NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

THESIS

Reliability Evaluation of Massive Parallel Architectures NVIDIA GPUs on GPGPUsim Simulator

Ioannis G. Avgeros
Dimitrios K. Gkyrtis

Advisor **Dimitrios Gkizopoulos**, Professor

ATHENS

February 2016

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αξιολόγηση Αξιοπιστίας Μαζικά Παράλληλων Αρχιτεκτονικών NVIDIA GPUs στον Προσομοιωτή GPGPUsim

Ιωάννης Γ. Αυγέρος
Δημήτριος Κ. Γκυρτής

Επιβλέπων **Δημήτριος Γκιζόπουλος,** Καθηγητής

ΑΘΗΝΑ

Φεβρουαριος 2016

THESIS

Reliability Evaluation of Massive Parallel Architectures NVIDIA GPUs on GPGPUsim Simulator

Ioannis G. Avgeros
A.M.: 1115201000036
Dimitrios K. Gkyrtis
A.M.: 1115201000032

ADVISOR: **Dimitrios Gkizopoulos,** Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αξιολόγηση Αξιοπιστίας Μαζικά Παράλληλων Αρχιτεκτονικών NVIDIA GPUs στον Προσομοιωτή GPGPUsim

Ιωάννης Γ. Αυγέρος
Α.Μ.: 1115201000036
Δημήτριος Κ. Γκυρτής
Α.Μ.: 1115201000032

ΕΠΙΒΛΕΠΩΝ: **Δημήτριος Γκιζόπουλος,** Καθηγητής

ABSTRACT

The aim of this study is to explore the effects of several hardware faults that happen during the program execution in a Graphics Processing Unit (GPU) after injecting the hardware faults into the GPGPU simulator's program code and executing multiple NVIDIA kernels out of the official NVIDIA GPU computing SDK. The fault injection is completely randomized. By randomized, we mean the process of choosing the register that will be fault injected, the bit that the fault is going to be injected along with the register's thread and the cycle, in transient fault cases.

This study also focuses on the effects that hardware faults has on the program's executional behavior, matching any errors or defects to a category, and analyzing the samples multiple executions results through diagrams and conclusions. There has been a connection on how an error can affect the program's execution based on the program's cycles, the number of the registers and the type of data (either integer values stored in integer variables or integer arrays) that each program uses.

SUBJECT AREA: Computer Architecture
KEYWORDS: Graphics Processing Units (GPUs), CUDA, Reliability Assessment, Fault Injection

ΠΕΡΙΛΗΨΗ

Ο σκοπός αυτής της πτυχιακής είναι η διερεύνηση επιπτώσεων των διαφόρων λαθών υλικού τα οποία παρουσιάζονται κατά την διάρκεια της εκτέλεσης του προγράμματος σε μία μονάδα επεξεργασίας γραφικών (GPU), αφού έχουνε εισαχθεί τα σφάλματα λογισμικού μεσα στον κώδικα του GPGPU προσομοιωτή και μετά την εκτέλεση πολλαπλών προγραμμάτων της NVIDIA από το επίσημο NVIDIA GPU Computing SDK.Τα σφάλματα εισάγονται τυχαία στον προσομοιωτή. Η τυχαιότητα αφορά το ποιος καταχωρητής είναι αυτός που θα επηρεαστεί από την εισαγωγή σφάλματος,το ποια θέση θα είναι αυτή που θα εισαχθεί και το ποιου νήματος θα είναι ο καταχωρητής όπως επίσης και σε ποιον κύκλο θα εισαχθεί το σφάλμα, στην περίπτωση προσωρινού σφάλματος.

Η πτυχιακή επικεντρώνεται επίσης στο πόσο επηρεάζουν τα σφάλματα υλικού την συμπεριφορά του προγράμματος κάτα την εκτέλεση, ταιριάζοντας κάθε λάθος ή ατέλεια σε μία κατηγορία και αναλύοντας τα αποτελέσματα των πολλαπλών εκτελέσεων μέσω διαγραμματών και συμπερασμάτων.Έγινε σύνδεση του κατά πόσο ένα σφάλμα μπορεί να ανιχνευθεί και να επηρεάσει την εκτέλεση του προγράμματος σε σχέση με τον αριθμό των κύκλων ενός προγράμματος, τον αριθμό των καταχωρητών του καθώς και το είδος των δεδομένων (είτε αριθμοί που αποθηκεύονται σε μια μεταβλητή για ακέραιους είτε ένας πίνακας) που το εκάστοτε πρόγραμμα χρησιμοποιεί.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Αρχιτεκτονική Υπολογιστών, Υλικό Υπολογιστών
ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: μονάδες επεξεργασίας γραφικών (GPUs),γλώσσα προγραμματισμού CUDA, μέτρηση αξιοπιστίας, εισαγωγή ελαττωμάτων

ΕΥΧΑΡΙΣΤΙΕΣ

Για τη διεκπεραίωση της παρούσας Πτυχιακής Εργασίας, θα θέλαμε να ευχαριστήσουμε των επιβλέποντα, καθ. Δημήτριο Γκιζόπουλο για την καθοδήγηση, τη συνεργασία και την πολύτιμη συμβολή του στην ολοκλήρωση της.

CONTENTS

FIGURES' INDEX

TABLES' INDEX

1 INTRODUCTION

1.1 Importance of GPU Programming

A many-core graphics processing unit (GPU), also occasionally called visual processing unit (VPU), is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display. GPUs are used in mobile phones, game consoles, personal computers, , workstations, embedded systems and supercomputers(in ascending order of computing power). Modern GPUs are very efficient for computer graphics and image processing, and their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of visual data is done in parallel. In a personal computer, a GPU can be present on a video card, or it can be embedded on the motherboard or—in certain CPUs—on the CPU die. The former case is called a “discrete” GPU while the latter is called a “fused” CPU/GPU architecture

General-purpose computing on graphics processing units (GPGPU) became practical after 2001 with introduction of programmable shaders and floating point support on graphics processors. GPU programming is ideal for problems involving multi-dimensional matrices and vectors. As time goes by, and by the natural laws applied to CPU evolution, slightly declining the expected computing power reached over the years, there has been an increased amount of interest for GPU to increase the performance of computing systems.

Increase in graphics hardware performance and improvements in programmability, has enabled Graphics Processing Units (GPUs) to evolve from a graphics-specific accelerator to a general-purpose computing device. Consequently, GPUs have enjoyed wide-spread adoption in various application domains, including scientific computing [1].

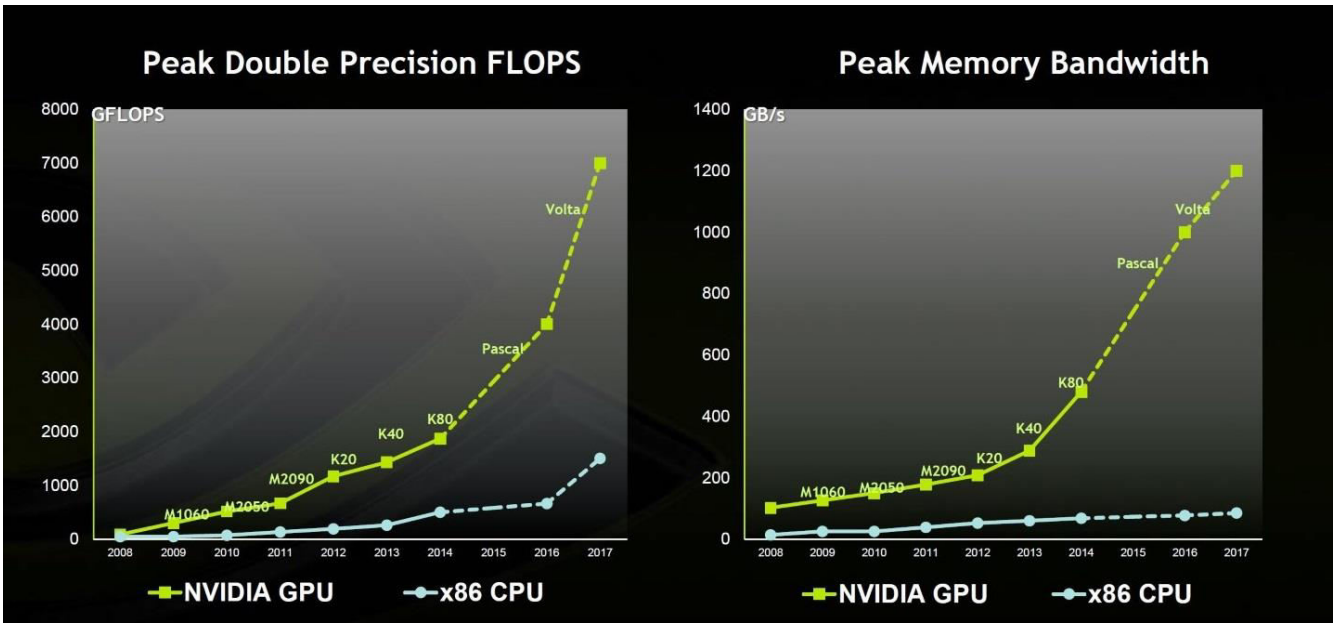


Figure 1: Graph showing the rapid evolution of GPU computation performance compared to the x86 CPU [2].

As GPUs are increasingly used to accelerate applications by allowing more flexibility and programmability, their fault tolerance is becoming much more important than before when they were used only for graphics. In high speed image rendering, the fault in some pixels is not noticeable by human eyes. But, nowadays that GPUs are used for general programming use (for DNA sequencing and other problems where computing correctness is critical) a hardware error [3] might lead to unpredicted behavior, or erroneous program output. Hardware faults may not lead to immediate program failure. A single, small hardware failure may often go undetected until it leads to more serious failures or it may be completely masked leaving program execution unaffected.

With the current microprocessor fabrication trends, i.e. smaller feature sizes, lower voltages and faster clock frequencies, microprocessors are becoming increasingly susceptible to hardware failures. Microprocessors can be protected against failures by implementing some form of redundancy [4, 5]. Thus, when a failure occurs, it is masked by the redundancy, keeping the microprocessor functioning as though the failure did not take place [6].

While graphics processing units (GPUs) have gained wide adoption as accelerators for general-purpose applications (GPGPU), the end-to-end reliability implications of their use have not been quantified. Fault injection is a widely used method for evaluating the reliability of applications. However, building a fault injector for GPGPU applications is challenging due to their massive parallelism, which makes it difficult to achieve representativeness being time-efficient. [7]

1.2 PTX

PTX (shortening for a low-level parallel thread execution) is a virtual machine and instruction set architecture (ISA). PTX provides a stable programming model and instruction set for general purpose parallel programming. PTX is an intermediate language, as already mentioned, that is designed to be portable across multiple GPU architectures. It gets compiled by the compiler component PTXAS into final machine code, also referred to as SASS, for a particular GPU chip and architecture. A PTX program specifies the execution of a given thread of a parallel thread array(PTA).

A cooperative thread array, or CTA, is an array of threads that execute a kernel concurrently or in parallel. Threads within a CTA can communicate with each other. To coordinate the communication of the threads within the CTA, one can specify synchronization points where threads wait until all threads in the CTA have arrived. Each thread has a unique thread identifier within the CTA. Programs use a data parallel decomposition to partition inputs, work, and results across the threads of the CTA. Each CTA thread uses its thread identifier to determine its assigned role, assign specific input and output positions, compute addresses, and select work to perform. Each thread identifier component ranges from zero up to the number of thread ids in that CTA dimension.

Threads within a CTA execute in SIMT (single-instruction, multiple-thread) fashion in groups called warps. A warp is a maximal subset of threads from a single CTA, such that the threads execute the same instructions at the same time. Threads within a warp are sequentially numbered. The warp size is a machine-dependent constant. Typically, a warp has 32 threads. Multiple CTAs may execute concurrently and in parallel, or sequentially, depending on the GPU chip. Each CTA has a unique CTA identifier (ctaid) within a grid of CTAs. Each grid of CTAs has a 1D, 2D, or 3D shape specified by the parameter nctaid. Each grid also has a unique temporal grid identifier (gridid).

