Using scheduling entropy amplification in CUDA/OpenMP code to exhibit non-reproducibility issues

David Defour

Using scheduling entropy amplification in CUDA/OpenMP code to exhibit non-reproducibility issues.

David Defour
LAMPS, Univ. of Perpignan, France.
Email: david.defour@univ-perp.fr

***Abstract*—Rounding error or cancellation that appears with each floating-point operations, combined with the lack of control over execution order in parallel code leads to numerical issues such as numerical reproducibility.**

In order to enhance the possibility to discover such numerical issue, in this article we propose a simple solution base on an index interposer and an index scrambler to amplify the possible combination of execution order.

I. INTRODUCTION

Determinism and numerical reproducibility on today's modern multicore architectures are affected by the hierarchical structuration of computation as well as execution concurrency. This can be encountered when targeting parallel programming, either GPU's (CUDA, SyCL or OpenCL), or OpenMP where several thousands of threads are executed and scheduled concurrently. On these system's thread scheduling is mostly unknown and can be considered unpredictable.

A common workaround is to enforce interaction between tasks or threads using memory consistency with synchronization mechanisms (locks, atomics or barriers). Although these can address the problem of memory consistency, they do not solve the problem of numerical reproducibility when dealing with floating-point numbers. Furthermore they can come with a significant performance penalty. This is a major issue as non-determinism of floating-point calculations in parallel programs causes validation and debugging issues, and may even lead to deadlocks [1], [2], [3], [4].

Numerical non-reproducibility of floating-point operations is due to the combination of two phenomena: rounding-error and the order in which operations are executed. If there exists numerous tools and solutions [5], [6] to analyse the impact of rounding errors, to our knowledge there are no tools devoted to analysing the impact due to the reorder of operations which can occur during parallel execution.

This problem can be depicted with the simplified following CUDA kernel which computes the sum of N floating-point numbers stored in table *i_val* according to their address *i_adr* in a table *res* located in global memory as it could be encountered with the bin counting or histogram problem such as those encountered in Nbody [7], Real-time simulation [8], accurate reduction scheme [9], or SQL query [10].

```
__global__ void GlobalSum(float *i_val,
           int *i_adr, float *res, int N){
  int gid =  blockDim.x*blockIdx.x+threadIdx.x;

  for(uint i=0; i<N; i+=GridDim.x*blockDim.x)
     atomicAdd(&res[i_adr[i+gid]],
           i_val[i+gid]);
}
```

Listing 1. Floating-point atomic accumulation

As we do not have information on the order in which threads will acquire access to the datum *Res*, the order in which the accumulation is actually done is unknown. For example, on a set of $N = 2^{16}$ values with a condition number[1] of 10^8 and a single output address, out of 1000 runs with 1 block of 1024 threads, 1000 different results can be obtained.

In this article, we propose to explore a Proof-Of-Concept to detect and quantify potential numerical issues due to the order in which floating-point operations are executed.

The rest of this article is organized as follows. Section II introduces the necessary background about floating-point arithmetic. Section III presents the execution model for GPU and OpenMP. Section IV describes the proposed solution to amplify scheduling scheme in order to expose potential numerical issues. Section V analyses some performance measurements on Nvidia CUDA and OpenMP executions.

[1]The condition number characterizes the numerical stability of a problem [11].

II. NUMERICAL NON REPRODUCIBILITY IN PARALLEL EXECUTION ENVIRONMENT

Floating-point (FP) numbers can represent a wide range of numbers with nearly-constant precision by approximating real numbers with a significant, an exponent, and a sign. The representation formats and operations are standardized by the IEEE-754 standard.

The results of floating-point operations have to be rounded. In the case of addition, this may lead to the absorption of the lower bits of the sum. For example the exact mathematical result of $(1 + 2^{100} - 2^{100})$ is equal to 1 whereas the computed result is either 0 or 1 depending on the order of operations. Thus, the final accuracy of floating-point summations depends on the order of evaluation. More details can be found in the main references related to floating-point arithmetic [11], [12]. This phenomenon is emphasized when executed in parallel environments.

There exist two main solutions to address the problem of numerical reproducibility. One is based on reducing rounding error and the second one is based on a deterministic execution.

