Performance-Reliability Trade-Off in Graphics Processing Units

Fernando Fernandes dos Santos, Paolo Rech

Performance-Reliability Trade-Off in Graphics Processing Units

Fernando F. dos Santos[†] Paolo Rech[*]
[†] Univ. Rennes, INRIA, Rennes, Brittany, France
[*]Department of Industrial Engineering, University of Trento, Povo, Italy

***Abstract*—We show that most performance improvements in GPUs increase the number of executions correctly completed before experiencing a failure. We consider four different performance improvements: architectural solutions, software implementations, compiler optimizations, and degree of parallelism.**

I. Introduction

Graphics Processing Units (GPUs) have evolved from being devices dedicated to gaming, graphics, and video rendering to flexible accelerators for a variety of High Performance Computing (HPC) and safety-critical applications, including autonomous vehicles. This market shift led to a burst in GPUs' computing capabilities and efficiency, significant improvements in the programming frameworks, and a sudden concern about their hardware reliability.

A common belief is that reliability and performance are conflicting proprieties with opposite requirements that should be traded-off, trying to achieve a sufficient reliability level without losing too much on performance or ensuring that the recovery from faults does not significantly impact the performance. In this paper, we demystify this false myth on GPUs. Our goal is to show how and why better performances, in most configurations, despite increasing the error rate, can still lead to a higher amount of correctly processed data.

We consider four common and effective ways to improve GPUs performance that are available to the programmer and do not require extra resources, specific hardening solutions, nor extra effort to be implemented: (1) software optimizations, i.e., algorithm efficient implementations, (2) degree of parallelism improvement, i.e., increase the number of parallel threads reducing the operations in each thread, (3) architectural solutions, such as tensor core and mixed-precision, and (4) compiler optimizations. Overall, we consider more than 50 different configurations.

Intuitively, the best performance is achieved on a GPU when its parallel capabilities are fully exploited. Nevertheless, when more operations are executed in parallel, the device error rate might increase. Thus a higher occupation of the GPU hardware delivers higher performance at the cost of a higher error rate. However, in the vast majority of the cases, for GPUs the performance gain grows faster than the error rate. In other words, the benefit of better using additional resources (shorter execution time) is, most of the time, higher than the possible drawback (higher FIT rate).

II. Reliability-Performance Evaluation Metrics

The primary metric to measure the reliability of a device is the Failure In Time (FIT) rate. The FIT rate depends on the amount of resources used for computation, i.e., the cross section, and their corruption probability of affecting the output, i.e., the Architectural or Program Vulnerability Factors (AVF or PVF) [1], [2]. The FIT rate, being by definition a *rate*, does not depend on the execution time. If the same amount of memory is exposed for a time interval t or $2t$, its FIT rate will not change. In fact, under a constant flux ($neutrons/cm^2/sec$), in $2t$, we expect twice the errors and a double neutrons fluence (n/cm^2) to hit the device, giving the same error rate. Similarly, executing x or $2x$ *sequential* (independent, for simplicity) instructions does not change the code FIT rate. On the contrary, if the additional x instructions are executed in *parallel* with the original sequence, the FIT rate is expected to double (same execution time, same fluence, but doubled the error rate). We use these premises to comment on how performance improvement can impact the GPU FIT rate and reliability. The interesting aspect, only apparently playing against GPUs, is that a slower execution *does not* increase the FIT rate while using more parallel resources or bigger hardware cores, with a potential benefit on GPUs performance, increases the FIT rate. We will demonstrate in Section V that on GPUs most of the common optimizations increase the performance faster than the error rate.

To combine error rate and performance, the Mean Instructions, Executions, or Work Between Failure metrics (MIBF, MEBF. MWBF) were introduced [3], [4]. The idea is to consider how many instructions, executions, or workloads can be correctly completed before the error occurrence. Considering a constant FIT rate, then the faster configuration will have a higher MIBF, MEBF or MWBF.