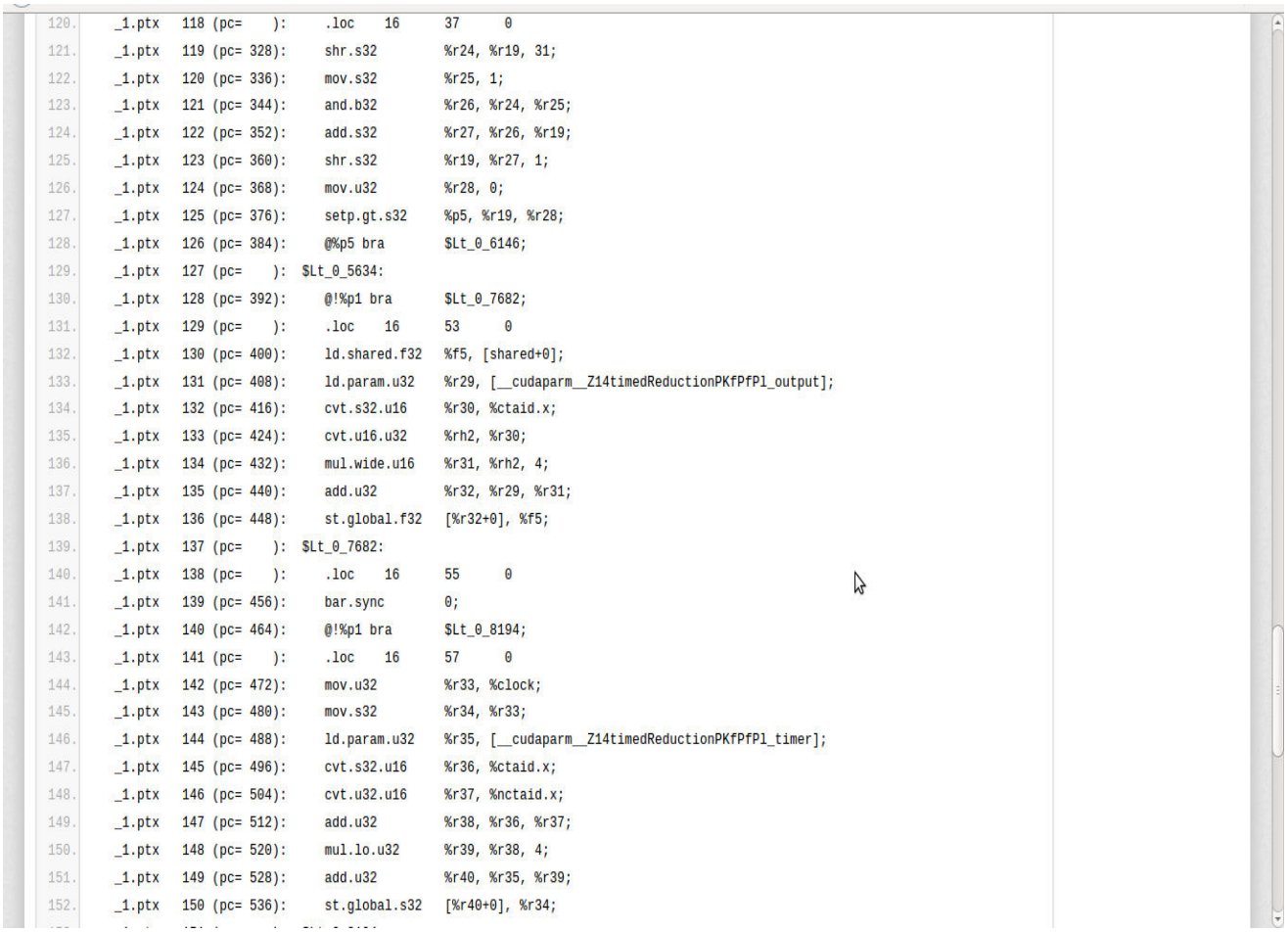
PTX contains a set of registers for general purpose uses. We will explain their use here:

a) ***%tid***: The thread identifier is a three-element vector tid, (with elements tid.x, tid.y, and tid.z) that specifies the thread's position within a 1D, 2D, or 3D CTA

b) ***%rh***: are “half registers”. They are 16 bit registers used to save space when we need to store data only in a short number of bits.

c) ***%ctaid***: CTA identifier within a grid. The %ctaid special register contains a 1D, 2D, or 3D vector, depending on the shape and rank of the CTA grid.

d) ***%r, %p, %f***: Registers used for local store of unsigned values, predicate logic(not only) and local store of floating variables respectively.



```
120.    _1.ptx   118 (pc=    ):     .loc    16      37      0
121.    _1.ptx   119 (pc= 328):     shr.s32         %r24, %r19, 31;
122.    _1.ptx   120 (pc= 336):     mov.s32         %r25, 1;
123.    _1.ptx   121 (pc= 344):     and.b32         %r26, %r24, %r25;
124.    _1.ptx   122 (pc= 352):     add.s32         %r27, %r26, %r19;
125.    _1.ptx   123 (pc= 360):     shr.s32         %r19, %r27, 1;
126.    _1.ptx   124 (pc= 368):     mov.u32         %r28, 0;
127.    _1.ptx   125 (pc= 376):     setp.gt.s32     %p5, %r19, %r28;
128.    _1.ptx   126 (pc= 384):     @%p5 bra        $Lt_0_6146;
129.    _1.ptx   127 (pc=    ):  $Lt_0_5634:
130.    _1.ptx   128 (pc= 392):     @!%p1 bra       $Lt_0_7682;
131.    _1.ptx   129 (pc=    ):     .loc    16      53      0
132.    _1.ptx   130 (pc= 400):     ld.shared.f32   %f5, [shared+0];
133.    _1.ptx   131 (pc= 408):     ld.param.u32    %r29, [__cudaparm__Z14timedReductionPKfPfPl_output];
134.    _1.ptx   132 (pc= 416):     cvt.s32.u16     %r30, %ctaid.x;
135.    _1.ptx   133 (pc= 424):     cvt.u16.u32     %rh2, %r30;
136.    _1.ptx   134 (pc= 432):     mul.wide.u16    %r31, %rh2, 4;
137.    _1.ptx   135 (pc= 440):     add.u32         %r32, %r29, %r31;
138.    _1.ptx   136 (pc= 448):     st.global.f32   [%r32+0], %f5;
139.    _1.ptx   137 (pc=    ):  $Lt_0_7682:
140.    _1.ptx   138 (pc=    ):     .loc    16      55      0
141.    _1.ptx   139 (pc= 456):     bar.sync        0;
142.    _1.ptx   140 (pc= 464):     @!%p1 bra       $Lt_0_8194;
143.    _1.ptx   141 (pc=    ):     .loc    16      57      0
144.    _1.ptx   142 (pc= 472):     mov.u32         %r33, %clock;
145.    _1.ptx   143 (pc= 480):     mov.s32         %r34, %r33;
146.    _1.ptx   144 (pc= 488):     ld.param.u32    %r35, [__cudaparm__Z14timedReductionPKfPfPl_timer];
147.    _1.ptx   145 (pc= 496):     cvt.s32.u16     %r36, %ctaid.x;
148.    _1.ptx   146 (pc= 504):     cvt.u32.u16     %r37, %nctaid.x;
149.    _1.ptx   147 (pc= 512):     add.u32         %r38, %r36, %r37;
150.    _1.ptx   148 (pc= 520):     mul.lo.u32      %r39, %r38, 4;
151.    _1.ptx   149 (pc= 528):     add.u32         %r40, %r35, %r39;
152.    _1.ptx   150 (pc= 536):     st.global.s32   [%r40+0], %r34;
```

Figure 2:A typical ptx file generated by the GPGPUsim simulator.

The aim of this thesis is to investigate and explore the effects of semi-automated fault injection into different programs using three types of fault injection also known as fault insertion testing from certain safety standards. The idea behind fault injection is to accelerate the occurrence of faults in the system to evaluate its behavior under the influence of anticipated faults, and to evaluate error handling mechanisms [8]. Before explaining the three types of fault injection, we categorize the type of hardware faults as permanent and transient.

a)***Permanent Faults***: : Faults that remain in a register throughout the execution of a program and beyond that; a bit of the register is persistently stuck at a specific value (0 or 1) throughout the execution of the program

b)***Transient Faults****:* Faults that happen in one cycle and exist until another value is being written over them. A bit is flipped from zero to one or from one to zero for one cycle during program's execution.

1.3 Fault Injection

1.3.1 Fault Injection Operators

The three operators we use to inject faults:

a) **XOR operator (bit flip)**: Flips (inverts) the value of a single bit in a random position of a register. The bit remains flipped until it is overwritten with a new value. This type is transient fault (also known as *soft error*).

number ^= 1 << x;

That will toggle bit x.

b) **OR operator**: Performs the "OR" operation in a random bit. "OR" is the operation that results in each input is 0 if both bits are 0. This type is a permanent fault.

number |= 1 << x;

That will set bit x. We are going to refer to this category as stuck-at-1.

c) **AND operator**: Performs the AND operations in a random bit AND is the operation that a true output results if one, and only one, of the inputs is true. This is also a permanent fault.

- number &= ~(1 << x);

That will clear bit x. We must invert the bit string with the bitwise NOT operator (~), then perform the AND operator on it. We are going to refer to this category as stuck-at-0.

1.3.2 Type of Errors

We have categorized the type of errors produced after the program's execution under the presence of a faulty bit (transient of permanent fault) to the five following categories:

Masked: There are cases were the injection fault does not have any effect in the program output, leaving the output exactly the same as in a fault-free program execution. It may happen because the register access did not alter the value or if in a sample execution, basic criteria are not met (specific cycle in TRANSIENT FAULT case). That case is called Masked.

Silent Data Corruption (SDC): The faulty register may have effects on the output of a program. This error requires extra debugging as there should be at least one check at the regular output of one program compared to the one that has been fault injected other.

Crash: The program is abnormally terminated. That means that the altered register holds, for example, the address of a variable or the index of an array.

Slowdown: The execution time of the program is more that the fault-free time, meaning that there is a growth in the number of cycles for program execution. The output of the program remains unchanged.

Speedup: There is a reduction in the execution time of the program, meaning by that, that there is reduce in the number of cycles the program did. The output of the program remains unchanged.

Timeout: The program falls in an infinite loop or other erroneous behavior which does not allow it to complete. For this thesis we define the “infinite loop” as the program executing for more than twice the normal (fault free) execution time.

These are the main 5 categories we have chosen to differentiate any irregular case in a sample execution while the program is fault-injected. Categories are mutually exclusive.

1.3.4 Fault Injection in the GPGPUsim Simulator

As far as our approach for the fault injection is concerned, we altered the GPGPU simulator C++ code to facilitate injection of permanent and transient faults. We began, by accessing all the files responsible for the simulation, especially those who were responsible for storing data into registers, those who were printing out the number of execution cycles and those who were handling the information of threads.

The function in charge for storing data into a register isthe ptx_thread_info::set_operand_value located under src/cuda-sim/instructions.cc. This function has four parameters:

- ***operand_info &dst***: A wrapper class containing a source operand for an instruction which may be either a register identifier, a memory operand (including displacement mode information), or an immediate operand.

- ***ptx_reg_t &data***: The data to be stored represented by union ptx_reg_t which holds multiple memory types (unsigned 32, 64 lower/upper bits et cetera).

- ***ptx_instruction *pI****:* Contains the full state of a dynamic instruction including the interfaces required for functional simulation.

- ***ptx_thread_info *thread***: Contains functional simulation state for a single scalar thread (work item in OpenCL). This includes the following:

- Register value storage
- Local memory storage
- Shared memory storage
- Program counter (PC)
- Call stack

- Thread IDs (the software ID within a grid launch, and the hardware ID indicating which hardware thread slot it occupies in timing model) The most important members of this class are the ones that hold information about the thread IDs:
  - unsigned m_hw_sid
  - unsigned m_hw_tid
  - unsigned m_hw_wid
  - unsigned m_hw_ctaid

They are multiple useful structures here for our study. The register is the “dst” parameter, the data to be stored is the data parameter and the thread information we were seeking is the thread parameter.

We had to follow different approaches regarding the faults we wanted to inject. All approaches have a common starting point. There is only one time we set the variables for the injection. The random cycle variable, random register variable, along with other crucial variables, all of them are initialized only once. This is achieved by the global, boolean variable randtime that after the first time that all required variables are initialized, they never change again. It is the only point where all three of the fault injection categories have the same approach.

The XOR operator requires to be used only once in a specific cycle. So, we need to measure the execution cycles. The print_simulation_time() located in src/gpgpusim_entrypoint.cc prints general statistics about the program, such as instructions per second, instructions executed and number of cycles. We execute the program once without fault injection and we store the amount of cycles. Back to the specific cycle aspect, the first time set_operand_value is called there is a rand function called, divided by the cycles we have plus one and keep the remainder into RandCycle variable, along with the RandRegister, which is the random index into the character 2 dimension array variable we have for the registers. Last, but not least we also keep the random position for bit-flip into RandPosition variable, which normally is a rand function divided by 64(number of bits for the data) and keeping the remainder

```
if(RandTime==1)
    {
        srand (time(NULL));
        RandTime=0;
        srand (time(NULL));
        randRegister=rand()%85;
        srand (time(NULL));
        randPosition=rand()%64;
        srand (time(NULL));
        randCycle=rand()%5637;
    }
```

Figure 3: The stuck-at-1 case of variable initialization.

The AND/OR operator (bit in a register stuck at 0 and 1, respectively) also requires the RandPosition variable and the RandRegister. What is also required is four extra variables, named mySid, myWid, myTid, myCtaid holding the random thread's ids with the register malfunction. The two operands affect only one register of one thread, so there comes a point where we need to identify and separate each unique thread.

```
if(RandTime==1)
    {
        srand (time(NULL));
        RandTime=0;
        mySid=(rand()%6)*3;
        srand (time(NULL));
        myTid=rand()%512;
        srand (time(NULL));
        myWid=rand()%16;
        srand (time(NULL));
        myCtaid=0;
        srand (time(NULL));
        randRegister=rand()%85;
        srand (time(NULL));
        randPosition=rand()%64;
    }
```

Figure 4: The stuck-at-1/stuck-at-0 case of RandTime 1.

After the initialization, we proceed with the injection. Due to the difference between transient fault and stuck-at-0/stuck-at-1, we follow two different strategies.

The first one is that when we have the transient fault case, we perform the following check:

If(flipBit==0 && strcmp(name.c_str(),registers[randRegister])==0 && gpu_sim_cycle==randCycle)

```
        if(gpu_sim_cycle==randCycle)
        {

           if(flipBit==0 && strcmp(name.c_str(),registers[randRegister])==0)

             {
                 flipBit=1;

                 setValue.u64 ^=  1 << randPosition;

             }
      }
       set_reg(dst.get_symbol(),setValue);
   }
```

Figure 5: Transient fault case injection along with the check we perform.

FlipBit is responsible for allowing only once the access to the register. When the code reaches the part for the injection, it changes the value of flipbit to 1. The name.c_str() is the name of the current register, and we compare it with the random register. The last check is the current cycle represented by the global variable gpu_sim_cycle, compared to ours RandCycle, as stuck-at-1 is used for injection in one cycle.

