Avoiding synchronization to accelerate a CFD solver in GPU

Ernesto Dufrechou*, Pablo Ezzatti* and Gabriel Usera†
*Instituto de Computación, Universidad de la República, Montevideo, Uruguay.
Emails: {edufrechou,pezzatti}@fing.edu.uy
† Instituto de Mecánica de los Fluidos e Ingeniería Ambiental
Universidad de la República, Montevideo, Uruguay.
Email:gusera@fing.edu.uy

***Abstract*—The caffa3d.MBRi is an open source, GPU-aware, general purpose incompressible flow solver, aimed at providing a useful tool for numerical simulation of real world fluid flow problems that require both geometrical flexibility and parallel computation capabilities to afford tens and hundreds million cells simulations. At the core of this tool there are a number of linear solvers that can be selected according to the characteristics of the problem to solve. For band matrices, the most efficient linear solver included in caffa3d.MBRi is the Strongly Implicit Procedure (SIP) solver. The parallelization of this solver follows the hyper-planes strategy, where the computations in one hyper-plane bare no dependencies and can be executed in parallel, while the hyper-planes have to be processed sequentially.**

In this work, we analyze this strategy to reach an efficient GPU implementation of the SIP solver for the caffa3d.MBRi. In particular, we design and implement a self-scheduling procedure to avoid the overhead of CPU-GPU synchronization implied by the hyper-planes strategy, outperforming the standard GPU implementation of the SIP by approximately 2×.

***Keywords*-Graphics processors; Strongly Implicit Procedure; Computational fluid dynamics; Asynchronous computations.**

I. Introduction

The solver caffa3d.MBRi [1], [2] is an open source, general purpose incompressible flow solver aimed at providing a useful tool for numerical simulation of real world fluid flow problems. Besides providing the flexibility necessary to process complex geometries, the model also aims to have the computational efficiency required to solve large-scale problems. For this reason, the caffa3d.MBRi is currently undergoing transformations oriented to improve the exploitation of modern parallel platforms.

The development of hardware platforms in the last decade was restrained by several physical limits. To mitigate this deceleration, one of the most widespread strategy is the leverage of many computational units concurrently. Aligned with this idea, Graphics Processing Units (GPUs) have evolved to become one of the most important parallel architectures nowadays, completely changing the landscape of High Performance Computing (HPC) infrastructures [3]. For this reason, it is crucial to adapt the parallel computing strategies of the caffa3d.MBRi to fully exploit the capabilities of these sort of devices.

The process of enabling the use of GPUs in the model has advanced significantly in the last years, with about three quarters of the different physical modules being already functional in GPU. To avoid memory transfers the model data is stored in the GPU global memory from the beginning of the simulations. Furthermore, to promote coalesced memory access and prevent the excessive need of atomic functions in several routines across the solver, structured grid field arrays are stored using a red-black layout. This strategy suits well most of the solver routines, and even simple linear solvers that can be optionally used in caffa3d.MBRi, like the Red-Black solver. However, the most efficient linear solver included in caffa3d.MBRi for the band matrices is the SIP Solver [4].

Departing from the work of Igounet et al., that presented an efficient GPU implementation of the SIP solver [5], and later evaluated that implementation on the caffa3d.MB model [6], [7], in this work we present a new implementation of the SIP for the caffa3d.MBRi. The new proposal leverages a special memory scheme that better suits the hyper-planes strategy that the SIP follows, since the red-black layout makes it difficult to attain an efficient memory access in the forward elimination and backward substitution phases of the solver. However, the main contribution of this work is a novel mechanism that seeks to reduce the cost of the synchronization that must take place between the processing of two consecutive hyper-planes. This mechanism uses atomic operations to control the progress of the computations, and leverages the particularities of the GPU execution model to avoid deadlock situations. Similar ideas have been proposed in recent years to decrease the overhead due to kernel launches in the context of sparse numerical linear algebra operations [8], [9], [10], [11].

The rest of the article is structured as follows. Section II summarizes the main aspects of the caffa3d.MBRi CFD model while Section III introduces the theoretical aspects of the SIP solver. We then describe our proposal in Section IV. After that, the experimental evaluation is showed in Section V. Finally, the principal conclusions arrived in our effort and the future lines of work are summarized in

Section VI.