III. GPUs Performance Improvement Strategies

In this Section, we discuss the reasons why, in GPUs, better performances can provide a higher number of corrected executions before the error occurrence even if they increase the error rate. The optimizations we consider, as shown in the cartoon of Figure 1, are:

Algorithm Implementation: The performance of an algorithm is improved maximizing the GPU occupancy and the locality of data. While an optimized algorithm provides significant performance gain, the optimization normally involves code modification and, thus, the executed operations differ,

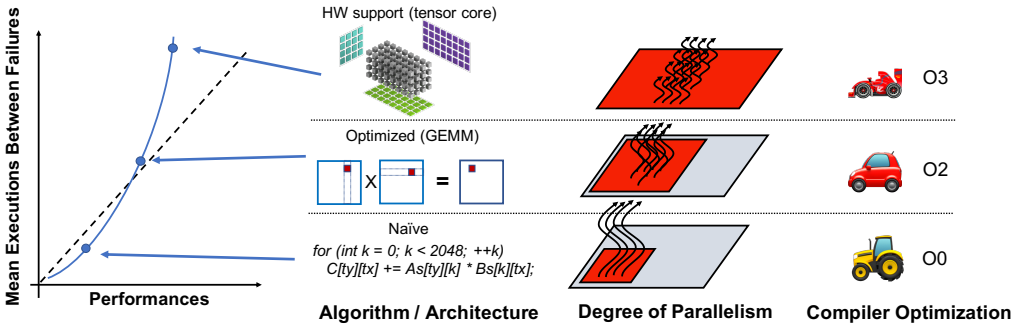

Fig. 1: Overview of the proposed study. Improving the performance of codes executed on GPUs by (1) optimizing the code, (2) taking advantage of dedicated architectural solutions, (3) increasing the Degree of Parallelism, or (4) using aggressive optimization flags will also impact the error rate (not necessarily for worse, as we will show). We aim to identify the configurations that provide a performance gain higher than the FIT rate increase, thus improving the MEBF.

impacting both the raw FIT rate (different functional units have different FIT rates) and the PVF (the probability for a fault to reach the program output). The optimizations normally increase the computation *density*, i.e., the number of operations executed in parallel and, then, increase also the FIT rate. This is confirmed by the experimental data in Section V-A.

Degree of Parallelism (DOP): when a workload needs to be executed on a GPU, the higher the number of parallel threads in which the workload is divided, the higher the DOP. A higher DOP improves performance but a higher number of parallel threads means that a larger area of the GPU is used for computation and, thus, the FIT rate increases.

Architectural Solutions: The GPU architecture includes dedicated functional units for strategic operations. This is the case of tensor cores [5] and mixed-precision [6]. A tensor core is a hardware unit that multiplies 4x4 matrices in 1 clock cycle [5]. Obviously, a tensor core is much larger than a multiplier or an adder [5] and, thus, it also has a higher error rate than a single ADD or MUL. The use of mixed-precision (32bit, 16bit, or lower precision) cores is a peculiar optimization as, besides being faster, lower precision cores are also smaller and, thus, are expected to reduce not just the execution time but also the FIT rate of the code.

Compiler Optimizations: the process of translating the C++ CUDA code into GPU executable is very complex. Even if the source code does not change, the compiler optimization modifies the machine code to be executed. The compiler modifies the instructions distribution and memory utilization, which changes the way the software is mapped in the underlying hardware. As we show in Section V-D, generally, the higher the optimization, the more reliable the execution.

IV. Evaluation Methodology

To evaluate the FIT/performances trade-off we consider several representative applications from different computing domains executed on three different GPU architectures.

A. Codes and Devices

The codes we test are: Matrix Multiplication (MxM and GEMM), LavaMD, Fast Fourier Transform (FFT), Sorting (Quicksort and Mergesort), Needleman-Wunsch (NW), and an object detection convolutional neural network (YOLO).

