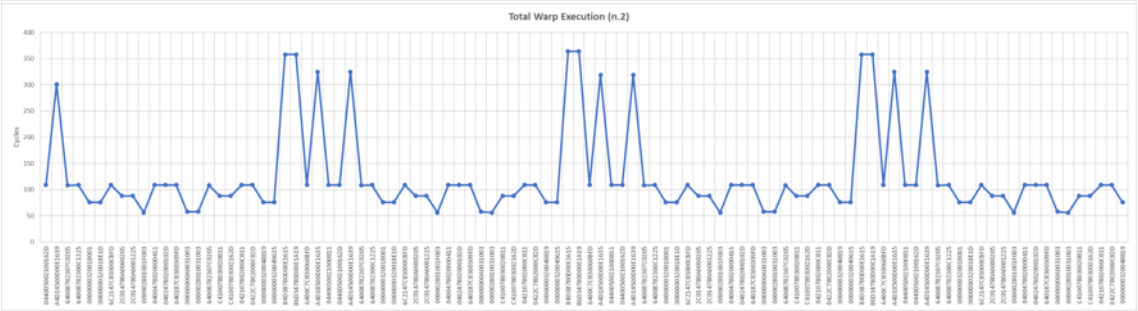
B.1 Fetch Unit

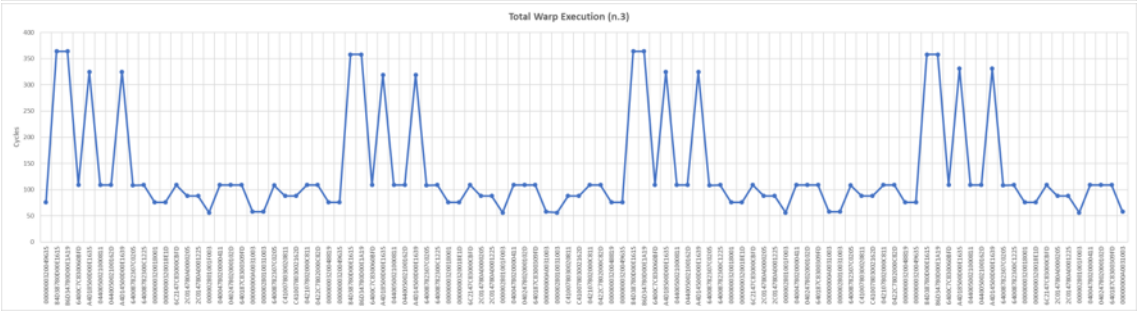


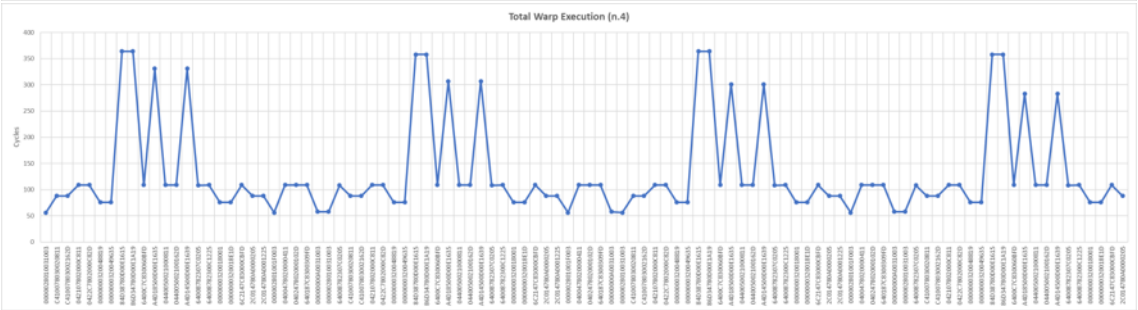

Fetch Unit Latency (n.4)
Fetch Unit Latency (n.5)
Fetch Unit Latency (n.6)
Fetch Unit Latency (n.7)
Fetch Unit Latency (n.8)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

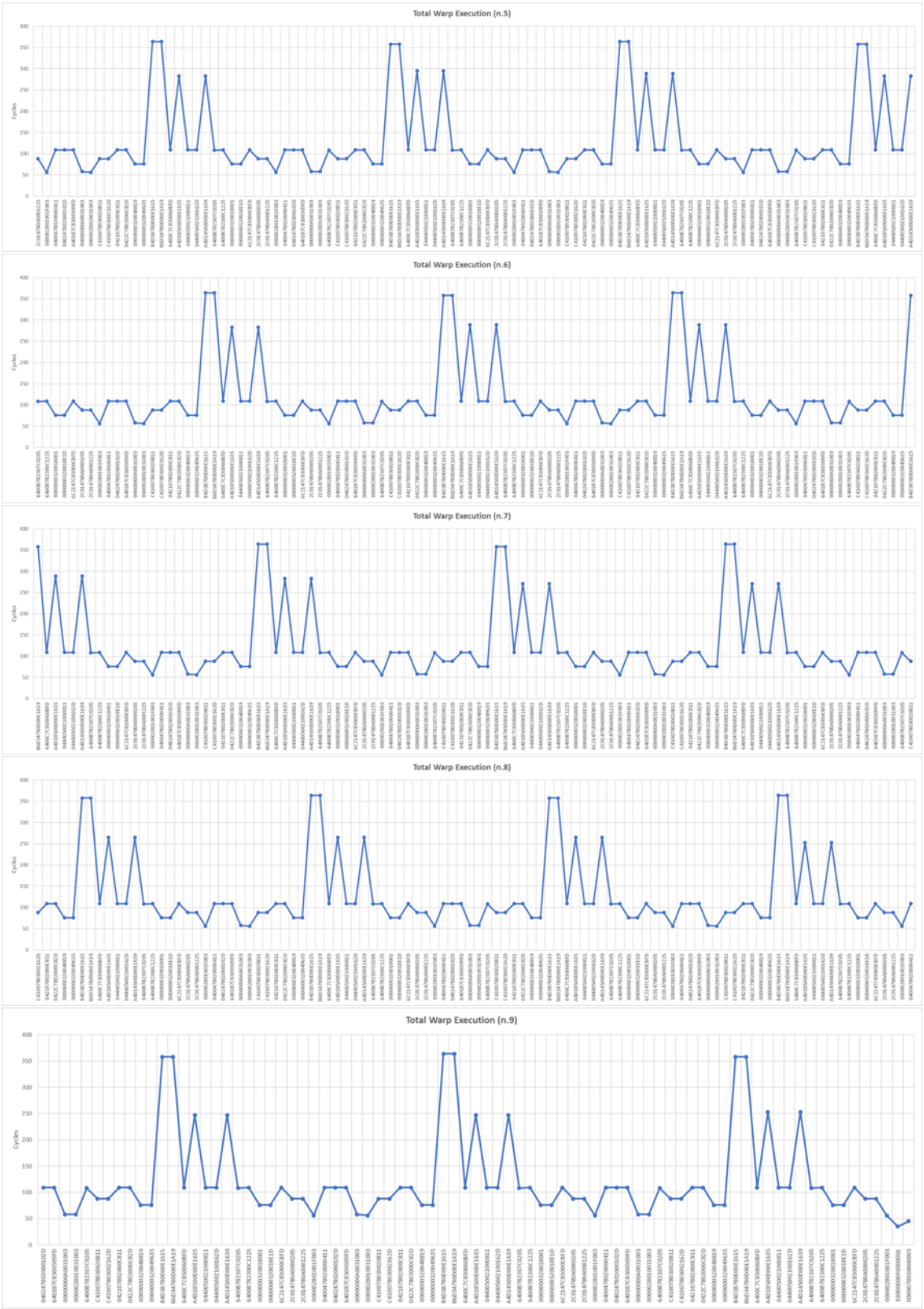

Fetch Unit Latency (n.9)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3


Total Warp Execution (n.1)
Cycles


Total Warp Execution (n.2)
Cycles


Total Warp Execution (n.3)
Cycles


Total Warp Execution (n.4)
Cycles


Total Warp Execution (n.5)
Total Warp Execution (n.6)
Total Warp Execution (n.7)
Total Warp Execution (n.8)
Total Warp Execution (n.9)
Cycles

B.2 Decode Unit

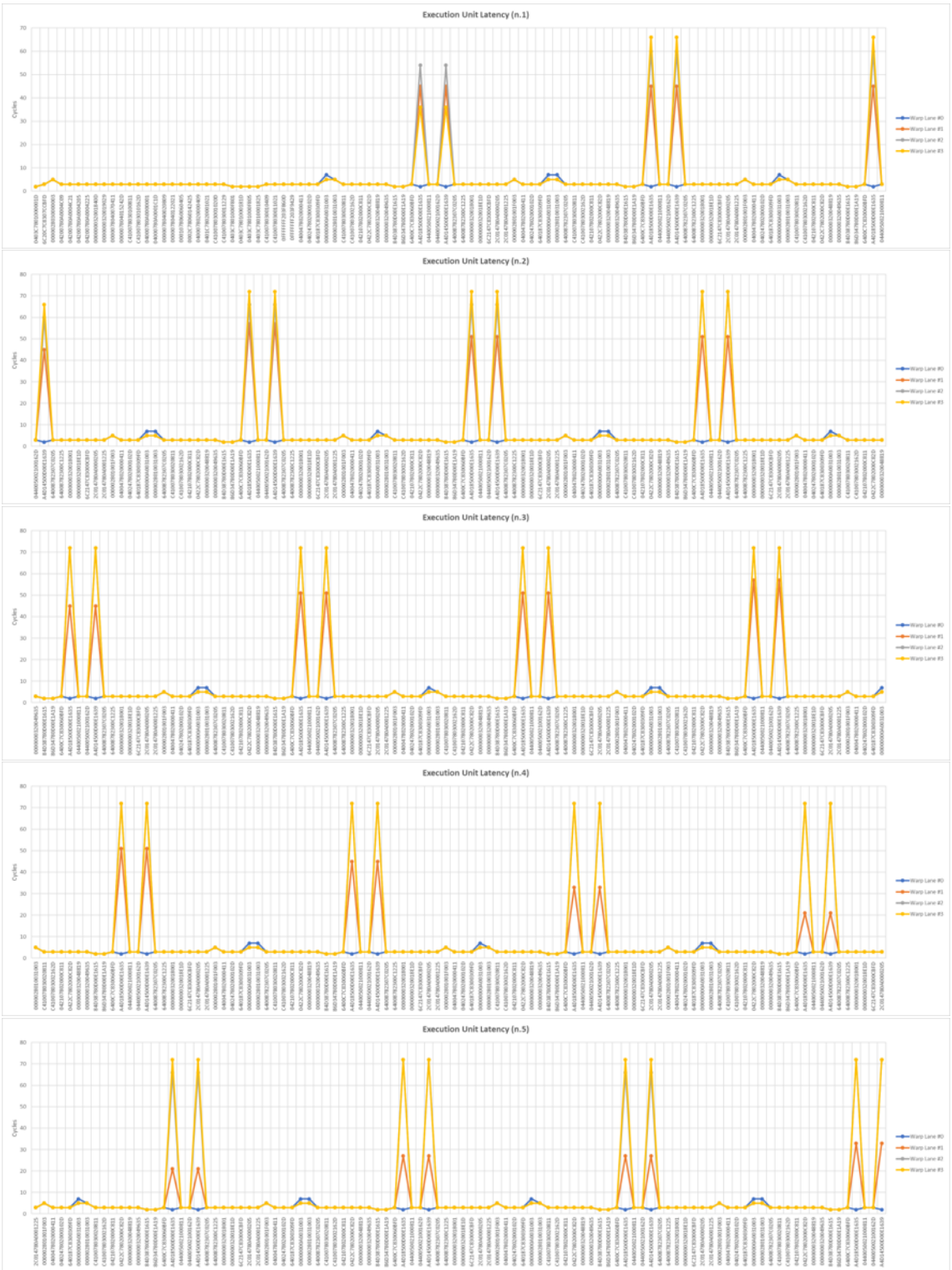

Decode Unit Latency (n.6)
Decode Unit Latency (n.7)
Decode Unit Latency (n.8)
Decode Unit Latency (n.9)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

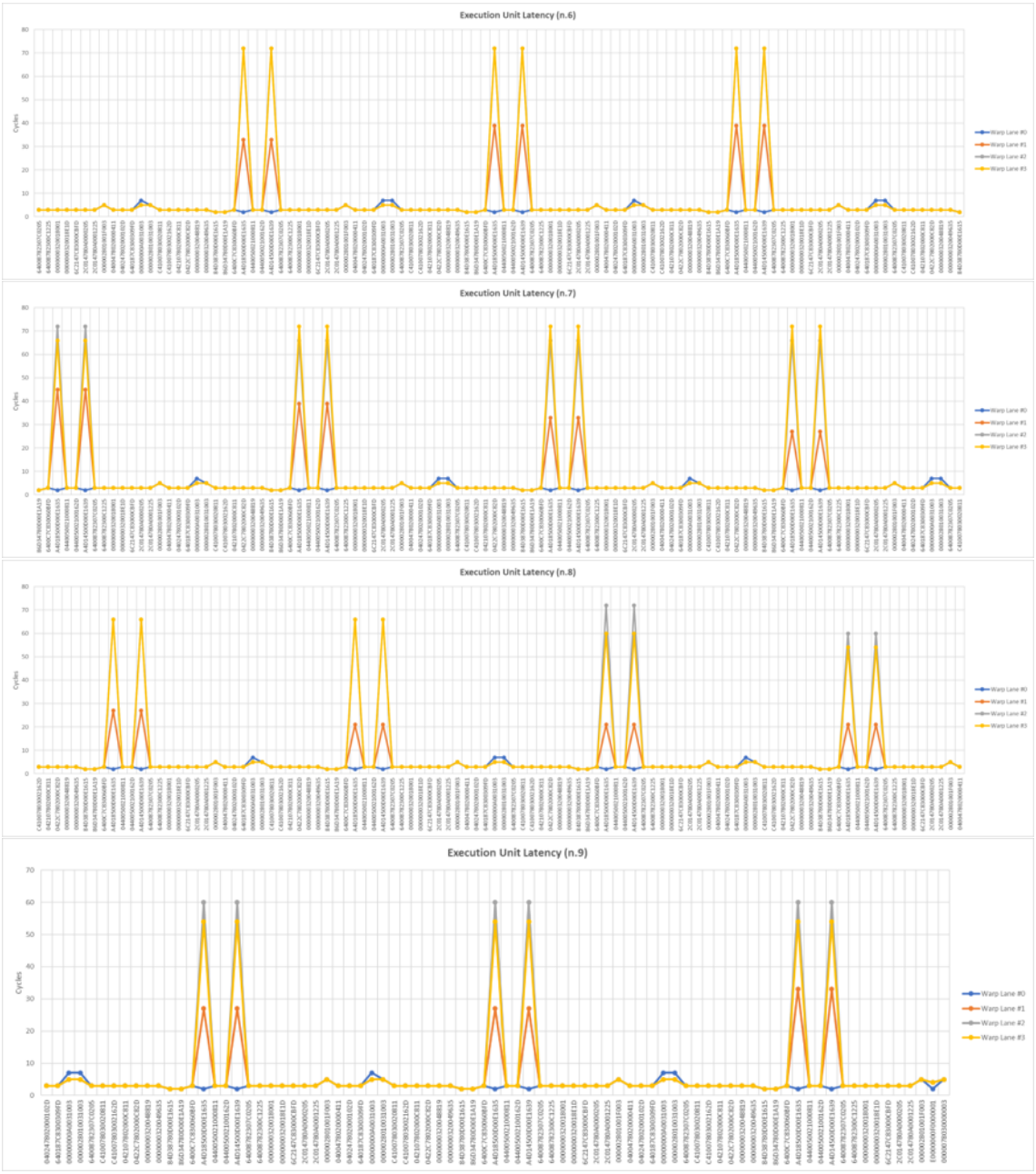
B.3 Read Unit


Read Unit Latency (n.1)
Read Unit Latency (n.2)
Read Unit Latency (n.3)
Read Unit Latency (n.4)
Read Unit Latency (n.5)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

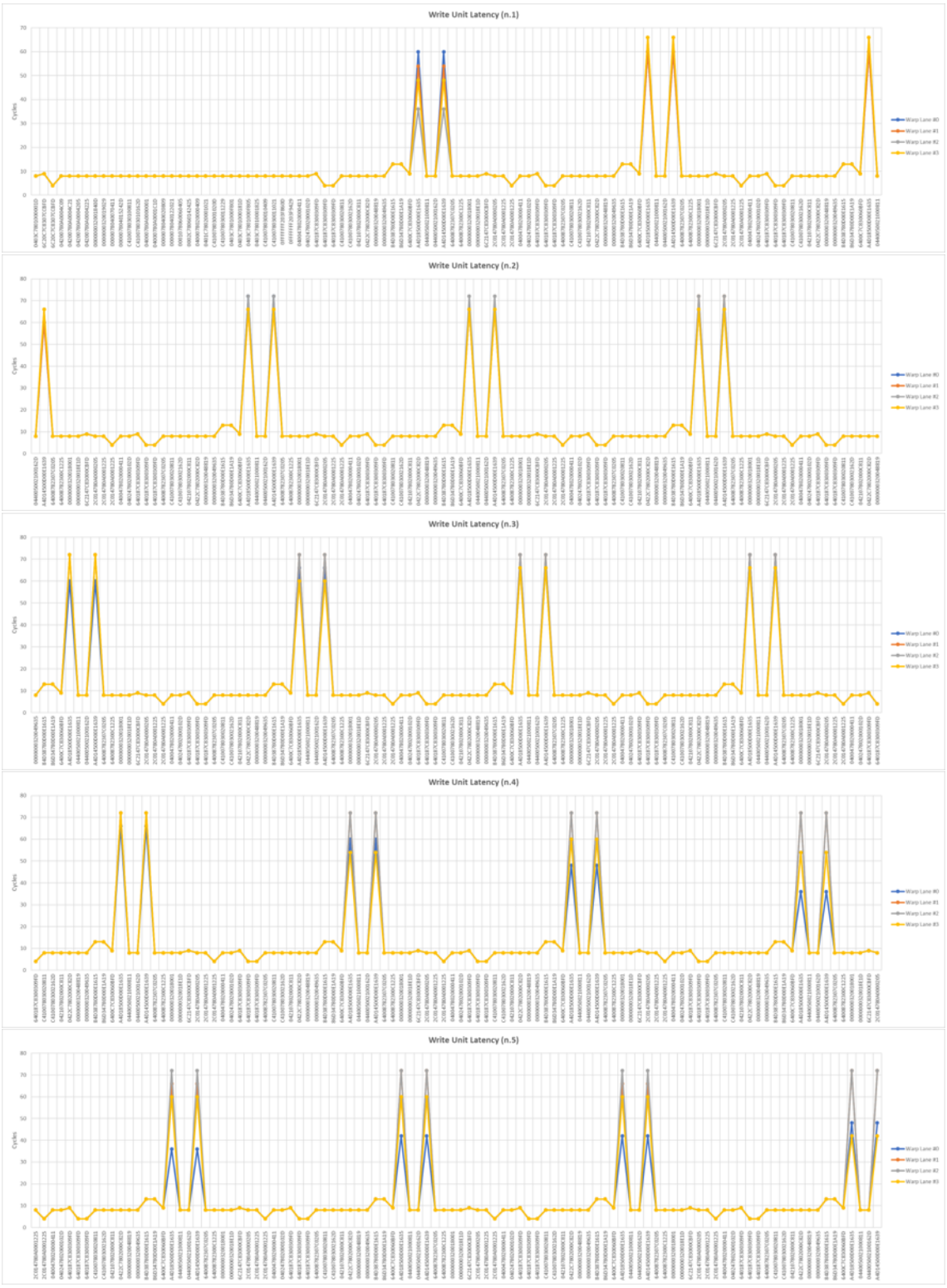

Read Unit Latency (n.6)
Read Unit Latency (n.7)
Read Unit Latency (n.8)
Read Unit Latency (n.9)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

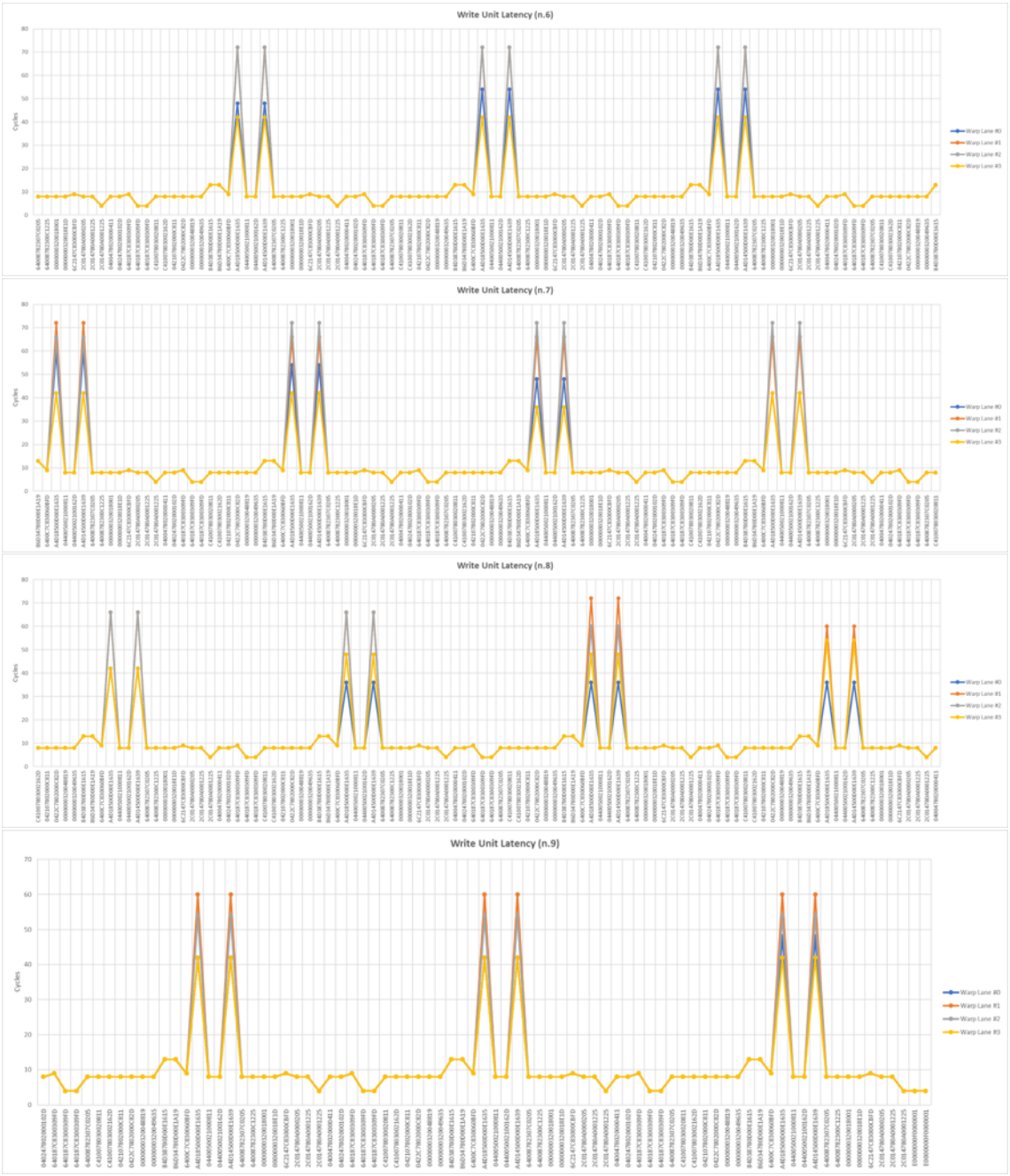
B.4 Execution Unit



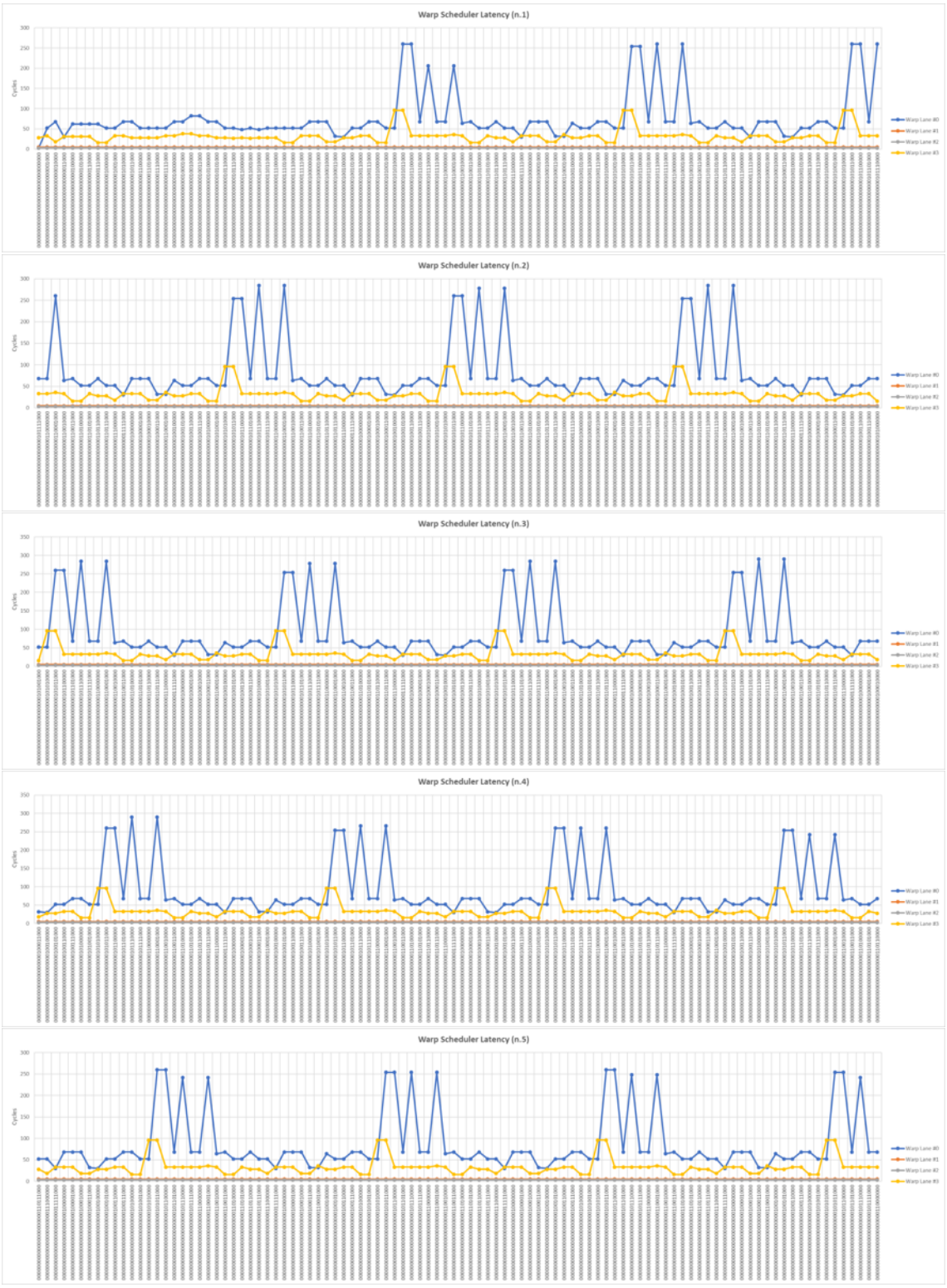

Execution Unit Latency (n.6)
Execution Unit Latency (n.7)
Execution Unit Latency (n.8)
Execution Unit Latency (n.9)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

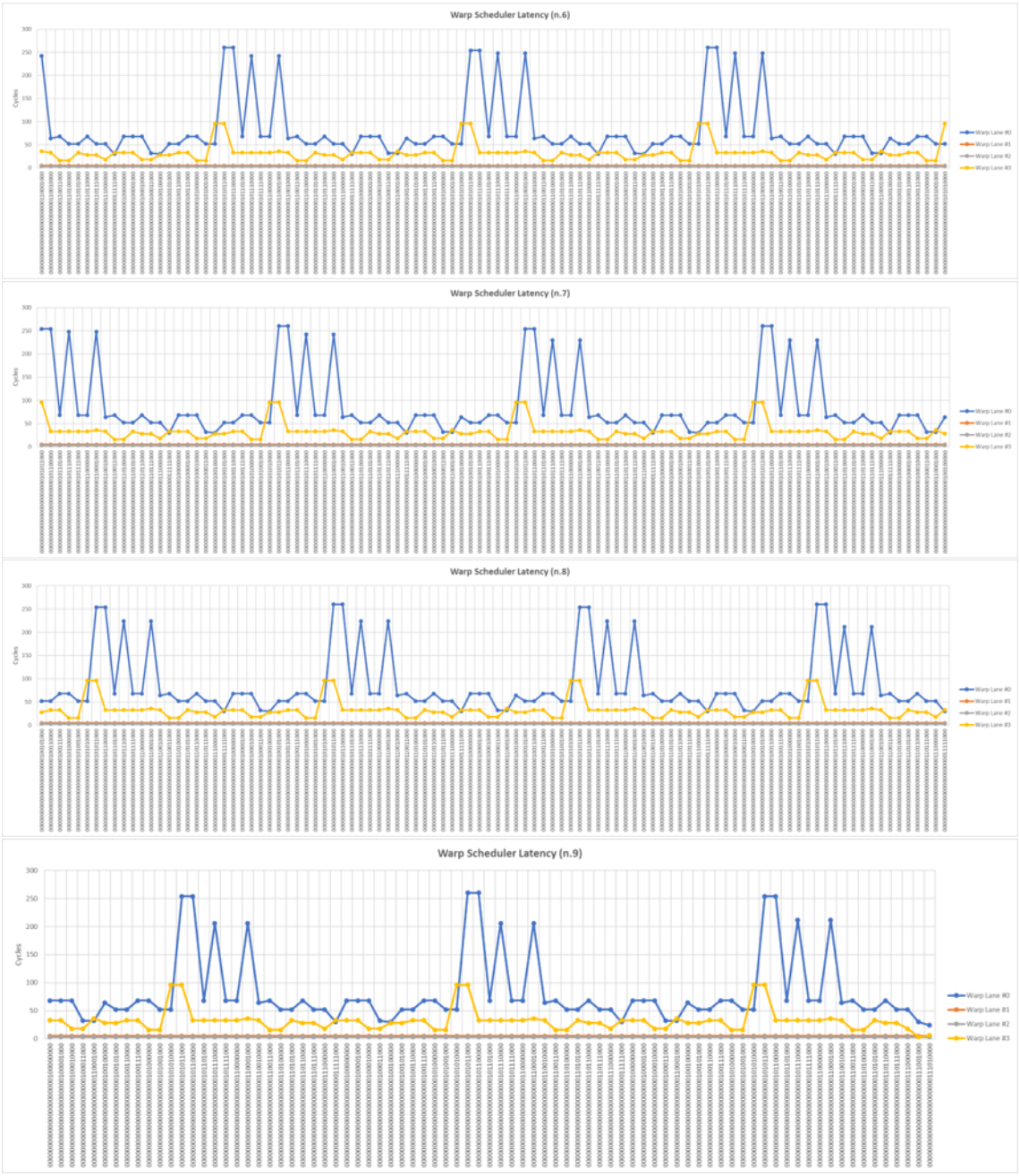
B.5 Write Unit




Write Unit Latency (n.6)
Write Unit Latency (n.7)
Write Unit Latency (n.8)
Write Unit Latency (n.9)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

B.6 Warp Scheduler





B.7 Warp Checker

Independently on the instruction and the warp lane, the warp checker always spends 2 clock cycles to complete the check operation.