On the other hand the stuck-at-1/stuck-at-0 case has slightly different check.

if(strcmp(name.c_str(),registers[randRegister])==0 && mySid==thread->m_hw_sid && myTid==thread->m_hw_tid)

There are two main checks. The first one is again comparing current register with the random one. The second one, though, checks whether the current thread matches with the random thread as the bit stuck in a register pertains to only one thread.

After the injection, there is a function call to set_reg. Set reg is a function responsible for assigning the value (here is setValue) to each register. The setValue is a variable of ptx_reg_t type. Ptx_reg_t is a union, with members that save memory data for each register, such as upper and lower bits, possible float or unsigned value etc. Set_reg copies the information calculated or passed from the set_operand_value function into the desired register.

Figure 6: : The check and the injection with the AND and the OR case (inside comments).

```
void ptx_thread_info::set_reg( const symbol *reg, const ptx_reg_t &value )
{
   assert( reg != NULL );
   if( reg->name() == "_" ) return;
   assert( !m_regs.empty() );
   assert( reg->uid() > 0 );
   m_regs.back()[ reg ] = value;
   if (m_enable_debug_trace )
      m_debug_trace_regs_modified.back()[ reg ] = value;
   m_last_set_operand_value = value;
}
```

Figure 7: set_reg function.

What we should add here is the approach we followed after the injection as far as the transient fault is concerned. The register with the fault injection must hold the value for only one cycle. There are two different cases where the register might be used again. The first one is the case where the register is the one that needs to be read in order to load its value from the memory, while the second one is the case where the register is the one that needs to be written in order to store the result of an instruction.

The simulator has already a function implemented for load register values from memory. It is called get_operand_value and the parameters of these functions are similar to the one for set_operand_value.

We need to store the thread information along with the previous value that would have been stored if we did not fault inject the register. This happens in the set_operand_value function:

```
if(gpu_sim_cycle==randCycle && strcmp(name1.c_str(),registers[randRegister])==0 && flipBit==0)
     {
     prevValue=data;
     mySid=thread->m_hw_sid;
     myTid=thread->m_hw_tid;
     myWid=thread->m_hw_wid;
     myCtaid=thread->m_hw_ctaid;
     }
```

Figure 8: Thread information and the value to be written in the register.

Every time the get_operand_value is called, we perform a check. The check consists of 2 basic blocks. The first one is whether we have already performed the injection, and if we did, if the current cycle is different from the one that the fault injection happened and makes sure that the write back operation of the injected value happens only once. The second block checks whether we have the same thread as the

one we fault injected and about having the same register. If the above conditional statements are satisfied, then we write the previous value in the register.

```
if(randCycle!=0 && gpu_sim_cycle>randCycle && getRandTime==1 && flipBit==1) // After
        {
        if(mySid==thread->m_hw_sid && myTid==thread->m_hw_tid && myWid==thread->m_hw_wid && thread->m_hw_ctaid==myCtaid &&
        strcmp(name.c_str(),registers[randRegister]) )
        {

                getRandTime=0;
                set_reg(op.get_symbol(),prevValue);

        }

        }
```

Figure 9: The basic check in get operand value function.

As far as the second case of re accessing a register is concerned, we didn't have to change anything. If the register is not used between the cycle that the fault injection happened, then consecutive writes on it will erase the previous value with the new one .We should mention here, though, that this is a pretty rare case as the register allocation ensures that there will be no dead variables, thus, no dead code . Dead variable is the variable that is never used after defined. Dead code is when the variable of a computation is never used.

An example of a dead code is as it follows:

```
x=y+1;
  y=1;
x=2*z;
```

The first x variable is a dead variable as it stores the addition result but it is never being actually used, while the next use of the x variables is again for storing operation, thus overwriting the previous value.

1.4 Noticeable Cases

There were a number of cases we came up a lot during our study, and we are going to explain and elaborate them here. Mostly each one of the presented samples projected the cases explained below.

Firstly, mostly in our programs we noticed that they obey a general rule, that the stuck-at-0 case has significantly less error cases (cases that belong into in any other category than the masked one) than the stuck-at-1 case .This happens mainly because the injection is random in a 64 bit variable. The contents of the registers are small numbers, usually using up to 7 or 8 bits. The random bit, thus, is more likely to be a zero rather one. stuck-at-0 case keeps a specific bit stuck into zero value through the whole program execution, so the most probable scenario we have here is a bit already in zero value not being altered by the stuck-at-0 case. We should call this one the "failed stuck-at-0", failed because no matter that the register was accessed and injected

with the stuck-at-0 operator, there was no change in register's value(Previous values of register remains unchanged after the stuck-at-0 operation).

Secondly, we should mention here that the GPGPU Simulator is a tested program that the number of cycles the program has remain stable in normal execution, without fault injection. The number of cycles is always the same, a focal point for us, as we needed the executions that had different cycles than the normal ones to be affected ONLY by the injection.

Thirdly, the registers are randomly selected and all have equal chances of getting picked for injection. For example, for 2000 execution of a program with 20 registers every register has a 5% chance to be selected and there is an equality in the number of each register getting selected as the fault-injected register.

As far as statistics are concerned, we decided to count the most severe fault that appear, as it was possible to have one execution with multiple errors. The most regular cases were the combination of SDC and slowdown, where we counted this case as an SDC and we therefore mention the slowdowns that happened along with the SDC. The severities of the cases are as follows:

1. Timeout/Crash
2. SDC
3. Slowdown
4. Masked

2 EXAMPLES

2.1 Sample Introduction

The NVIDIA GPU Computing SDK consisted of more than 50 programs. For the purposes of this study and due to the large number of executions we performed, the chosen programs had an average time of execution around 8-10 seconds. We selected the following benchmarks for our experiments:

- **VectorAdd**: Two vectors initialized and passed as arguments to kernel function where an addition on each element of the vector is performed, and the result is stored into a third vector.
- ***Clock*:** This CUDA function computes a standard parallel reduction and evaluates the time it takes to do that for each block.
- ***DwtHaar1D***: Implements the Haar wavelet transform in a discrete wavelet.
- ***CppIntegration***: Simple test kernel for device functionality .Parameter is memory to process (in and out).
- ***MatrixMul***: Performs matrix multiplication of two vectors and stores the output to a third one.
- ***SimpleStreams***: This sample illustrates the usage of CUDA streams for overlapping kernel execution with device/host memcopies.
- ***SimpleTemplates***:. This sample is a templatized version of the template project. It also shows how to correctly templatize dynamically allocated shared memory array
- ***Templates***: A simple template project that can be used as a starting point to create new CUDA projects.
- ***SimpleVoteIntrinsics***: Consisted of two kernels. Kernel #1 tests the across-the-warp vote (any) intrinsic. Kernel #2 tests the across-the-warp vote (all) intrinsic.

All of the above programs are tested through normal execution for keeping the number of execution cycles stable. The programs have an average of two and a half thousand cycles and mostly the programs have thirty to forty registers each.

$$\mathbf{A}+\mathbf{B}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2n}+b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \cdots & a_{mn}+b_{mn} \end{bmatrix}$$

Figure 10: Vector addition.

The original NVIDIA program created two random vectors. The vectors were passed as arguments from the host function to the device function. The device function, at first, calculates the index the thread is responsible to calculate and store the result of the addition. The condition check whether the index is less than the number of elements the vector have.

```
// Device code
__global__ void VecAdd(const int* A, const int* B, int* C, int N,int *accessed)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
    {
        C[i] = A[i] + B[i];
    }

}
```

Figure 11: The VectorAdd kernel function.

We altered the source code, reducing the amount of elements the vector contained from ten thousands to 512 elements. We removed the original function that initialized the two vectors with random elements, and replaced the values of each position with their index, e.g. A[0]=0, A[1]=1 et cetera, to check after every execution whether the result-vector contains the correct elements. A for loop added after the call to CUDA function to check all the elements in result-vector and find all possible errors. The loop iterates through the entire vector and checks whether the results in the specific index is two times the index (position 256, A[256]=256 and B[256]=256, expected result C[256]=512). If the result is different, we have a silent data corruption case of error.

```
while(i<N)
{
    if(h_C[i]!=2*i)
    {
        printf("ERROR:h_C[%d]=%d\n",i,h_C[i]);
        check=check+1;
    }
i++;

}
```

Figure 12: Silent Data Corruption check in VectorAdd sample.

2.2.1 Vector Add Results

2.2.1.1 VectorAdd Statistics

Registers

- %tid.x
- %ntid.x
- %ctaid,%p1
- %rh1,%rh2

Number of registers: 18

Cycles: 711

Table 1:VectorAdd Execution results.

	Transient Fault	Stuck-at-0	Stuck-at-1
Masked	1973	1981	1807
SDC	25*	19**	188***
Slowdown	2	-	5
Crash	-	-	-
Timeout	-	-	-
Injections	2000	2000	2000

*16 slowdowns happened at the same program execution.

**10 slowdowns happened at the same program execution.

***61 slowdowns happened at the same program execution.

Table 2: The slowdown table that shows the number of extra cycles and the number of sample executions that those extra cycles occurred.

Extra Cycles	+1	+2	+3	+5	+7	+9	+12	+24	+36
Number of executions	24	8	13	9	2	5	4	4	2

2.2.1.2 Chart

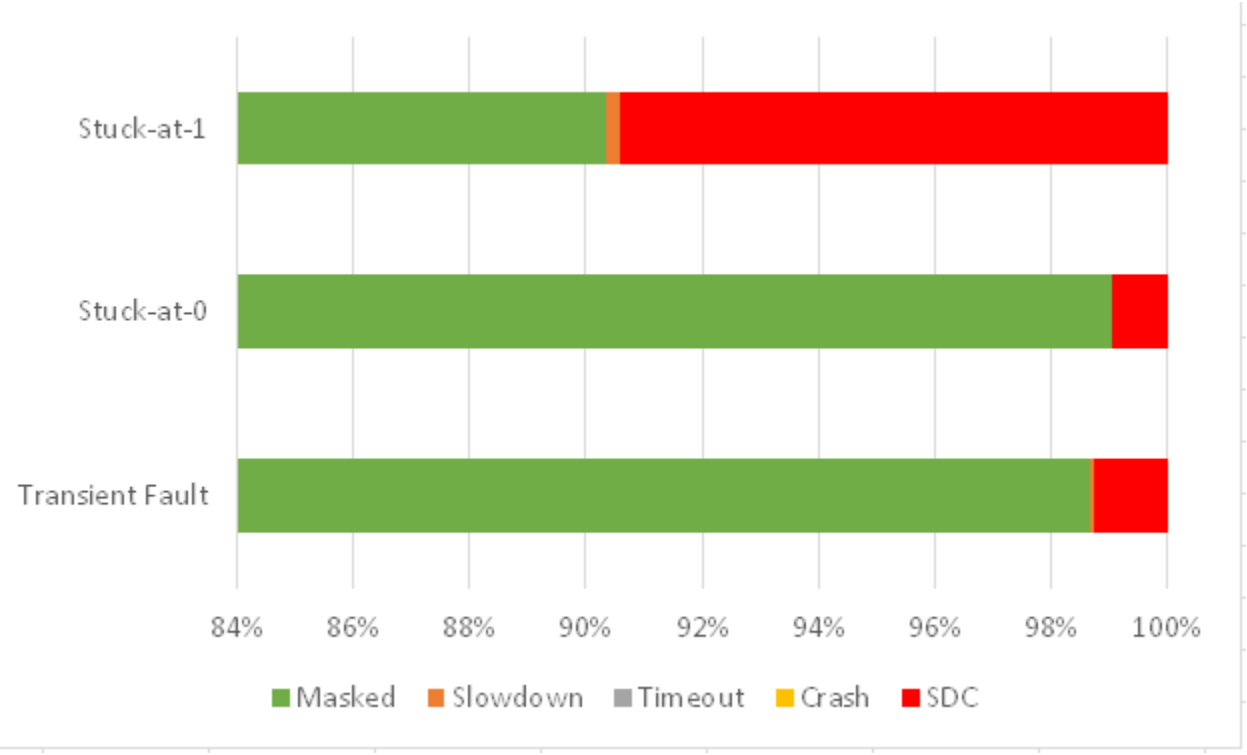


Figure 13: Chart for VectorAdd.

2.2.1.3 Comments

There were no crash or timeout cases in this program. This happened because there was a check inside kernel code that made sure that the program is not going to be over the bounds of the matrix that holds the addition result of the other two vectors. There is a general rule that applies to almost all the programs that the stuck-at-0 case has significantly less error cases than the stuck-at-1 case we have already mentioned.

Another useful notice is that one every three SDC faults we have a slowdown, where the program is executed .An SDC alters the value of a register and sometimes it cause the program to execute more cycles due to changed value of the one variable we have for keeping the index of the vectors. A change into that index may force the program to have more access cycles into the memory for retrieving data. In a normal execution, the program, based on locality by reference, would retrieve the specified object in the index along with the ones that are closer to it based on memory address. From the moment that we alter the index of the vector, the object may have not been already brought by the program and extra communication may exist with memory, contributing to the slowdown case.

What is also worth noted is that we have a big number of silent data corruption cases. As we can clearly see from the above chart (figure 2.4), almost one every 10 execution's the program execution ended with a result in the data that was different from what the original output was.