II. caffa3d.MBRi model

The solver caffa3d.MBRi is a solver designed for the simulation of real world fluid flow problems providing geometrical flexibility and the posibility of handling up to hundred-million cells simulations. Geometrical flexibility is provided in this model by using a block structured grid approach, combined with the immersed boundary condition method [12] to address even the most complex geometries with little meshing effort and preserving the inherent numerical efficiency of structured grids [13], [14].

The same block-structured framework provides the basis for parallelization through domain decomposition under a distributed memory model using the MPI library. A compact set of encapsulated calls to MPI routines provides the required high-level communication tasks between processes, or domain regions, each composed by one or several grid blocks.

The solver has been applied and validated in several different fields, including wind energy and wind turbines simulations [15], blood flow in arteries, and atmospheric pollutants transport [1]. For a full description of the solver capabilities please see [1].

A. *Caffa in GPU*

The solver is currently being ported to GPU, with about three quarters of the different physical modules being already functional in GPU. To minimize inefficiencies due to memory transfers between CPU and GPU, field arrays for velocity, pressure and other physical magnitudes are permanently stored at the GPU global memory, even at the expense of intense GPU RAM utilization. Currently about 1 GB RAM is required every 1.5 million cells in a typical simulation. Full field arrays are only transferred between CPU and GPU at the beginning of the simulation and before writing intermediate or final output to disk. For this purpose a copy of those fields is also permanently stored at CPU memory, which enables to overlap disk latency with ongoing computations. MPI communications between nodes are overlapped with computations in a similar fashion.

III. The SIP solver

The Strongly Implicit Procedure (SIP) [4] is an iterative method for band linear systems derived from the solution of elliptic Partial Differential Equations (PDEs) on regular grids. In these cases, the linear system can be expressed as $Ax = b$, where the matrix A penta- or hepta-diagonal.

The procedure can be described as a variant of the incomplete LU factorization, followed by the solution of the two corresponding triangular systems.

Algorithm 1 describes the main steps of the procedure. First a LU factorization without fill-in is computed, using a parameter α based on the properties of the PDE so that $\hat{L}$ and $\hat{U}$ are good approximations of L and U [16]. The value typically used for α is 1.8 (more details can be found in [16]).

Algorithm 1: Strongly Implicit Procedure (SIP)

```
1 Input: $A, b$
2 Output: $x$
   1: Do incomplete LU decomposition $\hat{L}\hat{U} \approx A$
   2: Calculate initial residual: $r_0 = b - Ax_0$
   3: while residual is not small enough do
   4:     Calculate vector $R_n$ (forward substitution):
          $R_n = \hat{L}^{-1} r_n$
   5:     Calculate $\delta x$(backward substitution): $\hat{U}\delta x = R_n$
   6:     Update solution: $x_{n+1} = x_n + \delta x$
   7:     Update residual: $r_{n+1} = b - Ax_{n+1}$
   8: end while
```

This step can be solved in sequential order beginning at the southwest corner of the grid (of size $N_i \times N_j$ in the 2-D case) following this computations:

$$\hat{L}^l_W = A^l_W/(1 + \alpha \hat{U}^{l-N_j}_N) \tag{1}$$

$$\hat{L}^l_S = A^l_S/(1 + \alpha \hat{U}^{l-1}_E) \tag{2}$$

$$\hat{L}^l_P = A^l_P + \alpha(\hat{L}^l_W \hat{U}^{l-N_j}_N + \hat{L}^l_S \hat{U}^{l-1}_E) - \hat{L}^l_W \hat{U}^{l-N_j}_N - \hat{L}^l_S \hat{U}^{l-1}_E \tag{3}$$

$$\hat{U}^l_N = (A^l_N - \alpha \hat{L}^l_W \hat{U}^{l-N_j}_N)/\hat{L}^l_P \hat{U}^l_E = (A^l_E - \alpha \hat{L}^l_S \hat{U}^{l-1}_E)/\hat{L}^l_P. \tag{4}$$

Using the matrices $\hat{L}$ and $\hat{U}$ obtained in the approximate factorization, the SIP method iterates on (2), (3), and (4) until residual is small enough. R^l can be easily computed by

$$R^l = (r^l - \hat{L}^l_S R^{l-1} - \hat{L}^l_W R_l - N_j)/\hat{L}^l_P.$$

This equation has to be solved considering a increasing order of the index l. After R^l is computed, δx can be obtained considering a decreasing order of the index l as:

$$\delta x^l = R^l - \hat{U}^l_N \delta x^l + 1 - \hat{U}^l_E \delta x^l + N_j.$$