B.8 Warp Generator

Time_interval	#Cycles
Total_warp_generation	7

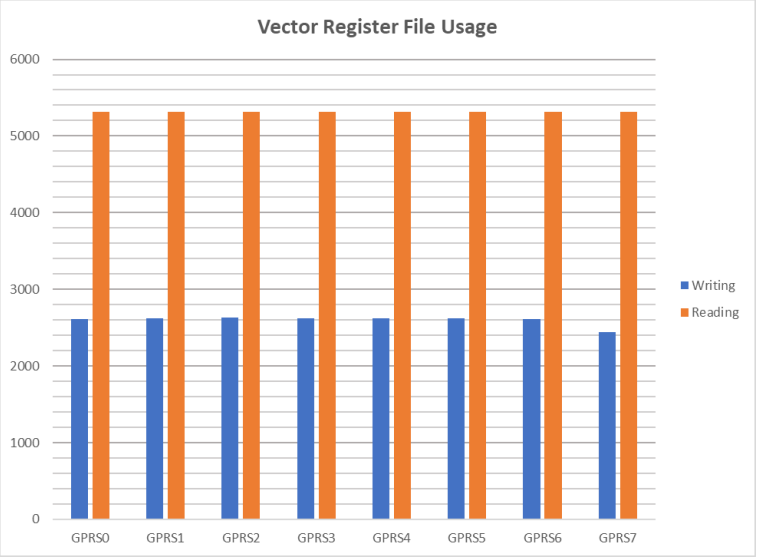
B.9 Streaming Multiprocessor Controller

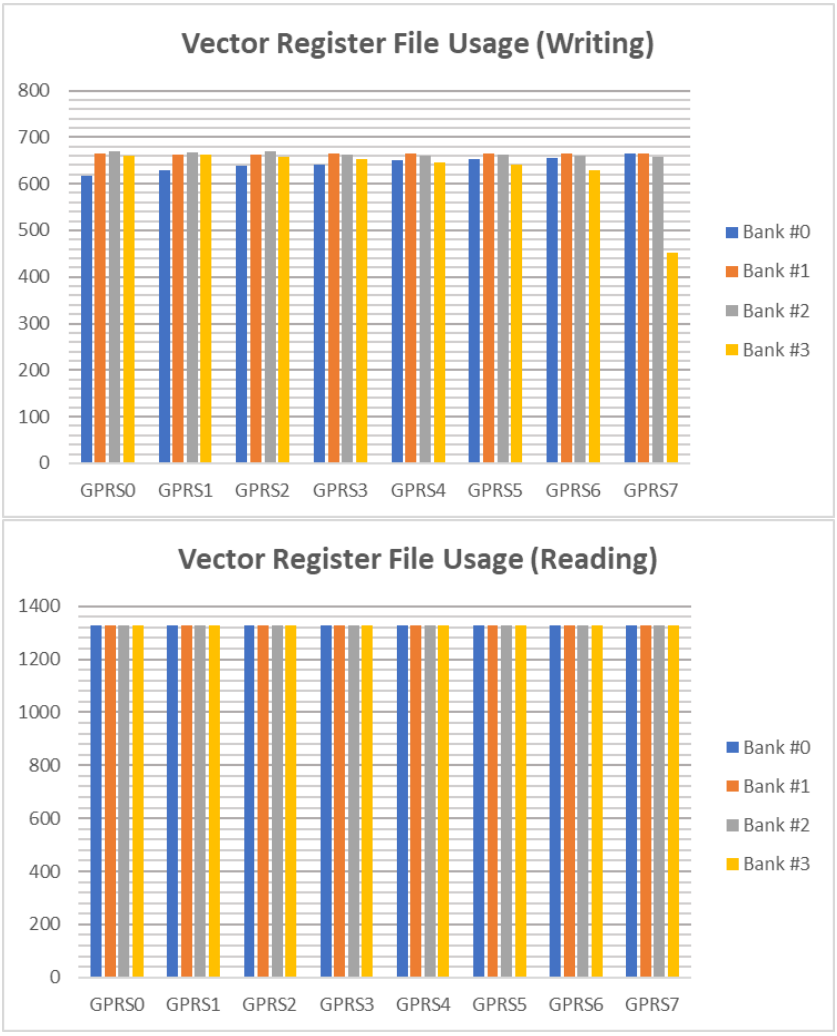
Time_interval	#Cycles
SM_task_execution	111332

B.10 Block Scheduler

Time_interval	#Cycles
Block_Scheduling	4

B.11 Vector Register File





B.12 Address Register File

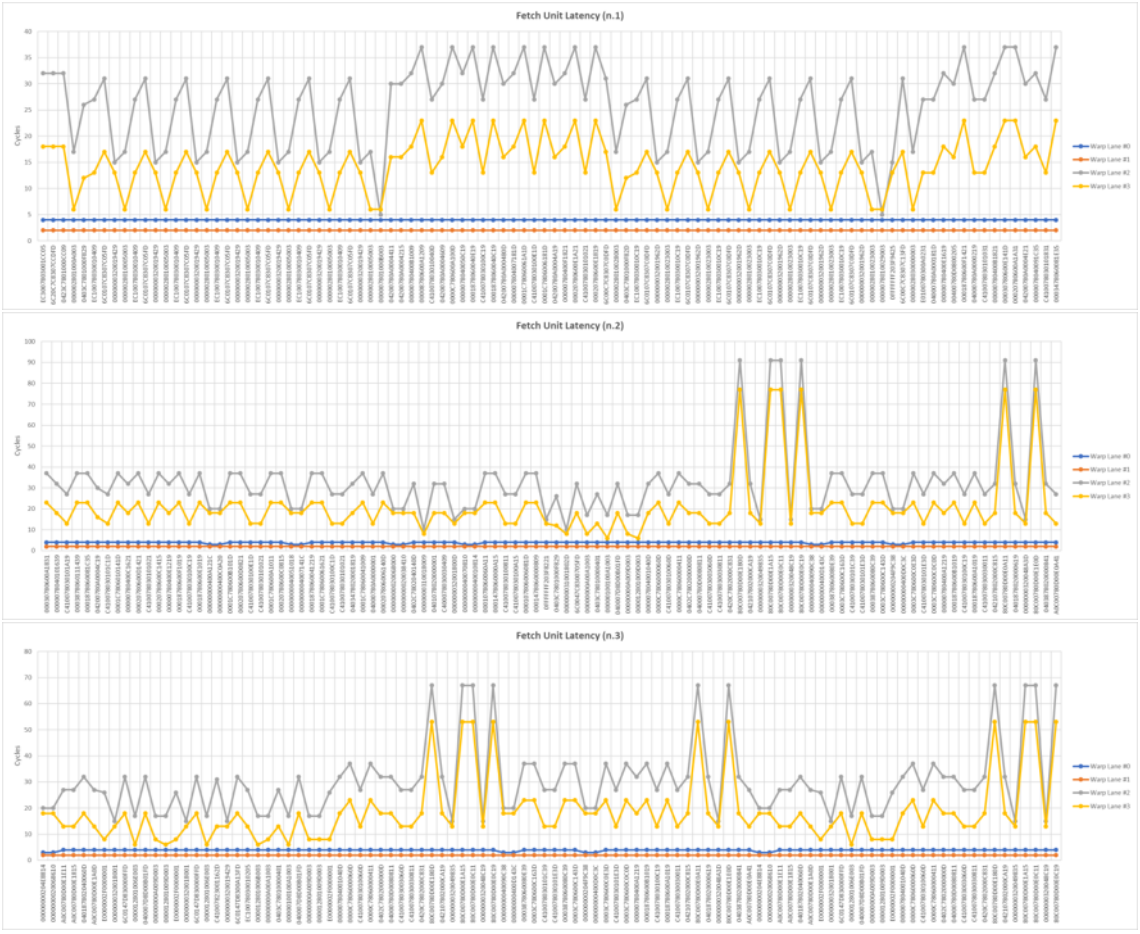
Module	# writings	# readings
ADDREG0	0	0
ADDREG1	0	0
ADDREG2	0	0
ADDREG3	0	0
ADDREG4	0	0
ADDREG5	0	0
ADDREG6	0	0
ADDREG7	0	0