We consider three GPUs architectures: Fermi (GTX480), Kepler (Tesla K20), and Volta (Tesla V100) NVIDIA GPUs. Fermi, belonging to an older GPU generation, has computing cores that support only float precision (FP32), while Kepler supports float and double (FP64) precisions and Volta GPUs support three IEEE754 float point precisions (FP64, FP32, and FP16) plus eight tensor cores.

In some configurations, we also compare the results when the available Single Error Correction Double Error Detection (SECDED) Error Correcting Code (ECC) is turned ON or OFF. We only compare different configurations on *one* device.

B. Neutrons Beam Experiments

To measure the FIT rate, we take advantage of controlled neutron beam experiments performed in the past few years. Beam experiments were performed at ChipIR facility of the Rutherford Appleton Laboratory (RAL) in Didcot, UK, and at the LANSCE facility of the Los Alamos National Laboratory (LANL) in Los Alamos, US. The available neutron flux at ChipIR and LANSCE are about $10^6 n/(cm^2/s)$ and $10^5 n/(cm^2/s)$, respectively, i.e. about 7-8 orders of magnitude higher than the terrestrial flux ($13 n/(cm^2 \times h)$ at

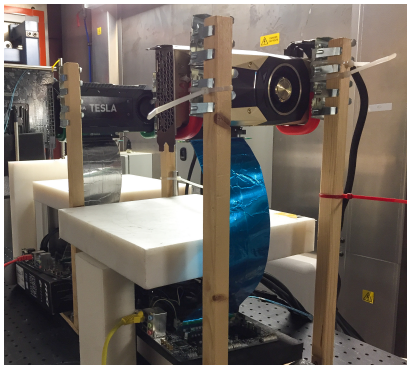

Fig. 2: Portion of the neutron beam setup at ChipIR.

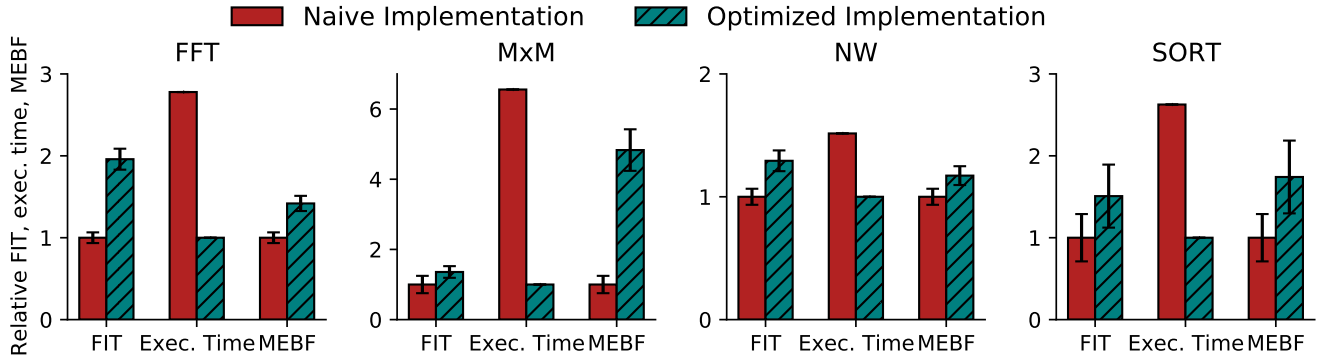

Fig. 3: Relative effects of changing the algorithm implementation on the Failure In Time rate, Execution time, and Mean Execution Between Failures for FFT, MxM, NW, and Sort executed on the Kepler K20 with ECC OFF.

sea level [7]). Since the terrestrial neutron flux is low, in a realistic application, it is highly unlikely to observe more than a single corruption during the program execution. We have carefully designed the experiments to maintain this property (observed error rates were lower than 1 error per 1,000 executions). Experimental data, then, can be scaled to the natural radioactive environment without introducing artifacts. Figure 2 shows a portion of the setup installed at ChipIR during one of the multiple test campaigns we performed.