A. *The hyperplanes strategy*

To compute the SIP in a parallel machine, the usual strategy is to organize the procedure by hyper-lines for the 2D case, or hyper-planes for the 3D case. these strategies are well known, and are described in detail in the technical report of Deserno et al. [17], which presents a thorough analysis of the implementation of the SIP method on shared memory multiprocessors. The parallelization strategy is based on that (in the 2D case) processing a certain point (i, j) of the grid, only requires the values in the positions

$(i, j-1)$ and $(i-1, j)$. This means that in step h of the procedure, those points of the grid for which $i+j=h$ present no data dependencies between each other and can be processed in parallel. These groups of points are called hyper-lines. The same is valid for the 3D case, only that now the points (i, j, k) for which $i+j+k=h$ are independent, forming hyper-planes. Therefore, the procedure to compute the SIP proceeds by increasing hyper-plane index in the *forward-substitution* stage, and by decreasing hyper-plane index in the *backward-substitution* stage, computing all the points that belong to the current hyper-plane concurrently.

IV. Proposal

Our GPU implementation of the SIP solver for the *caffa3d.MBRi* is composed by three main routines. `ComputeLuCoefficientsSIP` performs the initial approximate LU factorization, `SolveLUforFiForwardSIP` performs the forward substitution phase, and `SolveLUforFiBackwardSIP` performs the backward substitution.

To promote coalesced memory access and prevent excessive need of atomic functions in several routines across the solver a red-black scheme is adopted to store structured grid field arrays in GPU global memory. This strategy suits well most of the solver routines and even simple linear solvers that can be optionally used in caffa3d.MBRi, like the Red-Black linear solver. However, the red-black scheme is not suitable for the SIP Solver since in the three aforementioned routines, the memory is accessed by hyper-plane, and maintaining the red-black organization undermines the coalesced access mechanism. Thus, a special memory scheme has been adopted for the SIP solver routines, where the data corresponding to the grid points of one hyper-plane are contiguous in memory.

As the computations corresponding to the grid points of one hyper-plane depend on the computations of the previous hyper-plane, the hyper-planes have to be processed sequentially, and a synchronization is needed between the computation of two contiguous hyper-planes. Our baseline GPU version of the routines performs this synchronization by assigning a single GPU kernel to each hyper-plane, and launching them sequentially from a CPU thread. This will result in a queue of kernels in the default GPU stream that will be implicitly synchronized.

A. Overcoming the synchronization overhead

One drawback of the previous strategy is that the computations required for each hyper-plane are relatively simple and highly parallel, which makes these kernels very lightweight. Although the time consumed by a kernel launch is extremely small, it can be a significant percentage of the total runtime of a kernel that computes a single hyper-plane. Thus, in large-scale scenarios, where there can be potentially thousands of hyper-planes, the overhead due to kernel launches can be significant.

To reduce this overhead, we propose an explicit synchronization mechanism that works inside the kernel to avoid the implicit synchronization given by the multiple kernel launches. As the *__syncthreads()* primitive can be used to synchronize the threads within a block, our procedure only has to deal with the synchronization of thread blocks. Similar ideas were leveraged in the past by other authors [18], [19].

An outline of the proposed mechanism is the following:

- As it is common practice, we define a constant number of *block_size* threads per block. Therefore, a hyper-plane of n grid points is processed by $nb = \lceil n/\textit{block\_size} \rceil$ blocks of threads.
- Before invoking the main kernel, we initialize an array `blocks_x_hyp` in the global memory of the GPU with the nb value that corresponds to each of the hyper-planes. We perform this initialization using a simple kernel that sets this value for each hyper-plane concurrently. This is performed only once for each of the three steps of the SIP solver, so the computation time of this operation is practically negligible.
- In the main kernel, before processing the grid points of the next hyper-plane, each block must actively wait for the value corresponding to the current hyper-plane in the `blocks_x_hyp` array to become 0. This value will be decreased by every thread block that has a block index lower than nb for the current hyper-plane. We take advantage of the CUDA execution model, which removes a warp from execution whenever it must wait for the result of a global memory load, yielding the execution to another warp. This avoids a deadlock situation between different blocks in one multiprocessor when accessing the `blocks_x_hyp` array.
- Once the value corresponding to the current hyper-plane in `blocks_x_hyp` is 0, the hyper-plane has been processed in full, and the dependencies of the next hyperplane have the correct values. Then, the block advances one hyper-plane and updates the corresponding values.
- Finally, after all the threads in the block have performed their processing, the first thread decrements the value in `blocks_x_hyp` corresponding to the current hyper-plane through an atomic operation.