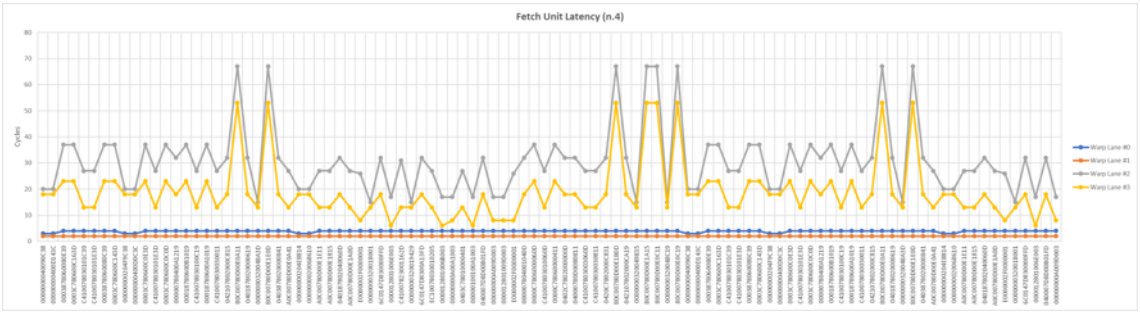
B.13 Predicate Register File

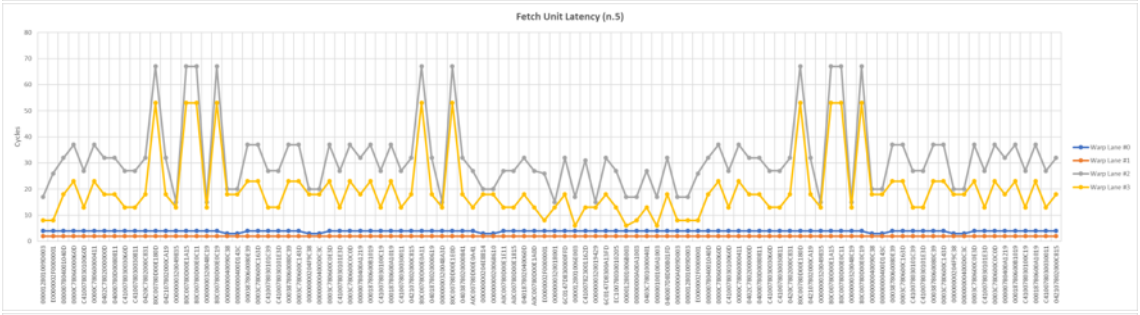
APPENDIX C

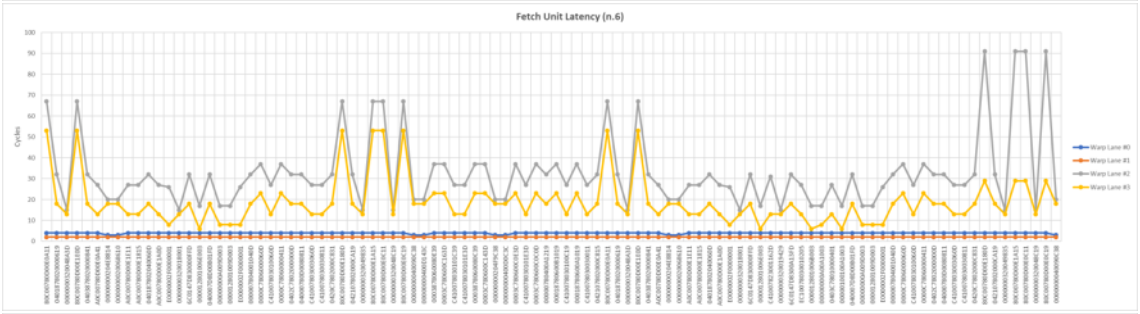
FFT

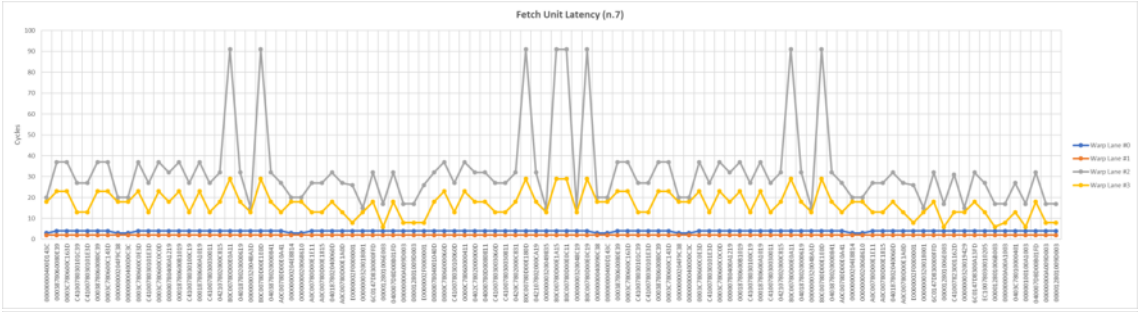
C.1 Fetch Unit

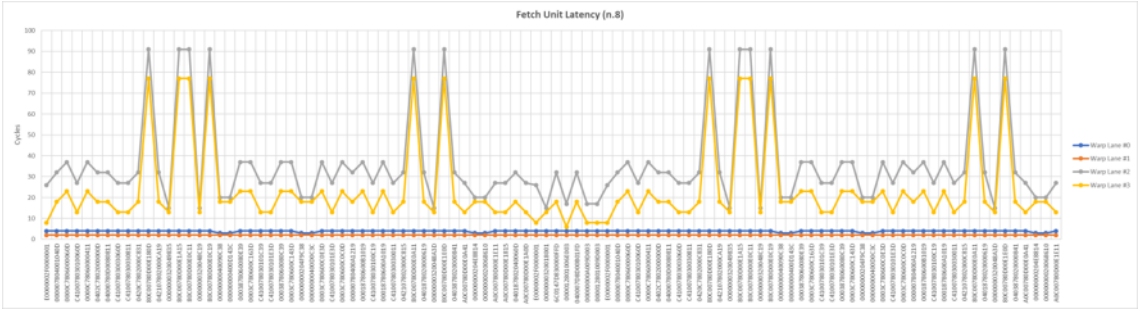


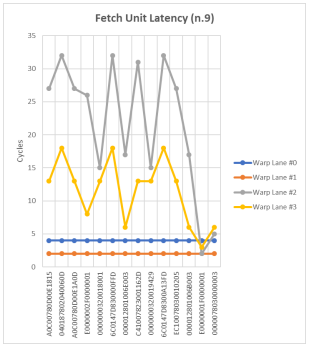

Fetch Unit Latency (n.4)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

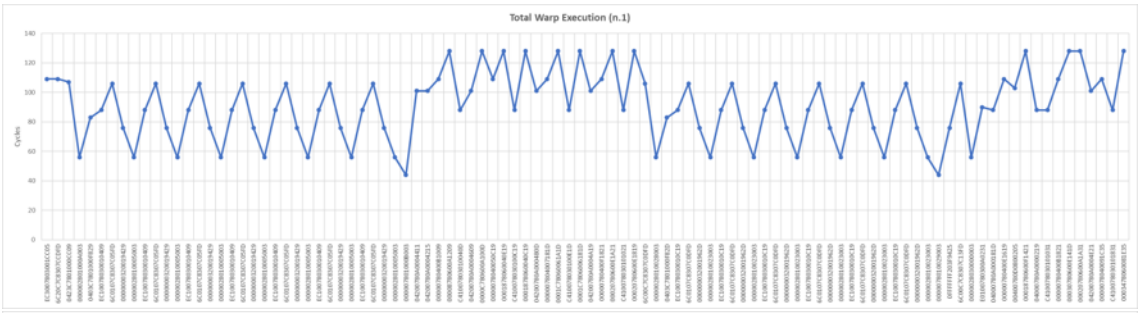

Fetch Unit Latency (n.5)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

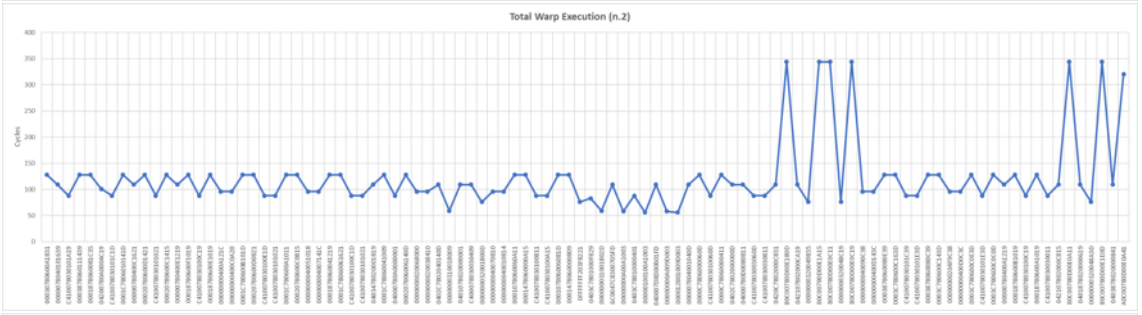

Fetch Unit Latency (n.6)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

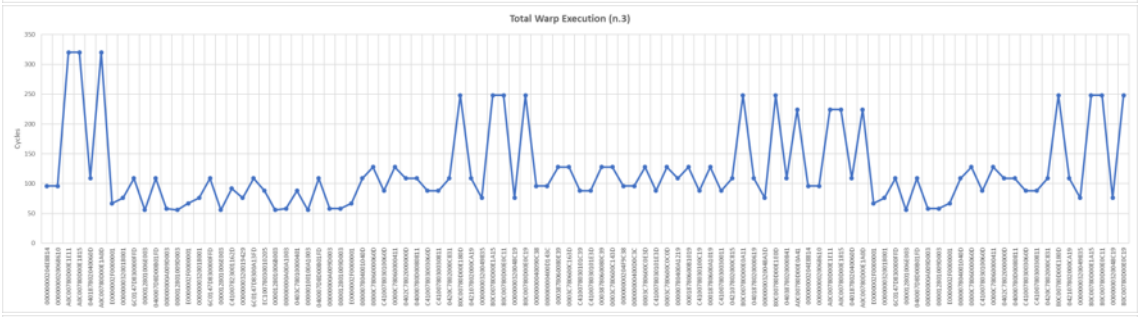

Fetch Unit Latency (n.7)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

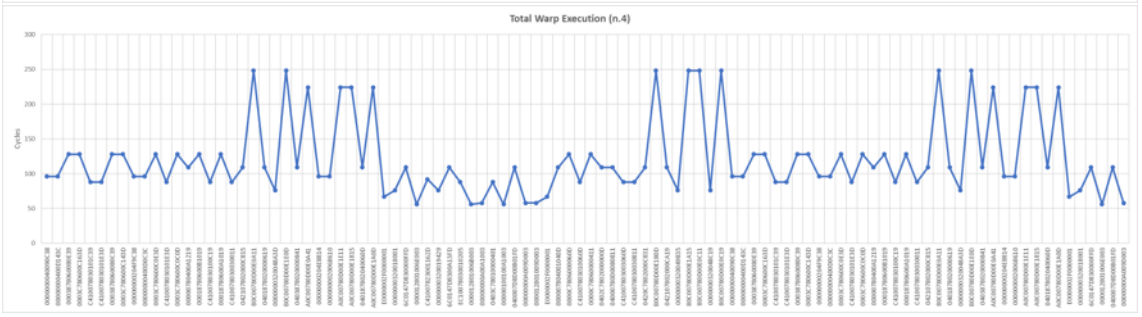

Fetch Unit Latency (n.8)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

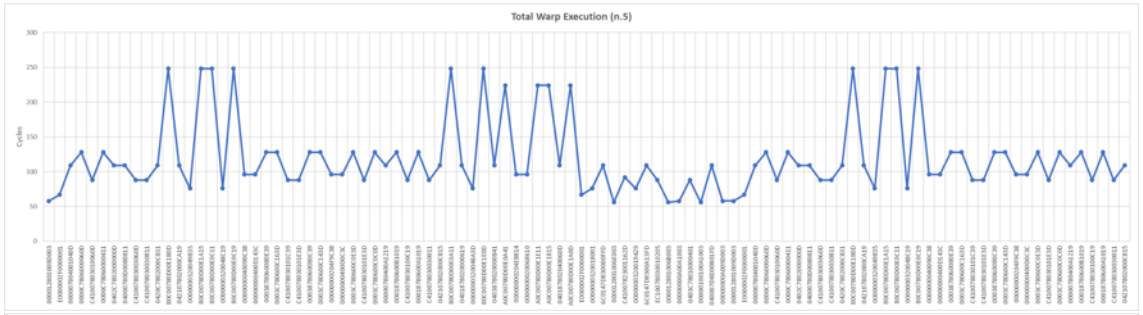

Fetch Unit Latency (n.9)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

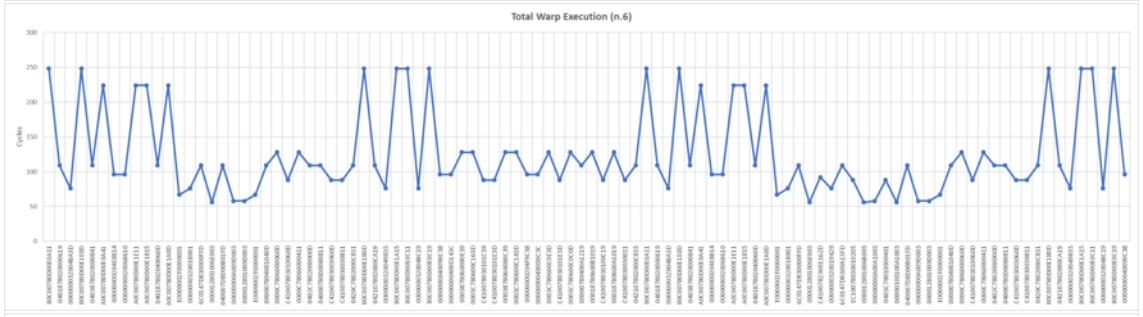

Total Warp Execution (n.1)
Cycles

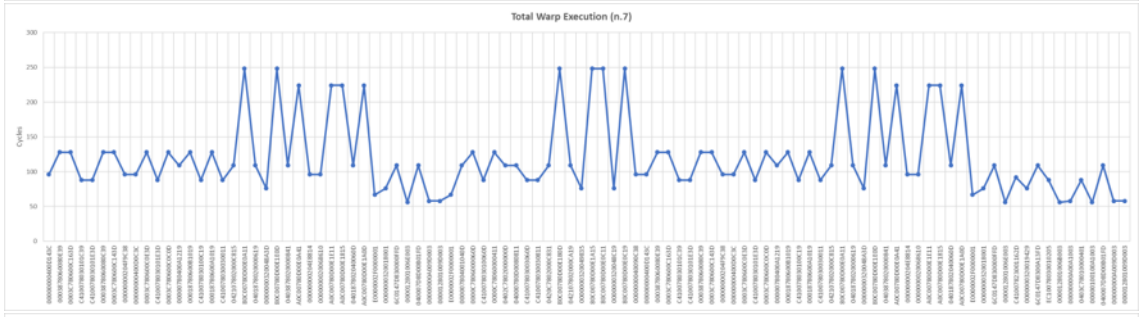

Total Warp Execution (n.2)
Cycles

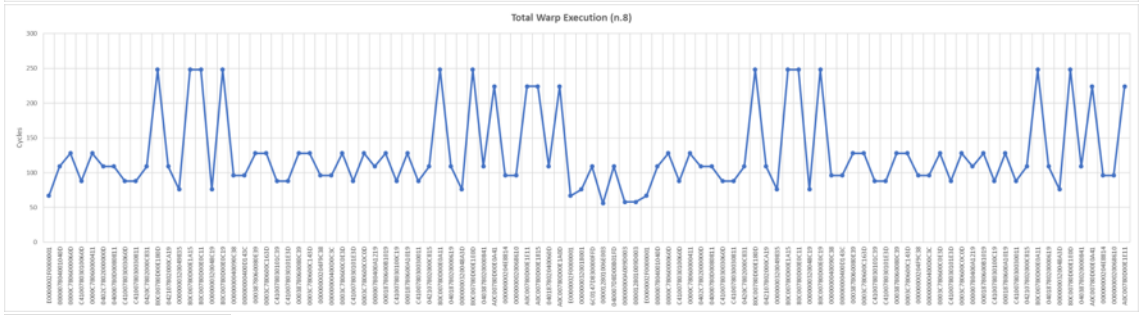

Total Warp Execution (n.3)
Cycles

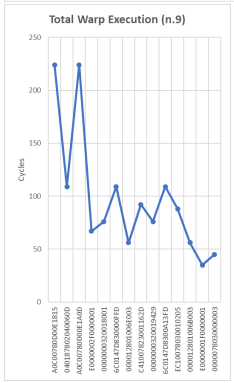

Total Warp Execution (n.4)
Cycles


Total Warp Execution (n.5)
Cycles

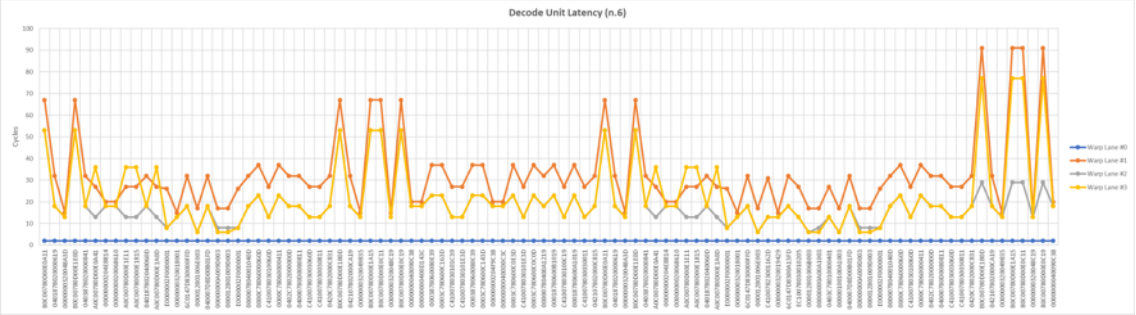

Total Warp Execution (n.6)
Cycles

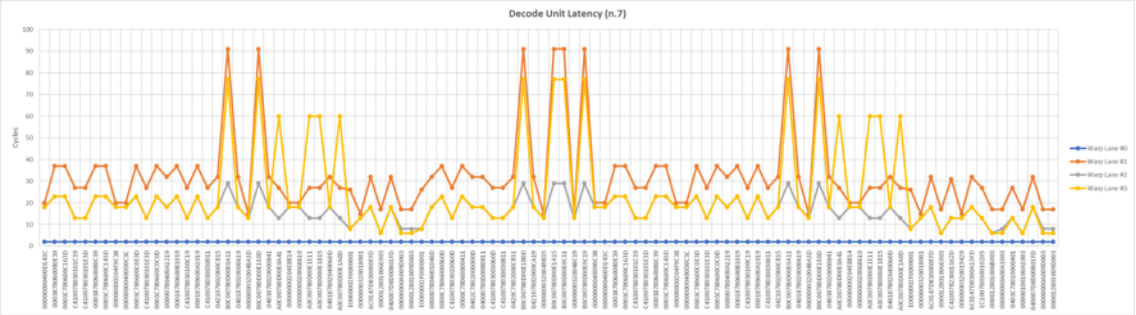

Total Warp Execution (n.7)
Cycles

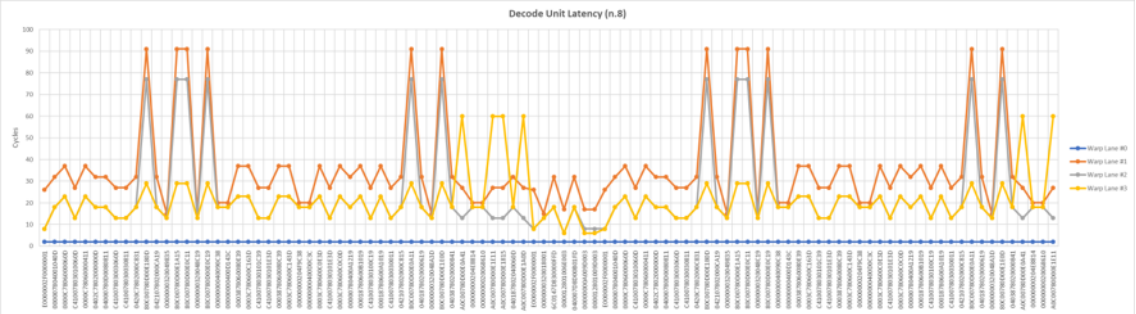

Total Warp Execution (n.8)
Cycles

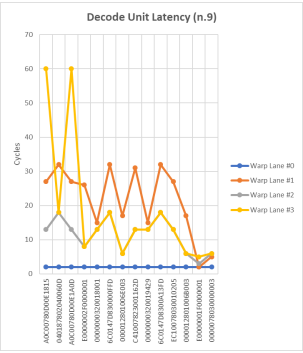

Total Warp Execution (n.9)
Cycles

C.2 Decode Unit


Decode Unit Latency (n.6)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

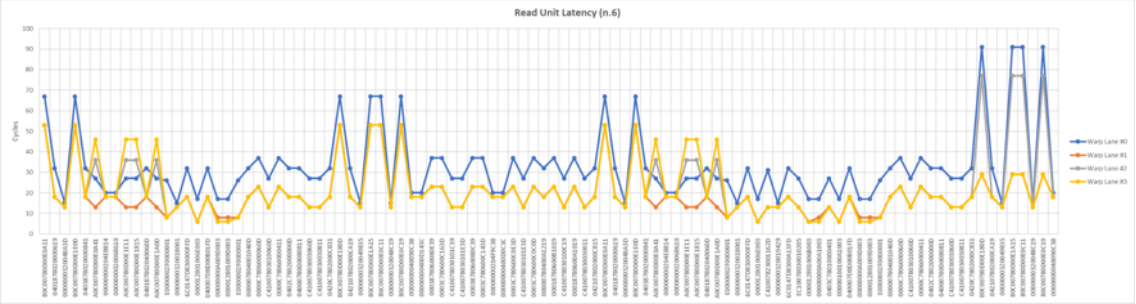

Decode Unit Latency (n.7)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

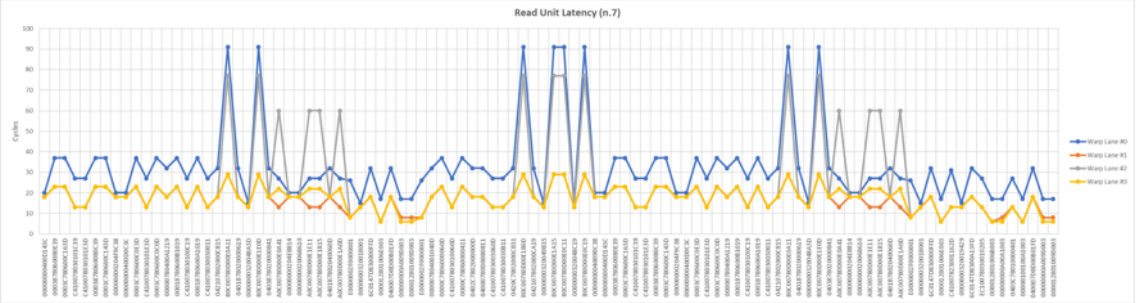

Decode Unit Latency (n.8)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

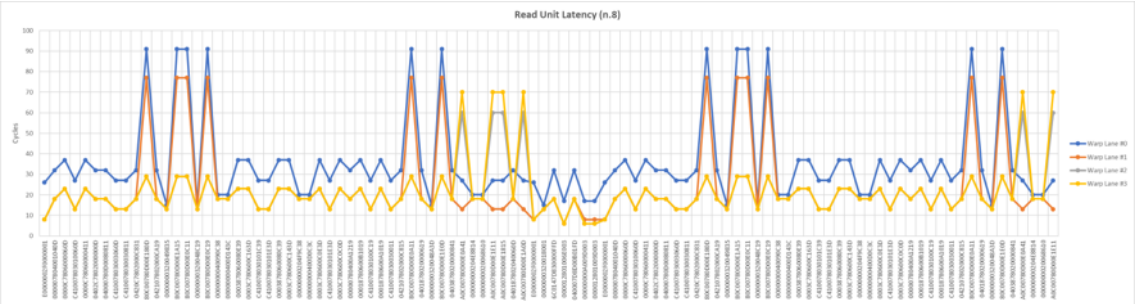

Decode Unit Latency (n.9)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

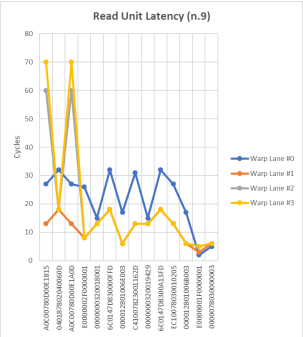
C.3 Read Unit


Read Unit Latency (n.1)
Read Unit Latency (n.2)
Read Unit Latency (n.3)
Read Unit Latency (n.4)
Read Unit Latency (n.5)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

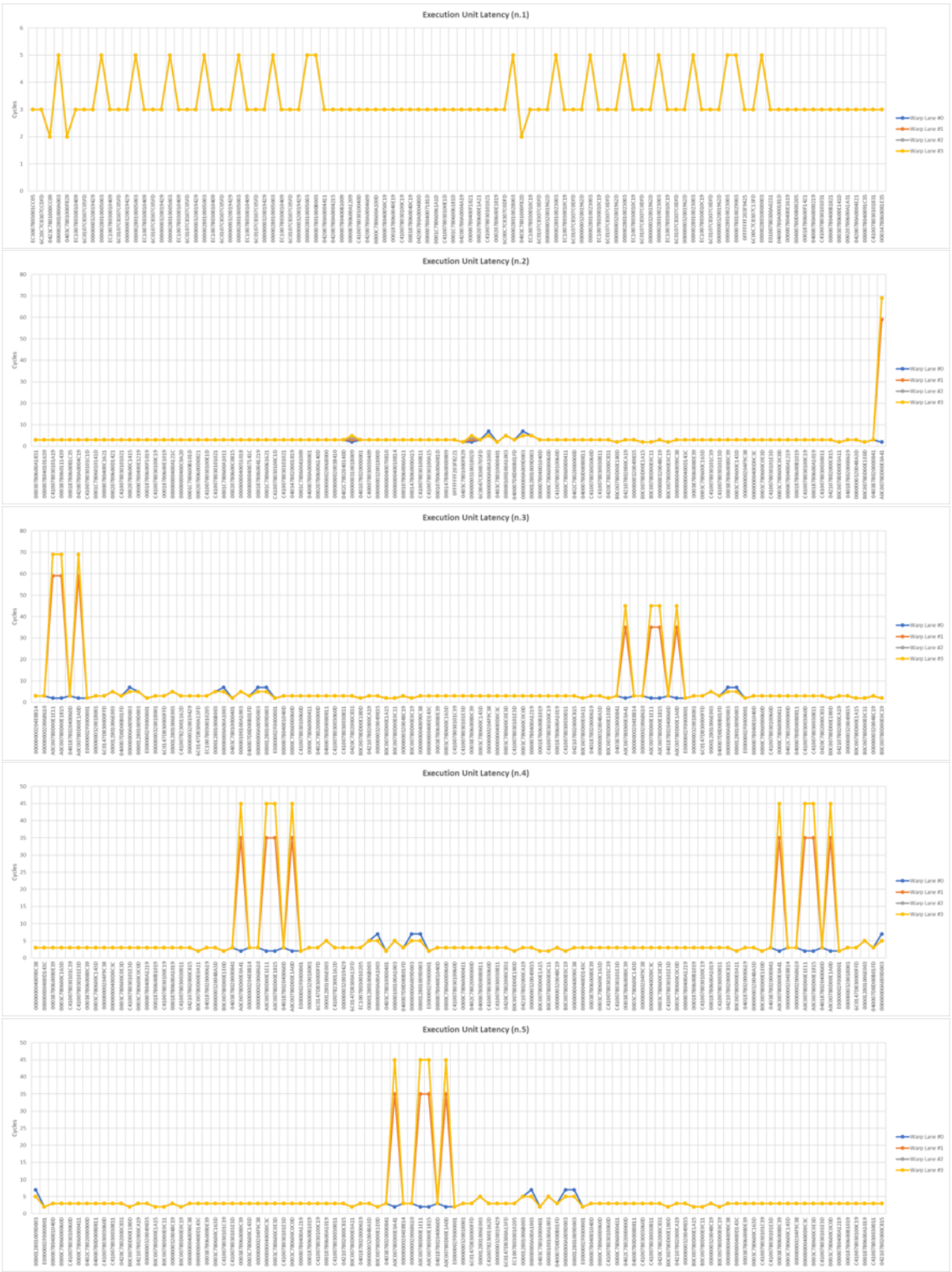

Read Unit Latency (n.6)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

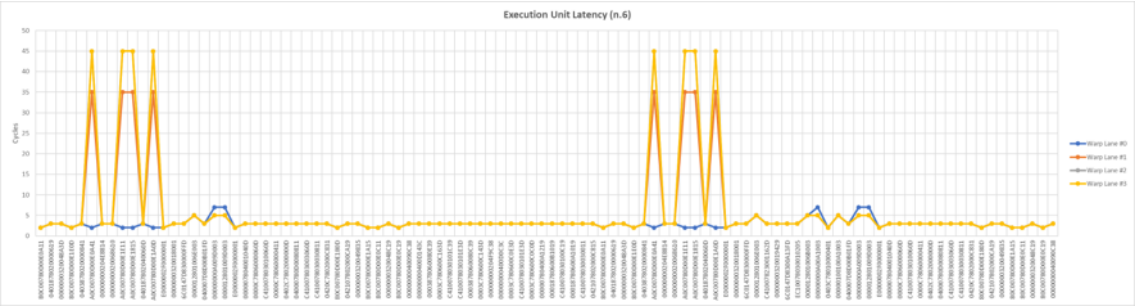

Read Unit Latency (n.7)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

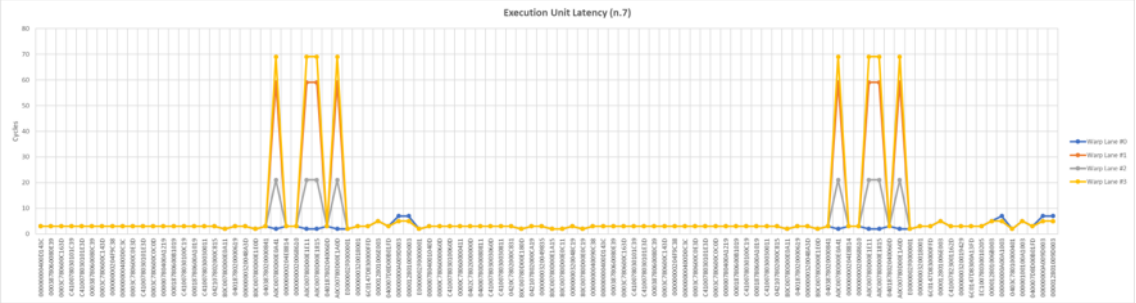

Read Unit Latency (n.8)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

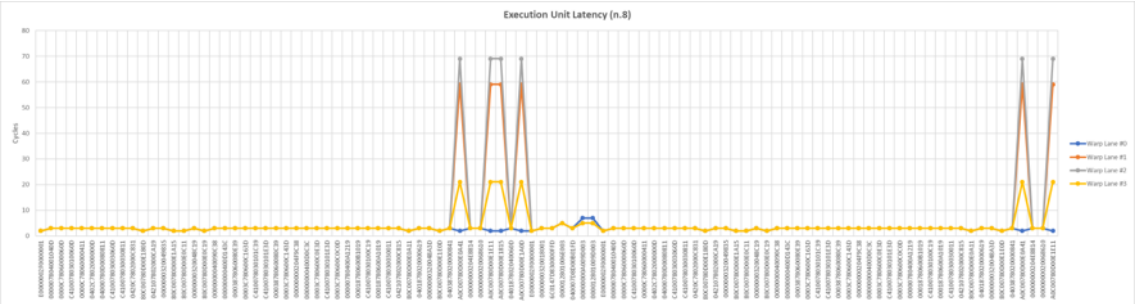

Read Unit Latency (n.9)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

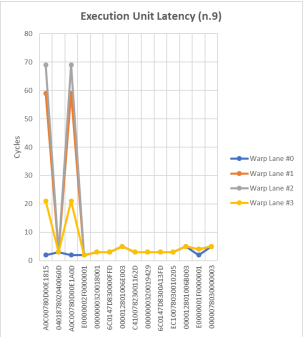
C.4 Execution Unit



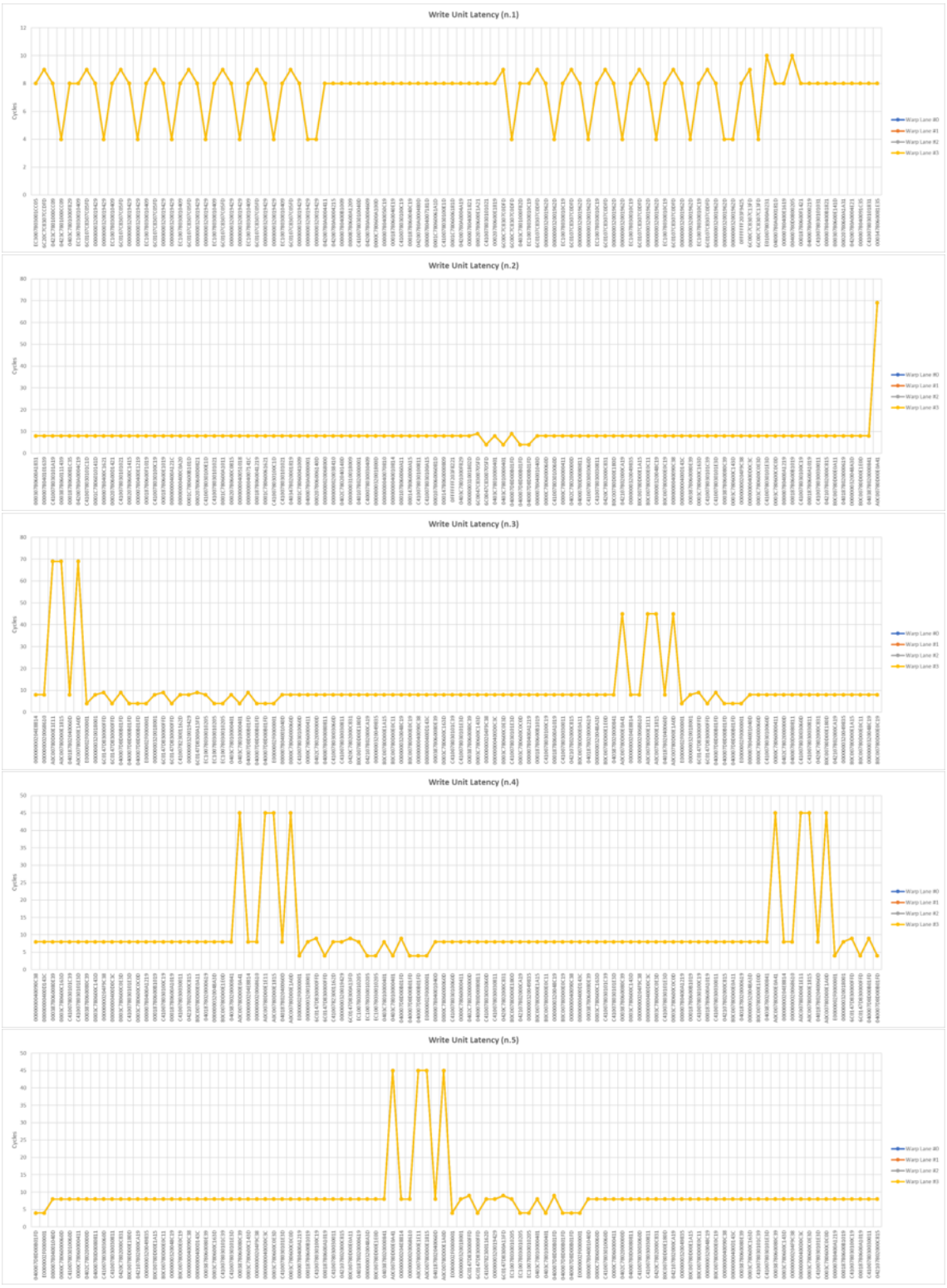

Execution Unit Latency (n.6)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

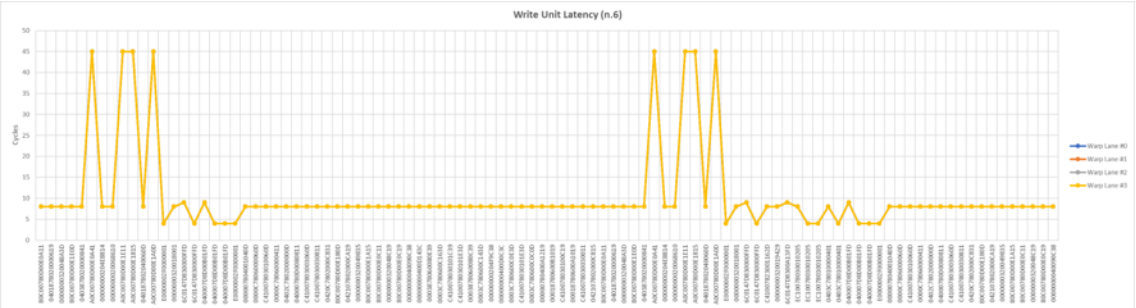

Execution Unit Latency (n.7)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

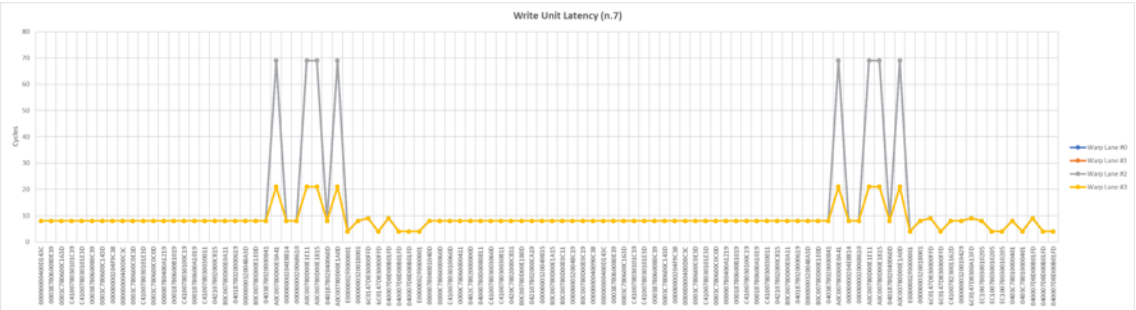

Execution Unit Latency (n.8)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

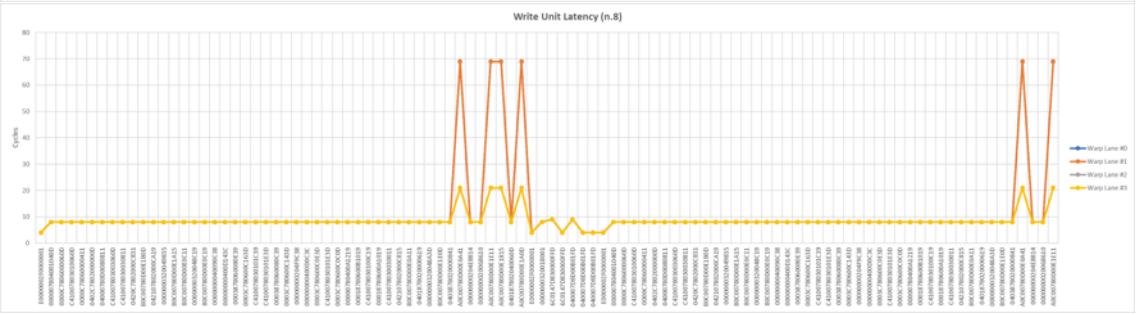

Execution Unit Latency (n.9)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

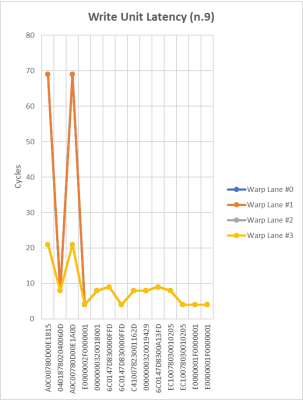
C.5 Write Unit



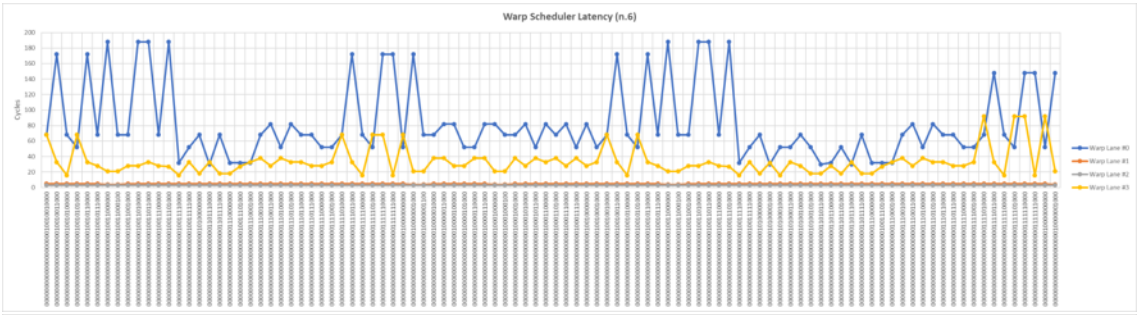

Write Unit Latency (n.6)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

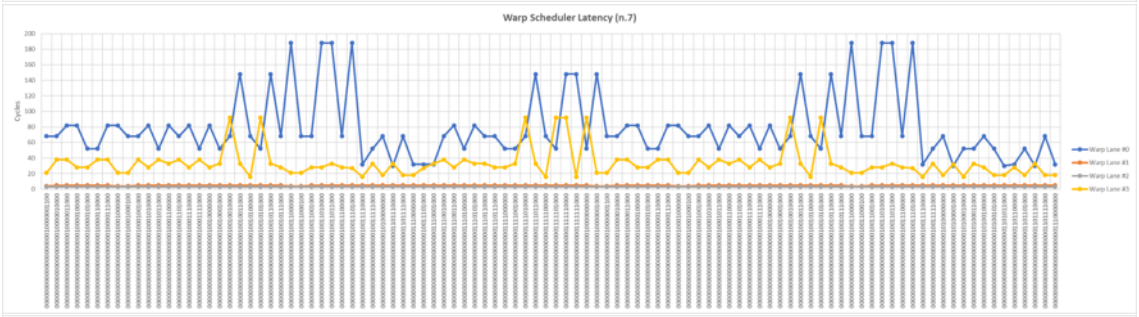

Write Unit Latency (n.7)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

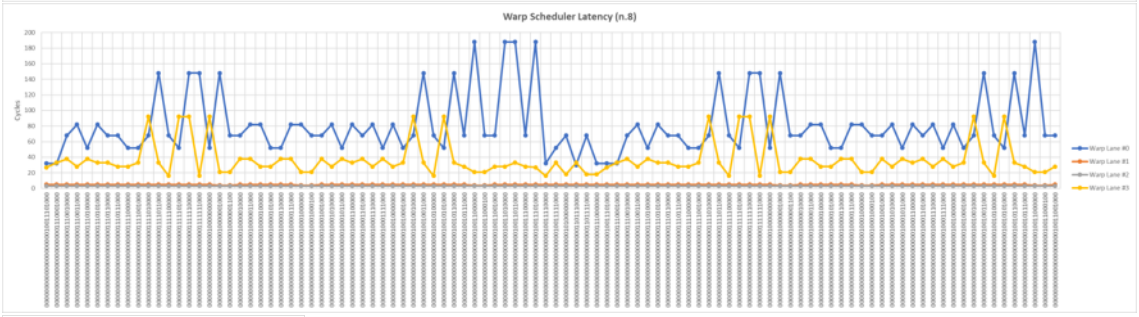

Write Unit Latency (n.8)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

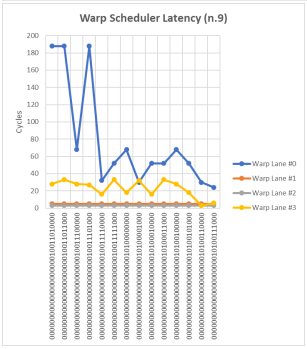

Write Unit Latency (n.9)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

C.6 Warp Scheduler









C.7 Warp Checker

Independently on the instruction and the warp lane, the warp checker always spends 2 clock cycles to complete the check operation.