A. Workaround based on reducing numerical error

Numerical reproducibility is particularly impacting regarding the summation problem which occurs during reduce scheme based on floating-point additions. For this particular problem, the solution consists in increasing the accuracy used for the accumulator. This can consist in increasing the accuracy of the accumulator (using binary64 in place of binary32 for example, or using multiprecision library [13]), compensated algorithm such as Kahan-Babuska summation [14], or a long accumulator [15], [16].

B. Workaround based on deterministic execution order

Another solution consists in enforcing an execution order. This order could be set either at software level or at execution level.

At software level, the programmer could set an order at design stage with the help of a loop iteration. However, in that case, the language, library and/or compiler could reorganize execution order (e.i. use of a *reduction* clause with OpenMP).

At execution level, it is possible to set an execution order. However, this solution could be challenging especially when no assumption can be made on the execution order of threads/block such as on GPUs. For example, on such architectures CUDA/OpenCL thread and block identifiers used at software level provide no information on the hardware scheduling order of threads and blocks. This has lead to numerous works focusing on efficient inter-block synchronization. For example, in [17] Volkov et al. propose a global software synchronization method that does not use atomic operations to accelerate dense linear-algebra constructs. In [18], [19], Xiao and Feng propose a mechanism for inter-block communication via global memory. In [20], Stuart and Owens are evaluating various implementations of barriers, mutexes and semaphores applied to Nvidia's GPU. Some solutions [21], [22], propose to use alternative thread and block identifier in place of the hardware generated one.

III. EXECUTION MODEL

Our motivation is to offer to the developers solutions to measure the numerical impact of scheduling, even those which are not possible at a given time. The goal is to determine if a given code is still valid when executed on a hypothetical architecture for example a future architecture and/or programming language/compiler with additional features.

In this section we first describe how parallelism is exploited at hardware level followed by software level and how it impacts scheduling.

A. Hardware level

Today's architecture exploit various level of data, instruction and thread parallelism.

1) Level N°0: Sub-Word Parallelism: On most processors, the register width is fixed at the architecture level. However, computation unit support multiple operand precision (FP16 and FP32), meaning that by lowering the precision the hardware follows a packed SIMD paradigm at register level.

2) Level N°1: SWAR: Similarly to what is done at level-0, architectures pack several data within a single register which correspond to SIMD Within A Register (SWAR). For example, ARM SVE is designed for variable length from 128 to 2048 bits by multiple of 128 bits. At this level, we often encounter cross-lane instruction with component swizzling ability. This corresponds to the ability to redirect individual components to and from individual processing units.

3) Level N°2: Warp: Instruction stream are scheduled simultaneously across the processing units of one or more SIMD blocks to form a subgroup called a wave (or sometimes wavefront or warp). The individual instruction within those are referred to as the lanes or threads of the wave. For example, the wavefront contains 32 threads for Nvidia GPU, and 64 threads for AMD GPUs.

4) Level N°3: SIMT: On architecture such as GPU, parallelism can be exploited using array processing contrarily to vector processing. This correspond to the concept of associative processor in Flynn's taxonomy. Spatio-temporal SIMT corresponds to the issuing of the same instruction multiple time but for different set of invocations over the individual lane of a SIMD block.

5) Level N°4: SMT: Some processor also embed the possibility to schedule instructions from another wave in order to hide long-latency instructions such as memory accesses. The same instruction stream is then issued across multiple invocation simultaneously in a lock-step fashion on the same core unit.

6) Level N°5: Core: Processor also embed several copy of the same hardware. This corresponds to core units, where every hardware resources are duplicated. In CUDA terminology, this corresponds to streaming multiprocessors (SMs). SM maintains a scoreboard for each warp to launch, depending on their type, instructions and a "fair" scheduling policy [23]. The number of core varies depending on the hardware generation as well as the version.

B. Software level

The architecture details mentioned in the previous section are usually not accessible directly, and advance have been made at the API level (either architectural or language level) to expose and give control of parallelism to the developer. In this section we describe how parallelism is exploited at software level and how it impacts scheduling.