The geometry of the grid implies that the number of grid points that lie in each hyper-plane varies. As in our proposal, a single kernel processes all the hyper-planes, the thread grid has to be configured considering the maximum number of grid points in one hyper-plane n_{max}. Thus, the total number of blocks launched is $n_{max}/\textit{block\_size}$. It is important to note that, because some of the first and last hyper-planes contain much less grid points than the average, some thread blocks will start idle. The CUDA execution model does not determine a specific order for the issue of thread blocks. Additionally only a subset of the thread blocks of the grid will be active at a given instant, while the rest of the blocks will be inactive, waiting for the active bocks to exit. This can imply that an active block that has to process hyper-plane h must wait for an inactive block to finish processing hyper-plane $h-1$, which would cause a deadlock. For this reason, instead of using the built-in block identifier, we use a global variable which each block has to atomically increment before proceeding with any other computation. This prevents the existence of an inactive block with a higher index that any active block.

V. Experimental evaluation

The experimental evaluation considers three variants of the solver: Caffa$_{CPU}$ performs all the computations on the CPU, Caffa$_{GPU}$ computes the SIP solver on the GPU using kernel launches to synchronize between hyper-planes, and Caffa$_{SF}$ computes the SIP solver on the GPU with our in-kernel synchronization strategy. First, we perform a comparison of total runtime between the three versions for a typical CFD problem, and then we study the two GPU versions more closely to compare the two synchronization strategies.

Before advancing to the experimental results, we describe the hardware and software, as well as the problems employed in our tests.

A. Test cases

The fluid dynamics setup chosen for the benchmarks computed in this work is that of the 3D turbulent backward facing step (BFS) at Re=5.0e+4, relative to channel height. This benchmark is frequently used due to the inherent complexity of the turbulent flow arising from sudden flow expansion downstream from the step.

Figure 1 presents an instantaneous stream-wise velocity field, normalized by the uniform inlet velocity, where the turbulent nature of the flow is captured. Non stationary simulations were run with dimensionless time step of dt*U/L = 1.0e-2, uniform inlet conditions in the upper half of the channel and null gradient developed outflow conditions downstream. A simple Smagorinsky LES turbulence model was applied as described in [2].

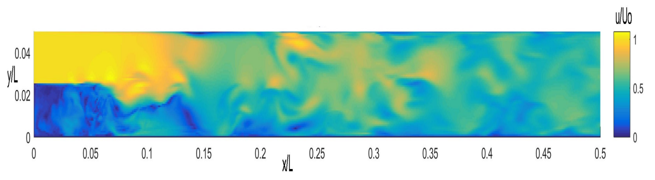

Figure 1: Instantaneous normalized stream-wise velocity field.

B. Experimental hardware platform

The hardware platform is composed by a Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz processor of 4 cores and 64GB of RAM, connected to a NVIDIA GeForce GTX 1080 Ti GPU. Table I summarizes the main characteristics of the GPU hardware and software.

Table I: Main characteristics of the employed GPU.

GPU Name	GeForce GTX 1080 Ti
CUDA Driver/ Runtime Version	9.0 / 9.0
CUDA Capability Major/Minor version:	6.1
Total amount of global memory:	11 GB
(22) Multiprocessors, (128) CUDA Cores:	3584 CUDA Cores
Max. memory bandwidth	484 GB/s

C. Experimental results

Table II shows the total runtimes obtained for the *backward facing step* problem by three variants of our solver, using different grid sizes. It can be observed that the baseline GPU implementation of the SIP solver achieves a remarkable acceleration with respect to the CPU variant. Additionally, the Caffa$_{SF}$ variant, which employs the proposed synchronization mechanism achieves a runtime reduction of 28%. However, considering that only three kernels have been enhanced with the proposed synchronization mechanism, this improvement is strongly significant.

Table II: Total runtime obtained for the *backward facing step* test case using different solver variants.

Grid Size	Variant	Time (s)
$512 \times 64 \times 64$	Caffa$_{CPU}$	4421.828
	Caffa$_{GPU}$	246.251
	Caffa$_{SF}$	192.326
$512 \times 128 \times 128$	Caffa$_{CPU}$	19779.911
	Caffa$_{GPU}$	595.749
	Caffa$_{SF}$	564.606
$1024 \times 64 \times 64$	Caffa$_{CPU}$	8975.414
	Caffa$_{GPU}$	461.965
	Caffa$_{SF}$	367.489

Now we proceed with a deeper inspection of the GPU routines. Table III dissects the runtime of the Caffa$_{GPU}$ variant for the employed test case, showing the performance of the most important kernels. This data was obtained using the tool *nvprof*, bundled with the CUDA Toolkit, executing only one outer iteration of the caffa3d.MBRi solver.