C.8 Warp Generator

Time_interval	#Cycles
Total_warp_generation	7

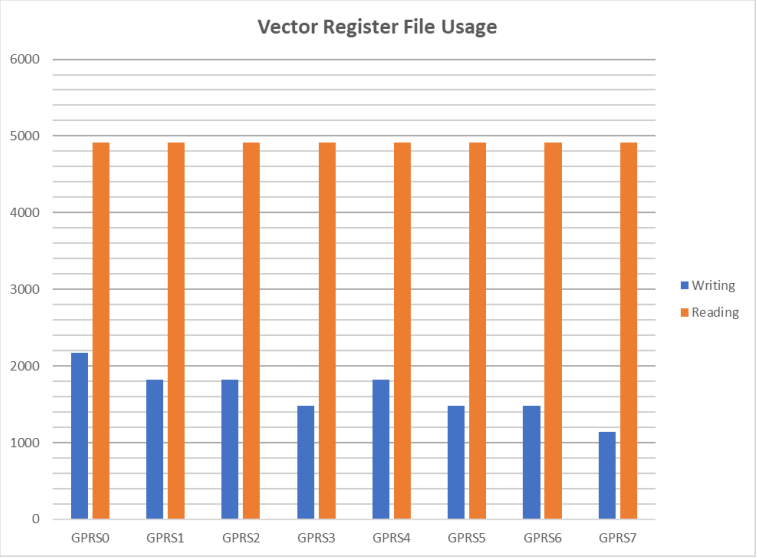
C.9 Streaming Multiprocessor Controller

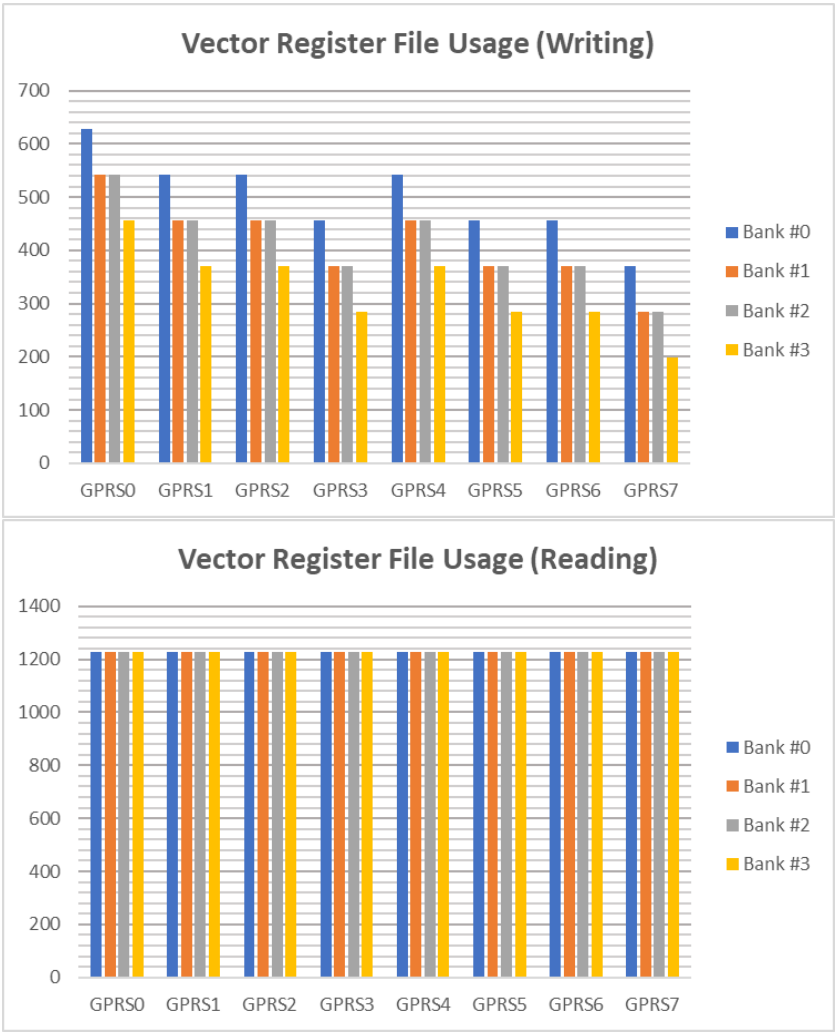
Time_interval	#Cycles
SM_task_execution	95578

C.10 Block Scheduler

Time_interval	#Cycles
Block_Scheduling	4

C.11 Vector Register File

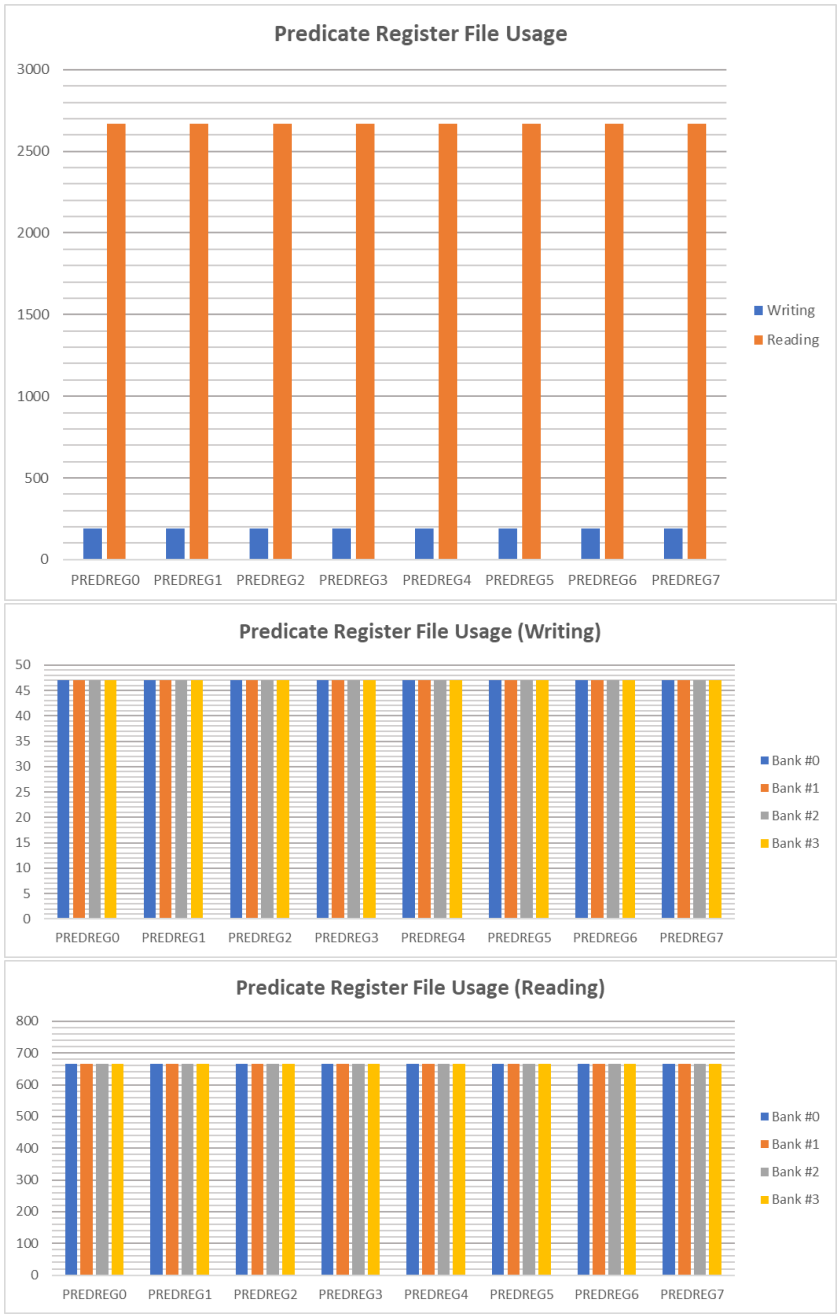




C.12 Address Register File

Module	# writings	# readings
ADDREG0	0	0
ADDREG1	0	0
ADDREG2	0	0
ADDREG3	0	0
ADDREG4	0	0
ADDREG5	0	0
ADDREG6	0	0
ADDREG7	0	0

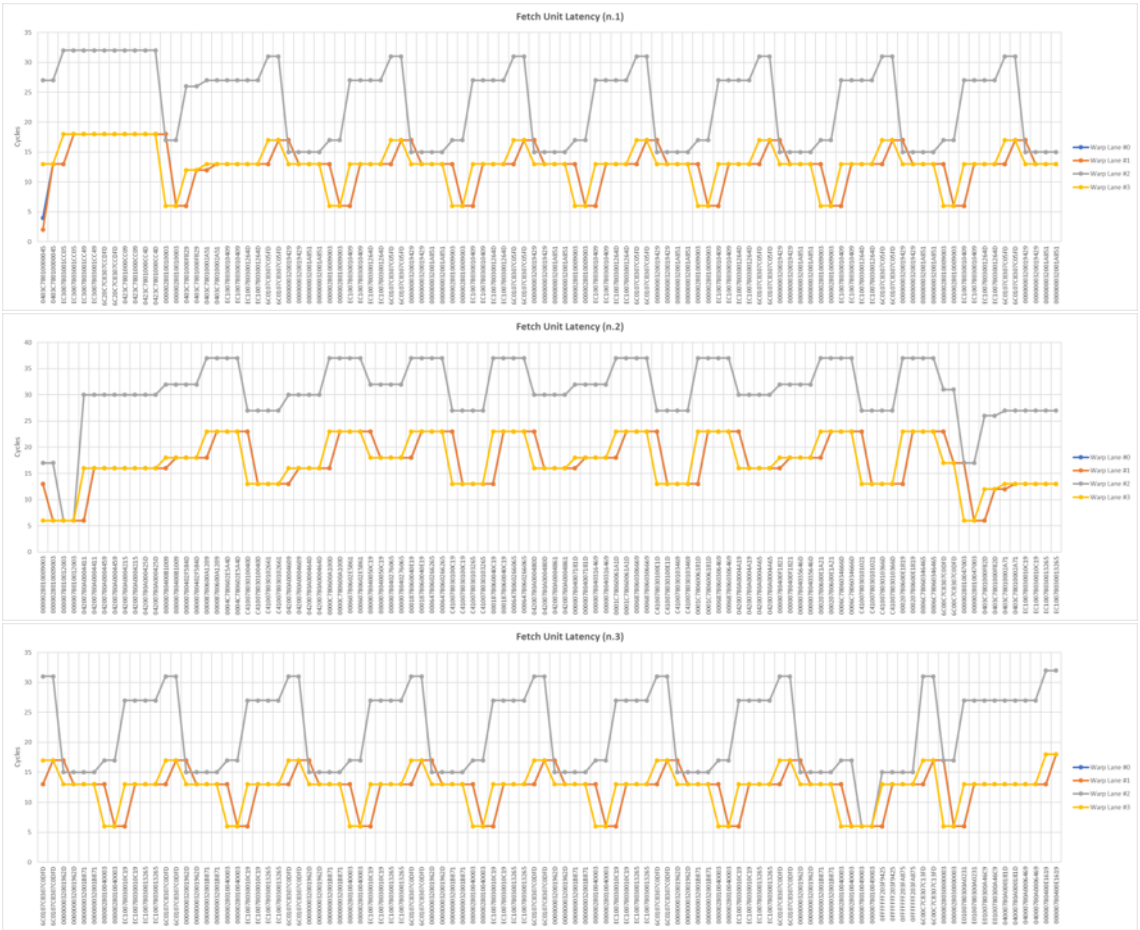
C.13 Predicate Register File

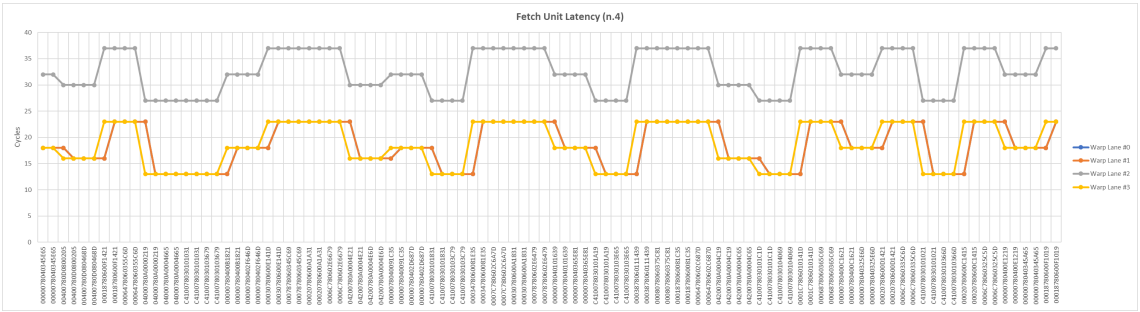


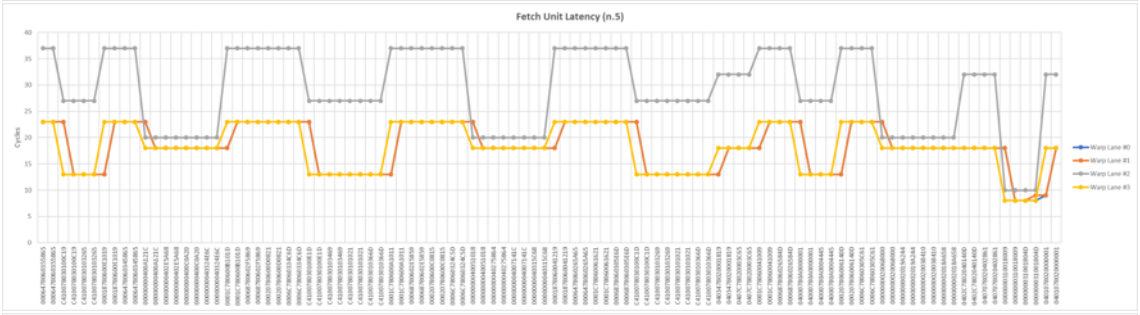
APPENDIX D

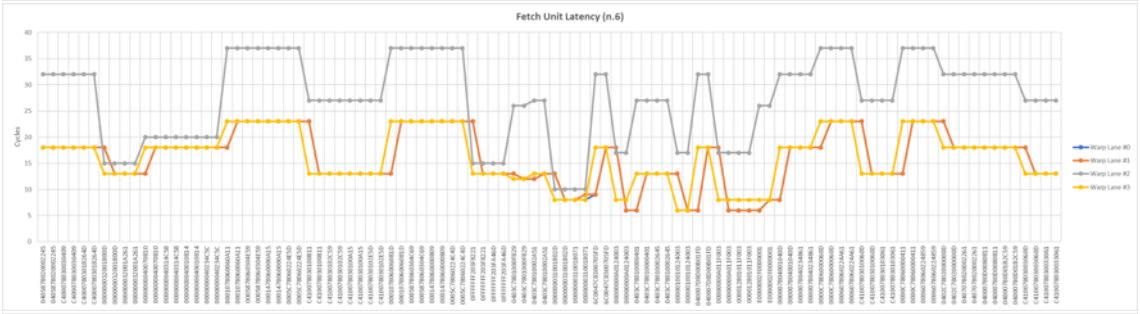
FFT with Resilient LOAD/STORE Instruction