In addition, this is an execution where the results of the transient fault case and the results of stuck-at-0 present many similarities, mostly regarding the number of faults for the SDC category. The vectorAdd calculates the memory offset for three different arrays. The majority of the registers hold address value. These values are big numbers with many digits, thus making the always zero bit to actually have an effect on the execution of the program and causing a lot of silent data corruption errors. In general, if a kernel's variables contain a lot of memory information, there is a high possibility the number of the silent data corruption in transient faults category and the stuck-at-0 to be similar.

```
_1.ptx    60 (pc=    ):  $LDWbegin__Z6VecAddPKiS0_PiiS1_:
_1.ptx    61 (pc=    ):     mov.u16         %rh1, %ctaid.x;
_1.ptx    62 (pc=   8):     mov.u16         %rh2, %ntid.x;
_1.ptx    63 (pc=  16):     mul.wide.u16    %r1, %rh1, %rh2;
_1.ptx    64 (pc=  24):     cvt.u32.u16     %r2, %tid.x;
_1.ptx    65 (pc=  32):     add.u32         %r3, %r2, %r1;
_1.ptx    66 (pc=  40):     ld.param.s32    %r4, [__cudaparm__Z6VecAddPKiS0_PiiS1__N];
_1.ptx    67 (pc=  48):     setp.le.s32     %p1, %r4, %r3;
_1.ptx    68 (pc=  56):     @%p1 bra        $Lt_0_1026;
_1.ptx    69 (pc=    ):     .loc    28      54      0
_1.ptx    70 (pc=  64):     mul.lo.u32      %r5, %r3, 4;
_1.ptx    71 (pc=  72):     ld.param.u32    %r6, [__cudaparm__Z6VecAddPKiS0_PiiS1__A];
_1.ptx    72 (pc=  80):     add.u32         %r7, %r6, %r5;
_1.ptx    73 (pc=  88):     ld.global.s32   %r8, [%r7+0];
_1.ptx    74 (pc=  96):     ld.param.u32    %r9, [__cudaparm__Z6VecAddPKiS0_PiiS1__B];
_1.ptx    75 (pc= 104):     add.u32         %r10, %r9, %r5;
_1.ptx    76 (pc= 112):     ld.global.s32   %r11, [%r10+0];
_1.ptx    77 (pc= 120):     add.s32         %r12, %r8, %r11;
_1.ptx    78 (pc= 128):     ld.param.u32    %r13, [__cudaparm__Z6VecAddPKiS0_PiiS1__C];
_1.ptx    79 (pc= 136):     add.u32         %r14, %r13, %r5;
_1.ptx    80 (pc= 144):     st.global.s32   [%r14+0], %r12;
_1.ptx    81 (pc=    ):  $Lt_0_1026:
_1.ptx    82 (pc=    ):     .loc    28      57      0
_1.ptx    83 (pc= 152):     exit;
_1.ptx    84 (pc=    ):  $LDWend__Z6VecAddPKiS0_PiiS1_:
```

Figure 14: PTX file for VectorAdd program.

As we can see from the above picture, the registers responsible for calculating the index (%r5) are eight out of 18. That means that almost half of the registers are directly involved into the index calculation. A change in one of those registers is highly possible to affect the number of cycles increasing the communication traffic with the memory. Furthermore, even the rest registers hold memory-related values, increasing the possibility to have a slowdown in our execution.

To sum up, the most important thing we would like to mention here is the strong correlation between the SDC error and the slowdown. There is a ratio 1/3 (1 slowdown every 3 SDC'S) resulted by the fact that almost all the registers hold memory related values.

2.3 Clock

This kernel computes a standard parallel reduction and evaluates the time it takes to do that for each block. The process of combining multiple parallel threads' results into one overall result is called reduction [9]. Here the reduction operation is addition, or sum, and we refer to the reduction as a sum-reduce. (Other programs would use other reduction operations as part of the same reduction pattern.)

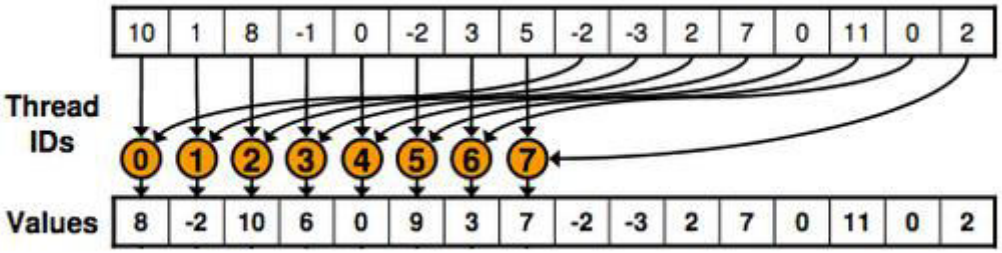


Figure 15: Example of parallel reduction.

In this code, parallel reduction is implemented as follows: half of the threads will perform the reading from global memory and writing to shared memory, as showed in the picture.

You execute a kernel, and now you want to reduce some values, you limit the access the code above to only to half of the total of threads running. Each step requires half the threads the previous required. The timing results are stored in device memory.

```
// in device memory.
__global__ static void timedReduction(const float * input, float * output, clock_t * timer)
{
    // __shared__ float shared[2 * blockDim.x];
    extern __shared__ float shared[];

    const int tid = threadIdx.x;
    const int bid = blockIdx.x;

    if (tid == 0) timer[bid] = clock();

    // Copy input.
    shared[tid] = input[tid];
    shared[tid + blockDim.x] = input[tid + blockDim.x];

    // Perform reduction to find minimum.
    for(int d = blockDim.x; d > 0; d /= 2)
    {
        __syncthreads();

        if (tid < d)
        {
            float f0 = shared[tid];
            float f1 = shared[tid + d];

            if (f1 < f0) {
                shared[tid] = f1;
            }
        }
    }
```

Figure 16: The clock kernel function.

After the call to the device function, the host's main function calculates, in a “for loop”, the minimum and maximum time and stores the result into two variables maxEnd and minStart.

```
for (int i = 1; i < NUM_BLOCKS; i++)
{
    minStart = timer[i] < minStart ? timer[i] : minStart;
    maxEnd = timer[NUM_BLOCKS+i] > maxEnd ? timer[NUM_BLOCKS+i] : maxEnd;
}
```

Figure 17: MaxEnd and minStart variables.

The check that shows whether the output is the expected one is shown in the next image:

```
if (maxEnd-minStart!=4263) printf("CHECK=1 WRONG TIME");
```

Figure 18: Check for SDC case.

The subtraction of maxEnd and minStart must produce 4263. If the data stored in a register holding minStart or maxEnd has changed, then the program prints the error message(SDC case).

2.3.1 Clock Results

2.3.1.1 Program Statistics

Registers:

- %r1->%r40
- %f1->%f5
- %p1->%p5

Number of registers: 50

Cycles: 4630

Table 3: Clock execution results.

	Transient Fault	Stuck-at-0	Stuck-at-1
Masked	1962	2000	1999
SDC	-	-	1**
Slowdown	12	-	-
Crash	26	-	-
Timeout	-	-	-
Number Of Injections	2000	2000	2000

.**Along with one slowdown.

Table 4: The slowdown table that shows the number of extra cycles and the number of sample executions that those extra cycles occurred.

Extra Cycles	7	9	98	137	180	586	719
Number of executions	2	4	3	1	1	1	1

2.3.1.2 Chart

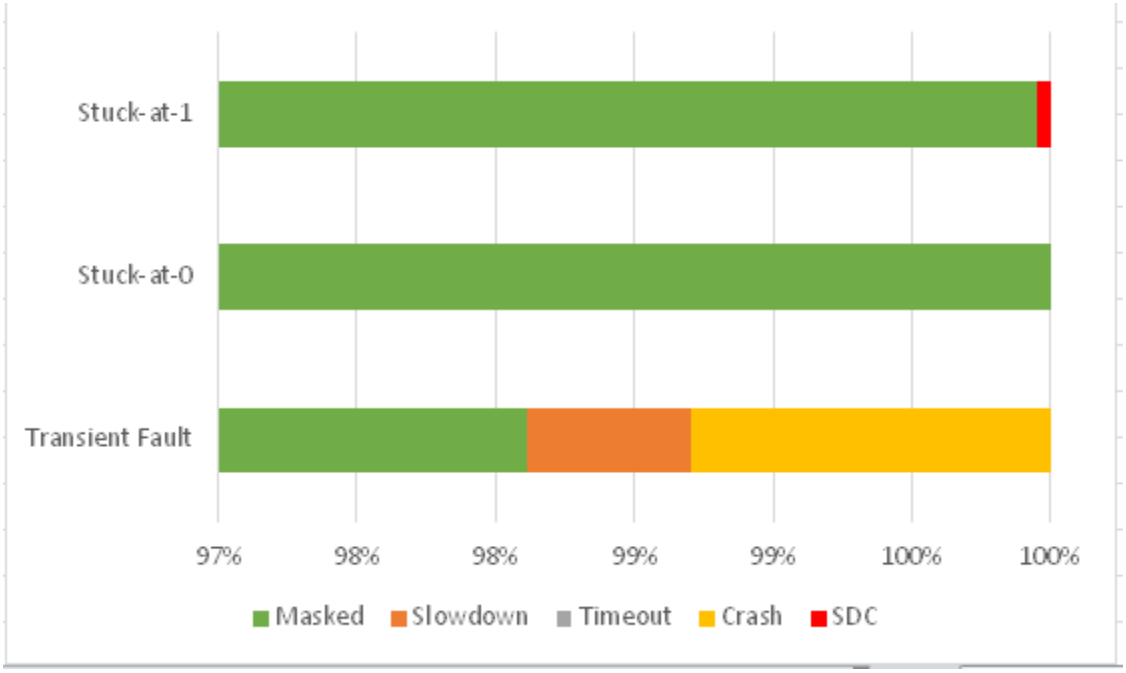


Figure 19: Chart for clock sample.

2.3.1.3 Comments

Clock is a program that computes the minimum and maximum time. It is quite obvious why the amount of SDC errors in those cases all combined are less than other programs. The injected fault would lead to an SDC error only if there is a big increase in the value of the registers that hold the max or minimum value or if the changed value of a temporary variable leads to altered maxEnd and minStart. Furthermore, we noticed that during our executions that clock operates with a large number of threads. Fault injecting a random thread that has access to a specific register (OR/AND cases) is very difficult and this is why we have only one error in a combined 4000 executions.

The few errors we had during the clock execution can also be imaged in the above chart (figure 2.9).The bars begin the percentage from 97 percent, leading to the observation that in all three cases the number of faults in an execution is below 3%.

Lastly, we should mention here that in the first case there are a big number of crash cases.. The 26 crashes happened mainly due to an implemented function called to clean the state of the program .From the moment that we check the kernel execution's result and compared to what it should be, the program called built in function cudaDeviceReset(). CudaDeviceReset causes the driver to clean up all state .The function is needed to ensure correct operation when the application is being profiled. Calling cudaDeviceReset causes all profile data to be flushed before the application exits.

2.4 DwtHaar1d

A **discrete wavelet transform** (DWT) is any wavelet transform for which the wavelets are discretely sampled. As with other wavelet transforms, a key advantage it has over Fourier transforms is temporal resolution: it captures both frequency *and* location information (location in time). For an input represented by a list of numbers, the Haar wavelet transform may be considered to pair up input values, storing the difference and passing the sum. This process is repeated recursively, pairing up the sums to provide the next scale, which leads to differences and a final sum.

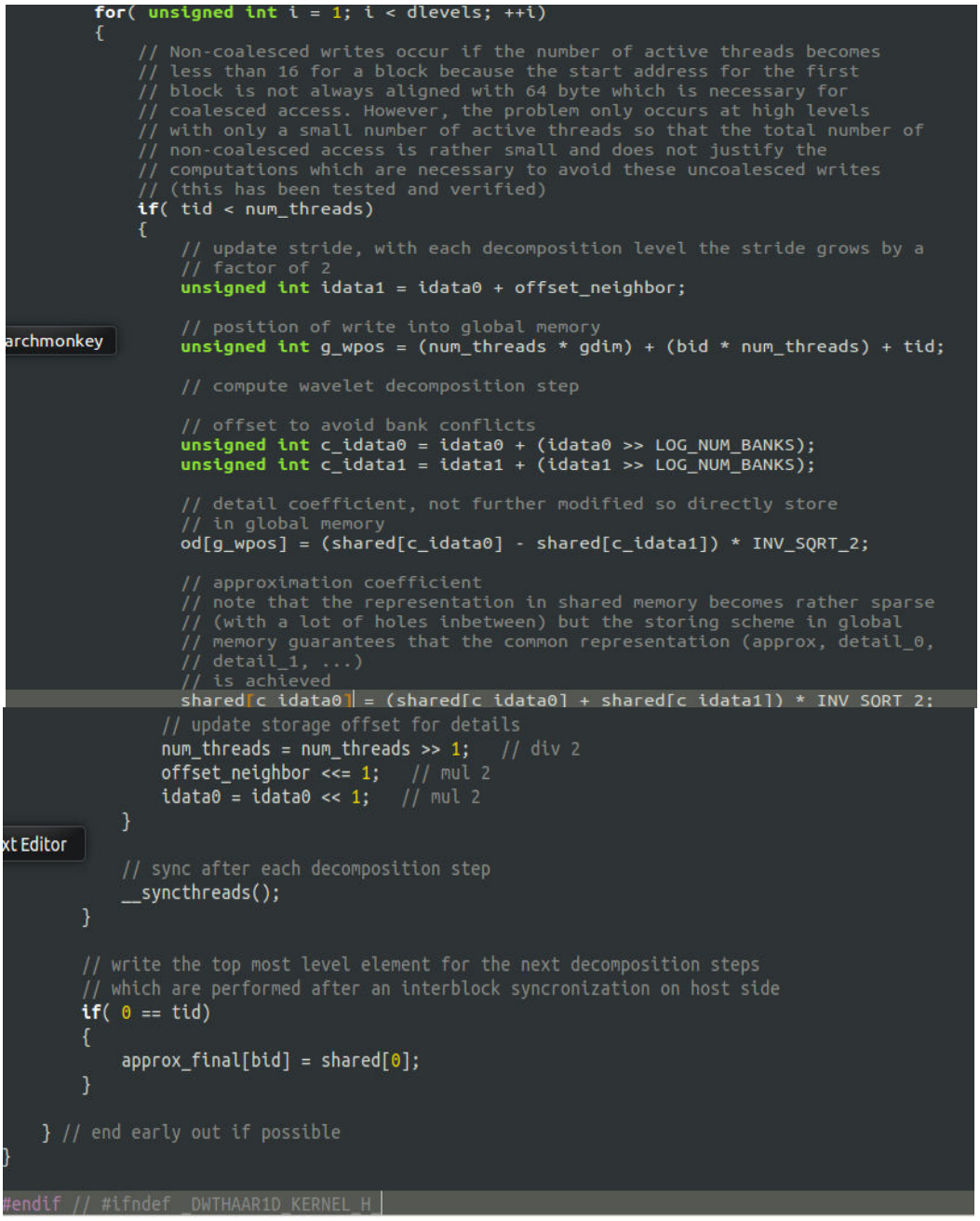


Figure 20:The DWTHaar1D kernel function.