1) OpenMP / Threads: OpenMP is a shared memory parallel programming model widely used at node level on today's supercomputers typically used to accelerate calculations within a MPI rank. It supports multiple style of parallelization based on threads (parallel regions), tasks (implicit and explicit) and work sharing (parallel loops). There are multiple synchronization constructs such as barriers, reductions, task dependencies, task wait, locks and critical sections.

With OpenMP program, numerical behavior could be impacted by the number of threads launched, the usage and/or the order in which threads are executed (middleware) and especially how reduction clause targeting floating-point reduction-identifier are handled. For this later point, the way a for loop operating a floating-point reduction behaves is dependant of the loop itself (possibility of using a reduction clause), the usage of task, SIMD lanes or Accelerator offloading (OpenMP 4.0).

If there are tools to help detect data race, however there are not accurate as they give false negatives (loops with dependencies) or false positives (ordered critical sections) and are not intended to detect numerical issues. Such tools are based on dynamic data race detection, cilkscreen [24] hybrid data race detection algorithm [25]. Regarding the specific case of OpenMP programs, there are analysis based on static analysis [26], tools such as Archer [27], Sword [28] or OMPT API based tools

2) CUDA/OpenCL: Task parallelism such as the one available in OpenMP proves to be useful to avoid work imbalanced and ease implementation of recursive construction. This sort of parallelism is not available with CUDA/OpenCL. Work distribution is solely done thanks the thread and block identifier.

Tasks executed on GPU either as CUDA or OpenCL code are divided in threads operating in SIMT mode and executed by specific hardware. We must distinguish the software and hardware organization of threads. From the developer point of view, threads are divided into three hierarchical levels: a *grid* of *blocks* of *threads*. The same code, or kernel, is executed by multiple threads running in parallel on different data. *Threads* are grouped in set of *block_size* elements in order to make so-called *blocks*. Blocks are packed in set of *grid_size* elements in order to make a so-called *grid*. Threads in a block and blocks of a grid are uniquely identified by their coordinates in the blocks and the grid. In this model, threads in a block and the blocks of a grid are virtually launched in parallel, which implies that no assumption shall be made regarding the execution order.

Blocks are dispatched among the available multiprocessor by the block schedulers. This step consists in launching a new block with a unique identifier according to available resources. The number of concurrent blocks depends on the number and version of SMs and the resources such as registers and shared memory required by the executed kernel. This step impacts determinism as no assumption can be made on how indexes are generated and is subject to variations from one run to another [29].

Traditional synchronization primitives used in parallel program such as mutexes, barriers, and semaphores are limited on GPU to intra-block synchronizations. Only atomic and fence operations provide basic support for inter-block communication. On Nvidia hardware, atomic instructions were introduced with compute capability 1.1, and with compute capability 2.0 for atomic addition operating on 32-bit floating-point values in global and shared memory. Atomic floating-point operations are necessary, first to provide the substrate for high

performance floating-point operations, and second, to preserve the memory consistency necessary to deal with thousands of in flight threads.

However, Block synchronization is challenging, as the CUDA programming model does not support it. The only safe solution consists on splitting kernels into subkernels, as a kernel launch involves an implicit synchronization barrier. Alternatively, resident kernel techniques take advantage of the fact that once launched, a block continue its execution until completion freeing resources only at the end.

Block barrier proposed by Feng in [18] is working only when the number of launched blocks is less than the number of blocks that could be executed concurrently on the hardware. In case this assumption is not met, deadlocks may occur.

Today's GPUs also offer grid nesting and synchronization, called dynamic parallelism. A parent grid of block of threads is able to launch kernels called child grids. Child grid launched with dynamic parallelism always complete before the parent grids that launch them. Synchronization at parent grid is possible and parent/child grid have a fully consistent view of global memory at grid level.

IV. PROPOSED SOLUTION

During software development, validation is usually done for a limited number of configurations and therefore scheduling combination. In this article we propose to amplify legal scheduling combination thanks to an accumulator identifier, an interposer and a scheduling amplifier (Figure 1).