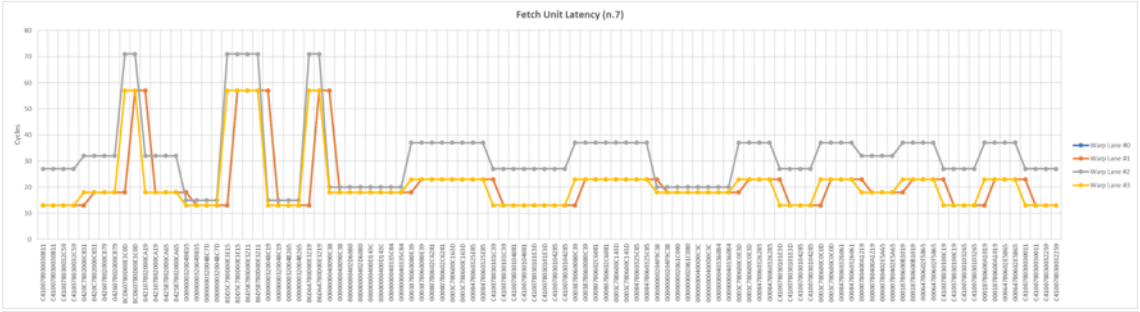
D.1 Fetch Unit

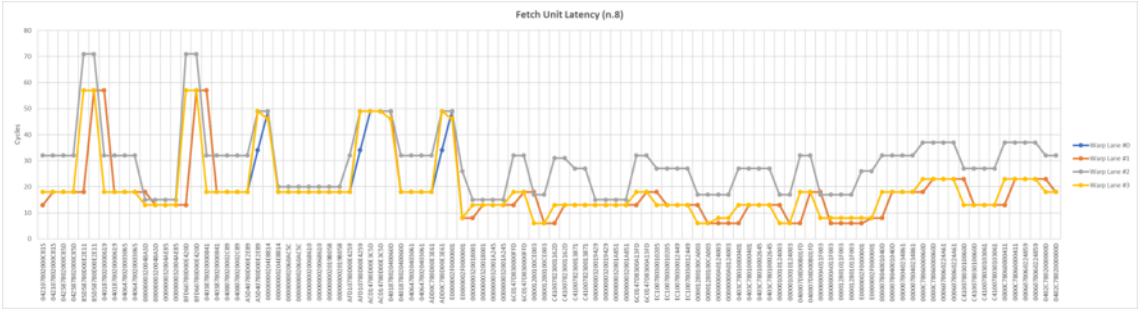


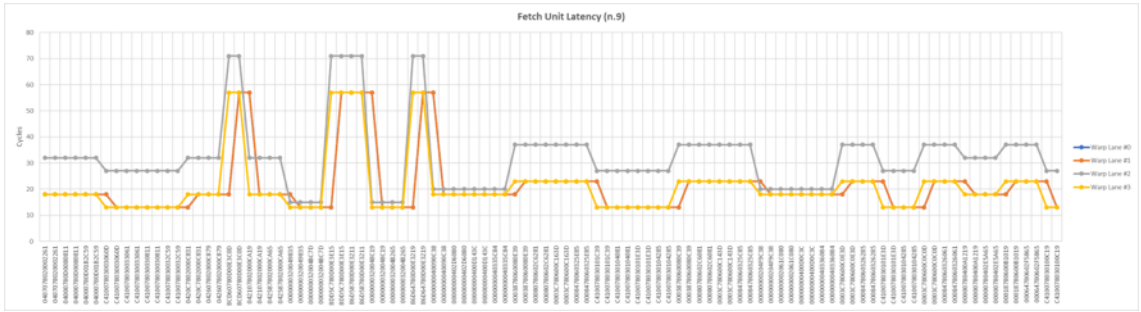

Fetch Unit Latency (n.4)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

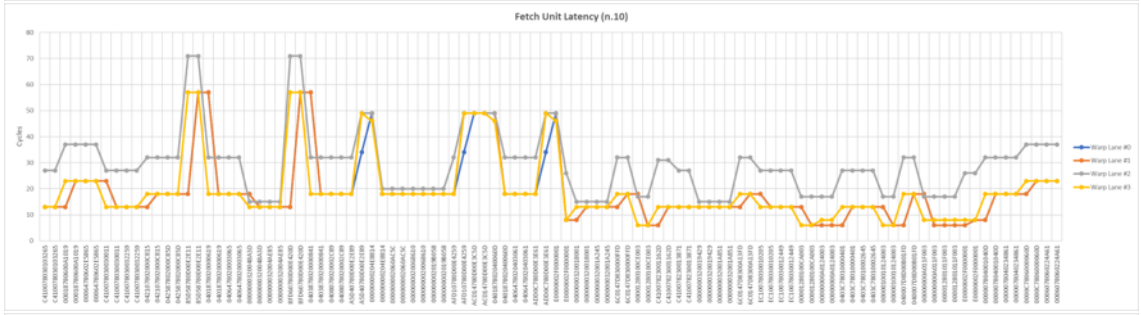

Fetch Unit Latency (n.5)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

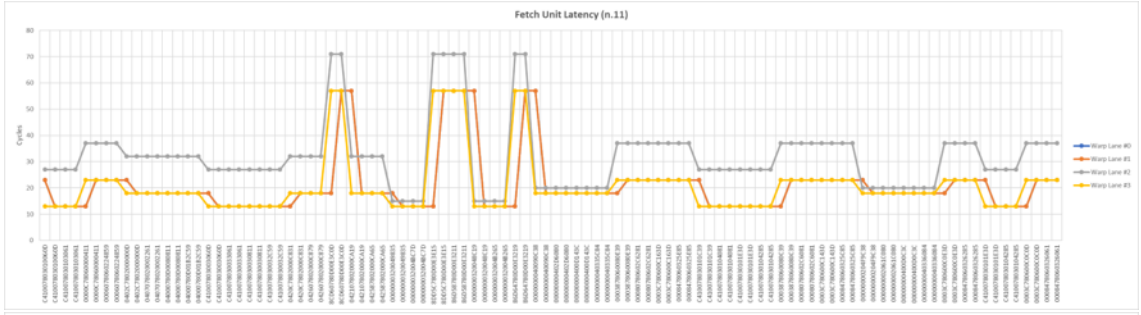

Fetch Unit Latency (n.6)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

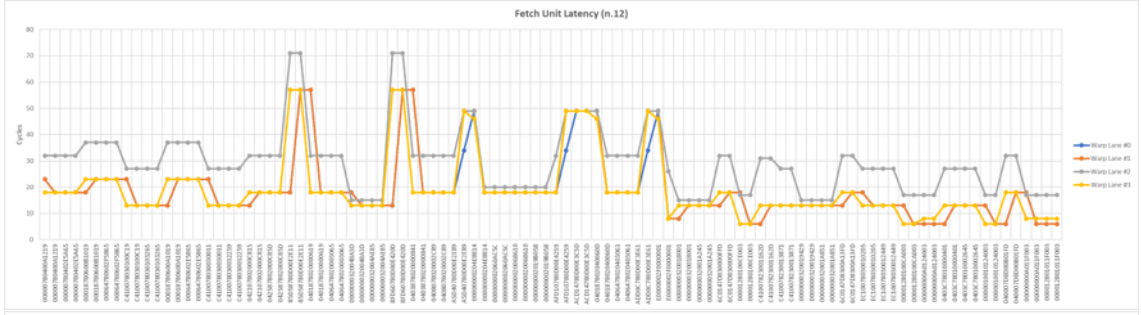

Fetch Unit Latency (n.7)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

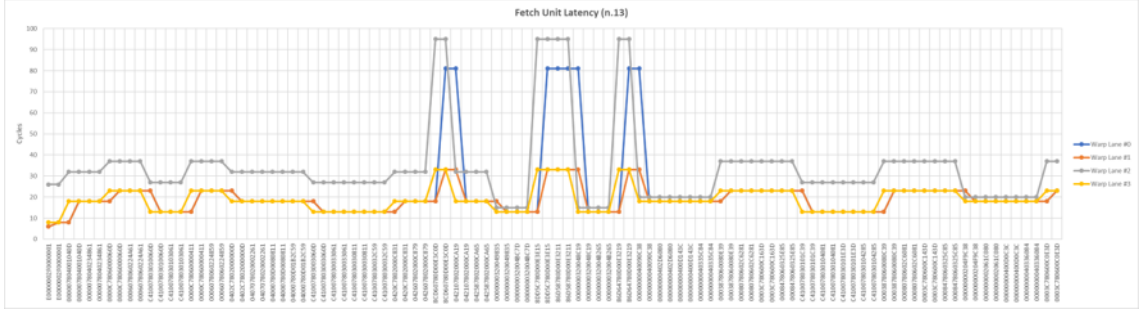

Fetch Unit Latency (n.8)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

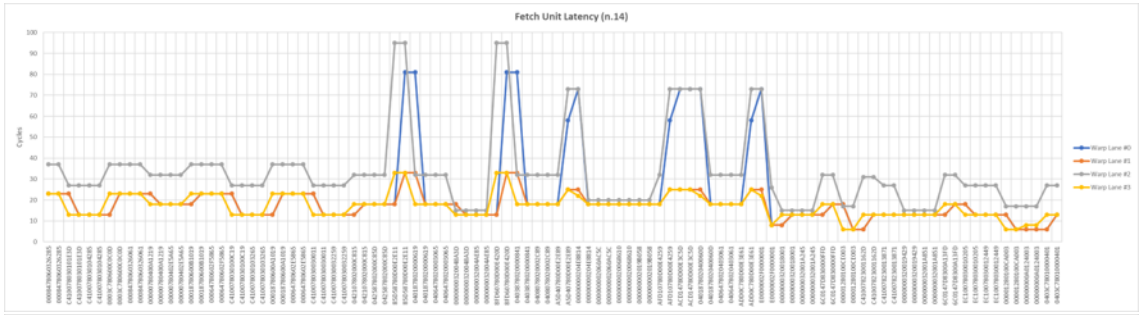

Fetch Unit Latency (n.9)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

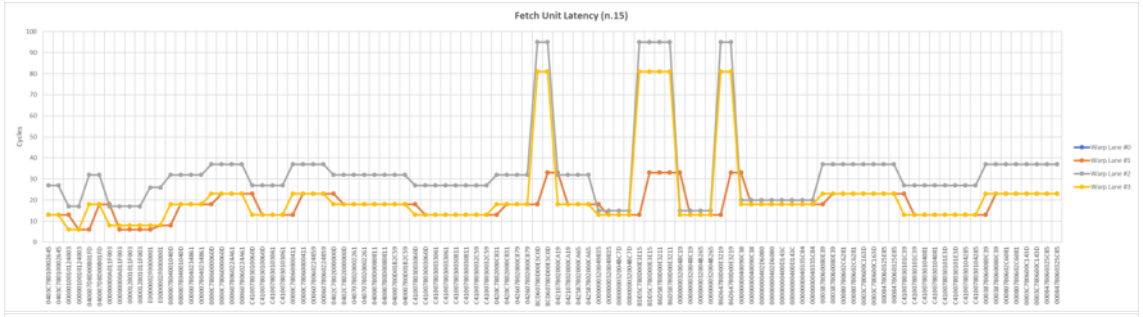

Fetch Unit Latency (n.10)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

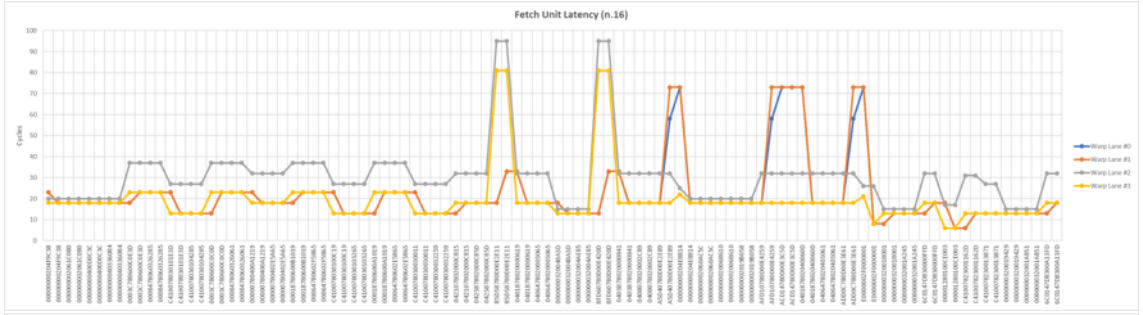

Fetch Unit Latency (n.11)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

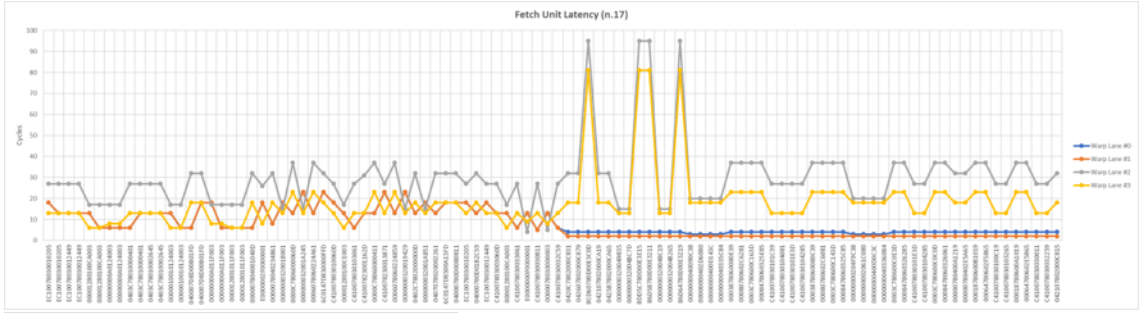

Fetch Unit Latency (n.12)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

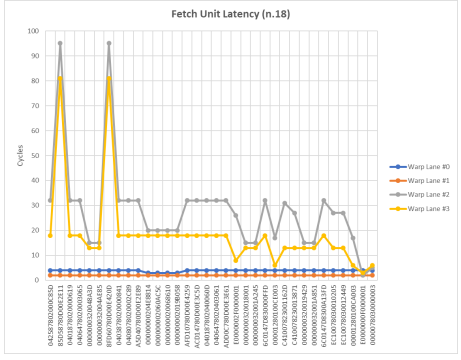

Fetch Unit Latency (n.13)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

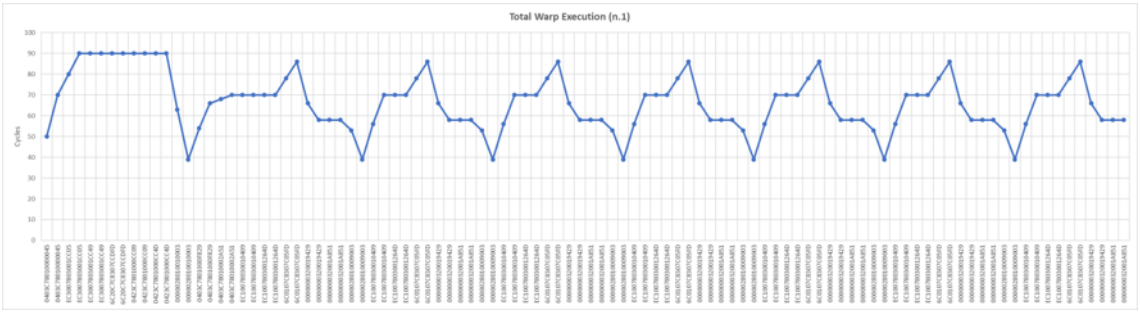

Fetch Unit Latency (n.14)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

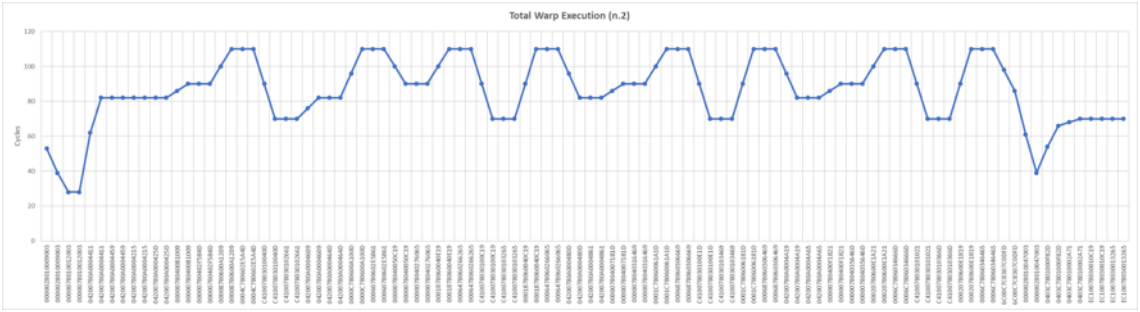

Fetch Unit Latency (n.15)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3


Fetch Unit Latency (n.16)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

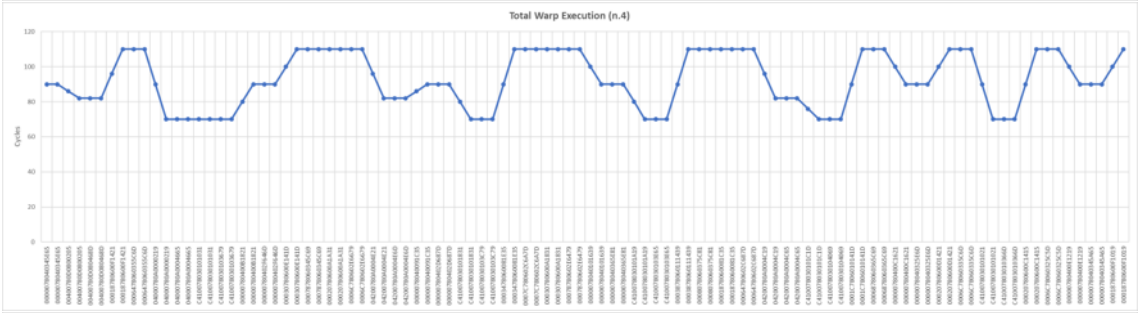

Fetch Unit Latency (n.17)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

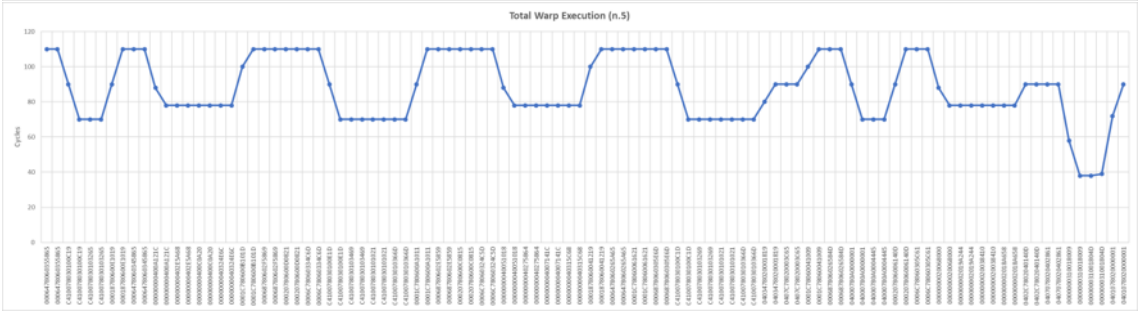

Fetch Unit Latency (n.18)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3


Total Warp Execution (n.1)
Cycles

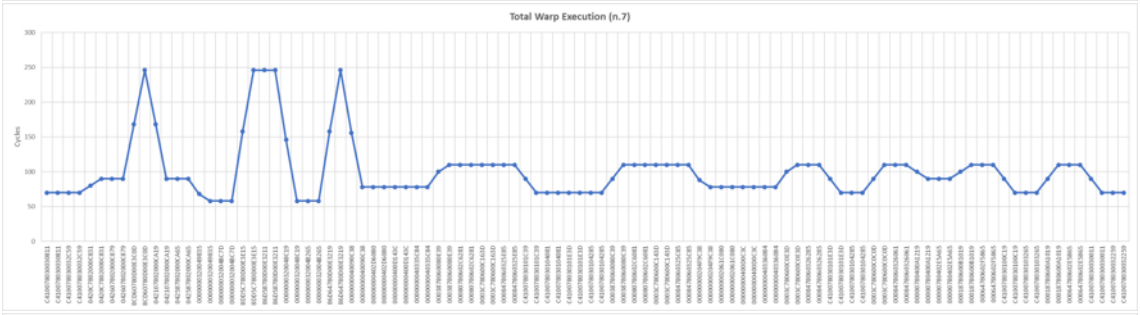

Total Warp Execution (n.2)
Cycles

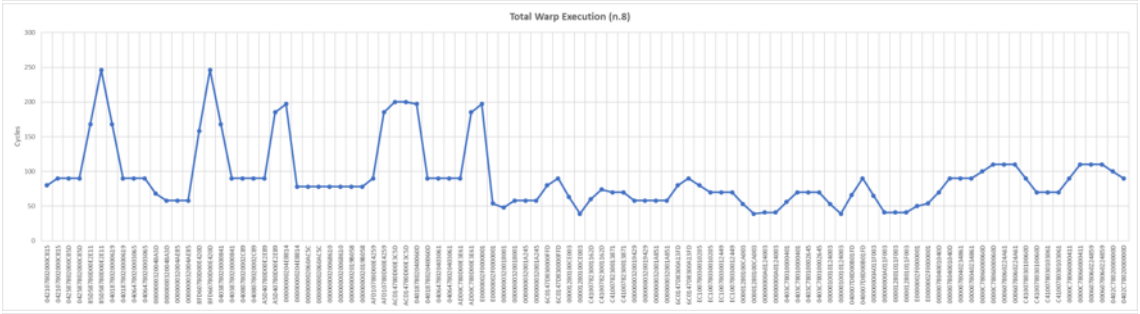

Total Warp Execution (n.3)
Cycles

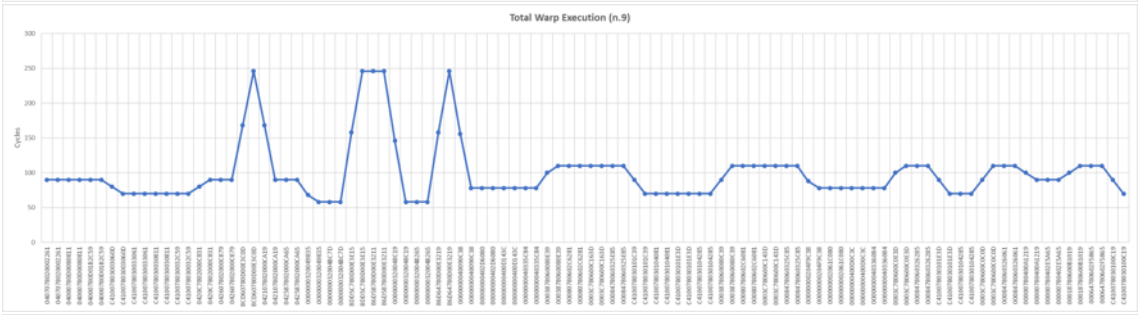

Total Warp Execution (n.4)
Cycles

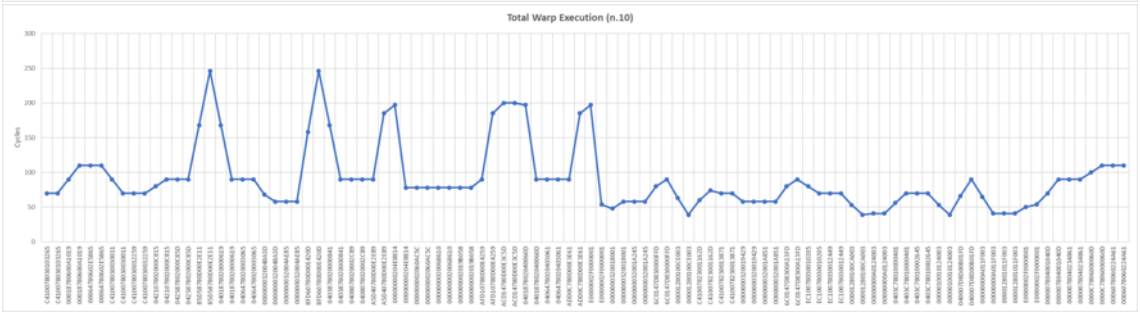

Total Warp Execution (n.5)
Cycles

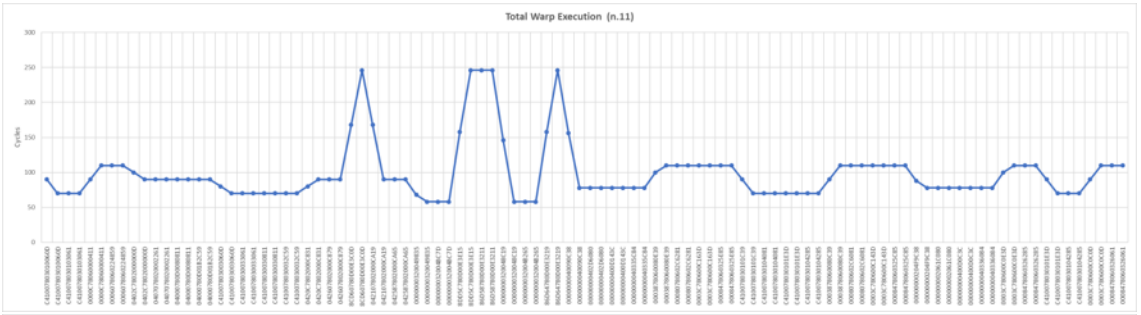

Total Warp Execution (n.6)
Cycles

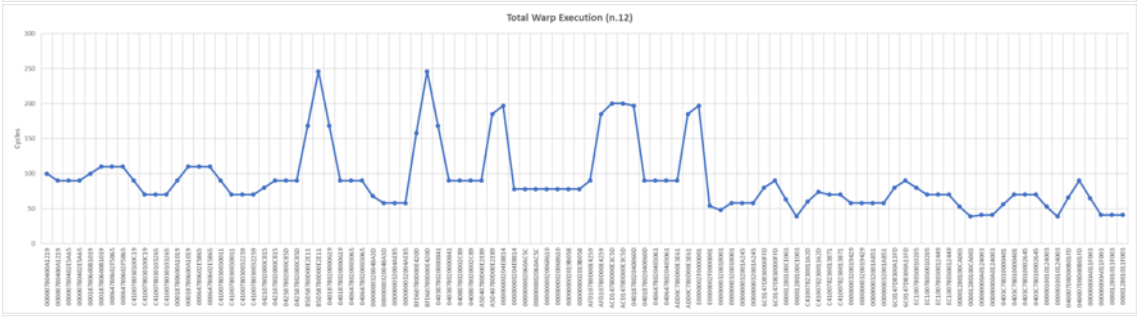

Total Warp Execution (n.7)
Cycles

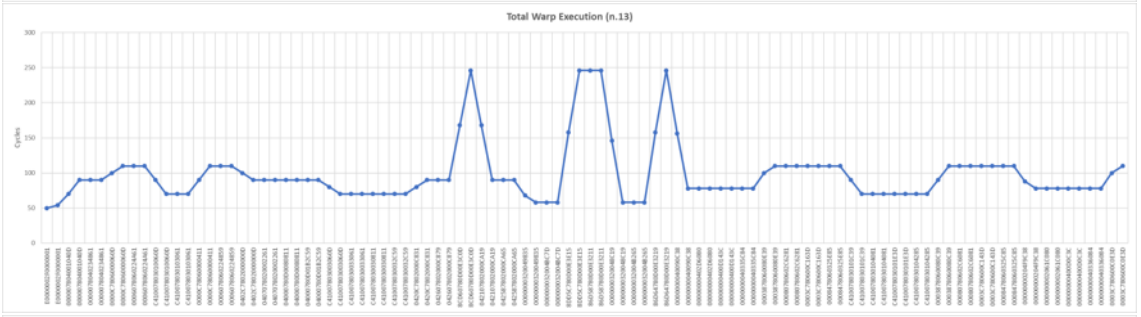

Total Warp Execution (n.8)
Cycles

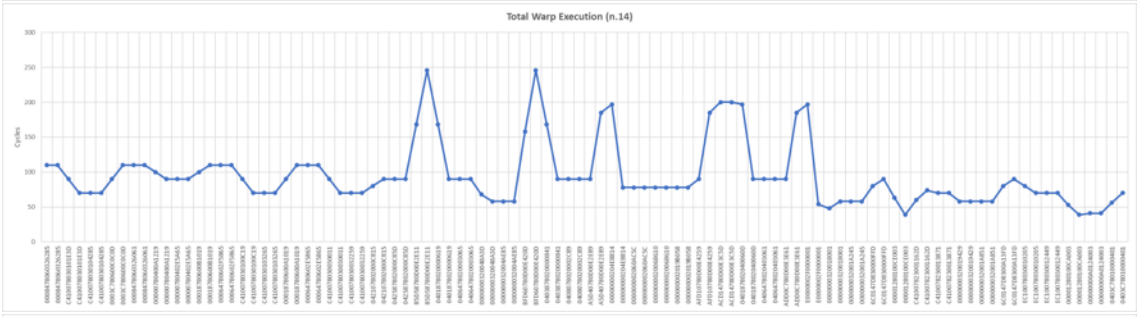

Total Warp Execution (n.9)
Cycles

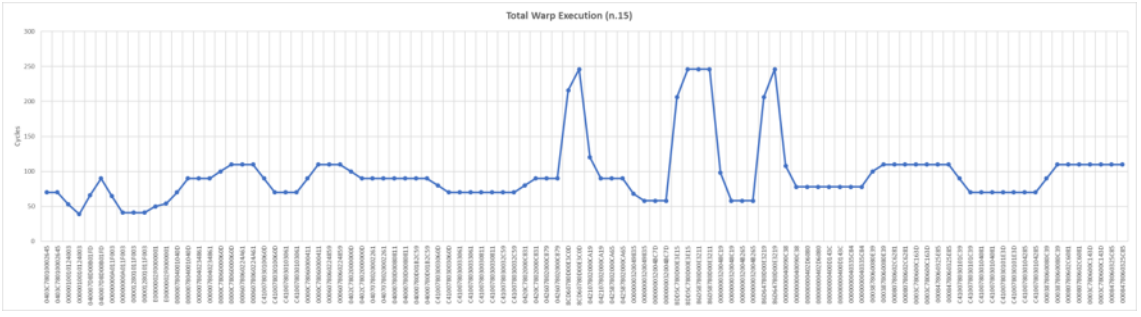

Total Warp Execution (n.10)
Cycles

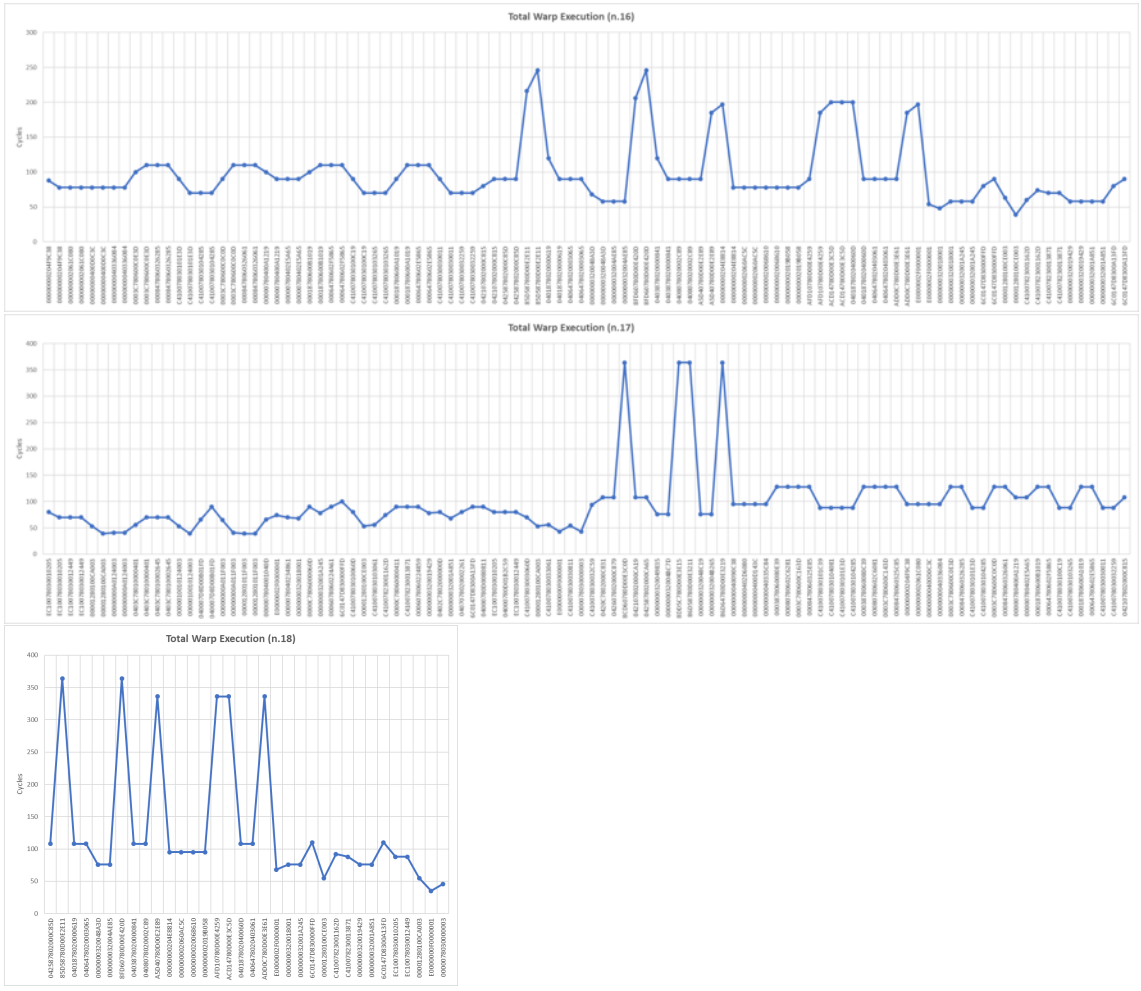

Total Warp Execution (n.11)
Cycles


Total Warp Execution (n.12)
Cycles

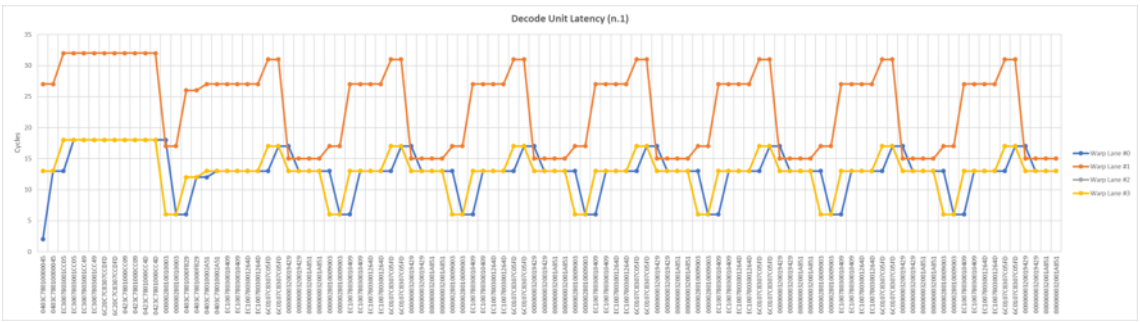

Total Warp Execution (n.13)
Cycles

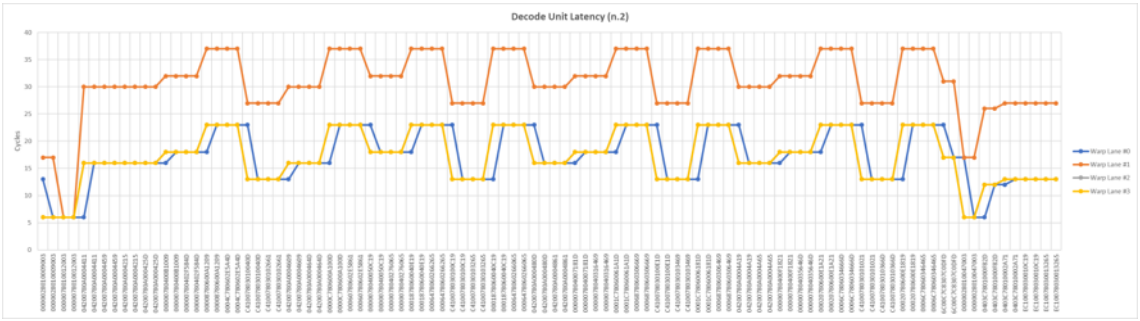

Total Warp Execution (n.14)
Cycles

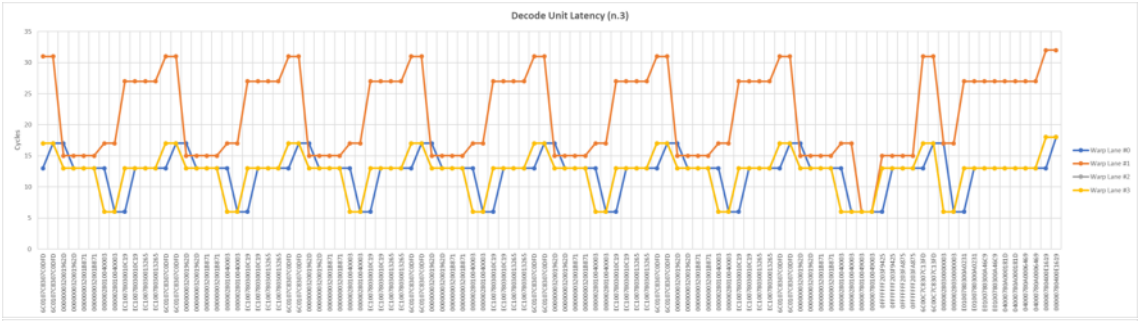

Total Warp Execution (n.15)
Cycles

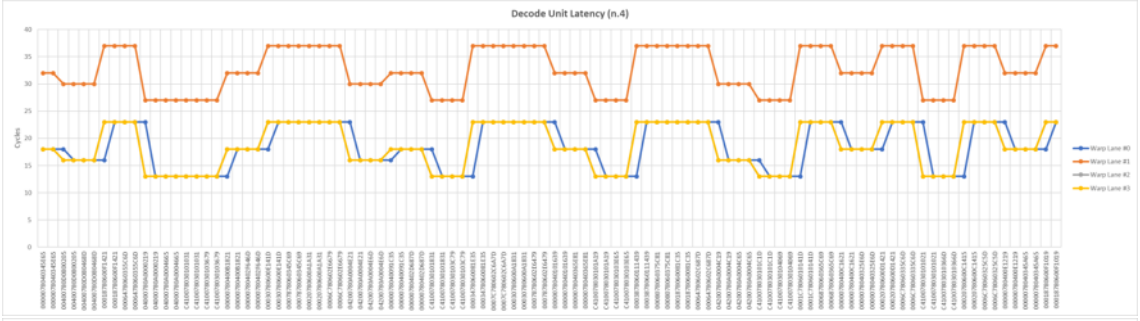


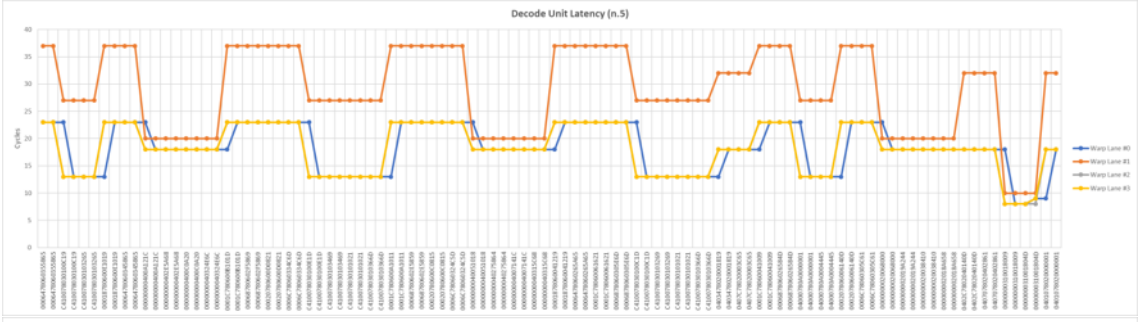
D.2 Decode Unit

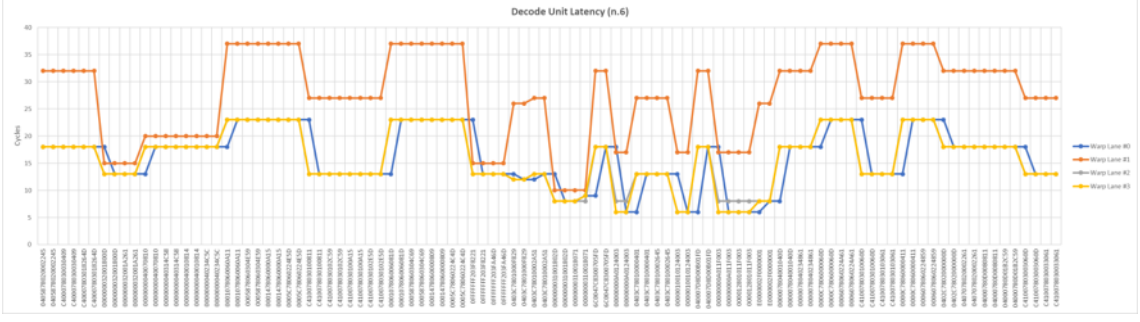