While ChipIR and LANSCE have been shown to provide similar FIT rates, we only compare configurations tested at the same facility to reduce data uncertainty. We always compare configurations that differ of only *one* characteristic. Thus, we can derive that the observed trend is strictly related to the characteristic that we have modified. Moreover, thanks to the application profile and analysis, we can provide a justification for the obtained results.

V. Performance and Error Rate Comparison

In this section, we characterize four different solutions to improve GPU performance: (a) algorithm optimizations, (b) degree of parallelism, to fully use the GPU parallel architecture, (c) architectural solutions, such as tensor core and mixed-precision, and (d) compiler optimization. For each evaluated solution, we consider the benefit in terms of performance and the impact on the error rate.

To compare the performance and the error rate of the different codes implementations, we plot the *relative* FIT rate and execution time of the tested configurations. That is, we divide the FIT rate (execution time) of each tested configuration by the lower FIT rate (execution time) we measured. All experimentally measured relative FIT rates and MEBF are shown with 95% confidence intervals.

A. Algorithmic Solutions

The same algorithm can be implemented in various ways and, on GPUs, it is also necessary to find the parallel implementation that delivers better performances. We have selected four common algorithms for GPUs: 512x512 FFT, 2Kx2K MxM, 16K NW, and 32M Sorting. For each, we have considered a naive implementation, i.e., not crafted to fully exploit the GPU, and the optimized implementation.

In the naive implementation of MxM, each thread is responsible for computing one output element, performing 2048 sums and additions, thus saturating the number of registers and caches available in the SM. The efficient implementation, also known as GEMM, is taken from the NVIDIA cuBLAS library and, by dividing the matrices in custom tiles sizes and applying specific algorithm policies, ensures that each thread works only on local data and increases the parallel occupancy of the SM [8]. For FFT, in the naive implementation, each thread computes an FFT on 512 points independently from the others, forcing most data to be stored in the main memory, increasing the memory latency. The efficient FFT implementation, developed by Volkov and Kazian [9], divides the 512 FFT computation among 64 threads (2 warps), using only shared data and avoiding synchronizations. For NW (sequencing of DNA), in the naive implementation, each algorithm step, performing a diagonal search, is done in a dedicated kernel while the optimized implementation organizes the input in matrices of 32x32 cells, and each group is assigned to a warp, reducing the number of kernels and improving the efficiency. For sorting, the naive implementation is quicksort, and the optimized one is mergesort. Quicksort suffers from long memory latencies as the whole input array does not fit into the caches, while mergesort, being recursive, better exploits GPU characteristics.

Figure 3 shows the relative FIT rate, execution time, and MEBF of the naive and optimized implementation of the four algorithms executed on Kepler K20 GPUs, with ECC OFF. As shown, the naive implementation is from $0.5\times$ (for NW) to $6\times$ (for MxM) slower than the optimized one. The FIT rate has the opposite trend for all the codes, with the optimized implementation having an up to $2\times$ higher error rate than the naive one (for FFT). The higher FIT rate is caused by the denser computation imposed by the optimizations.

A promising result is that, for all the considered optimizations, the performances benefit is higher than the FIT rate increase. The MEBF (number of executions completed