A. Accumulator Identification

One of the common source of numerical error is related with chain of floating-point accumulation. Longer is the chain of accumulated value, larger is the possibility of encountering numerical issues. Therefore, we propose to spot precisely in a code where are located long chain of accumulation. This step could be done semi-automatically by associating with each floating-point value the number of consecutive floating-point addition/-subtraction done to produce a given results. We have developed a pass based on a modified version of Nsan [30], where each floating-point number corresponds to a pair of number (V, N), with V the floating-point value and N the number of consecutive additions/subtractions. Each floating-point operation $R = A \cdot B$ is replaced in the Nsan framework by the following operation:

$(V_R, N_R) = (V_A, N_A) \odot (V_B, N_B)$

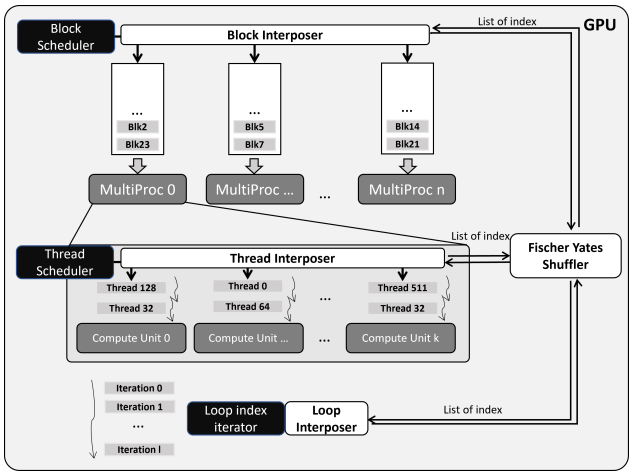

Fig. 1. Proposed configuration based on a multilevel interposer and Fischer-Yates Shuffler

where

$$V_R = V_A \odot V_B \text{ for } \odot \in \{+, -, \times, \div\}$$
$$N_R = \begin{cases} N_A \odot N_B & \text{for } \odot \in \{+, -\} \\ 0 & \text{for } \odot \in \{\times, \div\} \end{cases}$$

Programs are recompile with this modified version of Nsan and run on dataset. The output correspond for a given execution, to a list with the largest chains of accumulation which are either sequential, parallel or tree-based one.

B. Interposer

The interposer first objective is to intercept and build either set of index (block/thread) or iteration (e.i. for loops). This interposer collects information from the original code on how indexes are generated. This information includes the lower and upper bound index and the increment.

Once these information are collected, the interposer replace the original indexes with scrambled ones. If this could be done easily for loop index, and thread index, it is however challenging for accelerated devices (GPU) as user's do not have control over block and thread scheduling (hardware managed resources). In that case, the objective is to gain control over threads/block scheduling such that their execution regarding floating-point operations could potentially be scrambled.

1) Grid level: As mention in section III, if considering Nvidia's GPU, threads of blocks of a grid are scheduled as set of 32 consecutive threads, or warps. The solution has to be deadlock-free. For example the solution proposed by Feng in [18] is working only when

the number of launched blocks is less than the number of blocks that could be executed concurrently on the hardware. In case this assumption is not met, deadlocks may occur.

Therefore, we propose to use a software generated block identifier in place of the one offered by the hardware block scheduler. As we do not have any information on which block will be executed first, we propose to rely on a CPU generated array. This array will then be accessed based on the hardware block index whenever needed and the value stored within the array used as the new index. This solution allows us to offload to the CPU the generation of a good permutation of index. Notice that the memory required with this solution could potentially be very high as a grid can embed 2^{16} blocks for each of the 3 possible dimensions. However, in general the number of blocks used remains small enough (less than 2^{15}) to not impact memory. An alternative could be to generate scrambled index directly on the device, however this will requires global lock and atomic as in [21] which may impact performance.

2) Block level: At block level, thread index have to be generated directly on the hardware to avoid large memory overhead. Indeed, each block can embed up to 1024 threads each. In that case, this requires generating vectors of 1024 index on the CPU for each executed block potentially requiring $8.2^{10}.2^{16}.2^{16}.2^{16} = 2^{61}$ different combinations.