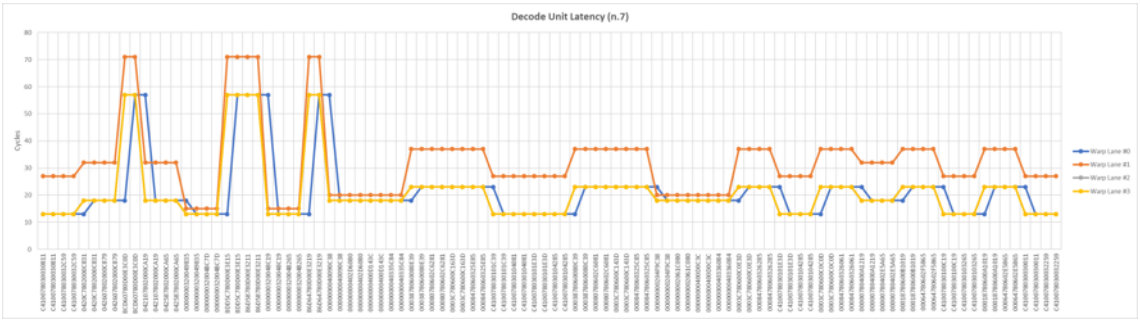

Decode Unit Latency (n.2)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

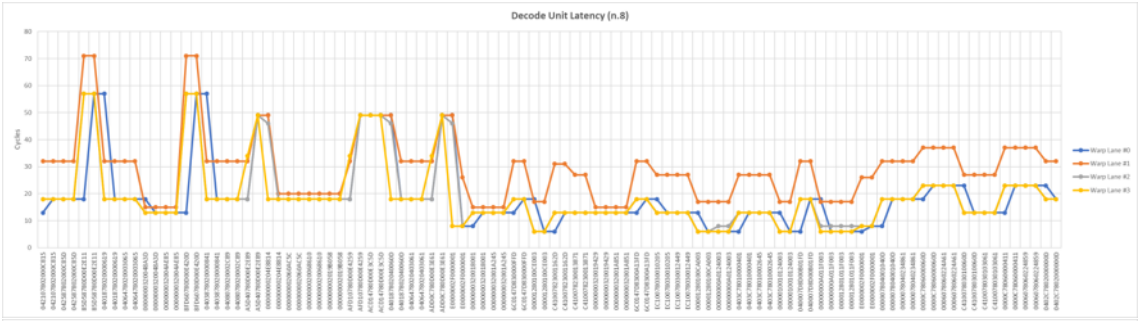

Decode Unit Latency (n.3)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

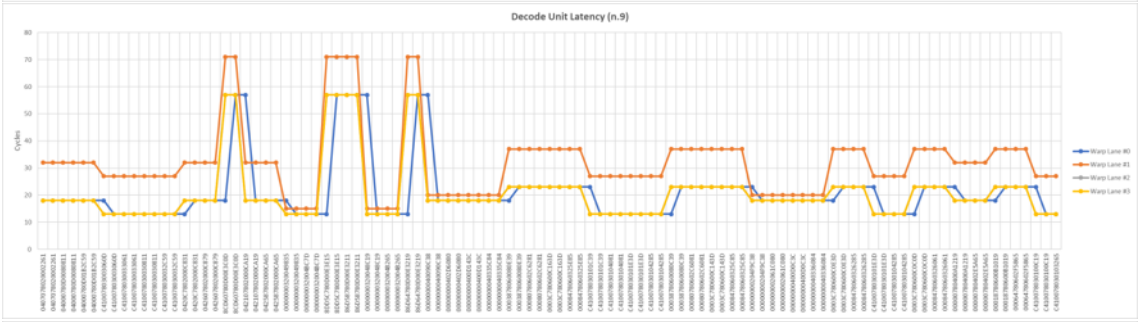

Decode Unit Latency (n.4)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

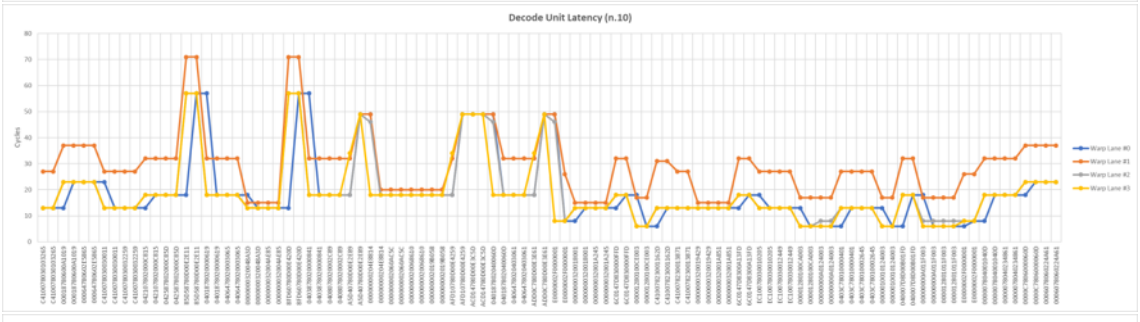

Decode Unit Latency (n.5)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

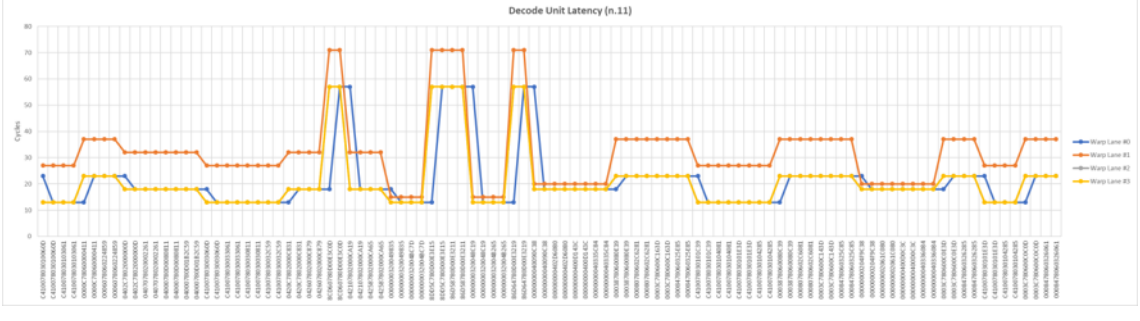

Decode Unit Latency (n.6)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

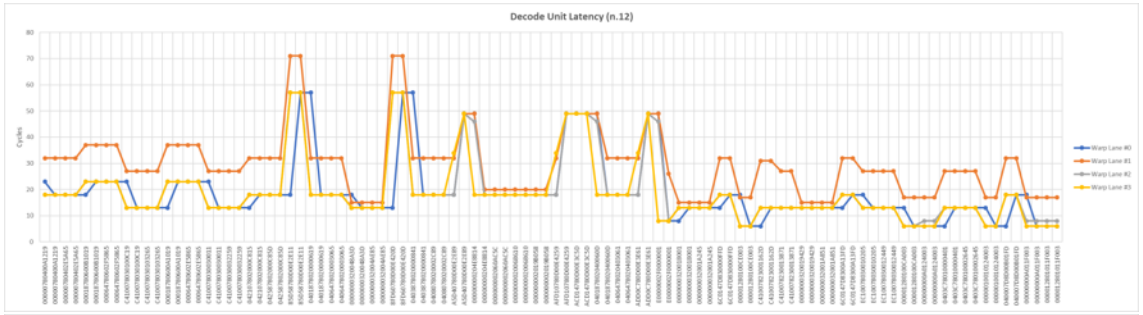

Decode Unit Latency (n.7)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

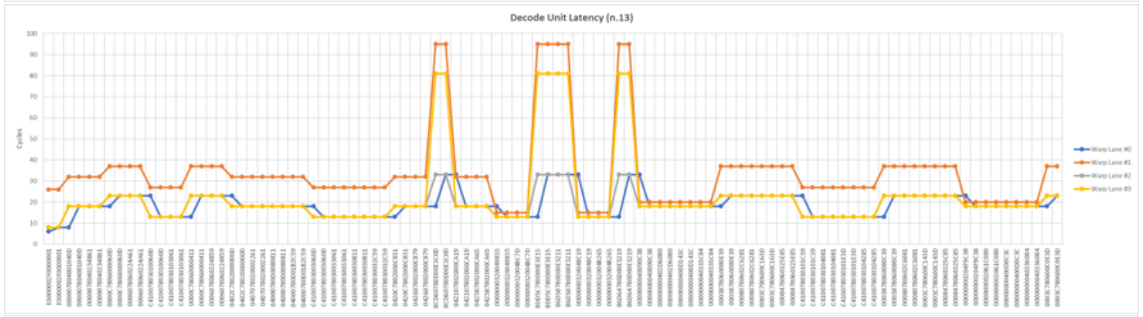

Decode Unit Latency (n.8)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

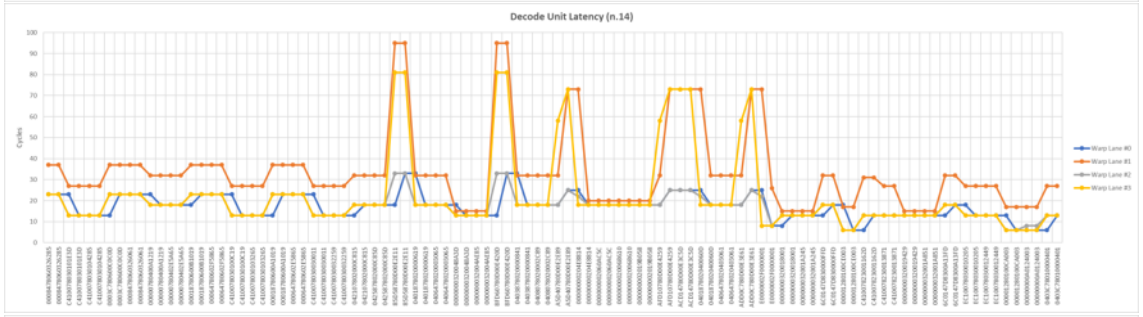

Decode Unit Latency (n.9)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

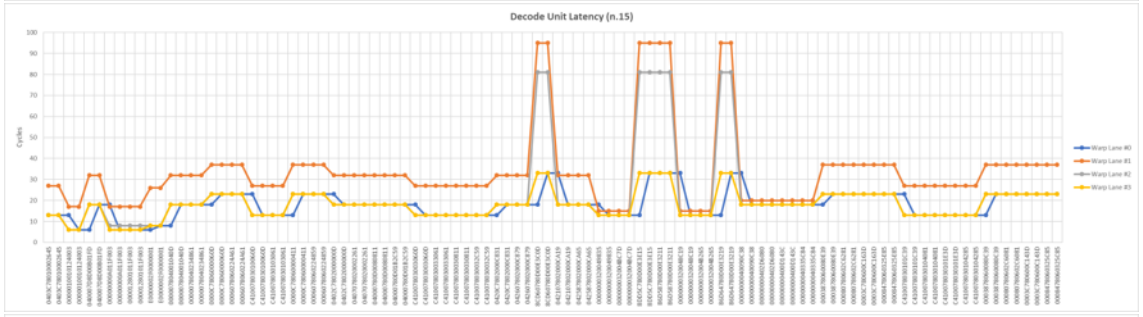

Decode Unit Latency (n.10)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

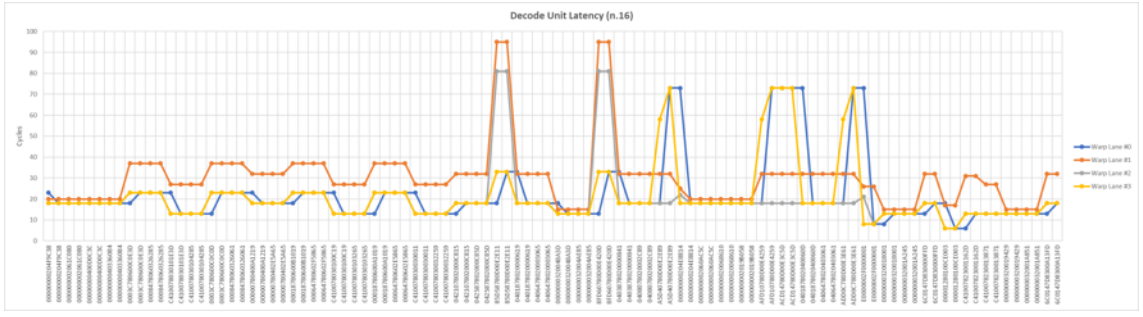

Decode Unit Latency (n.11)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

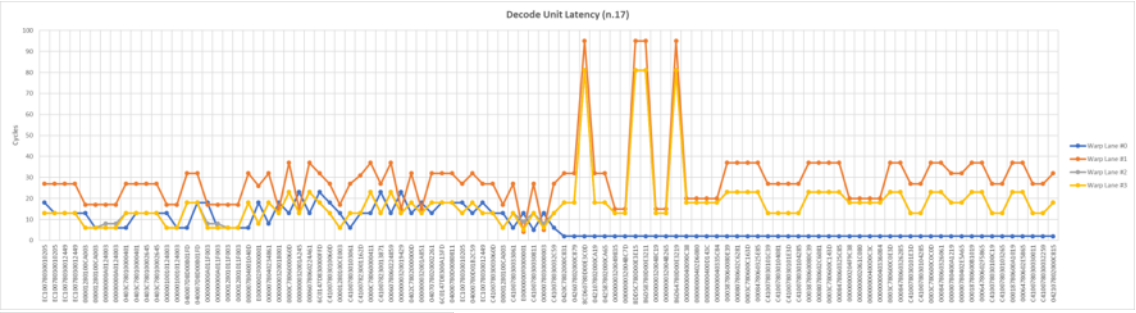

Decode Unit Latency (n.12)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

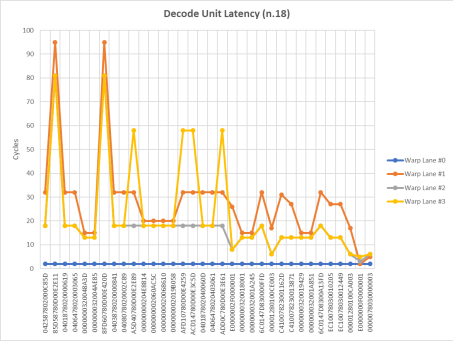

Decode Unit Latency (n.13)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3


Decode Unit Latency (n.14)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

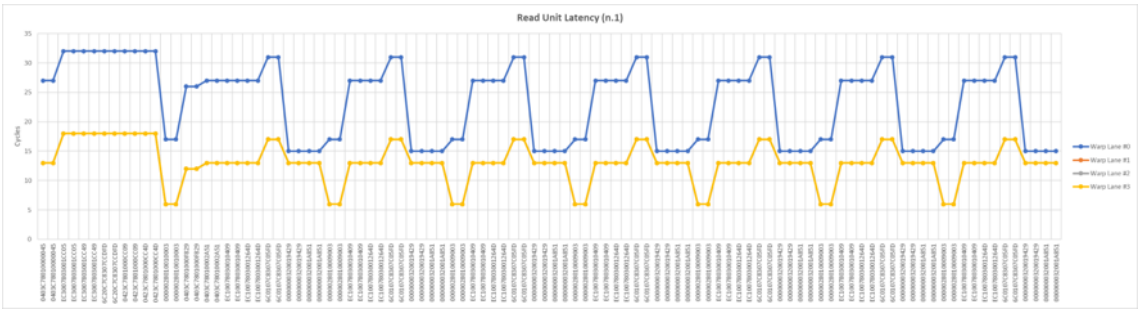

Decode Unit Latency (n.15)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

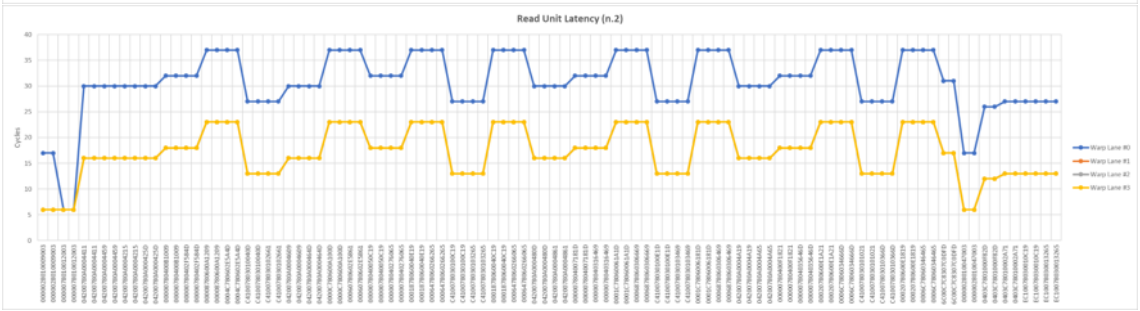

Decode Unit Latency (n.16)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

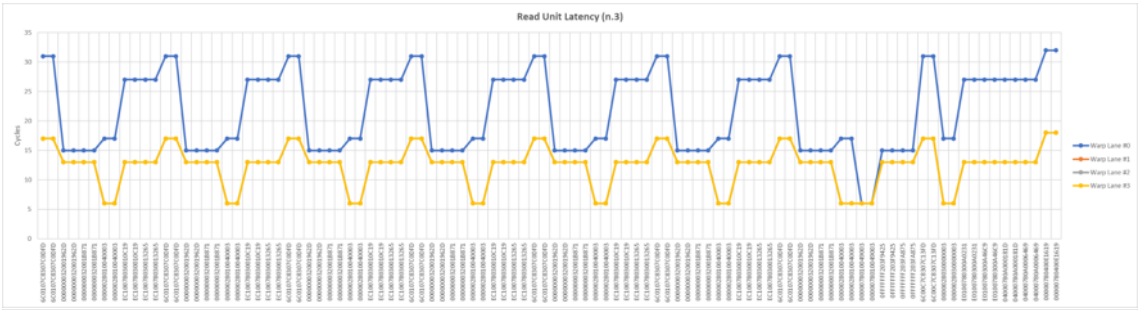

Decode Unit Latency (n.17)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

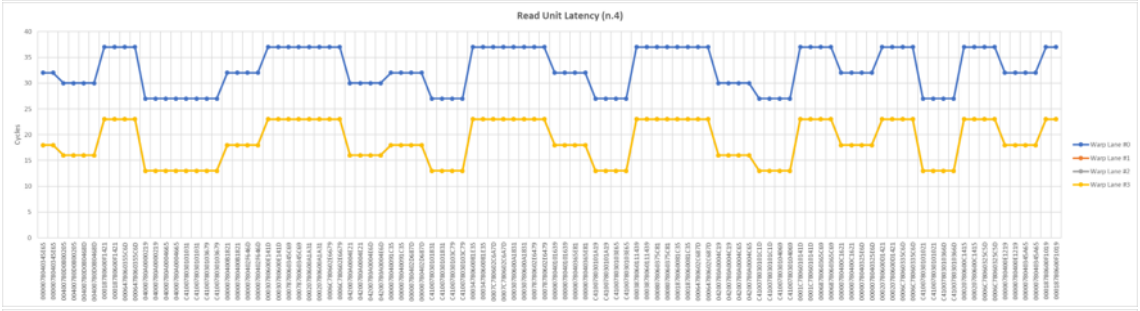

Decode Unit Latency (n.18)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

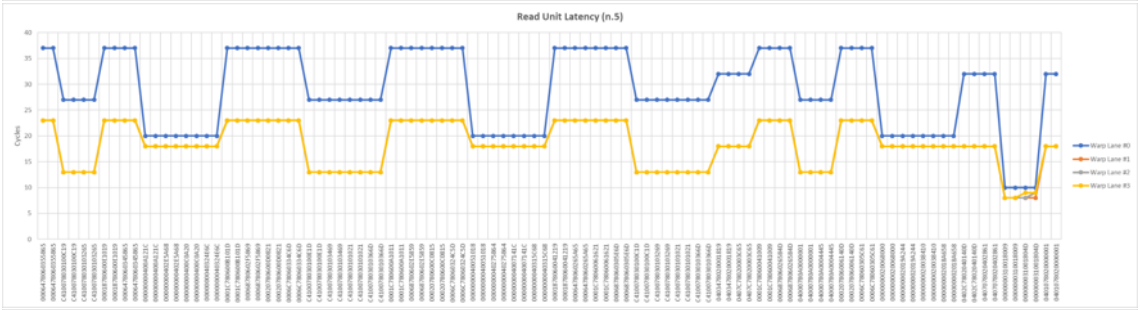
D.3 Read Unit

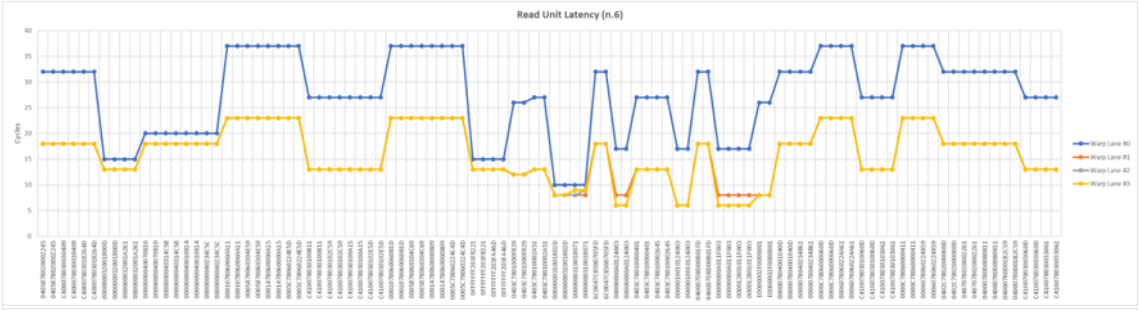

Read Unit Latency (n.1)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

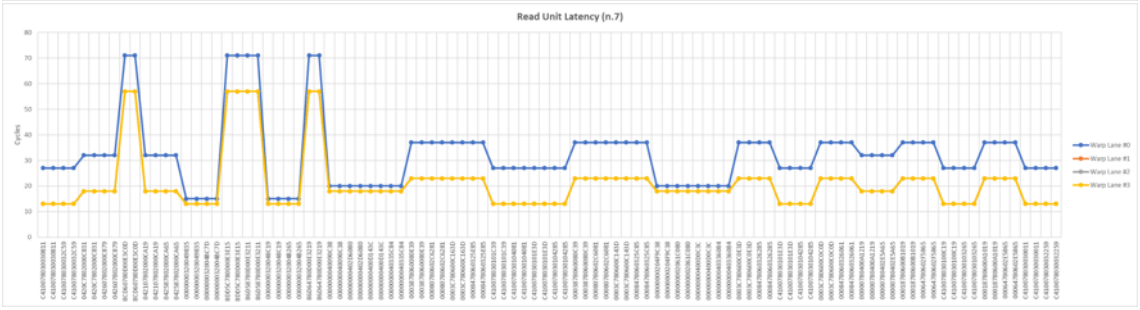

Read Unit Latency (n.2)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3


Read Unit Latency (n.3)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

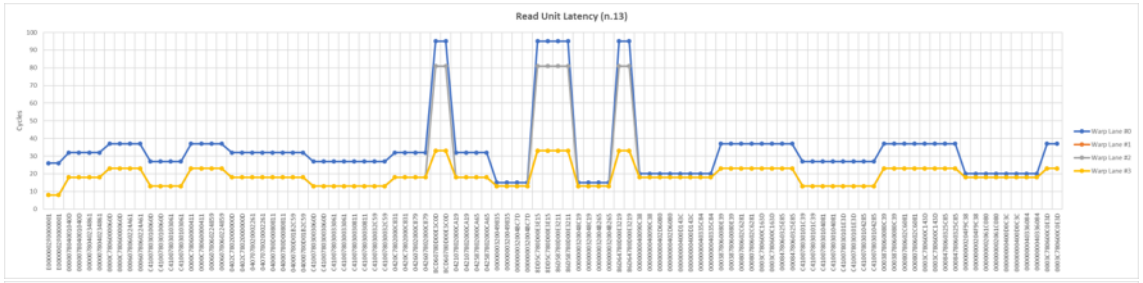

Read Unit Latency (n.4)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

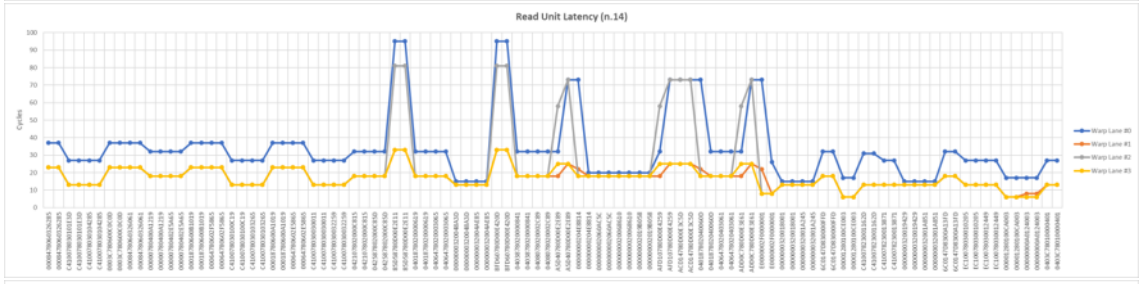

Read Unit Latency (n.5)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

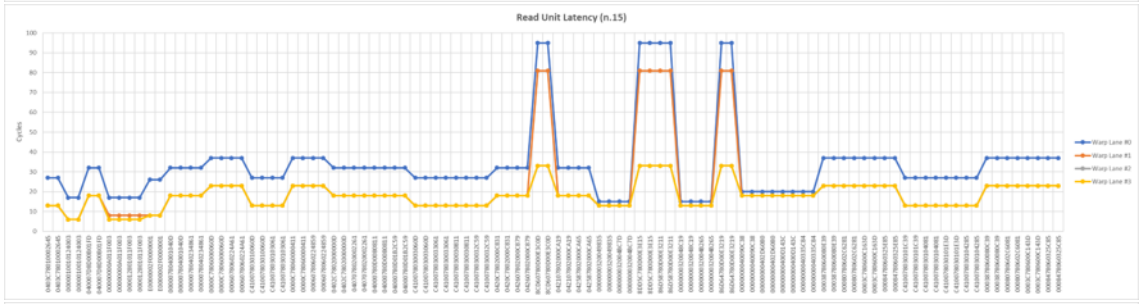

Read Unit Latency (n.6)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

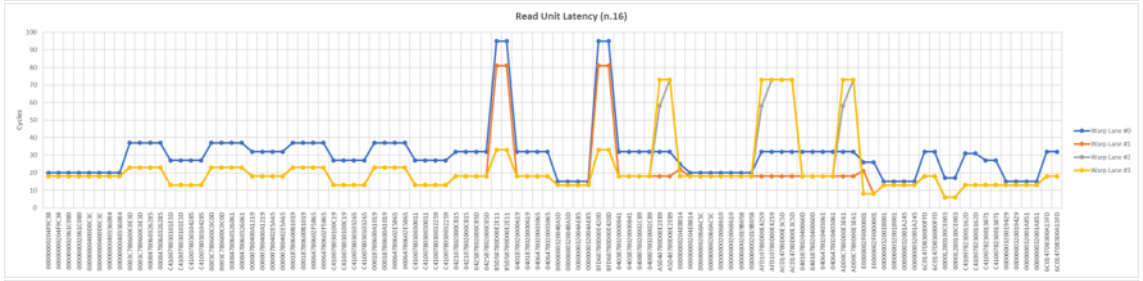

Read Unit Latency (n.7)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

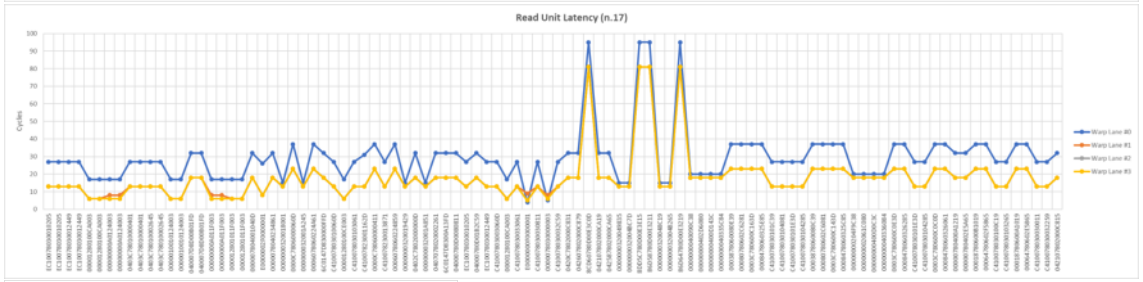

Read Unit Latency (n.8)
Read Unit Latency (n.9)
Read Unit Latency (n.10)
Read Unit Latency (n.11)
Read Unit Latency (n.12)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