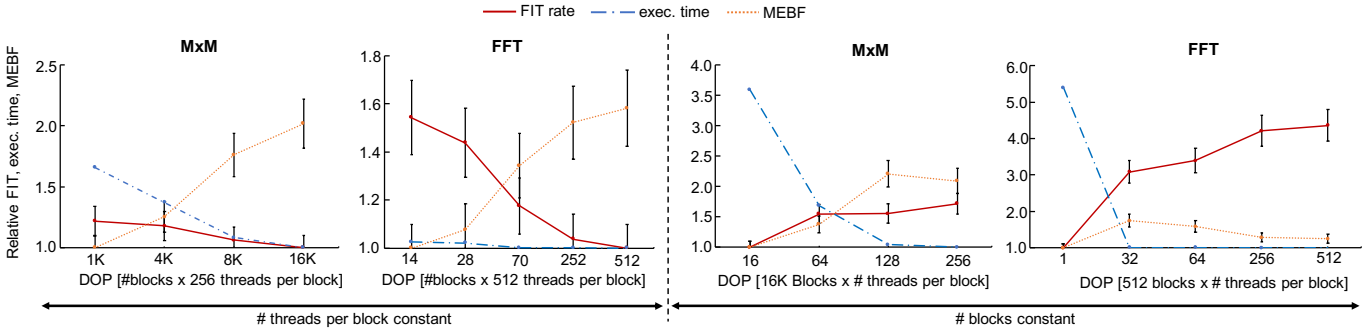

Fig. 4: Relative effects of varying the Degree of Parallelism on the FIT rate, execution time, and MEBF of MxM and FFT executed on the Fermi GPU with ECC OFF. We first maintain the number of threads per block constant (256 for MxM and 512 for FFT), increasing the number of blocks, and then keep the number of blocks constant (16K for MxM and 512 for FFT), increasing the number of threads per block. The workload is constant for all configurations.

between failures) is always higher for the optimized version, being up to $5\times$ higher for MxM. A optimized algorithm is then to be preferred, if the goal is to increase the amount of data correctly produced by the GPU. On average, the optimized version of the algorithm allows to correctly complete about $2.2\times$ more executions than the naive one.

B. Degree of Parallelism

To maximize the performances of a code on a GPU, it is fundamental to increase as much as possible the Degree of Parallelism (DOP). We consider two applications, 2048x2048 matrix multiplication (MxM) and a 512x512 FFT, implemented with increasing DOP. We keep the workload constant, gradually increasing the number of threads in which the workload is divided (reducing the number of operations each thread has to perform). We have used a Fermi GPU with the ECC turned OFF. Figure 4 shows how increasing the DOP influences the FIT, execution time, and MEBF of MxM and FFT. On the left side of Figure 4 we keep constant the number of threads per block (256 for MxM and 512 for FFT), increasing just the number of blocks. On the right side of Figure 4 we keep the number of blocks constant (16K for MxM and 512 for FFT), increasing just the number of threads per block. The number of blocks and threads per block has been chosen to fit the algorithm better and ease its coding.

1) Constant number of threads per block: On the left part of Figure 4 the DOP is increased keeping the number of threads per block constant to 256 for MxM and 512 for FFT. In both algorithms, the execution time and the FIT rate *decrease* when we increase the number of blocks. This should not surprise, given that even the configuration with the lowest number of blocks fully exploits the GPU hardware, and each block has 256 or 512 threads, ensuring that all 32 CUDA cores in each SM of the Fermi GPU are used. Thus, the DOP is increased without increasing the GPU exposed computing area but actually reducing the SM memory footprint (which influences the error rate as ECC is OFF). In fact, at most 15 blocks can be executed in parallel on the Fermi and, as we increase the DOP, each thread has less operations to perform and smaller memory requirements. As a result, a higher DOP with constant block size implies fewer operations executed and less memory stored in each SM, slightly reducing the FIT rate. Concurrently, the execution time is also reduced as a higher DOP better fits the GPU architecture. This leads to higher MEBF, with the configuration delivering higher performances having a MEBF that is $2\times$ and $1.6\times$ higher than the slower one for MxM and FFT, respectively.