Therefore, thread indexes are handled at block level and located in shared memory, accessible by each thread of a block. A the beginning of a block execution, one thread is in charge of scrambling the array of thread index. Once done, kernel execution can continue with the new index being replaced by the scrambled one. An overview of the code is given in listing 2.

```
#include <curand.h>
#include <curand_kernel.h>

__device__ dim3 *bckScblIdx;
__shared__ dim3 *trdScrblIdx;

// Use preprocessor to automaticaly use
// scrambled index
#define blockIdx.x (bckScrbIdx[blockIdx.x])
#define threadIdx.x (trdScrbIdx[threadIdx.x])

__device__  void
d_FisherYates(int *idx, int n){
    int i,j;

    for(i=n-1;i>0;i--){
        j = curand(&__MTPG_state[blockIdx.x]);
        j %= (i+1);
        swap( &idx[i], &idx[j]);
    }
}

__device__ void setup_kernel(){
    if (threadIdx.x==0){
        for(int i=0; i<blockDim.x; i++)
            tdScrblIdx[threadIdx].x = i;

        d_FisherYates(trdScrblIdx, blockDim.x);
    }
    __ syncthreads();
}
```

Listing 2. Thread reindexing

C. Scheduling amplifier

Once the set of possible index or iteration is build, we use a combination amplifier. This amplifier is based on Fischer-Yates shuffler which generate a random permutation of the input set. We selected the Sattolo's variant of the Fischer-Yates algorithm [31] as it produces in an efficient manner an unbiased permutation. With this variant the time taken is proportional to the number if input items thanks to an in-place shuffles.

D. Testing framework

As the amount of code modification could be potential high, we operate in the following order.

1) First, we alter work distribution. At OpenMP level, we change the number of thread and at accelerator level, we change the number of thread per block and block per grid. (Applying these modifications are not always possible. We noticed during our test, see section V-B that for some applications the number of thread is fixed by the applications and could not be altered).
2) Second, we change work scheduling. At OpenMP level, this is done by using a randomized scheduler [32], altering work distribution and at accelerator level this is done using the proposed solution.
3) Third, we alter loop iteration order as this modification is the most performance impacting modification.

V. Tests

In order to demonstrate how the proposed solution operates on real code, we measured the amplification of scheduling as well as the numerical variability on SHOC[33] and PARBOIL [34] benchmarks on a machine with an Intel Xeon E5645 with 6 cores and an Nvidia GeForce RTX2060 with 30 streaming multiprocessors.

A. Measure of scheduling amplification

We designed tests to demonstrate and quantify the benefit of our scheduling amplifier. To do so, we focused first on the order in which threads and block are processed.

We have tested our block and thread scrambler against the hardware block and thread scheduler. Figure 2 compares the scheduling with and without our scrambler at block level. Similar figures are obtained at thread level. The X axis corresponds to the execution test done one after another (as it would be done by a developer to manually test the numerical validity of an application). The Y axis corresponds to the possible scheduling. Each scheduling is encoded as a floating-point number in the range $[0.0, 1.0]$ using an arithmetic encoding of the sequence of index identifier according to their execution order [35]. The used coder started with equal probability for each index. The execution order is collected using an index gathered atomically by each thread/block. We choose the arithmetic encoding as it allows us to spot precisely the scheduling order distribution.

We can observe that without the entropy amplification, the scheduling combinations are very limited and does not cover well the scheduling possibility (the two graphics on the first line). In other words, it means that even though we cannot make assumption on how block and thread index are handled in hardware, there is a high correlation factor among them. By observing the two graphics on the second line, we can deduce that the Fished-Yates scrambling functions appears to provide less correlated scheduling combination with a larger spectrum of possibility coverage.

B. Measure of numerical variations

In order to demonstrate the benefit of the proposed solution on real program, we selected programs from the SHOC and PARBOIL benchmarks listed in Table I as those programs include floating-point calculations and in particular, floating-point accumulations.

For each of those benchmark, thanks to the accumulator identifier we have spotted at least one accumulator where the interposer could be used. We have then interfaced block/thread/loop index with the amplifier as described in section IV-C. Each program where run multiple times and we counted over 100 runs, the number of time we could observe numerical differences, as well as the numerical variability (norm2 of the relative error over each computed results) on the output for the following configurations:

1) Configuration 1: Use the same execution configuration for each run.

