# Accelerating Operational Earth Systems Models using GPUs

Portability of NEMO diagnostics to GPUs

Sergi Palomas Martinez

June 2019

**Resum–** Els models de ciències de la terra són àmpliament utilitzats en institucions meteorològiques i universitats per estudis de predicció climàtica. La complexitat de les equacions caòtiques i la quantitat de dades necessària per aconseguir una predicció acurada per les simulacions requereix una potència de càlcul només assolible en clústers com per exemple el Marenostrum 4. Un d'aquests models és NEMO, el framework per excel·lència a Europa utilitzat per estudis oceanogràfics. Un aspecte prioritari en aquest context és el rendiment del model. Tant per reduir "Time to solution" com pels costos associats durant l'execució. Per a preparar les sortides de NEMO, s'executen diagnòstics per a preparar les variables de sortida per als estudis de post processament, els quals fan el model més lent.
En aquest projecte s'analitza el rendiment dels diagnòstics de NEMO per a, més tard, poder ser implementats a noves arquitectures basades en GPUs mitjançant CUDA, al mateix temps que s'elimina aquesta part del camí crític. d'execució.

**Paraules clau–** BSC, HPC, NEMO, GPU, CUDA, MPI

**Abstract–** Earth science models are widely used in meteorological institutions and universities for weather and climate prediction studies. The complexity of the chaotic equations used and the amount of data needed to achieve a good accuracy on the simulation demand a computational power only found in clusters. One of these models is NEMO, which is the European framework for excellence used for oceanic studies and running on MareNostrum 4. A prime focus in this context is the computational performance of the model. So much to reduce "Time to solution" as for the costs associated during the execution. In NEMO, the execution time is extended due to the calculation of some diagnostics, which are used to prepare the output variables for the post-processing study.
In this project, the performance of NEMO diagnostics is analyzed. Afterward, the portability of them to new architectures based on GPUs through CUDA while removing this part from the critical path of the execution is discussed.

**Keywords–** BSC, HPC, NEMO, GPU, CUDA, MPI

## 1 INTRODUCTION

This project has been developed in the Barcelona Supercomputing Center, with the Earth Science department. The target application is NEMO. A state of the art global circulation model used in multiple European Institutions that reaches over a billion of computing hours per year. Thus, achieving a good performance is crucial to reduce the time and space (that is, money) needed. The main objective is to select and port one part of the code to GPU and study if this new architecture can be of benefit for NEMO.

Much success has been achieved using GPUs to accelerate existing applications in the fields of physics, data science and machine learning (REFERENCIAS). However, fewer attempts have been made in large applications like

• E-mail de contacte: sergi.palomas@e-campus.uab.cat
• Menció realitzada: Enginyeria de Computadors
• Treball tutoritzat per: Mario Acosta (BSC) i Ramón Grau Sala (CAOS)
• Curs 2018/19

NEMO and real Earth science models in general. The difficulties of maintaining two different versions of the same model and the cost of developing an operational GPU version are high. Additionally, a hybrid implementation requires to load balance GPU are CPU calculations, something which is not trivial for complex models run in thousands of cores using traditional parallel paradigms (MPI and OpenMP).

Although there are successful examples such as The Consortium for Small-scale Modeling (COSMO) and Weather Research and Forecasting (WRF), in the case of NEMO the difficulties have been bigger that the advantages of GPUs use until now. In the past, Nvidia did achieve to implement a hybrid version[1] in 2013. However, this version was never used in operational configurations due to the difficulties found by the sctientists to adapt the Nvidia implementation to the real simulations and the extra effort to maintain and develop the accelerator code. For this reason, newer versions of NEMO are released every year and the hybrid version was not maintained anymore.

In spite of the difficulties to implement a hybrid version of NEMO, the community is opened to find new and simple approaches to take advantage of GPU architectures and the reason why the post-processing is studied now. The diagnostics are part of this post-processing, they are processed on the CPU at the end of every time step for the configuration studied, taking almost 14% of the total execution time of NEMO, even though its results are not needed for the model anymore (next time step calculation) but only as an output.