The principal kernels related to the SIP are `SolveLUforFiBackwardSIP_GPUk` and `SolveLUforFiForwardSIP_GPUk`, which represent the forward and backward substitution phases of the solver. Additionally there are other kernels that represent previous and intermediate steps, such as `ComputeMomentumFluxesInnerFaces_GPUk`, `ComputeResInnerCellsSIP_GPUk`, `ComputeGradientInnerCellsFaces_GPUk` and

ComputeLuCoefficientsSIP_GPUk kernels. Added together, these kernels represent more than 63% of the total runtime. However, it is important to note that each execution takes only a few microseconds (except for ComputeMomentumFluxesInnerFaces_GPUk).

Conversely, the time implied by the transfer of parameters and the kernels launches, shown in Table IV, are considerable. In fact, the time taken by each kernel launch (4.7920us on average) can be even higher than the average kernel runtime of ComputeResInnerCellsSIP_GPUk, SolveLUforFiBackwardSIP_GPUk and SolveLUforFiForwardSIP_GPUk. This means that the overhead due to kernel launches is significant.

To visualize this situation, Figure 2 shows the timeline generated by the Nvida Visual Profiler for the forward substitution kernel. This confirms that the operations which require one kernel launch by hyper-plane suffer from a significant overhead originated by the calls to the CUDA API. For this reason, it is important to reduce the number of kernel launches in these cases.

To obtain a higher bound of the improvement that is attainable by reducing the cost of synchronization between hyper-planes, we first performed a test that completely eliminates the synchronization, disregarding the numerical result. In this test, we execute CUDA grids with different number of blocks, where each block has 1024 threads, unifying the processing of those hyper-planes with fewer cells than the total number of threads.

The results of this test can be observed in Table V, which shows that the improvement obtained by removing the synchronization behaves linearly, and that the maximum acceleration that is obtained is of approximately 2.7×.

We now proceed to evaluate the mechanism to reduce the cost of synchronizations between hyper-planes. We test this strategy in three of the kernels that use the hyper-plane layout, which are ComputeResInnerCellsSIP_GPUk, SolveLUforFiBackwardSIP_GPUk and SolveLUforFiForwardSIP_GPUk. The same strategy can be applied to ComputeLuCoefficientsSIP_GPUk but we leave it out of this experiments because it is called much fewer than the other three routines.

Tables VI and VII summarize the output of the profiling tool of Nvidia (*nvprof*) for the execution of the aforementioned test case. Comparing these results with those obtained for the CAFFA$_{GPU}$ variant, a minor increase in the total execution time of the three kernels can be noted. This is expected because now, besides performing the same computations that CAFFA$_{GPU}$, the kernels of the CAFFA$_{SF}$ variant contain the logic required by the synchronization between hyper-planes. Additionally, the results show how the time dedicated to kernel launches is severely reduced, decreasing from 589.85ms to 43.257ms, representing an improvement of approximately 13×.

The notorious increase of the time taken by the cudaDeviceSynchronize API call is because the time of that call is that comprehended between the call to cudaDeviceSynchronize and the end of all kernels and transfers that are queued on the device. In the CAFFA$_{GPU}$ variant, the call to cudaDeviceSynchronize occurs after the call to the kernel that computes the last hyper-plane, and thus comprehends only its execution time, while on the CAFFA$_{SF}$ variant, the call occurs after the call to the kernel that computes all the hyper-planes, which has a much longer execution time.

Figure 3 illustrates the behavior of the kernel that performs the forward substitution of the SIP (SolveLUforFiForwardSIP_GPUk) for the two GPU-aware variants. It can be observed that the synchronization mechanism of variant CAFFA$_{SF}$ implies a runtime reduction of 1.9× for this operation. A similar effect can be observed on the other kernels that use the hyper-planes layout, although the acceleration obtained for the group of three routines is of 1.7×. This is mainly because the average duration of the kernels corresponding to ComputeResInnerCellsSIP in the CAFFA$_{GPU}$ variant, is slightly higher than in the other two routines, which decreases the benefit of reducing the synchronization time.