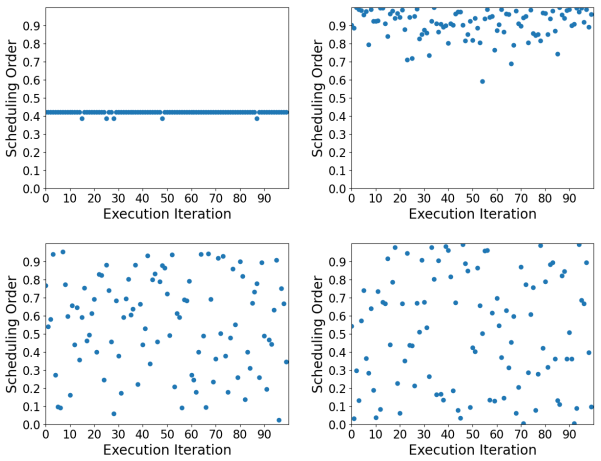

Fig. 2. Measure of the entropy for block scheduling with the hardware scheduler solely (first line), and combined with the entropy amplifier (second line), for 8 (first column) and 320 (second column) launched block over 100 executions.

TABLE I
SELECTED BENCHMARKS FROM SHOC AND PARBOIL.

Name	Description
Cutcp	Compute short-range electrostatic potentials induced by point charges in a 3D volume.
LBM	Lid-Driven Cavity fluid dynamics simulation using the Lattice-Boltzmann Method.
MRI-Q	Matrix computation used in a 3D magnetic resonance image reconstruction.
MRI-GRID	Compute for a 3D grid the contribution of every data point onto its neighboring grid points.
SGEMM	A register-tiled matrix-matrix multiplication, with default column-majored layout.
SPMV	Sparse-Matrix Dense-Vector Product.
Stencil	Seven point stencil.
MD	Molecular dynamics.
Reduction	Sum reduction operation using single precision floating-point data.
NeuralNet	Training of neural networks using backpropagation.

2) Configuration 2: Change the execution configuration (number of threads, block, ...) between each run.
3) Configuration 3: Change work scheduling between each run.
4) Configuration 4: Change loop indexing between each run.

Results are reported in table II. Out of these experiments, we made the following observations:

- With the exception of omp_cutcp, no numerical variations could be observed by executing the same program multiple time (configuration 1 column), as a developer would do during development phase.

TABLE II
OBSERVED NUMERICAL VARIABILITY OVER 100 RUNS FOR VARIOUS CUDA/OPENMP PROGRAMS ACCORDING TO 4 DIFFERENT EXECUTION CONFIGURATIONS.

	Conf. 1		Conf. 2		Conf. 3		Conf. 4	
	error	# diff.	error	# diff.	error	# diff.	error	# diff.
cuda_cutcp	0	0	-	-	0	0	7,52E+00	12482
omp_cutcp	3,02E-01	11970	4,70E-01	12253	7,05E-01	12135	-	-
cuda_MRI-GRID	0	0	0	0	0	0	5,65E-03	59
omp_MRI-GRID	0	0	9,83E-05	23	7,27E-04	32	1,6	87
cuda_MRI-Q	0	0	0	0	0	0	1,23E+00	64427
omp_MRI-Q	0	0	0	0	0	0	1,17E+00	64461
cudasdk_Reduction	0	0	7,30E-03	9	0	0	9,24E-02	100
cudasdk_Jacobi	0	0	4,79E-03	3	6,90E-01	100	8,87	100
cuda_SGEMM	0	0	0	0	0	0	1,75E-03	20482
omp_SGEMM	0	0	0	0	0	0	1,75E-03	20482
cuda_SPMV	0	0	0	0	0	0	3,69E-06	11948
omp_SPMV	0	0	0	0	0	0	1,16E-07	11948
(cuda/omp)_Stencil	0	0	0	0	0	0	0	0
(cuda/omp)_lbm	0	0	0	0	0	0	0	0