In this project, only the diagnostics of the model are taken into account and they do not affect the NEMO core engine. Therefore, the costs associated with a hybrid version mentioned do not represent for the future maintaining and developing the load balance of between CPU and GPU should be always easy, taking into account that the diagnostics calculations are not in reality part of the critical path of NEMO execution.

In this work, firstly, Section 2 describes the methodology, enviroment and model configuration used for the project. Section 3 summarize the analysis done for NEMO, descrive the diagnostics that have more impact to the model and selects the target one to port to GPU. Then, in Section 4 the CUDA implementation for the diagnostic and for a reduction procedure are discussed and different optimizations evaluated. Finally, in Section 5 the conclusions of the work are presented and future implementations are considered. The CUDA code developed can be found in the Appendix section.

## 1.1 Context

Barcelona Supercomputing Center (BSC-CNS) is the national supercomputing center in Spain, specialized in high-performance computing (HPC) and in charge of MareNostrum, one of the most powerful supercomputers in Europe. This project arises from the necessity to increase the performance and reduce the time-to-solution in a context in which scientific models are required to run in parallel and are growing in complexity and precision every year. It has been proposed by the Earth model performance analysis team which is part of the Computational Earth Science group.

## 1.2 NEMO

The Nucleus for European Modelling of the Ocean[1] (NEMO) is a state-of-the-art modeling framework used for oceanographic research, climate studies, seasonal forecasting and also for operational oceanography. It is maintained by a pan-European community and employed in multiple research centers in Europe and projects for the World Climate as the Coupled Model Intercomparison Project (CMIP[2]). It consists of 3 main components:

- NEMO-OCE: models the ocean thermodynamics and solves the primitive equations.
- NEMO-ICE: models the sea-ice thermodynamics
- NEMO-TOP (Tracers in the Ocean Paradigm): models the on,offline oceanic tracers transport and biogeochemical processes.

It includes a set of scripts and tools to use the model. CPP keys (at compile time) and NAMELISTS I/O are used to select and configure the experiment to run. An I/O server (XIOS[3]) is implemented to allow easy and flexible control of the output behavior and enabling parallel I/O operations. At the end of every time-step, a set of diagnostics can be executed to prepare and deliver results of interest for the user. Different resolutions exist depending on the size of the mesh used to map the Ocean named ORCA. For instance, the lowest resolution is ORCA2 with a grid size of x = 182, y = 149, z = 31. Very little compared with the super-high resolution named ORCA0036 with a grid size of x = 12962, y = 9173, z = 75 which has started to be used as an operational configuration in supercomputers like MareNostrum 4.

Fig. 1: Masked world ocean mesh

The definitions for the longitude (x), the latitude (y) and in deep (z) of the global mesh are outside the scope of this project but can be found in the reference section [2].

[1] M. Milakov (2013). Accelerating NEMO with OpenACC. NVIDIA GTC 2013

[2] CMIP. https://www.wcrp-climate.org/

[3] XIOS. https://forge.ipsl.jussieu.fr/ioserver

The current parallel implementation of NEMO is using MPI where the global grid is divided into multiple subdomains (domain decomposition) in the *x* and *y* coordinates.

### 1.3 Objectives

The main idea is to study and implement the portability of the most computer-hungry diagnostics of NEMO to GPU. Ideally, given that the model does not need the output of a diagnostic to continue its execution and CUDA kernel calls can be asynchronous (i. e. we can overlap CPU and GPU computation), we could completely remove the overhead that the execution of diagnostics adds to NEMO by using the GPU as an extra H/W device.
First of all, a performance analysis of the model and for the diagnostics will be mandatory to select the best target. Afterwards, an evaluation of each one to understand its behavior and how can they fit into a GPU will be necessary. Once the target diagnostic has been chosen, different GPU implementations will be done and optimizations for the CUDA kernels and data transfers discussed. Finally, a comparison between the CPU version will be provided. Summerizing, the main tasks are:

- Analyze the impact of NEMO diagnostics
- Select a target diagnostic to improve
- Propose solutions based on GPU
- Evaluate and compare the new implementation

The future objective, outside the scope of this work, is to adapt the implementations proposed and take the considerations exposed here into account to successfully develop a novel version of NEMO capable to run the core on the CPU the diagnostics asynchronously on the GPU.