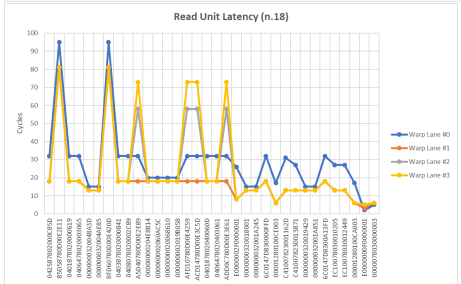

Read Unit Latency (n.13)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3


Read Unit Latency (n.14)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

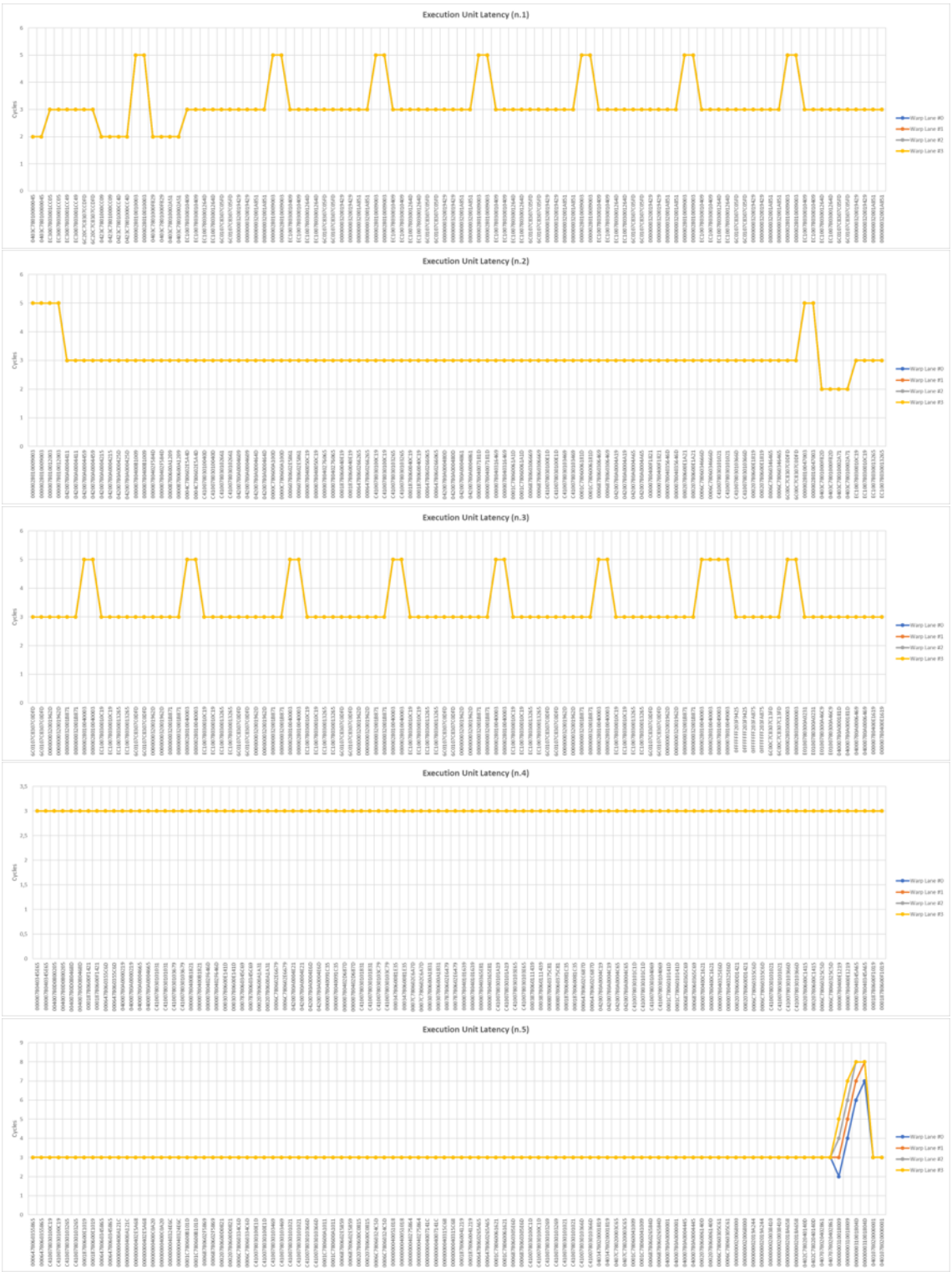

Read Unit Latency (n.15)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

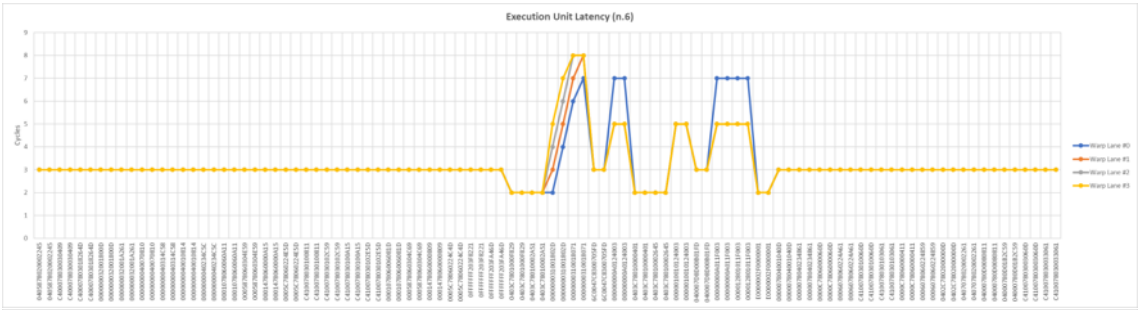

Read Unit Latency (n.16)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

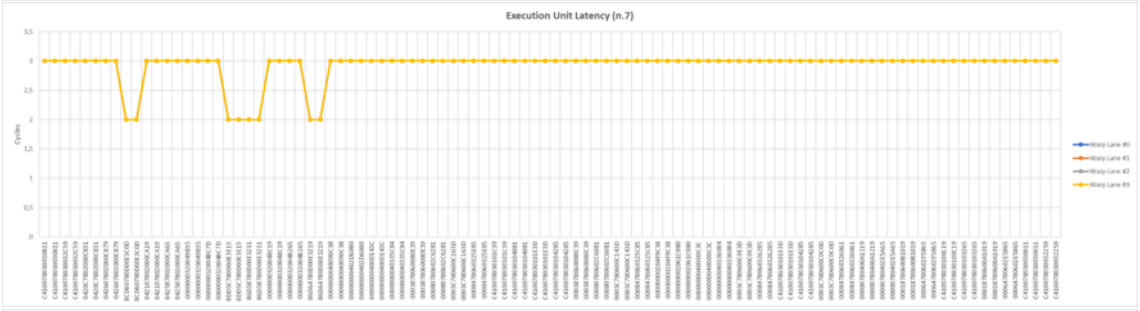

Read Unit Latency (n.17)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

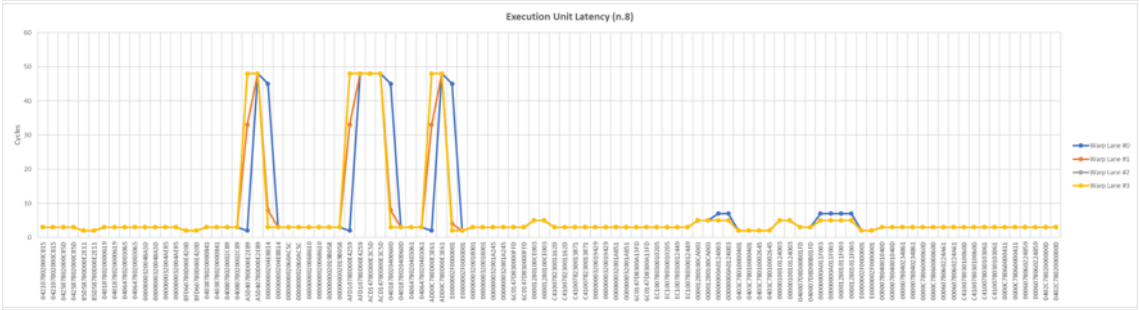

Read Unit Latency (n.18)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

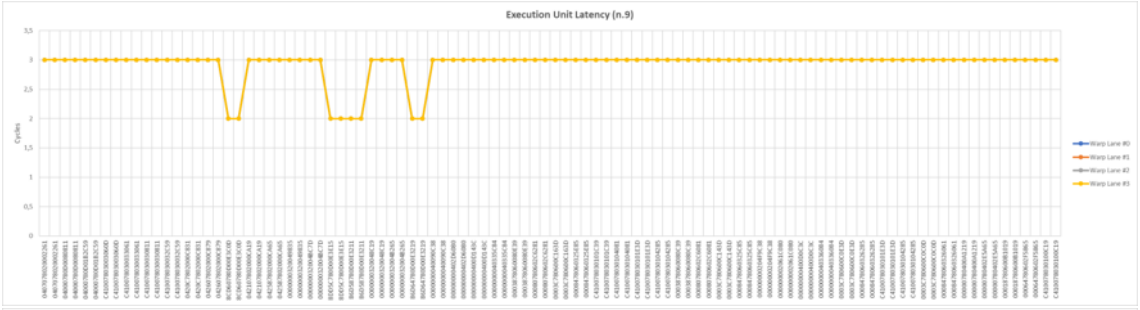
D.4 Execution Unit

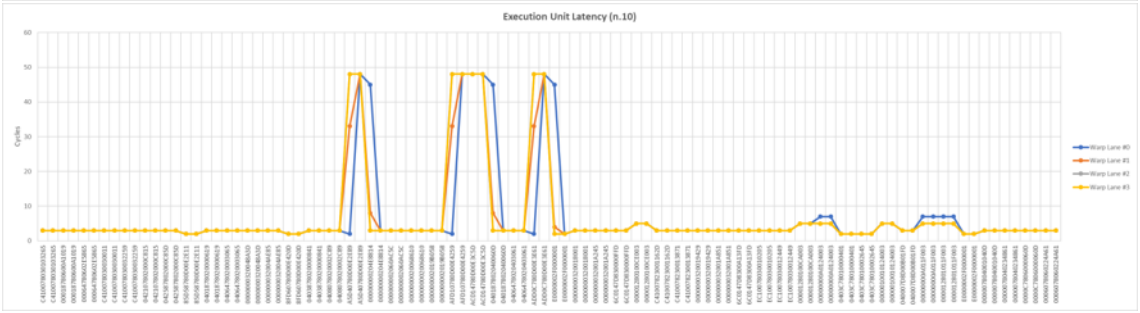


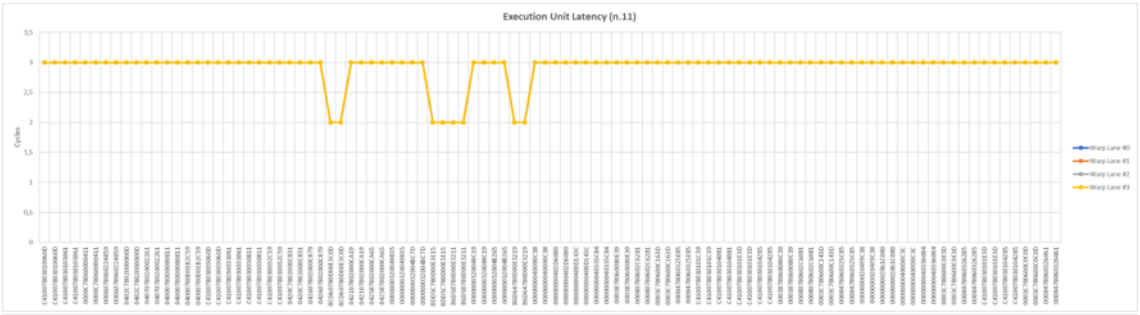

Execution Unit Latency (n.6)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

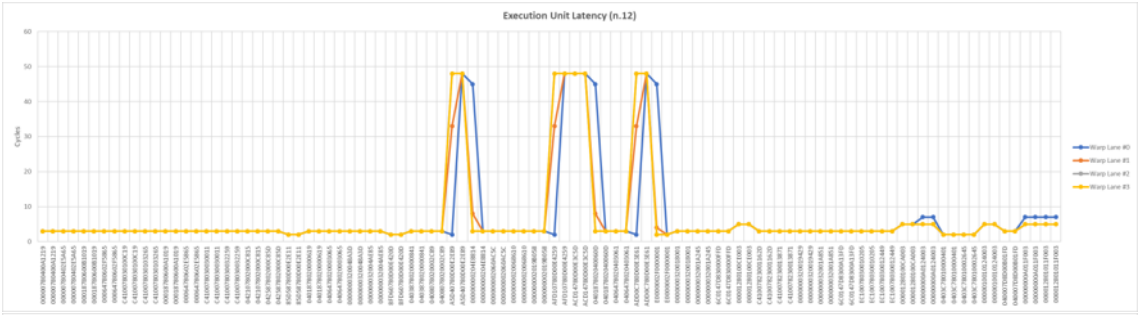

Execution Unit Latency (n.7)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

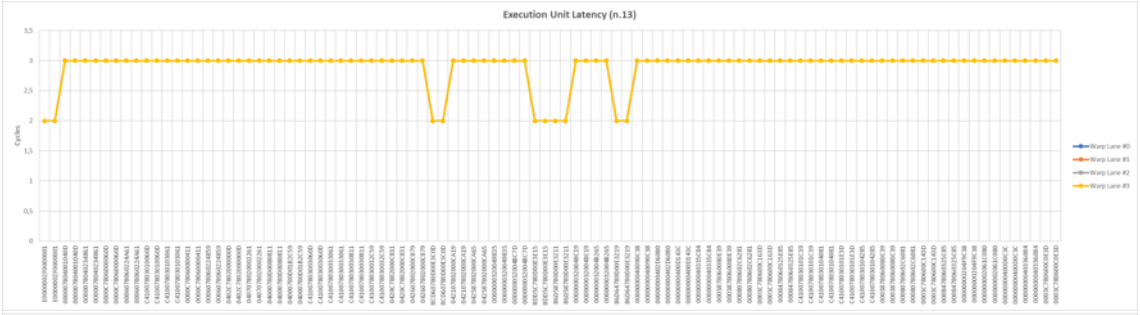

Execution Unit Latency (n.8)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

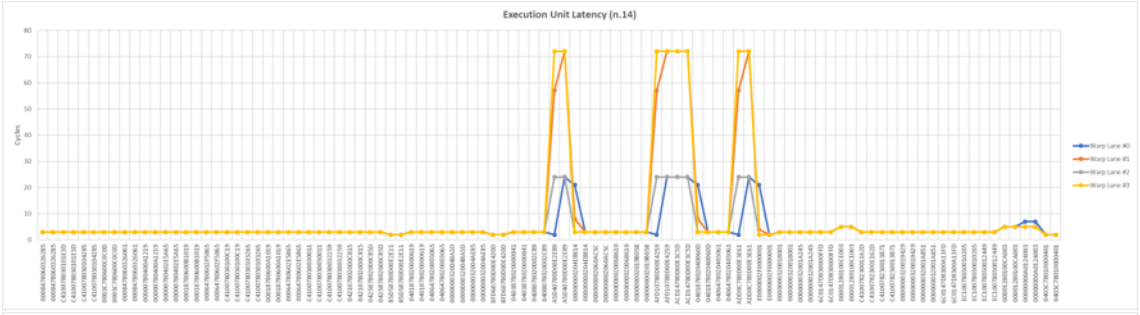

Execution Unit Latency (n.9)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

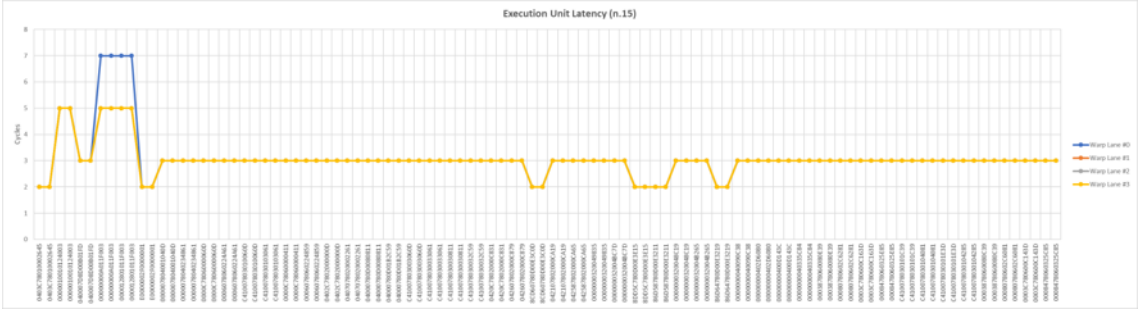

Execution Unit Latency (n.10)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3


Execution Unit Latency (n.11)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

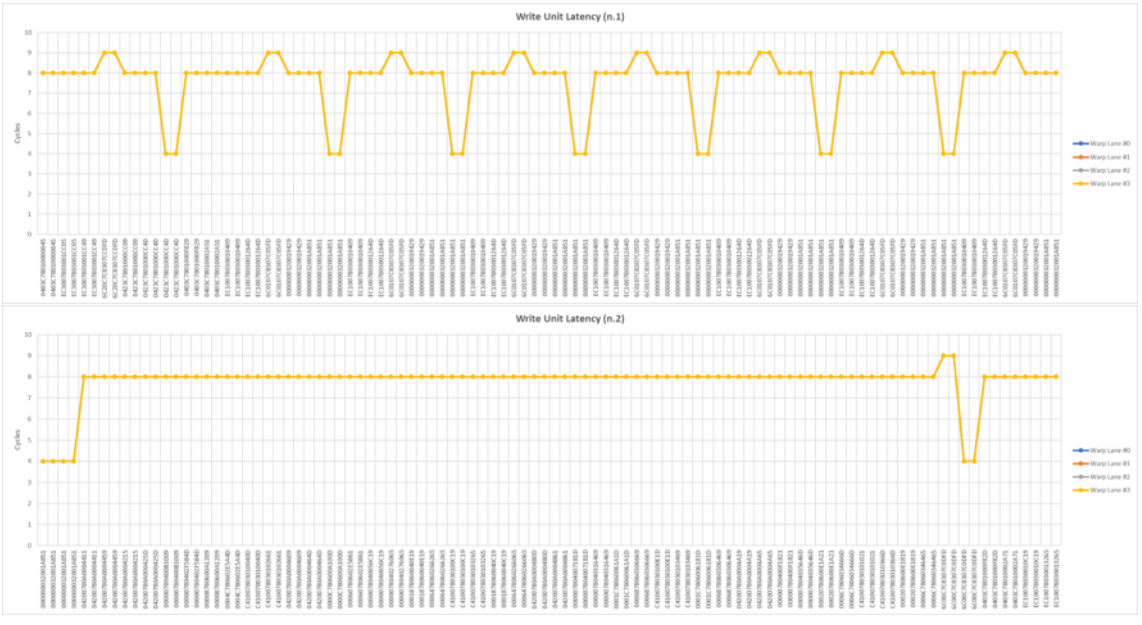

Execution Unit Latency (n.12)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

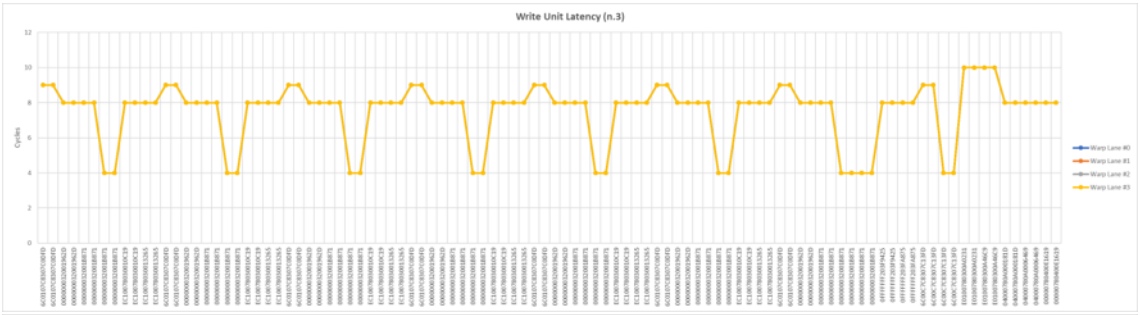

Execution Unit Latency (n.13)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

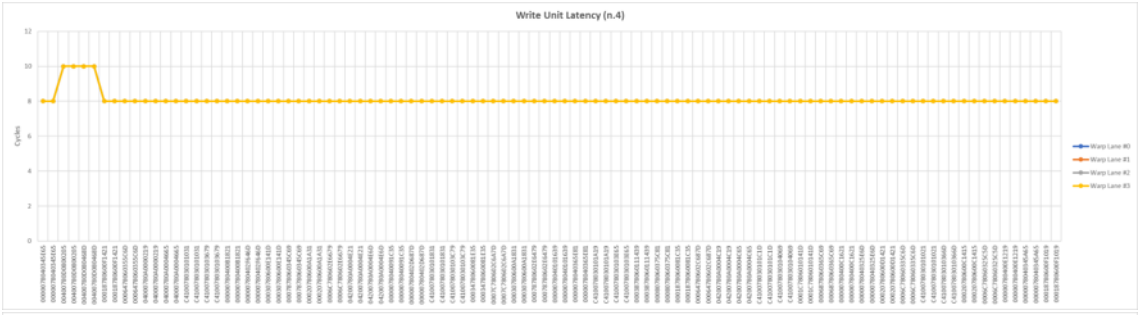

Execution Unit Latency (n.14)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

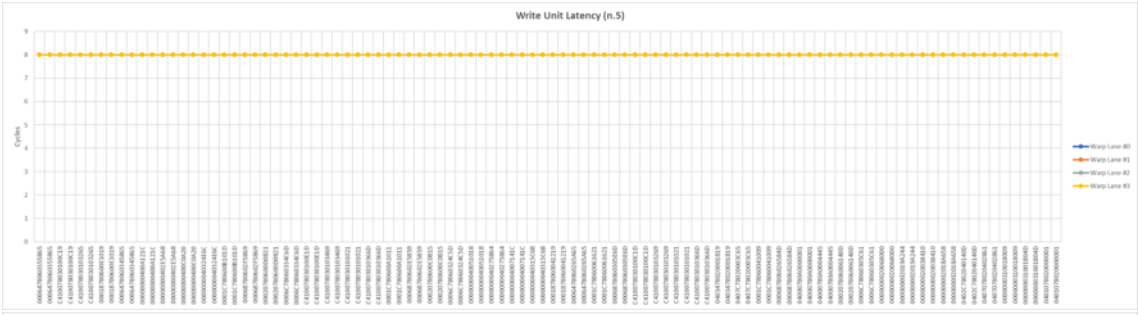

Execution Unit Latency (n.15)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