- Changing the execution configuration (configuration 2), leads to numerical variations only for a few programs. Those observations were hindered for the cuda version as the number of valid configurations to test is limited.
- It was not possible to alter the execution configuration, (configuration N°2 and 4), for cutcp for the cuda and omp version respectively. This is due to the way these two programs were design: the number of thread per block and block per grid is hard-coded or due to dependencies between loop iterations.
- Altering work scheduling (configuration 3) leads to an increase of the norm2 error for Jacobi or MRI-GRID which relies on atomic floating-point accumulations.
- The largest number of differences, norm2 error, and number of benchmark impacted were observed for configuration 4 which corresponds to interfacing the amplifier with scrambled loop index.
- Despite the fact that stencil and lbm benchmark uses floating-point accumulators, it was not possible to observe numerical variability. In that case, it is due to the coding style adopted by the developer. For these two benchmarks, the chain of accumulation is hardcoded and independent of any index. However, it was possible to observe numerical variation using the *–fast-math* compiler option, which allow the compiler to reorganise sequence of operations.

VI. CONCLUSION

In this article, we were concerned by exploring solutions to amplify numerical variability linked with the order of operations which occurs especially during floating-point reduction pattern for parallel execution either in OpenMP or CUDA.

We have observed that, exposing numerical variability the traditional way, with multiple execution, or by changing execution configuration settings, may not work, or expose a sufficient large enough numerical error to detect numerical bug.

We proposed a solution, consisting of an accumulator identifier based on llvm to locate chain of floating-point additions, an interposer to replace indexes by software generated one, and an index amplifier based on a Fischer-Yates shuffler. We have tested this solution against OpenMP and CUDA benchmarks, and demonstrated that this solution could help developers to detect numerical issues by amplifying scheduling combination.

As future work, we are planning to explore how to take into account other sources of numerical errors such as simdization, matrix operations, or usage of MPI primitives.

ACKNOWLEDGEMENT

Funding: This work was supported by the ANR-20-CE46-0009 InterFLOP project

REFERENCES

[1] L. S. Blackford, A. Cleary, J. Demmel, and I. Dhillon, "Practical experience in the dangers of heterogeneous computing," *Lecture Notes in Computer Science*, vol. 1184, pp. 57–64, 1996.

[2] S. Keum, R. G. Jr., J. Gao, X. Yang, and T. Kuo, "Effect of parallel computing environment on the solution consistency of cfd simulations—focused on ic engines." *Engineering*, vol. 9, pp. 824–847, 2017.

[3] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide, Ch. 6. Accuracy and stability*. SIAM, 1997.

[4] Q. Meng, A. Humphrey, J. A. Schmidt, and M. Berzins, "Preliminary experiences with the uintah framework on intel xeon phi and stampede," in *Extreme Science and Engineering Discovery Environment: Gateway to Discovery, XSEDE13, San Diego, CA, USA - July 22 - 25, 2013*, N. Wilkins-Diehr, Ed. ACM, 2013, pp. 48:1–48:8. [Online]. Available: http://dl.acm.org/citation.cfm?id=2484762

[5] C. Denis, P. D. O. Castro, and E. Petit, "Verificarlo: Checking floating point accuracy through monte carlo arithmetic," in *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*. Los Alamitos, CA, USA: IEEE Computer Society, jul 2016, pp. 55–62. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ARITH.2016.31

[6] F. Jézéquel and J. M. Chesneaux, "Cadna: a library for estimating round-off error propagation," *Comput. Phys. Commun*, vol. 178, no. 12, pp. 933–955, 2008.

[7] (2012, november) N-body: Fp atomics v. recomputation. [Online]. Available: http://www.cudahandbook.com/2012/11/n-body-fp-atomics-v-recomputation/

[8] J. Allard, S. Cotin, F. Faure, P.-J. Bensoussan, F. Poyer, C. Duriez, H. Delingette, and L. Grisoni, "Sofa an open source framework for medical simulation," in *Medicine Meets Virtual Reality (MMVR'15)*, Long Beach, USA, February 2007.

[9] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamarić, D. H. Ahn, and G. L. Lee, "Determinism and reproducibility in large-scale HPC systems," in *Informal Proceedings of the 4th Workshop on Determinism and Correctness in Parallel Programming (WoDet 2013)*, 2013.

[10] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2010, Pittsburgh, Pennsylvania, USA, March 14, 2010*, ser. ACM International Conference Proceeding Series, D. R. Kaeli and M. Leeser, Eds., vol. 425. ACM, 2010, pp. 94–103.