VI. Final remarks and future work

The caffa3d.MBRi is a useful tool for numerical simulation of real world fluid flow problems that is currently been transformed to exploit the computational power of massively parallel processors. In this context, we developed a GPU version of the SIP, which is the most efficient linear solver available in this CFD tool for regular grids, following the hyper-planes strategy, which is a variant of the well-known wavefront parallelism pattern.

The analysis of the baseline GPU variant of the SIP revealed that a potential drawback of this strategy is the considerable overhead implied by the kernel launches, which are required to synchronized the computations between consecutive hyper-planes. Although the time taken by the setting of parameters and the kernel launch is usually negligible, the small volume of computation required by each hyper-plane allows this overhead to have a significant impact on the performance of the entire simulation.

We proposed a synchronization mechanism for the GPU implementation of the SIP that works within the kernel, avoiding the need to launch multiple kernels to synchronize the computations. The strategy makes use of atomic operations to control the progress of the computations, and leverages the particularities of the GPU execution model to avoid deadlock situations.

The synchronization strategy proposed represents a considerable improvement with respect to the implicit synchronization GPU variant, reaching accelerations close to 2× on

Table III: Total runtime, percentage of the total runtime, number of launching, and maximum, minimum and average of each launch.

Name	Time(%)	Time	Calls	Avg	Min	Max
ComputeResInnerCellsSIP_GPUk	16.55%	117.33ms	38280	3.0650us	1.4080us	7.5520us
SolveLUforFiBackwardSIP_GPUk	14.97%	106.13ms	38220	2.7760us	1.2800us	7.7130us
SolveLUforFiForwardSIP_GPUk	12.94%	91.700ms	38280	2.3950us	1.2800us	6.0490us
ComputeGradientInnerCellsFaces_GPUk	9.44%	66.930ms	150	446.20us	240.30us	764.06us
ComputeMomentumFluxesInnerFaces_GPUk	4.81%	34.079ms	30	1.1360ms	1.0708ms	1.1792ms
ComputeLuCoefficientsSIP_GPUk	4.21%	29.811ms	6380	4.6720us	1.2800us	6.3680us

Table IV: Total runtime, percentage of the total runtime, number of launching, and maximum, minimum and average of each call to the CUDA API.

Name	Time(%)	Time	Calls	Avg	Min	Max
cudaLaunchKernel	**48.16%**	**589.85ms**	**123080**	**4.7920us**	**4.4130us**	**281.81us**
cudaMemcpy	15.18%	185.89ms	736	252.57us	5.0010us	12.990ms
cudaDeviceSynchronize	14.76%	180.71ms	3798	47.581us	1.8500us	7.5785ms
cudaStreamCreate	11.88%	145.50ms	2	72.748ms	15.367us	145.48ms
cudaDeviceReset	7.32%	89.602ms	1	89.602ms	89.602ms	89.602ms

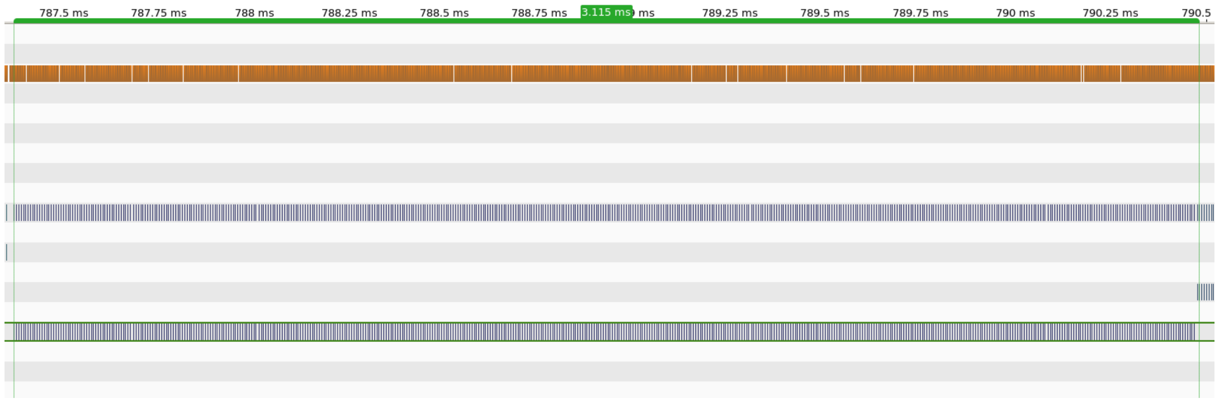

Figure 2: Detail of the timeline generated by Nvidia Visual Profiler that evidences the overhead of synchronization between hyper-planes (gaps between vertical bars) for the `SolveLUforFiForwardSIP_GPUk` kernel.