Large signals are subdivided into sub-signals with 512 elements and the wavelet transform for these is computed with one block over 10 decomposition levels. The resulting signal consisting of the approximation coefficients at level X is then processed in a subsequent step on the device. This requires interblocking syncronization which is only possible on host side.

The basic of all Wavelet transforms is to decompose a signal into approximation (a) and detail (d) coefficients where the detail tends to be small or zero which allows / simplifies compression. The following "graphs" demonstrate the transform for a signal of

length eight. The index always describes the decomposition level where a coefficient arises. The input signal is interpreted as approximation signal at level 0. The coefficients computed on the device are stored in the same scheme as in the example. This data structure is particularly well suited for compression and also preserves the hierarchical structure of the decomposition.

After the kernel call, we compare the computed solution and the reference.

2.4.1 DWTHaar1D Results

2.4.1.1 Program statistics

Registers

- %r1->%r58
- %tid
- %ctaid
- %f1->f21
- %p1->p4
  Number of registers: 85

Cycles: 5637

Table 5: DWTHaar1D execution results.

	Transient Faults	Stuck-at-0	Stuck-at-1
Masked	1997	1990	1959
SDC	3	9	37*
Slowdown	-	1	4
Crash	-	-	-
Timeout	-	-	-
Number Of Injections	2000	2000	2000

*Along with 19 slowdowns.

Table 6: The slowdown table that shows the number of extra cycles and the number of sample executions that extra cycles occurred.

Extra Cycles	4	7	11	17	19	94	245	289	386	592	638	1121	1425	1739
Number of executions	4	3	3	3	2	1	1	1	1	1	1	1	1	1

2.4.1.3 Chart

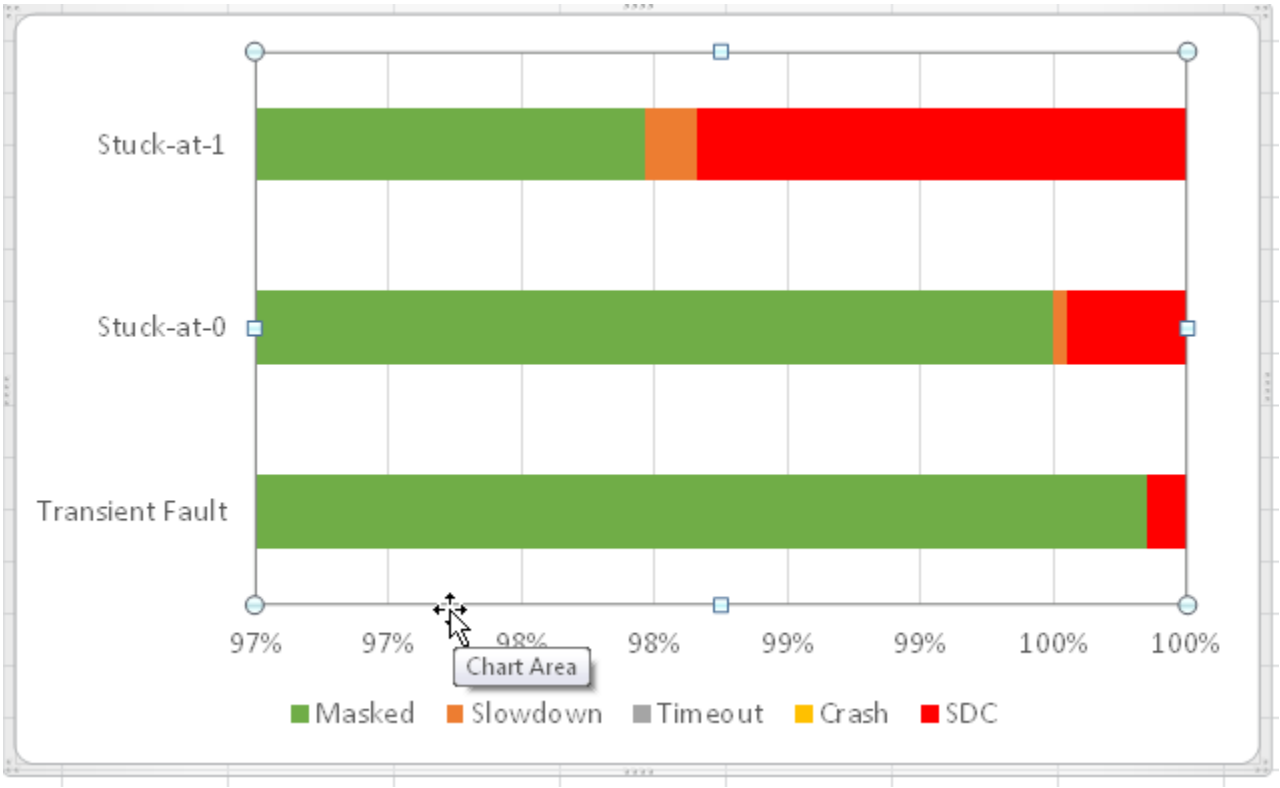


Figure 21: Chart for DWTHaar1D.

2.4.1.3 Comments

The transient fault column has only 3 SDC's which is explained by the fact we have 85 registers and 5637 cycles, making the possibility to match a specific register being used in a specific cycle reach almost zero per cent(here it is 0.15% possibility to have an SDC fault).

There is an increase in the number of SDC cases, moving upwards the chart image. The red bar increases as we go from the transient fault case to the stuck-at-0 case and finally to the stuck-at-1 case.

2.5 MatrixMul

The next program is about matrix multiplication. In mathematics matrix multiplication is a binary operation that takes a pair of matrices, and produces another matrix. Numbers such as the real or complex numbers can be multiplied according to elementary arithmetic. Computing matrix products is both a central operation in many numerical algorithms and potentially time consuming, making it one of the most well-studied problems in numerical computing. Various algorithms have been devised for
computing C = AB, especially for large matrices.

$$(\mathbf{AB})_{ij} = \sum_{k=1}^{m} A_{ik}B_{kj} \, .$$

Thus the product AB is defined only if the number of columns in A is equal to the number of rows in B, in this case *m*. Each entry may be computed one at a time. Sometimes, the summation convention is used as it is understood to sum over the repeated index *k.*

The sample demonstrates the matrix multiplication. It has been written for clarity of exposition to illustrate various CUDA programming principles. It had no goal of implementing the most efficient generic kernel for matrix multiplications, that they are not part of this thesis.

```
template <int BLOCK_SIZE> __global__ void
matrixMulCUDA(float *C, float *A, float *B, int wA, int wB)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep  = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep  = BLOCK_SIZE * wB;
    float Csub = 0;

    // Loop over all the sub-matrices of A and B
    // required to compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
         a <= aEnd;
         a += aStep, b += bStep)
    {

        // Declaration of the shared memory array As used to
        // store the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Declaration of the shared memory array Bs used to
        // store the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from device memory
        // to shared memory; each thread loads
        // one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
#pragma unroll

        for (int k = 0; k < BLOCK_SIZE; ++k)
        {
            Csub += As[ty][k] * Bs[k][tx];
        }

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to device memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

Figure 22: : MatrixMul kernel code.

2.5.1 MatrixMul Results

2.5.1.1 Program statistics

Registers:

- %r1->%r56
- %f1->%f50
- %ctaid.x,%ctaid.y
- %p1,%p2
- %tid.x,%tid.y
- %rh1
- Number of registers:113

Cycles: 57901

Table 7: MatrixMul execution results.

	Transient Fault	Stuck-at-0	Stuck-at-1
Masked	1992	1981	1959
SDC	7	14	19*
Slowdown	1	2	17
Crash	-	2	5
Timeout	-	1	-
Number of Injections	2000	2000	2000

*Along with 14 slowdowns.

Table 8: The slowdown table that shows the number of extra cycles and the number of sample executions that those extra cycles occurred.

Extra Cycles	Number of Executions	Extra Cycles	Number of Executions
1	4	1456	1
2	5	1567	1
3	6	1587	1
15	2	1769	1
25	1	1932	1
32	1	1997	1
45	1	2482	1
48	1	2789	1
80	1	3456	1
112	1		
231	1		
278	1		
587	1		
634	1		
765	1		
811	1		
1078	1		
1134	1		

2.5.1.2 Chart

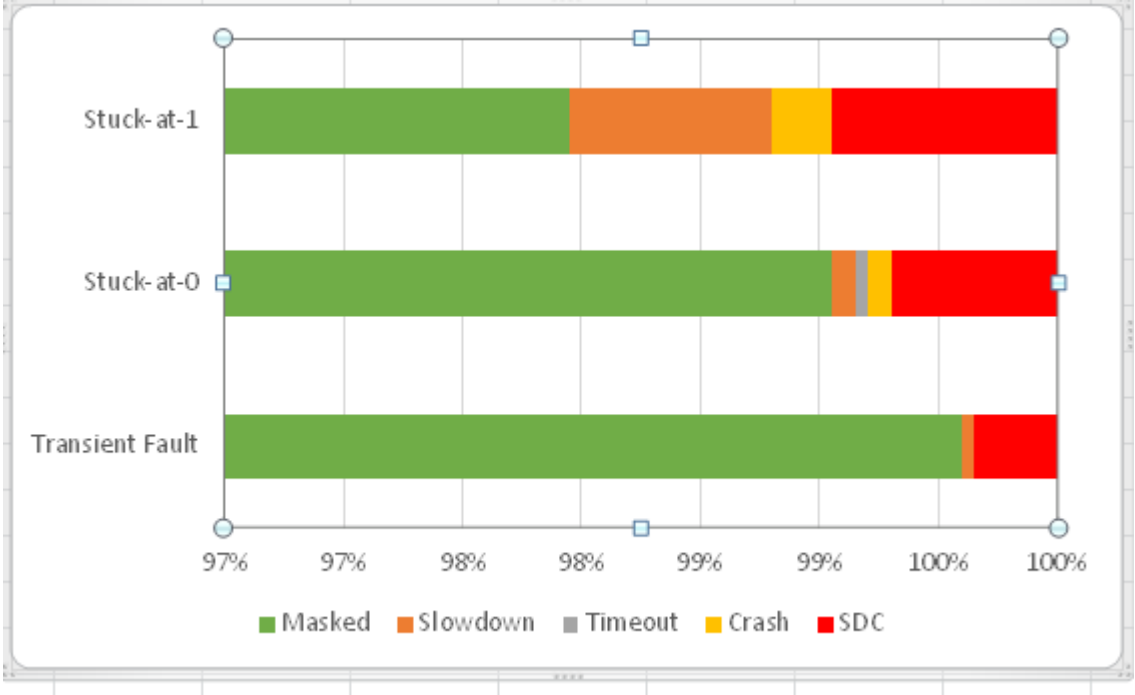


Figure 23: Chart for MatrixMul.

2.5.1.3 Comments

Due to the large number of cycles and registers, we expected that in all categories of fault injection there would be only a small number of faults that would lead to an error.

Worth noted here is the fact that in the stuck-at-0 column we got all types of errors. The complexity of the program, which explores many fundamental aspects of the CUDA programming language in purpose of computing the result of multiplying two matrices, affects our cause with the production of one timeout and two crashes. We set a rule about the timeout error that if a program run for twice the time it was supposed to, that would be the case of a timeout. The timeout appeared because the bit stuck with the 0 value was the register containing the index value inside the main loop. There was a timeout because every time the index added a value, the bit stuck into zero restrained the possible values the index would get, limited the maximum value to a one lesser than the upper boundary of the for loop. As far as the crashes are concerned, they appeared because the register hold address value, and the AND led to a segmentation fault. In the stuck-at-1 column, we once again find a connection between the number of silent data corruption faults and the slowdowns.

Despite the fact that in this program we have multiple error cases, the number of faults as a total is below 2% in all three categories. Furthermore, it is the first program that the number of slowdowns and the number of silent data corruption cases is almost the same.

2.6 SimpleTemplates

Next sample is simple templates. This sample is a templatized version of the template project. It also shows how to correctly templatize dynamically allocated shared arrays.