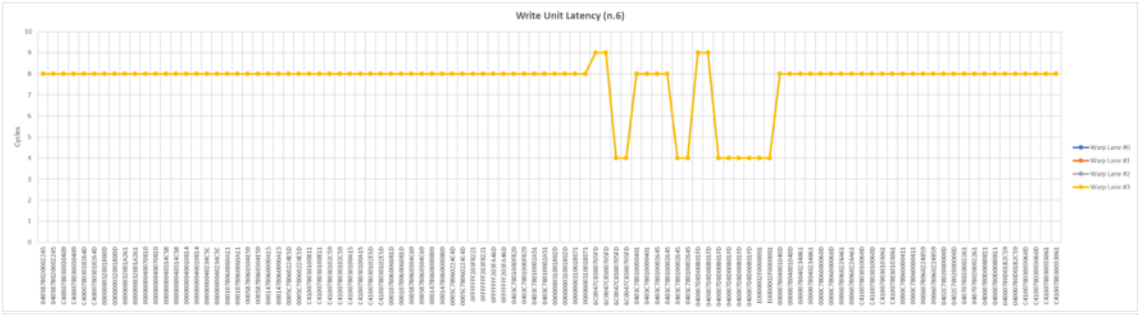
D.5 Write Unit

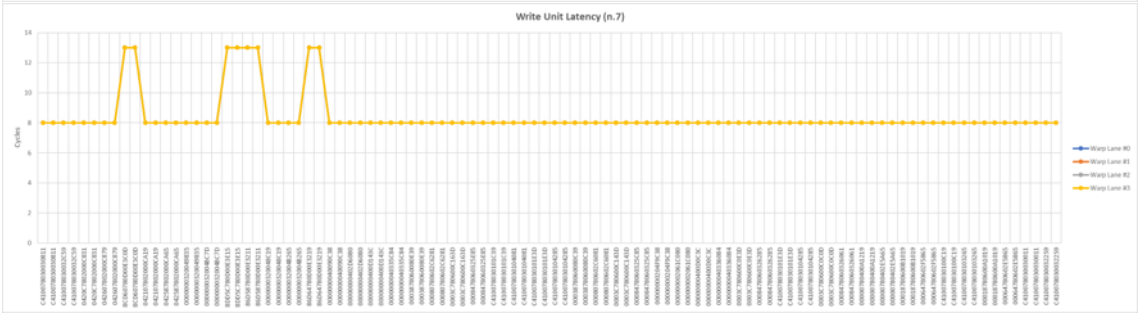


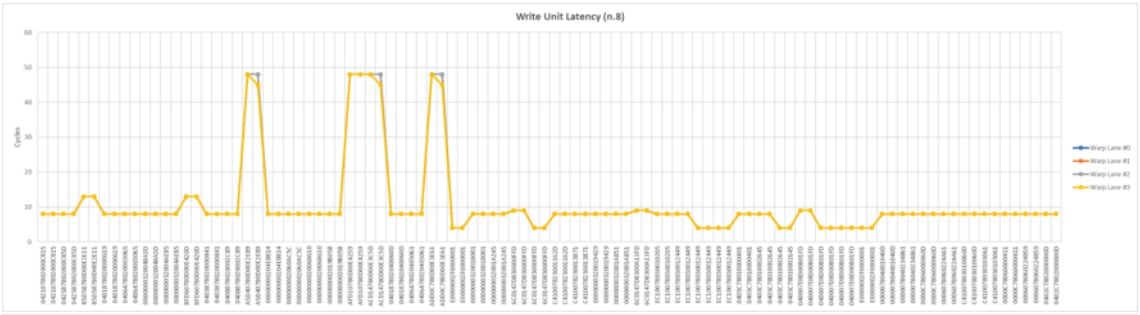

Write Unit Latency (n.3)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

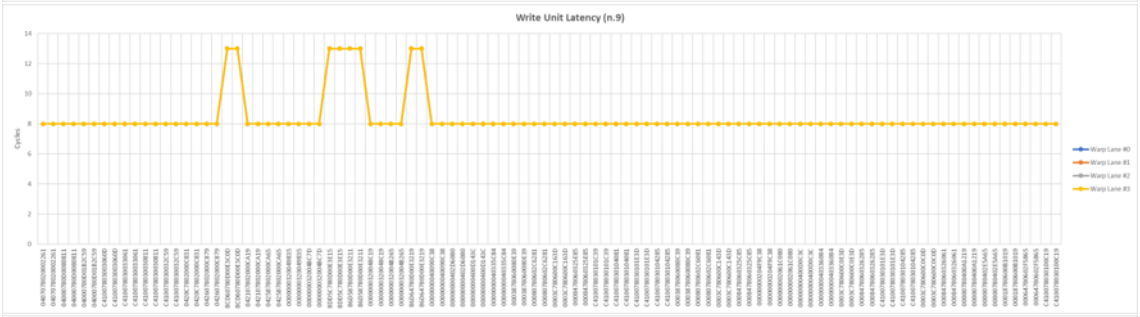

Write Unit Latency (n.4)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

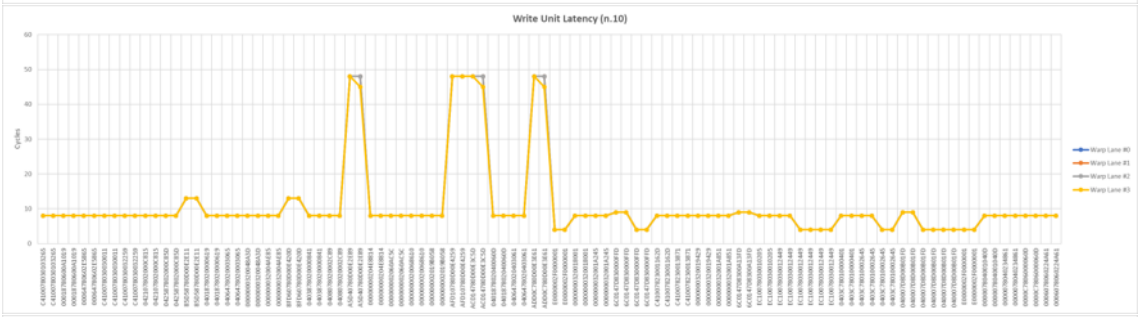

Write Unit Latency (n.5)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

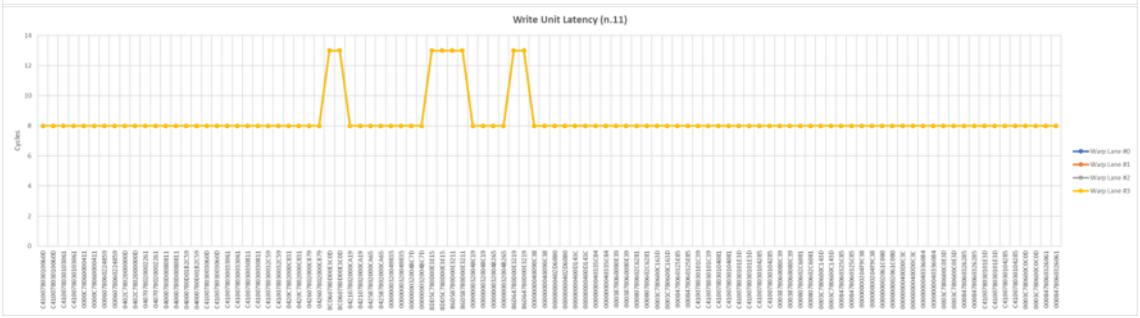

Write Unit Latency (n.6)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

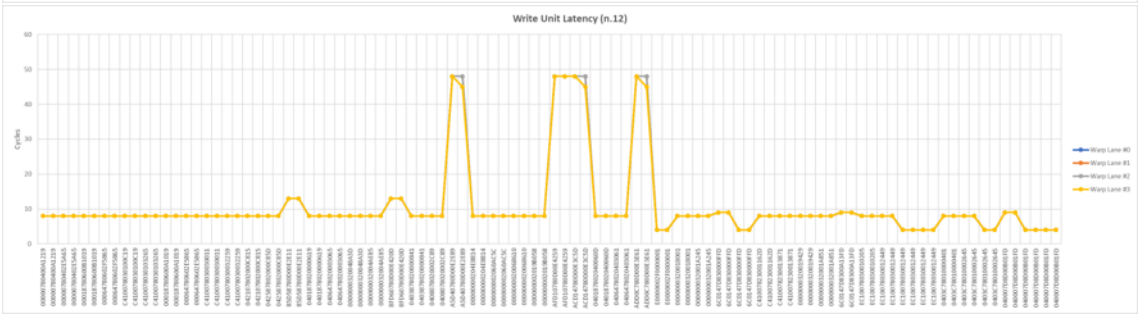

Write Unit Latency (n.7)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

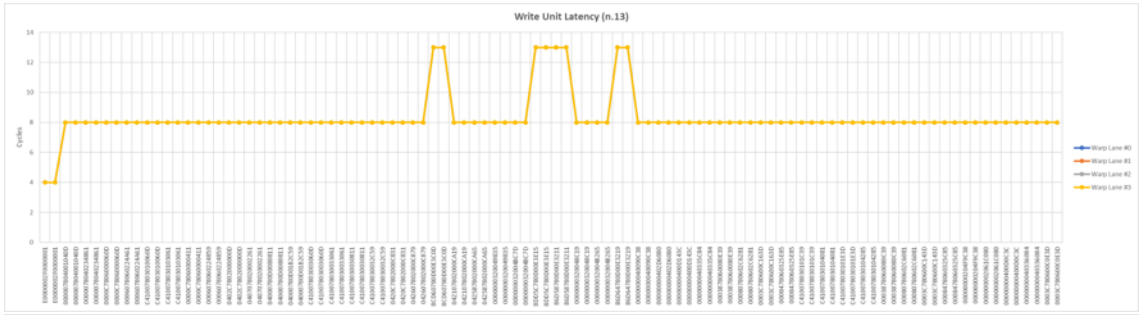

Write Unit Latency (n.8)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

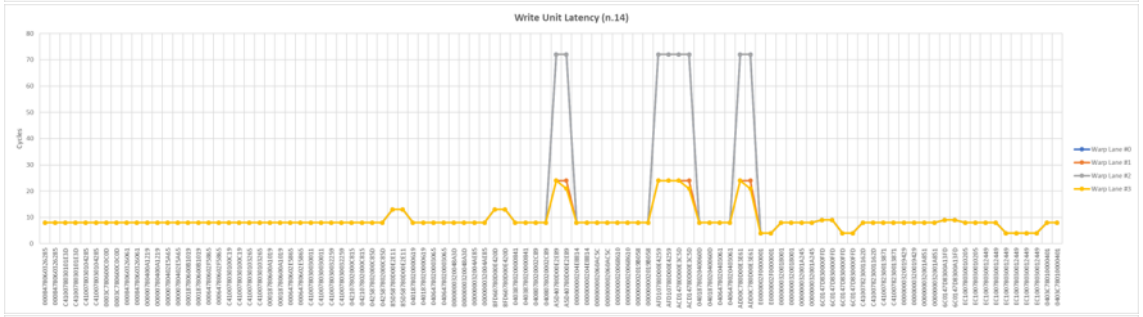

Write Unit Latency (n.9)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

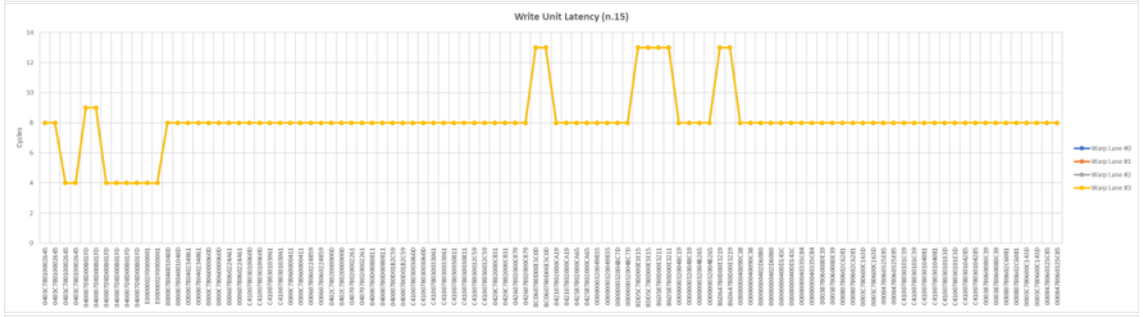

Write Unit Latency (n.10)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

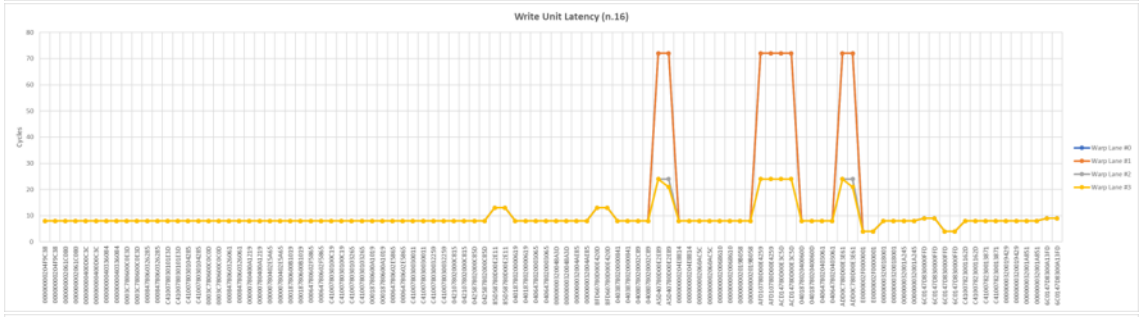

Write Unit Latency (n.11)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

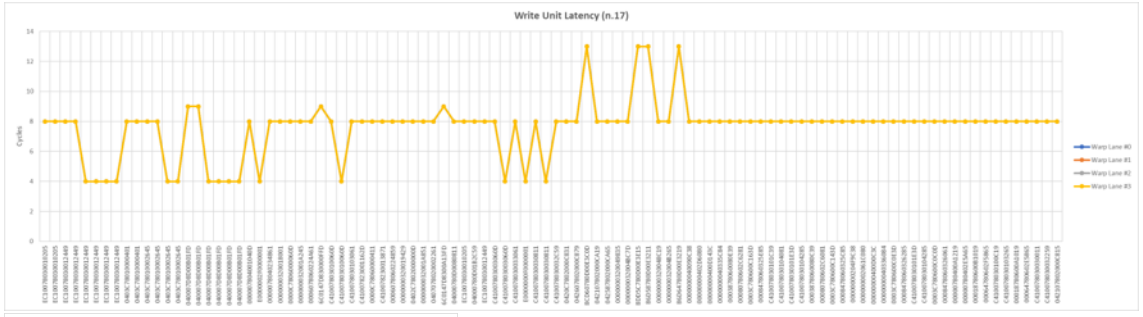

Write Unit Latency (n.12)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

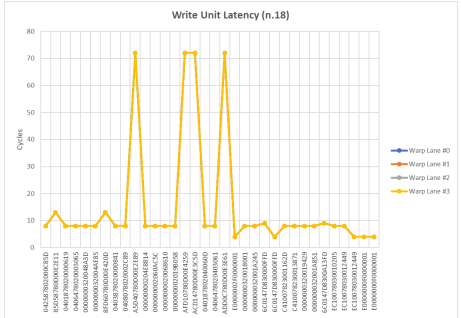

Write Unit Latency (n.13)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3


Write Unit Latency (n.14)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

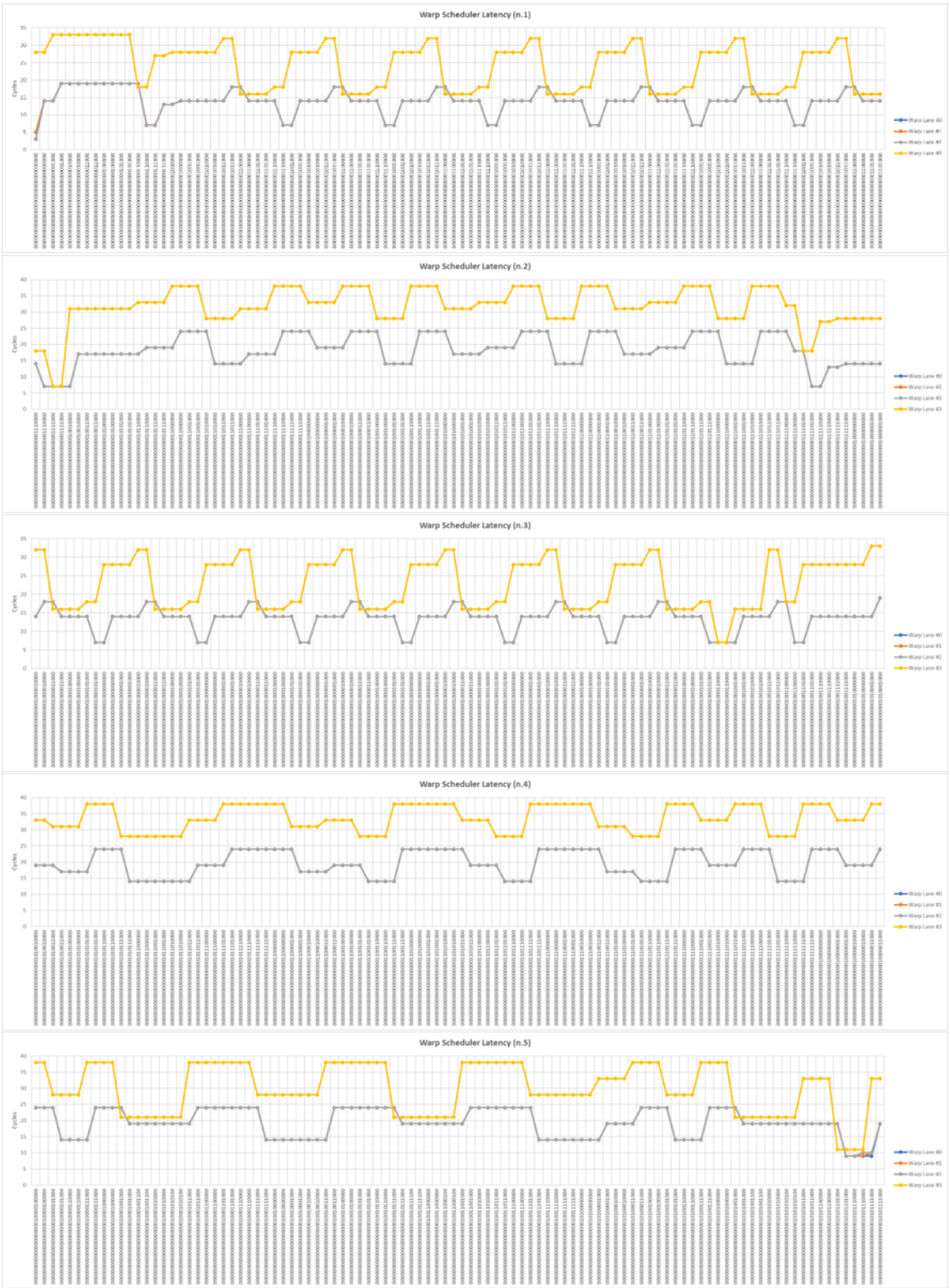

Write Unit Latency (n.15)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

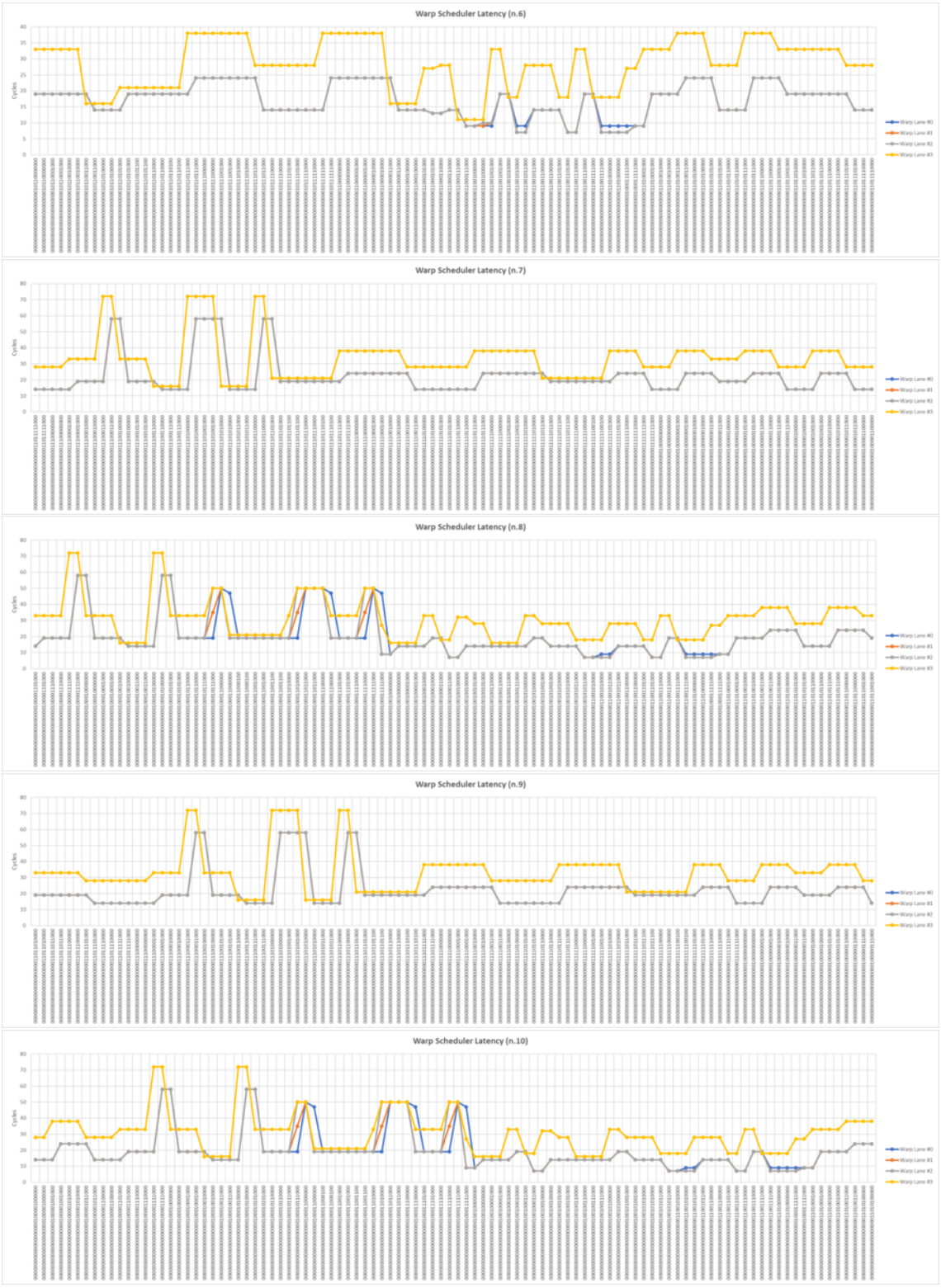

Write Unit Latency (n.16)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3


Write Unit Latency (n.17)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3


Write Unit Latency (n.18)
Cycles
Warp Lane #0
Warp Lane #1
Warp Lane #2
Warp Lane #3

D.6 Warp Scheduler




Warp Scheduler Latency (n.6)
Warp Scheduler Latency (n.7)
Warp Scheduler Latency (n.8)
Warp Scheduler Latency (n.9)
Warp Scheduler Latency (n.10)


Warp Scheduler Latency (n.11)
Warp Scheduler Latency (n.12)
Warp Scheduler Latency (n.13)
Warp Scheduler Latency (n.14)
Warp Scheduler Latency (n.15)
Cycles

D.7 Warp Checker

Independently on the instruction and the warp lane, the warp checker always spends 2 clock cycles to complete the check operation.

D.8 Warp Generator

Time_interval	#Cycles
Total_warp_generation	10

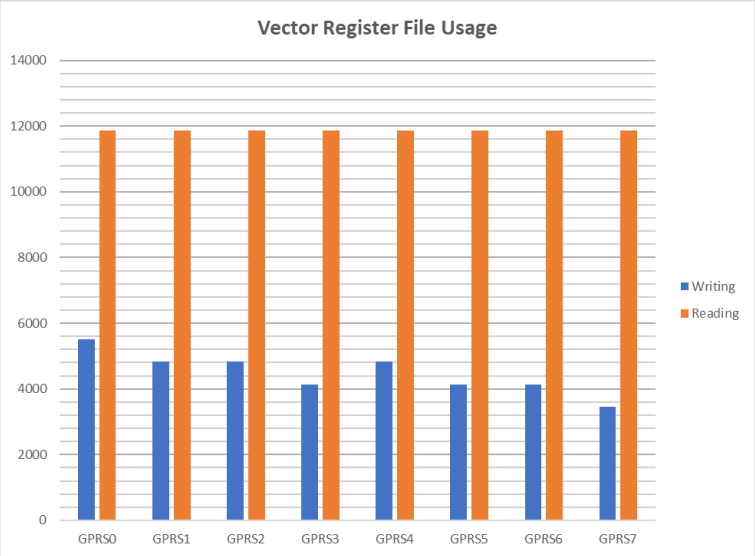
D.9 Streaming Multiprocessor Controller

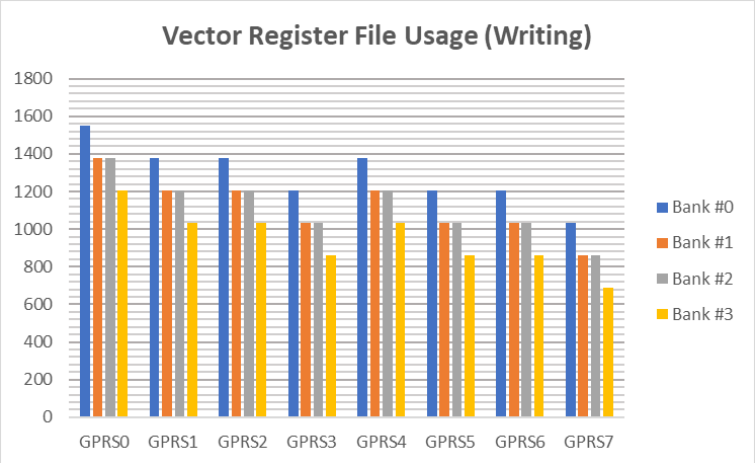
Time_interval	#Cycles
SM_task_execution	157696

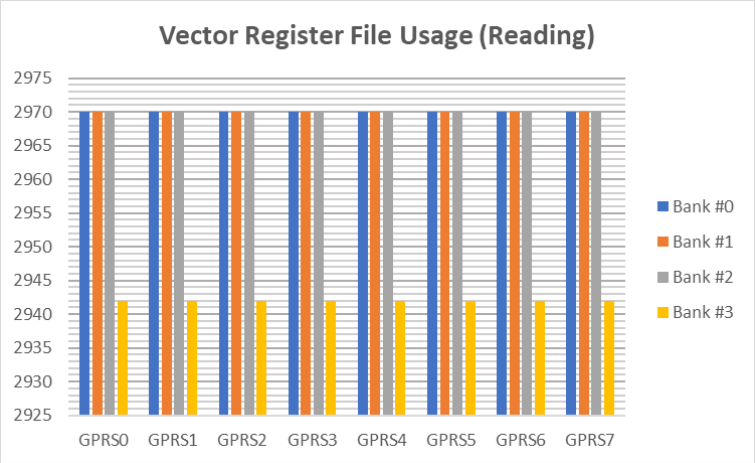
D.10 Block Scheduler

Time_interval	#Cycles
Block_Scheduling	4

D.11 Vector Register File



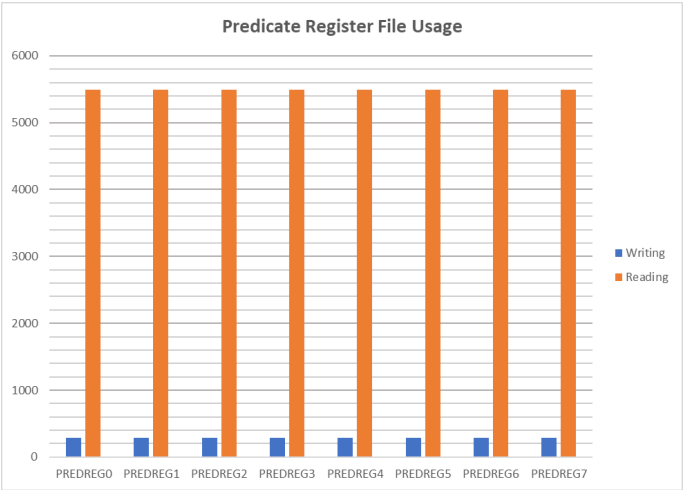


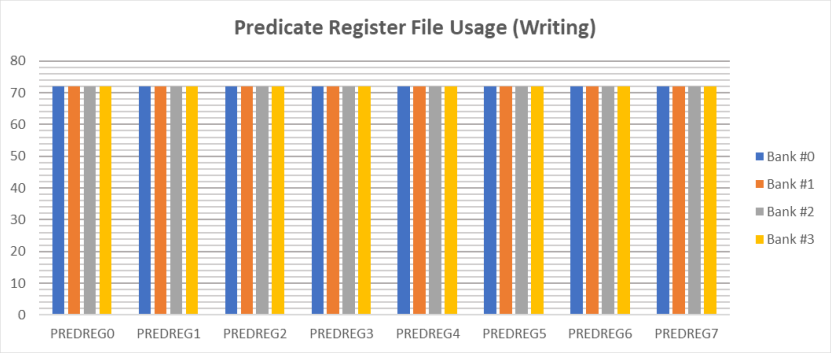


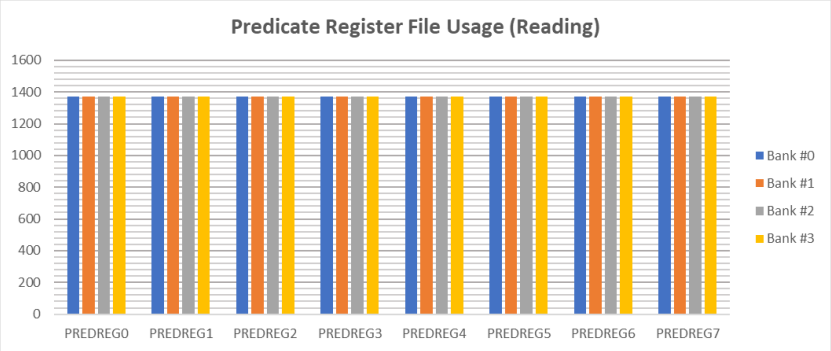
D.12 Address Register File

Module	# writings	# readings
ADDREG0	0	0
ADDREG1	0	0
ADDREG2	0	0
ADDREG3	0	0
ADDREG4	0	0
ADDREG5	0	0
ADDREG6	0	0
ADDREG7	0	0

D.13 Predicate Register File




Predicate Register File Usage (Writing)
80
70
60
50
40
30
20
10
0
PREDREG0
PREDREG1
PREDREG2
PREDREG3
PREDREG4
PREDREG5
PREDREG6
PREDREG7
Bank #0
Bank #1
Bank #2
Bank #3


Predicate Register File Usage (Reading)
1600
1400
1200
1000
800
600
400
200
0
PREDREG0
PREDREG1
PREDREG2
PREDREG3
PREDREG4
PREDREG5
PREDREG6
PREDREG7
Bank #0
Bank #1
Bank #2
Bank #3