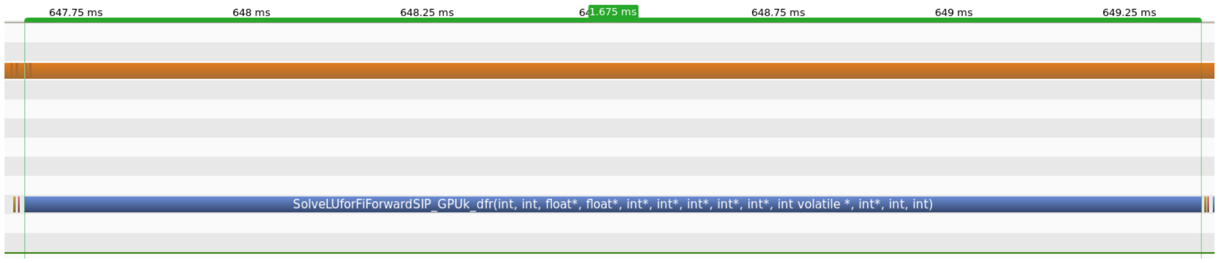
647.75 ms
648 ms
648.25 ms
64
1.675 ms
648.75 ms
649 ms
649.25 ms
SolveLUforFiForwardSIP_GPUk_dfr(int, int, float*, float*, int*, int*, int*, int*, int*, int volatile *, int*, int, int)

Figure 3: Timeline generated with the Nvidia Visual Profiler for the `SolveLUforFiForwardSIP` kernel with the CAFFA$_{SF}$ variant.

some of the kernels that work with the hyper-plane layout.

As future work, we intend to perform a deeper study on the individual kernels, including the design of storage layouts that allow a more efficient data reuse within the kernels. We are also interested on accelerating other routines that gain importance after the acceleration of those that work with the hyper-plane layout, including those that follow the red-black memory organization. Additionally, it would be interesting to compare our asynchronous SIP implementation with one based on the Cooperative Groups technique, available in lastest CUDA generations.

REFERENCES

[1] M. Mendina, M. Draper, A. Kelm Soares, G. Narancio, and G. Usera, “A general purpose parallel block structured open source incompressible flow solver,” *Cluster Computing*, vol. 17, no. 2, pp. 231–241, 2014.

Table V: Effect of disabling the synchronization between hyper-planes for the routine `SolveLUforFiForwardSIP_GPUk`, using blocks of 1024 threads.

Blocks	Merged hyper-planes	%	Time (ms)
0	1	0.39	3.45
1	44	17.25	3.12
2	63	24.71	2.96
3	77	30.20	2.85
4	90	35.29	2.76
5	100	39.22	2.67
6	110	43.14	2.58
7	119	46.67	2.50
8	127	49.80	2.44
9	135	52.94	2.37
10	144	56.47	2.31
11	154	60.39	2.20
12	164	64.31	2.11
13	177	69.41	1.99
14	191	74.90	1.85
15	210	82.35	1.68
16	254	99.61	1.26

Table VI: Total runtime, percentage of the total runtime, number of launching, and maximum, minimum and average of each launch.

Name	Time (%)	Time (ms)	Calls	Avg (ms) Min (ms) Max (ms)
ComputeResInnerCells	14.64	92.543	60	1.542 1.373 2.283
SolveLUforFiBackward	12.76	80.681	60	1.344 1.149 2.281
ComputeGradient...	10.59	66.944	150	0.446 0.240 0.765
SolveLUforFiForward	10.26	64.885	60	1.081 0.946 1.628
ComputeMomentum...	5.40	34.108	30	1.136 1.072 1.178
ComputeLuCoefficients	4.88	30.832	6380	0.004 0.001 0.006

Table VII: Total runtime, percentage of the total runtime, number of launching, and maximum, minimum and average of each call to the CUDA API.