```
__global__ void
testKernel( T* g_idata, T* g_odata)
{
  // Shared mem size is determined by the host app at run time
  SharedMemory<T> smem;
  T* sdata = smem.getPointer();

  // access thread id
  const unsigned int tid = threadIdx.x;
  // access number of threads in this block
  const unsigned int num_threads = blockDim.x;

  // read in input data from global memory
  // use the bank checker macro to check for bank conflicts during host
  // emulation
  sdata[tid] = g_idata[tid];
  __syncthreads();

  // perform some computations
  sdata[tid] = (T) num_threads * sdata[tid];
  __syncthreads();

  // write data to global memory
  g_odata[tid] = sdata[tid];
}
```

Figure 24: SimpleTemplates kernel code.

2.6.1 SimpleTemplates Results

2.6.1.1 Program statistics

Registers:

- %r1->%r12
- %f1->%f5
- %tid.x
- %ntid.x

- %rh1

Number of registers: 19

Cycles: 1417

Table 9: SimpleTemplates execution results.

	Transient Fault	Stuck-at-0	Stuck-at-1
Masked	1997	1964	1713
SDC	3	27	211
Slowdown	-	9	76
Crash	-	-	-
Timeout	-	-	-
Number of Injection	2000	2000	2000

Table 10: The slowdown table that shows the number of extra cycles and the number of sample executions that those extra cycles occurred.

Extra Cycles	Number of Executions	Extra Cycles	Number of Executions
1	27	158	1
2	18	196	1
3	11	211	1
9	7	256	1
11	2	289	1
14	3	305	1
29	1	311	1
35	1	378	1
67	1		
82	1		
91	1		
97	1		
105	1		
108	1		
124	1		

2.6.1.3 Chart

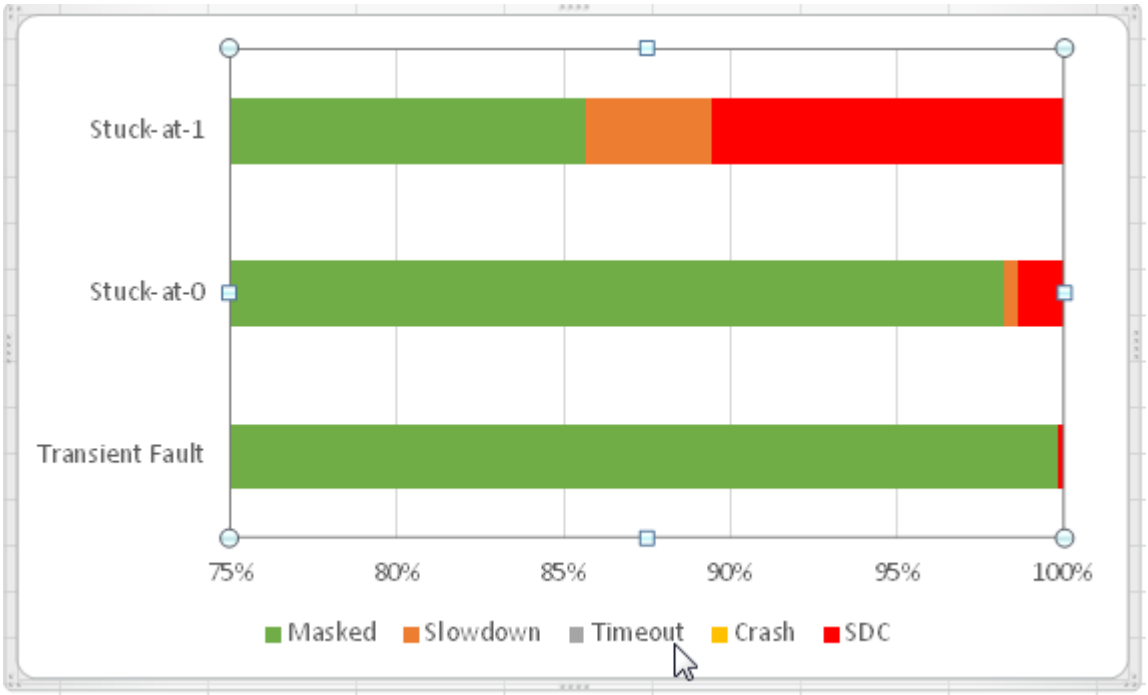


Figure 25: Chart for simple templates.

2.6.1.3 Comments

In this sample we had a large number of faults that produced an error, especially in OR case. The explanation is purely mathematical. The program is a simple program with 19 registers. Thus, especially in stuck-at-1 and in stuck-at-0 case, there is a big possibility a thread uses the register in a cycle, and the injection has an immediate effect on the execution of the program.

2.7 Simple Streams

The next NVIDIA program we use was the simple streams. This sample illustrates the usage of CUDA streams for overlapping kernel execution with device/host memory copies. The kernel is used to initialize an array to a specific value, after which the array is copied to the host (CPU) memory. To increase performance, multiple kernel/memory copy pairs are launched asynchronously, each pair in its kernels are serialized. Thus, if n pairs are launched, streamed approach can reduce the memory copy cost to the (1/n)th of a single copy of the entire data set. Additionally, this sample uses CUDA events to measure elapsed time for CUDA calls. Events are a part of CUDA API and provide a system independent way to measure execution times on CUDA devices with approximately 0.5 microsecond precision. Elapsed times are averaged

over nreps repetitions (10 by default). After the kernel is launched and the function is executed, then we compare the result we have with the result we should and expected to have. Each element should have been incremented by a total of npres*niterations times, and this is what we compare to . Regarding the result, a message appears whether the program's output is normal or altered.

```
__global__ void init_array(int *g_data, int *factor, int num_iterations)
{
   int idx = blockIdx.x * blockDim.x + threadIdx.x;

    for(int i=0;i<num_iterations;i++)
            g_data[idx] += *factor;     // non-coalesced on purpose, to burn time
}
```

Figure 26:SimpleStreams kernel.

2.7.1 SimpleStreams Results

2.7.1.1 Program Statistics

Registers:

- %r1->%r14
- %ctaid.x
- %p1,%p2
- %tid.x
- %ntid.x
- %rh1,%rh2

Number of registers:21

Cycles:16504

Table 11: SimpleStreams execution results.

	Transient Fault	Stuck-at-0	Stuck-at-1
Masked	1992	1988	1941
SDC	7*	9**	44***
Slowdown	1	3	7
Crash	-	-	-
Timeout	1	-	8
Injection	2000	2000	2000

*Along with *Along with 7 slowdowns.

**Along with 3 slowdowns.

****Along with 17 slowdowns.

Table 12: The slowdown table that shows the number of extra cycles and the number of sample executions that those extra cycles occurred.

Extra Cycles	Number of Executions	Extra Cycles	Number of Executions
1	3	255	1
2	5	273	1
3	4	281	1
9	3	305	1
10	3	376	1
13	3	379	1
18	2	405	1
27	1	465	1
39	1		
55	1		
79	1		
103	1		
137	1		
188	1		
207	1		

2.7.1.2 Chart

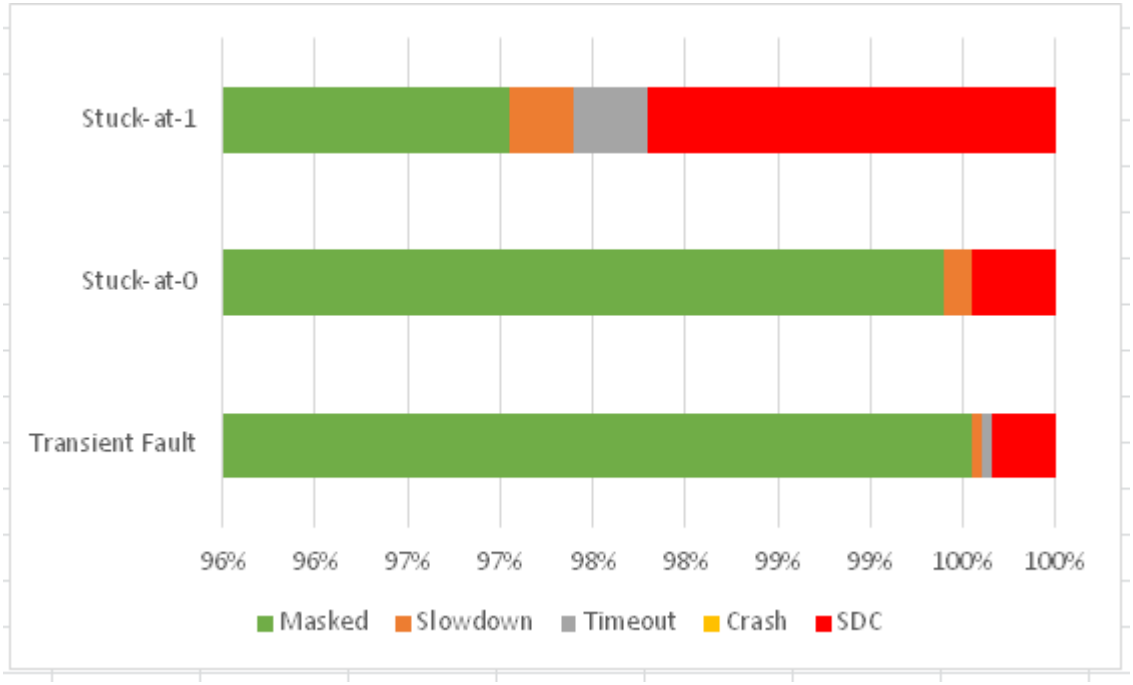


Figure 27: SimpleStreams chart.

2.7.1.3 Comments

In this sample, the results of fault injection are those we expected. There is a small number of errors in the transient fault column since we have a program with 21 registers(as we have seen,a program with medium amount of registers) and 16500 cycles. The possibility of matching a specific register to a specific cycle is around 0.04 %, with the main cause being the number of cycles.

Regarding the other columns, the stuck-at-0 has few errors as the main registers hold small numerical values. The program's execution results (as seen on the chart) have a maximum 3% errors in all cases. The stuck-at-1 column has 44 silent data corruption cases, along with many timeouts (it is the sample with the most timeout cases-grey bar in all charts), compared to previous samples, mainly attributed to the small amount of registers. Many of those associate to the computation of the main loop index as operands and highly affect the sample in terms of timeouts.

2.8 SimpleVoteIntrinsics

SimpleVoteIntrinsics is a simple program which demonstrates how to use the Vote (any, all) intrinsic instruction in a CUDA kernel. It explores two basic warp voting functions, VOTE any and VOTE.All. There are three kernels in this sample. The first one tests the across-the-warp vote (any) intrinsic.(VoteAnyKernel1). The second one tests the across the warp vote (all) intrinsic(VoteAnyKernel2). And the third one is a directed test for the across-the-warp vote (all) intrinsic (VoteAnyKernel3).

```
__global__ void VoteAnyKernel1(unsigned int *input, unsigned int *result, int size)
{
        int tx = threadIdx.x;

        result[tx] = any(input[tx]);
}

// Kernel #2 tests the across-the-warp vote(all) intrinsic.
// If ALL of the threads (within the warp) of the predicated condition returns
// a non-zero value, then all threads within this warp will return a non-zero value
__global__ void VoteAllKernel2(unsigned int *input, unsigned int *result, int size)
{
        int tx = threadIdx.x;

        result[tx] = all(input[tx]);
}

// Kernel #3 is a directed test for the across-the-warp vote(all) intrinsic.
// This kernel will test for conditions across warps, and within half warps
__global__ void VoteAnyKernel3(bool *info, int warp_size)
{
    int tx = threadIdx.x;
    bool *offs = info + (tx * 3);

    // The following should hold true for the second and third warp
    *offs = any((tx >= (warp_size * 3) / 2));
    // The following should hold true for the "upper half" of the second warp,
    // and all of the third warp
    *(offs + 1) = (tx >= (warp_size * 3) / 2? true: false);
    // The following should hold true for the third warp only
    if(all((tx >= (warp_size * 3) / 2))) {
      *(offs + 2) = true;
```

Figure 28: SimpleVoteIntrinsics kernel.

2.8.1 SimpleVoteIntrinsics results

2.8.1.1 Program Statistics

- **Registers:**
  - %r1->%r22
  - %p1->%p6
  - %tid.x
  - %rh1

  Number of registers:31

- **Cycles**:2144

Table 13: SimpleVoteIntrinsics execution results.

	Transient Fault	Stuck-at-0	Stuck-at-1
Masked	1995	1994	1969
SDC	4	6*	31**
Slowdown	1	-	-
Crash	-	-	-
Timeout	-	-	-
Injection	2000	2000	2000

*Along with one slowdown.

**Along with 20 slowdowns.

Table 14: The slowdown table that shows the number of extra cycles and the number of sample executions that those extra cycles occurred.

Extra Cycles	Number of Executions
1	6
2	7
3	5
11	2
15	2

2.8.1.2 Chart

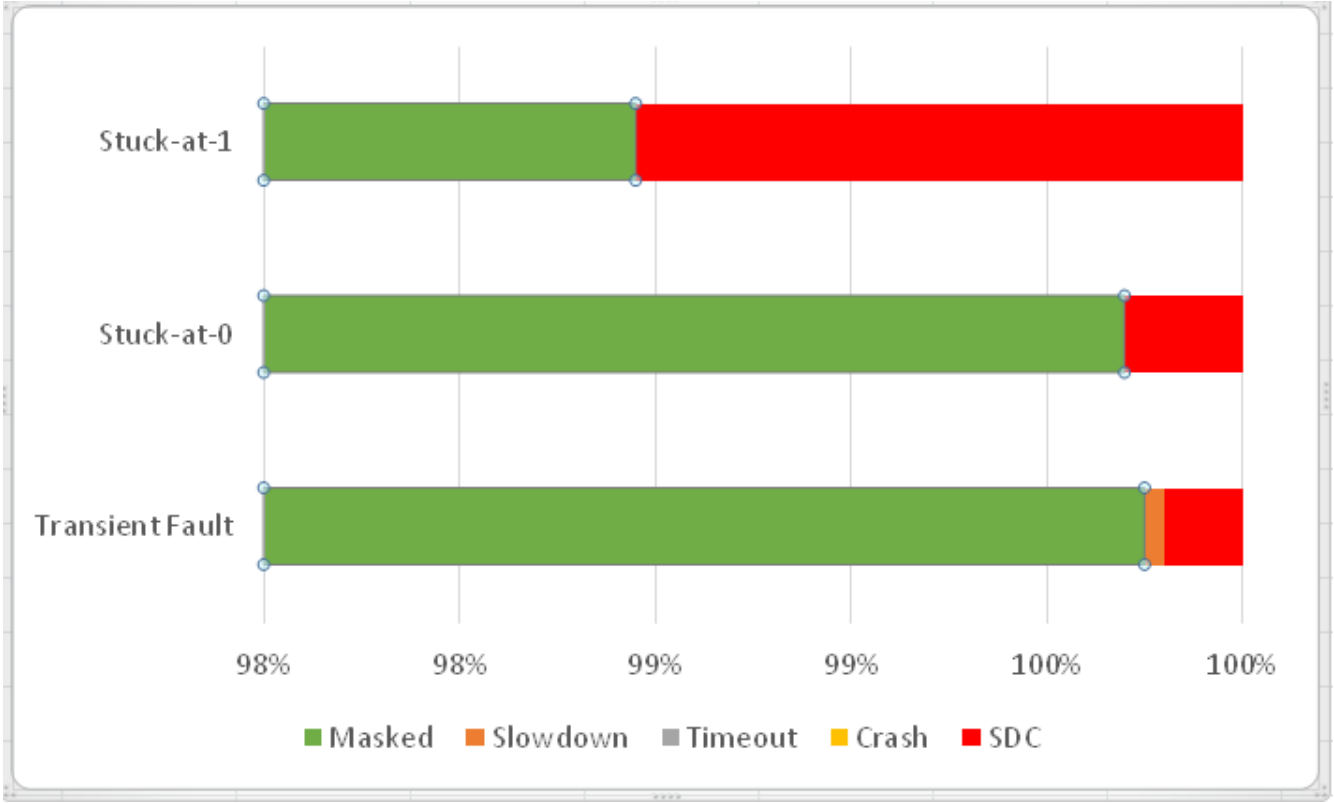


Figure 29: SimpleVoteIntrinsics chart.

2.8.1.3 Comments

The programs have a 2% error in all cases. The main fault appearing here is the SDC, contributing to almost 1% percent of total program execution. Despite the small number of registers and the medium number of cycles, the sample has only 2 %. The reason behind that is the massive amount of threads running, making the fault injection for cases OR/AND very difficult.

2.9 Templates

This sample is a more complex implementation of the simple templates sample, to which we referred previously.