## 2 METHODOLOGY

### 2.1 Envrionment and configuration

The next tables summarize the model configuration and contains an environment overview for MareNostrum 4, CTE IBM Power9[4] clusters and for the Nvidia Tesla V100 used:

Table 1: MODEL CONFIGURATION

| Configuration name | ORCA025 NEMO standalone |
|---|---|
| Grid | ORCA |
| Resolution | 2 degrees (Low resolution) x=1442 y=1050, z=75 |
| Modules | OCE OPA |
| Time step | 900 seconds |
| Run length | 1 week (672 time-steps) |
| Nº processes | 144 Nemo 48 Xios (3+1 nodes) |
| Domain per process | x=87, y=105, z=75 |

[4]Power9 User guide: https://www.bsc.es/user-support/power.php

Table 2: MARENOSTRUM 4 ENVIRONMENT

| Overview | 3456 Nodes with 2 x Intel Xeon Platinum 8160 CPU with 24 cores (**48 cores per node**) |
|---|---|
| Compiler | Intel 2017.4 |
| Network | 100 Gbit/s Intel Omni-Path HFI Silicon 100 Series PCI-E adapter |

Table 3: POWER9 ENVIRONMENT

| Overview | 54 Nodes with 2 x IBM Power9 8335-GTH 2.4GHz (3.0 GHz on turbo, 20 cores and 4 threads/core) each |
|---|---|
| Compiler | PGI/18.4 |
| GPUs | 216 (4 per node) NVIDIA V100-SXM2-16GB |
| CUDA | CUDA version 9.1 (driver 418.39) |

Table 4: NVIDIA TESLA V100 OVERVIEW

| Complete name | NVIDIA Tesla V100 SXM2 16GB |
|---|---|
| Compute capability | 7.0 |
| Memory BW (GB/s) | 898.05 |
| Global memory (GB) | 16.91 |
| Number of SM | 80 |
| Shared memory per SM (B) | 98304 |
| Shared memory per Block (KB) | 49152 |
| Max threads per block | 1024 |
| Mex threads per block dimension | 1024, 1024, 64 (x, y, z) |

The next figure represents the connection architecture between Processors and GPUs within a single CTE IBM Power9 node [3].

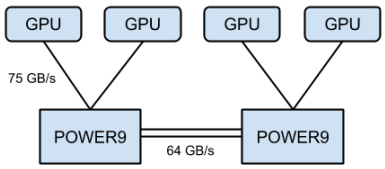


Fig. 2: Power9 intra-node GPU connection

This illustrates that in a node, every socket is physically connected to two GPUs through a network of 75 GB/s of bandwidth. If the link is to a GPU attached to the other Power9 processor of the node, the bandwidth is limited to 64 GB/s due to the connection between them.

### 2.2 Work

As in other related works[4], the first step will be to **analize the application** and record basic information about its execution. The number of cores and configuration used for the

study are fixed matching the definitions in Table 1. Therefore, the same experiment and long runs will be used to get consistency on the performance metrics. After comparing the time spent in each diagnostic with respect to the total execution time of NEMO and performing a computational study, the best target diagnostic will be selected. Afterwards, an example model will be created to mime the original source code and a new implementation based on CUDA will be developed. The next step will be to optimize the new GPU implementation and ultimately test how it performs and validate the results.