[11] N. J. Higham, *Accuracy and stability of numerical algorithms*, 2nd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 2002.

[12] J.-M. Muller and al., *Handbook of floating-point arithmetic*. Birkhäuser, 2010.

[13] T. Grandlund, "GNU MP: The GNU Multiple Precision Arithmetic Library," http://gmplib.org.

[14] I. Babuška, "Numerical stability in mathematical analysis," in *Proc. IFIP Congress*, ser. Information Processing 68. North-Holland , Amsterdam, The Netherlands, 1969, pp. 11–23.

[15] U. W. Kulisch, *Computer arithmetic and validity*, 2nd ed., ser. de Gruyter Studies in Mathematics. Berlin: Walter de Gruyter & Co., 2013, vol. 33, theory, implementation, and applications.

[16] S. Collange, D. Defour, S. Graillat, and R. Iakymchuk, "Full-Speed Deterministic Bit-Accurate Parallel Floating-Point Summation on Multi- and Many-Core Architectures," INRIA, DALI–LIRMM, LIP6, ICS, Tech. Rep. HAL: hal-00949355, Feb. 2014.

[17] V. Volkov and J. W. Demmel, "LU, QR and Cholesky factorizations using vector capabilities of GPUs," Department of Electrical Engineering and Computer Science, University of California, Berkeley, LAPACK Working Note 202, May 2008.

[18] W. chun Feng and S. Xiao, "To GPU synchronize or not GPU synchronize?" in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, 2010, pp. 3801–3804.

[19] S. Xiao and W. chun Feng, "Inter-block GPU communication via fast barrier synchronization," in *IPDPS*. IEEE, 2010, pp. 1–12.

[20] J. A. Stuart and J. D. Owens, "Efficient synchronization primitives for GPUs," *CoRR*, vol. abs/1110.4623, 2011.

[21] D. Defour and C. Collange, "Reproducible floating-point atomic addition in data-parallel environment," in *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2015, pp. 721–728.

[22] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley, 2010.

[23] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, pp. 56–69, March 2010.

[24] M. Feng and C. E. Leiserson, "Efficient detection of determinacy races in cilk programs," in *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 1–11. [Online]. Available: https://doi.org/10.1145/258492.258493

[25] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," *ACM SIGPLAN Notices*, vol. 38, no. 10, pp. 167–178, Oct. 2003.

[26] F. Ye, M. Schordan, C. Liao, P.-H. Lin, I. Karlin, and V. Sarkar, "Using polyhedral analysis to verify OpenMP applications are data race free," in *2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness)*, 2018, pp. 42–50.

[27] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. S. 0001, G. L. Lee, J. Protze, and M. S. Müller, "ARCHER: Effectively spotting data races in large OpenMP applications," in *IPDPS*. IEEE Computer Society, 2016, pp. 53–62.

[28] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, I. Laguna, G. L. Lee, and D. H. Ahn, "SWORD: A bounded memory-overhead detector of OpenMP data races in production runs," in *IPDPS*. IEEE Computer Society, 2018, pp. 845–854.

[29] D. Defour, "Measuring predictability of nvidia's GPU schedulers: Application to the summation problem," in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, 2015, pp. 17–24.

[30] C. Courbet, "NSan: A floating-point numerical sanitizer," *CoRR*, vol. abs/2102.12782, 2021.

[31] S. Sattolo, "An algorithm to generate a random cyclic permutation," *Information Processing Letters*, vol. 22, no. 6, pp. 315–317, 1986.

[32] F. M. Ciorba, C. Iwainsky, and P. Buder, "OpenMP loop scheduling revisited: Making a case for more schedules," *CoRR*, vol. abs/1809.03188, 2018.

[33] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units, (3rd GPGPU'10)*, ser. ACM International Conference Proceeding Series, vol. 425, Mar. 2010, pp. 63–74.

[34] J. A. Stratton, C. Rodrigues, I. jui Sung, N. Obeid, L. wen Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, "University of illinois at urbana-champaign center for reliable and high-performance computing," Tech. Rep., May 22 2012.

[35] G. G. L. Jr., "An introduction to arithmetic coding," *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 135–149, Mar. 1984.