2) Constant number of blocks: a different trend is observed when the DOP is increased keeping the number of blocks constant (16K for MxM and 512 for FFT) and increasing the number of threads per block (right figures). For both MxM and FFT, when the number of threads per block is increased, the FIT rate increases, and the execution time decreases. This is because increasing the number of threads per block imposes higher memory and scheduler requirements in each SM (which increases the FIT) and reduces the number of sequential operations of each thread (which reduces the execution time). As shown in Figure 4, using fewer computing cores than the 32 available per SM reduces the FIT rate but jeopardizes performances. Passing from 16 to 64 threads per block for MxM and from 1 to 32 for FFT, in fact, increases the FIT rate of $1.5\times$ and $3.2\times$, respectively, but reduces the execution time to $1/2$ and $1/5$, respectively. Intuitively, as 32 CUDA cores are available per SM, instantiating less than 32 threads per block underutilizes the GPU hardware reducing the FIT rate but compromising the GPU parallel efficiency. As shown in Figure 4, underutilizing the GPU is *not* a good reliability solution as the FIT reduction is not sufficient to compensate for the performance degradation.

When the number of threads per block saturates the available cores, as shown in the right side of Figure 4, the FIT rate is basically constant for MxM (number of threads per block from 64 to 256) and, for FFT, the FIT increase is of about $0.5\times$ (threads per block are from 32 to 512). The (slight) FIT increase is caused by the higher strain in the scheduler (more threads to manage) and the different memory distribution. For FFT, the FIT increases faster as threads interact with each other (butterfly modules), while in MxM, as there is no interaction

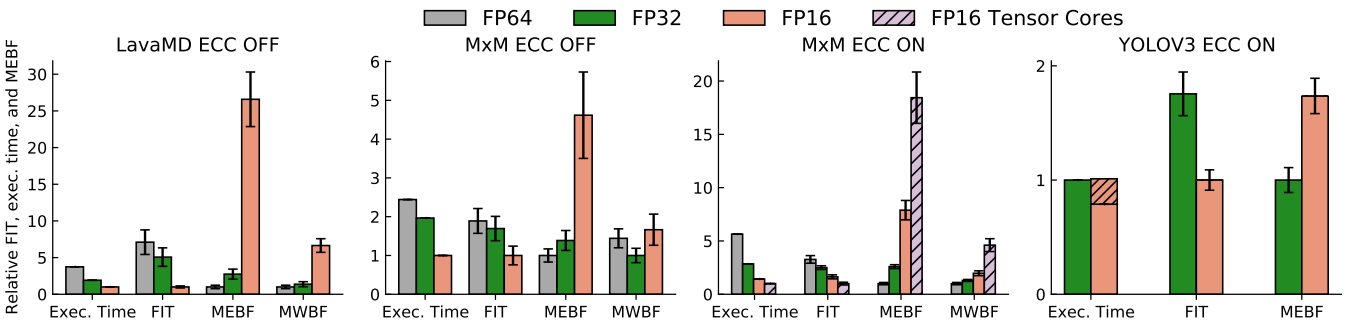

Fig. 5: Relative impact of using mixed-precision or tensor core in the Failure In Time, Execution Time, and Mean Executions or Work Between Failures of the tested Volta V100 GPU. For the FP16 YOLOv3 we have highlighted the time wasted in the casting of data.

between threads, it is easier for the scheduler to dispatch parallel threads.

What we can derive from the proposed DOP analysis is that underutilizing the GPU is never a good idea as, even if the FIT rate is reduced, the performances are jeopardized. When the GPU resources are saturated, the FIT rate normally grows slower than the execution time reduction, increasing the MEBF. Particular attention needs to be given to the block size. Having too many threads in a block can lead to memory latencies or scheduler overcharges that reduce the benefit brought by the faster execution. As a general rule, we can state that the number of threads per block should be engineered to fully exploit the GPU parallelism without saturating the SM register/cache.

C. Architectural Solutions

The GPUs architects are making available dedicated hardware resources to improve the efficiency and performances of key operations. This is the case of *tensor core* and *mixed-precision* functional units. Lower precision units are smaller, faster, and more efficient. The user can select the precision of the operations to tune the hardware utilization (and the execution time) with the application's needs. A tensor core is a hardware unit that performs a 4×4 FP16 matrix multiplication in *one* clock cycle.