```
__global__ void
testKernel( float* g_idata, float* g_odata)
{
  // shared memory
  // the size is determined by the host application
  extern  __shared__  float sdata[];

  // access thread id
  const unsigned int tid = threadIdx.x;
  // access number of threads in this block
  const unsigned int num_threads = blockDim.x;

  // read in input data from global memory
  // use the bank checker macro to check for bank conflicts during host
  // emulation
  SDATA(tid) = g_idata[tid];
  __syncthreads();

  // perform some computations
  SDATA(tid) = (float) num_threads * SDATA( tid);
  __syncthreads();

  // write data to global memory
  g_odata[tid] = SDATA(tid);
}
```

Figure 30: Templates kernel function.

2.9.1 Templates Results

2.9.1.1 Program Statistics

Registers:

- %r1->%r8
- %f1->%f5
- %tid.x
- %ntid.x
- %rh1

Number of registers: 19

Cycles: 629

Table 15: Template execution results.

	Transient Faults	Stuck-at-0	Stuck-at-1
Masked	1995	1944	1512
SDC	3	53*	481**
Slowdown	2	3	7
Crash	-	-	-
Timeout	-	-	-
Injection	2000	2000	2000

*Along with 9 slowdowns.

**Along with 276 slowdowns.

Table 16: The slowdown table that shows the number of extra cycles and the number of sample executions that those extra cycles occurred.

Extra Cycles	Number of Executions	Extra Cycles	Number of Executions	Extra Cycles	Number of Executions
1	27	65	1	[170-270]	45
2	24	71	1	[271-351]	38
3	21	82	1	[352-502]	34
5	15	85	1	[502-567]	40
9	10	96	1		
11	5	101	1		
13	7	104	1		
15	4	111	1		
19	2	115	1		
25	2	124	1		
32	1	131	1		
37	1	136	1		
46	1	148	1		
53	1	153	1		
55	1	157	1		
59	1	162	1		
63	1	169	1		

2.9.1.2 Chart

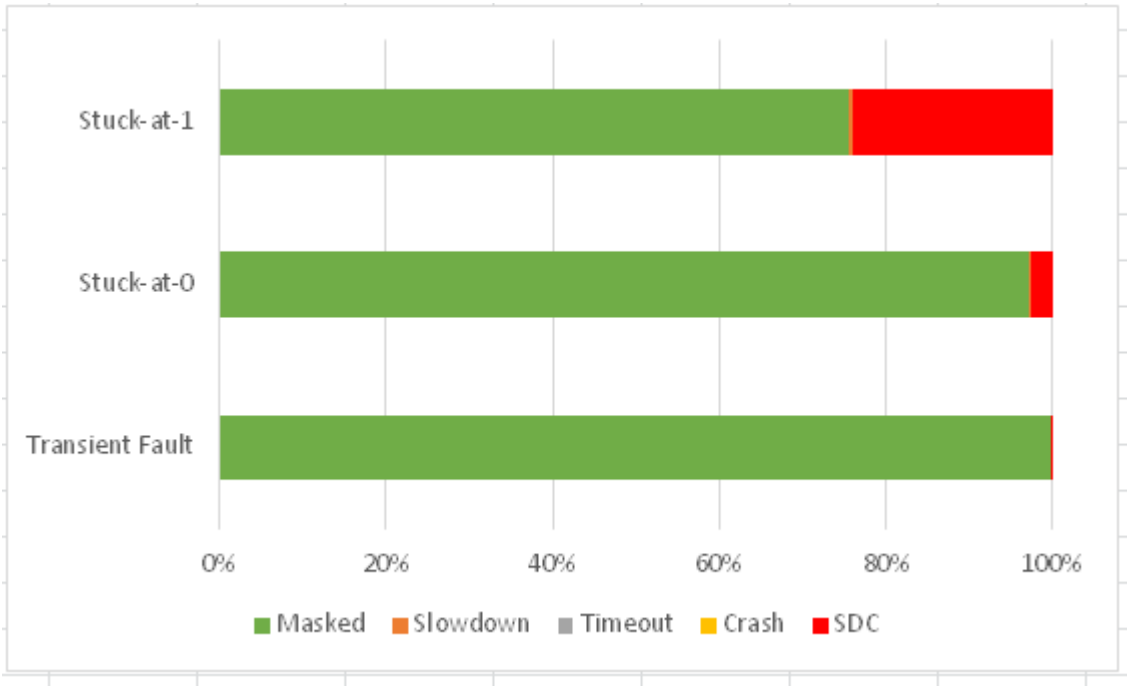


Figure 31: Templates chart.

2.9.1.2 Comments

What is obvious after 2000 executions here is that due to the small number of registers and cycles, the program has multiple silent data corruption cases. This, along with the fact that stuck-at-1 case affects specially variables with small numerical values, explains why we have 481 silent data corruption errors. The big amount of errors reflects on the chart as it is the only case that the error percentage begins from 0% percent, whereas other sample charts begin from a value greater than 90 %.

2.10 CPPIntegration

This sample is an example of integration CUDA functions into an existing application/framework. It consists of two kernels. The first one is a test kernel for device functionality (kernel). The second one is a kernel that demonstrates one templatized variable that can be used in the c++ code. This variable is actually a memory to be processed.

```
__global__ void
kernel( int* g_data )
{
    // write data to global memory
    const unsigned int tid = threadIdx.x;
    int data = g_data[tid];

    // use integer arithmetic to process all four bytes with one thread
    // this serializes the execution, but is the simplest solutions to avoid
    // bank conflicts for this very low number of threads
    // in general it is more efficient to process each byte by a separate thread,
    // to avoid bank conflicts the access pattern should be
    // g_data[4 * wtid + wid], where wtid is the thread id within the half warp
    // and wid is the warp id
    // see also the programming guide for a more in depth discussion.
    g_data[tid] = ((((data <<  0) >> 24) - 10) << 24)
                | ((((data <<  8) >> 24) - 10) << 16)
                | ((((data << 16) >> 24) - 10) <<  8)
                | ((((data << 24) >> 24) - 10) <<  0);
}
```

Figure 32: CppIntegration first kernel.

```
__global__ void
kernel2( int2* g_data )
{
    // write data to global memory
    const unsigned int tid = threadIdx.x;
    int2 data = g_data[tid];