- **Select a diagnostic** comparing the time spent in each one and with the model. Also, a behavior study of them will be done to choose the best that can fit into a GPU while being aware of the underlying architecture and possible code reutilization. Finally, a detailed analysis of the selected target diagnostic using the BSCTOOLS Extrae and Paraver will be done to get further details on its execution (function profile).

- Due to the complexity of the model, it is hard to develop code directly. Therefore, an **example model** will be used. That is a very much simplified version where the tasks of developing and testing are far easier. Basically, the data is generated randomly and conserving only the structures and computation to be ported to GPUs. The CPU code is employed in order to compare the performance and to validate the results.

- **GPU version implementation**: After the analysis, once we have selected which diagnostic to work on, we can take one stape ahead and port the CPU original version to GPU using CUDA.

- **Kernel optimization**: Different approaches for the kernel and for the data transfers will be discussed along with architectural limitations comparisons.

- **Test** the GPU implementation: Verify that the results after the optimization does not differ from the CPU ones and compare the performance of both versions.

## 3 DIAGNOSTICS ANALYSIS

Up to 8 different diagnostics have been activated using keys when compiling, modifying the NAMELISTS provided or defining variables as outputs in the IO server (XIOS) configuration files, to use the most typical diagnostics employed by scientists. Only the most time-consuming ones are shown. The results illustrated in Fig.2 have been collected using CPU timers and averaging the execution time after 1-week simulation (672 time steps).

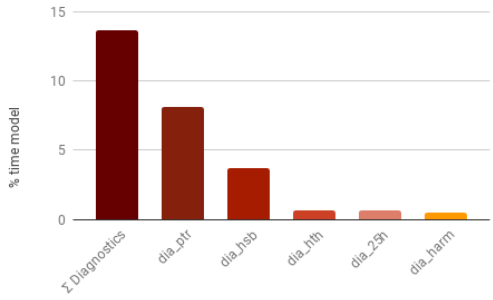


Fig. 3: Percentual time on diagnostics

We see that almost 14% of the total execution time is spent in diagnostics.

- **dia_ptr**: Meridional transports and zonal means
- **dia_hsb**: Compute the ocean global heat content, salt content and volume conservation
- dia_ohc: Specific depth ocean heat content
- dia_25h: For the harmonic analysis of tidial constituents. It outpus 25hr means for shelf seas
- dia_harm: Compute the Courant numbers and output to ascii file

Dia_ptr and dia_hsb are the diagnostics that more impact have, taking more than 10% of the total NEMO execution time. Therefore, in the following sections, we study more in depth both.

### 3.1 Meridional transports and zonal means

This diagnostic is activated setting 'ln_diaptr' to 'true' on the NAMELIST and called twice every time step. It controls its execution with an if-else statement depending on whether it is called from the main step function or from the transports procedure 'tra_adv'. Consequently, different codings are executed according to the case.
It checks whether a set of variables are activated or not on the I/O server. For each one, if defined on the XML files of XIOS, it runs the respective code. Thus, if nothing is defined on XIOS the diagnostic does nothing by itself. It also defines three new functions:

- ptr_sjk: "zonal" mean computation of a field (global MPI communication)
- ptr_sj: "zonal" and vertical sum computation of a "meridional" flux array
- dia_ptr_hst: Wrapper for heat and salt transport calculations

Figure 4 shows the percentual time with respect to a complete time step. It is pertinent to note that the aforementioned functions (ptr_sjk, ptr_sj and dia_ptr_hst) are called inside the dia_ptr routine. The same goes for the event dia_ptr_sjk_mpp_mpi, that represents the time in a collective MPI called from dia_sjk function. In both cases, the time spent in the inner procedure is not taken into account in the outer one. Therefore, the sum of all columns is equivalent to the value of dia_ptr shown in Figure 3.