Figure 5 shows, for the Volta V100 GPU, the relative FIT rate, execution time, MEBF, and MWBF for LavaMD, and Matrix Multiplication (MxM) implemented in FP64, FP32, and FP16. MxM is also executed with ECC ON and, in FP16, using tensor core. We also consider YOLOV3 executed in FP32 and FP16. To have a fair comparison, YOLO was *not* retrained. We simply cast the FP32 data to FP16 and execute the convolutions. The cast operations reduces of about 23% the benefit, in terms of performances, of executing the CNN in lower precision. In Figure 5, we also show the Mean Work Between Failure (MWBF) to have a fair comparison between the different precisions (an FP64 execution produces $4\times$ more work than an FP16 execution).

From Figure 5, it is evident that both reduced precision and tensor core improve performances. On average, the FP64 implementation has a $3.94\times$ higher execution time than the FP16 implementation. As mentioned, for YOLO, the execution time of FP32 and PF16 is similar due to data casting. A purely FP16 execution would have a 23% lower execution time. Interestingly, the FIT rate follows the same trend of the execution time, with the FP64 implementation having, on average, a $4\times$ higher FIT rate compared to the FP16 implementation. Reducing the precision of the operations significantly improves the MEBF, and even if we consider the MWBF ($4\times$ more bits are output for FP64 than for FP16), the reduced precision execution outperforms the other configurations. When ECC is enabled (only shown for MxM), the benefit in the FIT rate is maintained, indicating that the lower error rate is not solely attributed to the lower amount of memory (with ECC ON memory faults are masked) but also to the smaller sizes of the lower precision functional units. Whenever possible, the use of lower precision units is definitely recommended on GPUs.

Tensor core, as shown for GEMM with ECC ON in Figure 5, has a 63% lower FIT rate compared to the software implementation of FP16 matrix multiplication. The tensor core circuit, despite being bigger than an adder and multiplier, is slightly more reliable than the combination of adders, multiplications, and the loop control variables required to implement MxM in software. Tensor core is particularly efficient in improving the GPU performances, providing (in the FP16 implementation) a $2\times$ higher MEBF (and MWBF) than the FP16 software MxM. Whenever possible, then, the use of a tensor core is highly recommended.

D. Compiler Optimizations

Figure 6 shows the relative execution time, FIT rate, and the MEBF for the Matrix Multiplication (MXM) compiled with four different optimizations, O0, O1, O3+MinRf (O3 restricting the thread register usage to the minimum), and O3. We show the value for each metric relative to the lower value between the tested configurations. That is, the lower value for each metric in each graph is always 1. Results are obtained on a Kepler K20 GPU with ECC ON. We choose to test

the compiler optimization with ECC enabled to focus on the computation errors.

The static compiler optimizations, by changing the number and kind of machine instructions of each thread, can impact the reliability of a code running on a GPU. The O0 flag produces the least optimized machine code, including many instructions that could be simplified or reordered. The O0 version has the highest number of instructions, being 290% higher than the O3 version, the O0 version also includes $21\times$ more memory movements instructions than the O3 and $9\times$ more integer instructions than the O3 version. As shown in Figure 6, these inefficient set of instructions increases both the execution time and the error rate, consequently reducing the MEBF.

The execution time decreases significantly when the first optimization level is applied (O1), but the FIT rate remains similar to O0. As discussed in Section II, the FIT rate does not depend on the execution time but on how the code uses the resources. O1 significantly reduces the number and organization of machine instructions compared to O0. In particular, the stalls caused by instruction dependence on the code generated with O1 are still higher than the code generated with O3 (19% higher). O1 only organizes the instructions differently, reaching a similar execution time than O3. As the execution time for O1 is almost half the O0 one, and the FIT remains similar, the MEBF for O1 is $\sim 2x$ the O0 one.