    // use integer arithmetic to process all four bytes with one thread
    // this serializes the execution, but is the simplest solutions to avoid
    // bank conflicts for this very low number of threads
    // in general it is more efficient to process each byte by a separate thread,
    // to avoid bank conflicts the access pattern should be
    // g_data[4 * wtid + wid], where wtid is the thread id within the half warp
    // and wid is the warp id
    // see also the programming guide for a more in depth discussion.
    g_data[tid].x = data.x - data.y;
}
```

Figure 33: CppIntegration second kernel.

2.10.1 CPPIntegration Results

2.10.1.1 Program Statistics

Registers

- %r1->%r21
- %tid.x
- %rh1

Number of registers:22

Cycles:1247

Table 17: CppIntegration execution results.

	Transient Fault	Stuck-at-0	Stuck-at-1
Masked	1992	1969	1892
SDC	7*	31**	108**
Slowdown	1	-	-
Crash	-	-	-
Timeout	-	-	-
Injections	2000	2000	2000

*Along with one slowdown.

**Along with 1 slowdown.

***Along with 65 slowdowns.

Table 18: The slowdown table that shows the number of extra cycles and the number of sample executions that those extra cycles occurred.

Extra Cycles	Number of Executions	Extra Cycles	Number of Executions
1	10	68	1
2	8	89	1
3	7	112	1
5	9	179	1
6	6	211	1
12	5	211	1
15	3		
19	2		
20	2		
23	1		
29	1		
31	2		
38	1		
41	1		
47	1		
51	1		
55	1		
63	1		

2.10.1.2 Chart

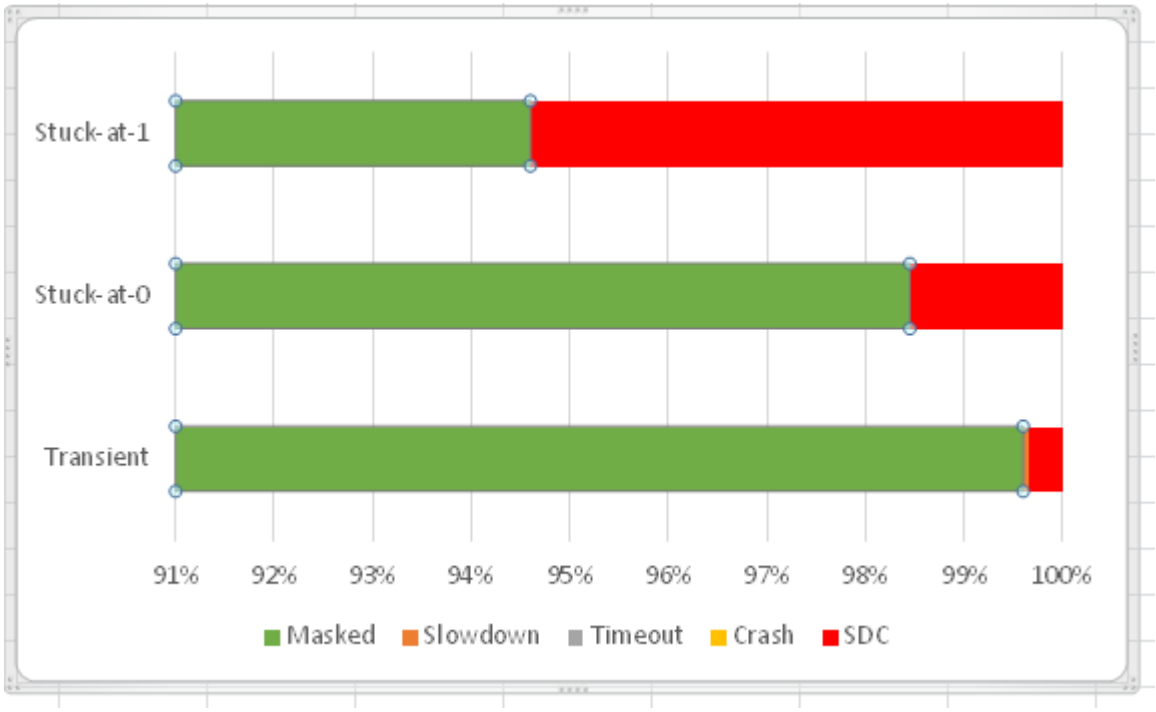


Figure 34: CPPIntegration chart.

2.10.1.3 Comments

We can see here that transient fault category has surprisingly few errors compared to the stuck-at-1/stuck-at-0 cases for the number of registers and cycle number of the execution. Taking in consideration the fact that the program has small number of cycles and registers, makes the transient fault case having only 8 errors even stranger. The reason why this is happening is because the program sets a register in very few cycles, in approximately, 10 percent of the programs cycle. As a result, worsening the possibility of a match between a register and a cycle with our corresponding random initialized variables.

2.11 General Charts

Below we present 3 general charts for all benchmarks. They present the percentage of the transient faults,stuck-at-0 and stuck-at-1 respectively compared to the total non-masked cases we encountered.

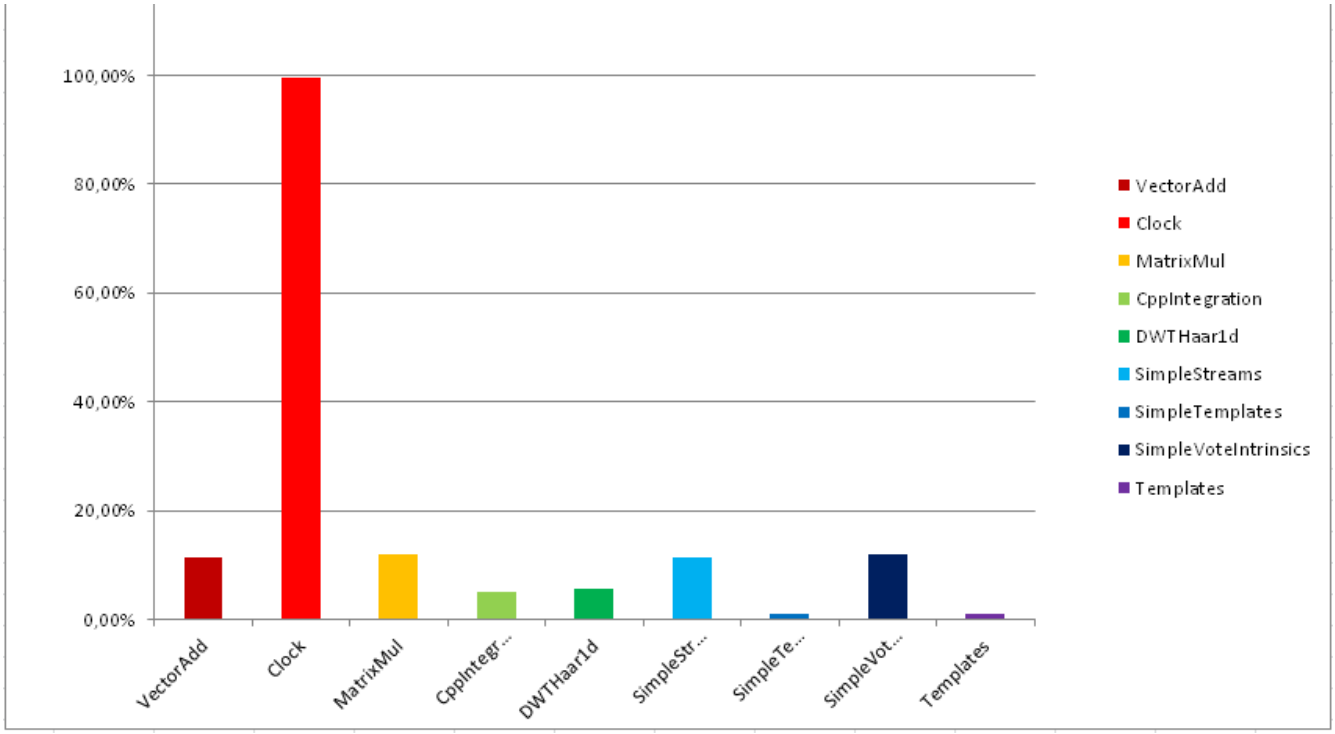


Figure 35: Percentage of transient faults according to total non-masked cases.

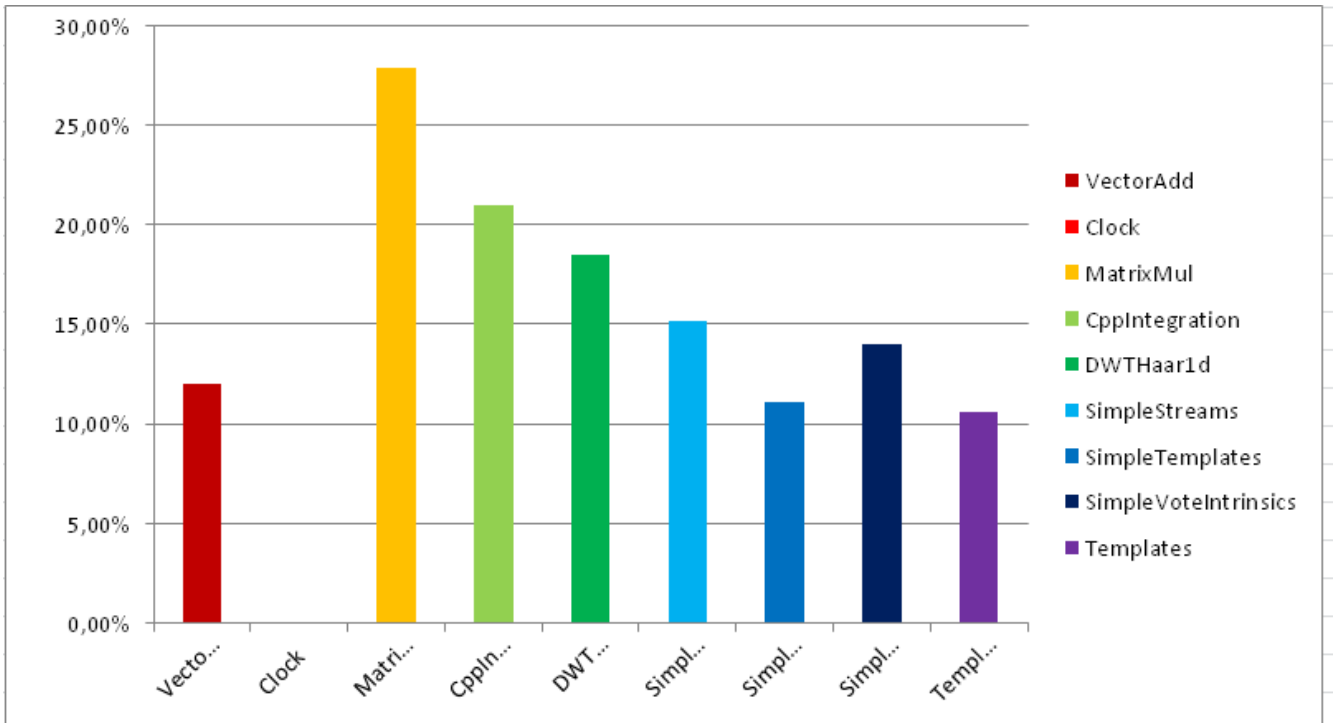


Figure 36: Percentage of stuck-at-0 faults according to total non masked cases.

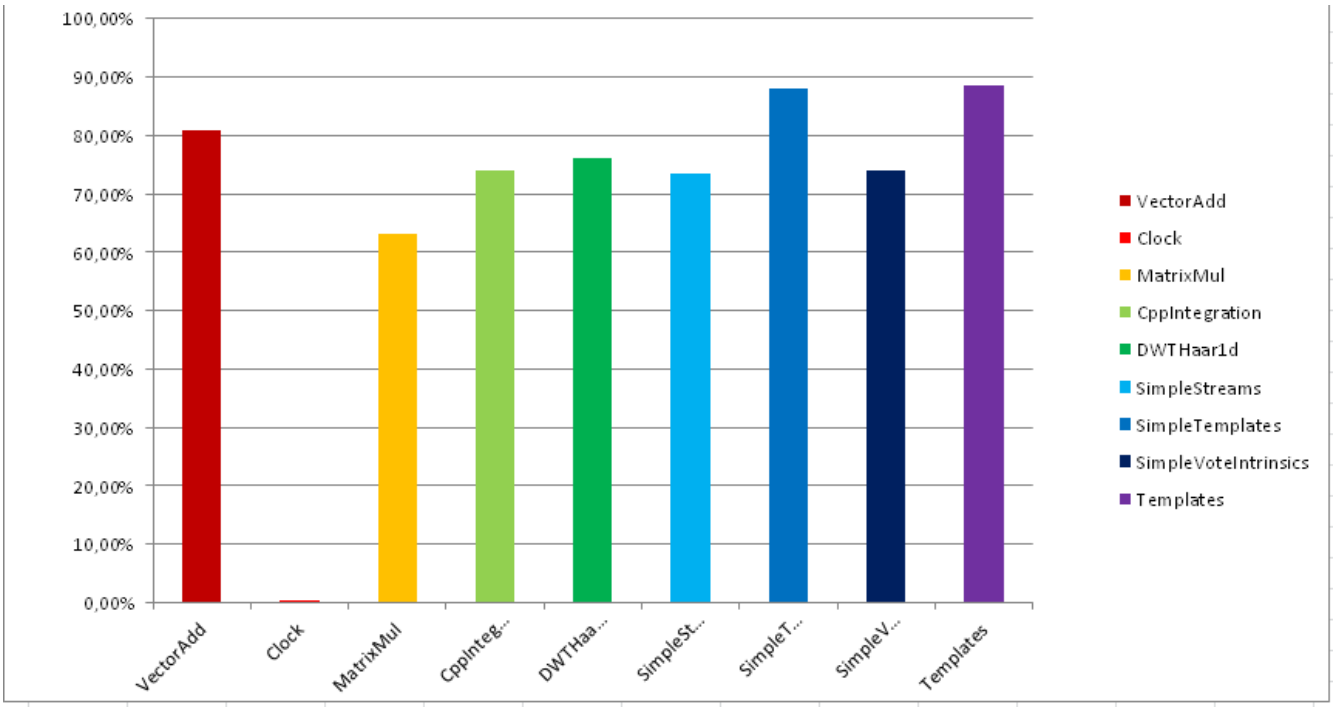


Figure 37: Percentage of stuck-at-1 faults according to total non masked cases.

3 SUMMARY

There are few points coming out as a result after finishing this thesis, which we believe worth to be mentioned, especially in this sector of the document.

The stuck-at-1 and stuck-at-0 case were kind of a compliment one to the other . The 64 bit variables is formed with a big number of zeros towards the beginning bits of the number stored in every register. As a result, the OR operator is more likely to fall into a position that holds zero numerical value .The bit alters its value, so there is a difference between the value before the fault injection and the value after . Registers that operate with different values than the normal are, obviously, more likely to cause an error during the execution of a sample and this explains why in all cases the amount of errors with the OR operator is higher than the amount of errors with the AND operator.

In addition, we noticed that there is a strong connection in many programs between the silent data corruption and the slowdown. As already explained a SDC changes the value of a register and sometimes it cause the program to execute more cycles due to changed value of the one variable we have for keeping the index of the vectors. Mostly, the programs we studied consisted of man loop that iterated over a vector. A change into that index may force the program to have more access cycles into the memory for retrieving data .In a normal execution, the program, based on locality by reference, would retrieve the specified object in the index along with the ones that are closer to it based on memory. From the moment that we altered the index of the vector, the object may have not been already brought by the program and extra communication may exist with memory , contributing to the slowdown case.

The XOR operator is the one causing always the least errors. The two main requirements are that the GPU cycle must be the same as the one randomly chosen inside set_operand_value and that the register chosen must be in the specific instruction that is executed by the simulator at that time. This is something that lessens the possibility of an access into register, as it requires the program to have a small number of cycles(possibly bellow 1000 cycles) and a small number of registers(possibly below 20 registers per PTX code produced).

The number of timeout cases are limited. The timeout case is the most difficult error to occur, as it needs to change the value of the executing loop in a value that is over the upper boundary of the loop. It is up to the program’s nature on how the program is affected by the number of registers, how many of them contribute to the index computation, how the program handles over the bounds indexing etc.

ABBREVIATIONS-ACRONYMS

GPU	Graphics Processing Unit
GPGPU	General-Purpose computation on Graphics Processing Units
SDK	Parallel Thread Execution
CTA	Cooperative Thread Array
SIMT	Single Instruction Multiple Threads
SDC	Silent Data Corruption
DWT	Direct Wavelet Transform
ΕΚΠΑ	Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

REFERENCES

[1] Devesh Tiwari, Saurabh Gupta, James Rogers, Don Maxwell, Paolo Rech, Sudharshan Vazhkudai, Daniel Oliveira, Dave Londo, Nathan DeBardeleben, Philippe Navaux, Luigi Carro, and Arthur Bland, "Understanding GPU Errors on Large-scale HPC Systems and the Implications for System Design and Operation," *North Carolina State University*, 2015;http://www4.ncsu.edu/~dtiwari2/Papers/2015_HPCA_Tiwari_GPU_Reliability.pdf

[2] Sajjad Hussain,Pascal and Volta – NVIDIA Unveiled the Computing Power of Next-Gen GPUs, http://tech4gamers.com/pascal-volta-nvidia-unveiled-computing-power-next-gen-gpus/

[3] Naoya Maruyama and Akira Nukada, "A High-Performance Fault-Tolerant Software Framework for Memory on Commodity GPUs*," Computer Science*; doi=10.1.1.383.1623

[4] Bo Fang , Pattabiraman, K. ; Ripeanu, M. ; Gurumurthi, S., "GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications "Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on), 2007; doi:10.1109/SCIS.2007.367670.

[5] T. Austin, V. Bertacco, S. Mahlke, and Y. Cao, "Reliable Systems on Unreliable Fabrics," IEEE Design Test of Computers, vol. 25, pp. 322 –332, july-aug. 2008

[6] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith, "Configurable Isolation: Building High Availability Systems with Commodity Multi-Core Processors," in Proceedings of the 34th annual international symposium on Computer architecture, ISCA '07, (New York, NY, USA), pp. 470–481, ACM, 2007.

[7] Lizandro Dami´an Solano Quinde,"Parallelization & checkpointing of GPU applications through program transformation" ,2012, Iowa State University, http://www.osti.gov/scitech/servlets/purl/1082971

[8] Rickard Svennignson ,"Model-Implemented Fault Injection for Robustness Assessment", 2011, KTH School of Industrial,http://www.diva-portal.org/smash/get/diva2:460561/FULLTEXT01.pdfMarc

[9] Scott Michael Denton,"Optimizing Parallel Reduction Operations ",http://www.osti.gov/scitech/servlets/purl/188648,1985, Northeast Louisiana University