Name	Time (%)	Time (ms)	Calls	Avg (ms) Min (ms) Max (ms)
cudaStreamCreate	40.70	522.52	2	261.260 0.011 522.510
cudaDeviceSynchronize	32.72	420.15	3798	0.110 0.001 7.586
cudaMemcpy	14.39	184.75	736	0.251 0.005 12.942
cudaDeviceReset	6.40	82.158	1	82.158 82.158 82.158
cudaLaunchKernel	3.37	43.257	8660	0.004 0.004 0.267

[2] G. Usera, A. Vernet, and J. Ferré, “A parallel block-structured finite volume method for flows in complex geometry with sliding interfaces,” *Flow, Turbulence and Combustion*, vol. 81, no. 3, pp. 471–495, 2008.

[3] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors, Third Edition: A Hands-on Approach*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.

[4] H. L. Stone, “Iterative solution of implicit approximations of multidimensional partial differential equations,” *SIAM Journal on Numerical Analysis*, vol. 5, no. 3, pp. 530–558, 1968. [Online]. Available: http://www.jstor.org/stable/2949703

[5] P. Igounet, P. Alfaro, M. Pedemonte, and P. Ezzatti, “A GPU implementation of the SIP method,” in *2011 30th International Conference of the Chilean Computer Science Society*, Nov 2011, pp. 195–201.

[6] P. Igounet, P. Alfaro, G. Usera, and P. Ezzatti, “Towards a finite volume model on a many-core platform,” *Int. J. High Perform. Syst. Archit.*, vol. 4, no. 2, pp. 78–88, December 2012. [Online]. Available: http://dx.doi.org/10.1504/IJHPSA.2012.050987

[7] ——, “GPU acceleration of the caffa3d.MB model,” in *Computational Science and Its Applications – ICCSA 2012*, B. Murgante, O. Gervasi, S. Misra, N. Nedjah, A. M. A. C. Rocha, D. Taniar, and B. O. Apduhan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 530–542.

[8] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter, “A synchronization-free algorithm for parallel sparse triangular solves,” in *European Conference on Parallel Processing*. Springer, 2016, pp. 617–630.

[9] W. Liu, A. Li, J. D. Hogg, I. S. Duff, and B. Vinter, “Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides,” *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, 2017.

[10] E. Dufrechou and P. Ezzatti, “Solving sparse triangular linear systems in modern gpus: A synchronization-free algorithm,” in *26th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2018, Cambridge, United Kingdom, March 21-23, 2018*, 2018, pp. 196–203.

[11] ——, “A new GPU algorithm to compute a level set-based analysis for the parallel solution of sparse triangular systems,” in *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018*, 2018, pp. 920–929.

[12] C.-C. Liao, Y.-W. Chang, C.-A. Lin, and J. McDonough, “Simulating flows with moving rigid boundary using

immersed-boundary method," *Computers and Fluids*, vol. 39, no. 1, pp. 152–167, 2010.

[13] Lilek, S. Muzaferija, M. Perić, and V. Seidl, "An implicit finite-volume method using nonmatching blocks of structured grid," *Numerical Heat Transfer, Part B: Fundamentals*, vol. 32, no. 4, pp. 385–401, 1997.

[14] C. Lange, M. Schäfer, and F. Durst, "Local block refinement with a multigrid flow solver," *International Journal for Numerical Methods in Fluids*, vol. 38, no. 1, pp. 21–41, 2002.

[15] M. Draper, A. Guggeri, M. Mendina, G. Usera, and F. Campagnolo, "A large eddy simulation-actuator line model framework to simulate a scaled wind energy facility and its application," *Journal of Wind Engineering and Industrial Aerodynamics*, vol. 182, pp. 146–159, 2018.

[16] E. Krause, "Ferziger, j. h.; perić, m.: Computational methods for fluid dynamics. berlin etc., springer-verlag 1996. xiv, 356 pp., dm 74,00. isbn 3-540-59434-5," *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik*, vol. 77, no. 2, pp. 160–160, 1997. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/zamm.19970770220

[17] F. Deserno, G. Hager, F. Brechtefeld, and G. Wellein, "Basic optimization strategies for cfd-codes," *Regionales Rechenzentrum Erlangen, Technical report*, 2002.

[18] J. G. Gomez Luna, L. Chang, I. Sung, W. Hwu, and N. Guil, "In-place data sliding algorithms for many-core architectures," in *2015 44th International Conference on Parallel Processing*, Sep. 2015, pp. 210–219.

[19] S. Yan, G. Long, and Y. Zhang, "Streamscan: Fast scan algorithms for gpus without global barrier synchronization," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: ACM, 2013, pp. 229–238. [Online]. Available: http://doi.acm.org/10.1145/2442516.2442539