O3 is the most aggressive optimization that can be applied. We consider the O3 both when the register per thread is limited to 16 (45% fewer registers than only O3 code) and when all registers can be used. The execution time for the two versions are similar (O3+MinRF has a 5% higher execution time), and O3+MinRF has 5.5% more instructions than the O3 version, necessary to perform the register spill to the memory. These additional instructions increase the FIT rate. We recall that, as the ECC is enabled, the extra register footprint of O3 compared to O3+MinRF does not increase the FIT rate.

O3 compiled code provides the optimal resource utilization, with the smallest number of machine instructions to reach the code solution (in O3, only 13% of the stalls are caused by instruction dependency). Consequently, O3 has the lowest FIT rate compared to the other configurations, and the MEBF of O3 can be $6.1\times$ higher than O0. Compiler optimizations, then, can significantly impact the reliability of a code, and better resources utilization can reduce the FIT rate while improving performances. Higher compiler optimizations, then, are definitely to be preferred.

VI. Conclusions

In this paper, we have evaluated the impact in the final code reliability of the most common optimizations available to GPU developers and architects. We have considered software optimizations, Degree Of Parallelism, compiler optimizations, and architectural solutions to speed up a set of applications executed on GPUs belonging to three different generations. The results we have presented, based on extensive neutron beam experiment campaigns, are highly encouraging, as the faster execution does not necessarily increase the GPU error rate and, even if it does, the execution time is reduced much more than the FIT rate increase. To improve reliability, then, it is not mandatory to add extra hardware or specific hardening solutions. It is actually not even necessary to sacrifice performance as, in most configurations, we can have it all: a faster configuration that can also produce more correct data. In particular, the use of dedicated hardware functional units and compiler optimizations are particularly efficient in improving the reliability-performance trade-ff.

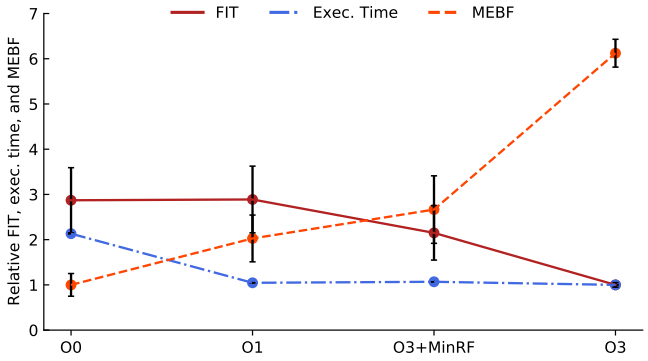

Fig. 6: Relative differences of compiler optimizations of Failure In Time, Execution Time, and Mean Executions Between Failures for MxM executed on Kepler GPU with ECC ON.

Finally, we have only provided relative FIT rates. However, we can mention that the reliability of the tested GPUs have significantly improved through the generations. Overall, the same code on a Kepler GPU has an error rate about one order of magnitude *lower* than the Fermi, and the same code on the Volta would be almost one order of magnitude more reliable than on the Kepler. The improved reliability is much higher than the one reported for similar technologies and even higher than the reported one for the memory structures of NVIDIA GPUs. If we also consider the improvement in performances and efficiency between the different GPU generations, the number of correctly completed executions are likely to increase of almost 2 orders of magnitudes. The intense research to improve GPUs' reliability has borne fruit and paves a promising path for the future.

References

[1] S. S. Mukherjee *et al.* in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.
[2] V. Sridharan *et al.* in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, 2009.
[3] C. Weaver *et al.* in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, 2004.
[4] G. Reis *et al.* in *32nd International Symposium on Computer Architecture (ISCA'05)*, 2005.
[5] Z. Jia *et al. arXiv*, 2018.
[6] NVIDIA tech. rep., NVIDIA, 2017.
[7] JEDEC tech. rep., JEDEC Standard, 2006.
[8] V. Volkov *et al.* in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.
[9] V. Volkov *et al. preprint on University of California, Berkeley*, 2011.