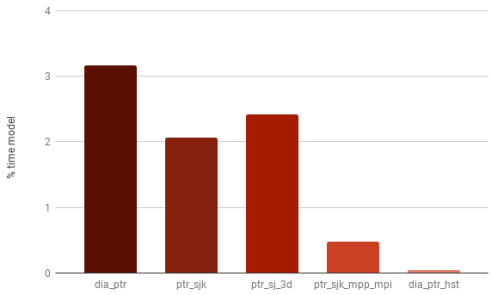


Fig. 4: Percentual time on dia_ptr

Another important consideration is the number of calls. We already mentioned that dia_ptr is called twice every time step. Furthermore, ptr_sjk function (and dia_ptr_sjk_mpp_mpi) is called 40 times and ptr_sj up to 50 times per time step.

## 3.2 Heat and salt budgets

This diagnostic is activated when the key 'dia_hsb' is added at compilation time. It is responsible for the calculation of the ocean heat content, salt content and volume conservation. Called every time step from the main routine and composed by 3 subroutines: dia_hsb_run for the computation itself, dia_hsb_rst to R/W in restart files and dia_hsb_init for initialization purposes.

We focus on the **dia_hsb_run** subroutine. It computes de deviation of heat content, salt content and volume at the current time step from their values at the beginning (nit000).

The variables involved are:

- surf (s): 2D array with the surface dimension. i and j ($m^2$)
- e3t_n (m): 3D array for the vertical scale factor. (m)
- mask (binary): 3D array to filter the sea points
- hc_ini = 3D array with the initial heat of the ocean
- sc_ini = 3D array with the initial salinity of the ocean
- tsn: 4D array with the temperature or salinity (controlled with the 4th dimension) of the ocean in the current time step

The next pseudocode illustrates the control flow of the diagnostic:

```
Heat & salt budgets (dia_hsb)
  FOR each local vertical point jk DO
      vv = VolumeVariation(:, :, jk)
  END FOR
  Gloval_vv = Global reduction(vv)
  FOR each local vertical point jk DO
      hv = HeatVariation(:, :, jk)
  END FOR
  Gloval_vv = Global reduction(hv)
  FOR each local vertical point jk DO
      sv = SalinityVariation(:, :, jk)
  END FOR
  Gloval_vv = Global reduction(sv)
  FOR each local vertical point jk DO
      v = Volume(:, :, jk)
  END FOR
  Gloval_vv = Global reduction(v)
End Heat & salt budgets
```

**Volume:**
The product between the surface and its corresponding i and j points of e3t_n and the mask give the volume of every point for the ocean.

$$Volume = \int_{i,j,k}^{jpi,jpj,jpk} m(i,j,k) \cdot s(i,j) \cdot mask(i,j,k) \tag{1}$$

**Volume variation:**
For the volume variation, every point of 3D array is the by-product of multiplying a mask by the result of the multiplication of the corresponding point of the t-vertical scale factor by the surface (that is, the volume) in the current time step substracted by the initial value (initial t-vertical scale factor x initial surface). Mathematically expressed as follows:

$$Volume\ variation = \int_{i,j,k}^{jpi,jpj,jpk} \big(m(i,j,k) \cdot s(i,j) - m\_ini(i,j,k) \cdot s\_ini(i,j)\big) \cdot mask(i,j,k) \tag{2}$$

**Heat/Salinity deviation:** Almost the same goes for the temperature and salinity content deviation. The only difference is that it multiplies the volume by another 3D array with the values of the heat or salinity, depending on the case. This applies for both the current time step and for the initial state ($init\_heat$ and $init\_salinity$) , used to compute the deviation.

$$\begin{aligned} Heat\ content &= m \cdot s \cdot heat - init\_heat \\ Salinity\ content &= m \cdot s \cdot salinity - init\_salinity \end{aligned} \tag{3}$$

Once each of the 3D array results (*v, vv, hv* and *sv*) have been computed, a global reduction procedure (*Global reduction*) is needed to get the final result. Therefore, all points of the four matrices (volume, volume variation, heat and salt deviation) are added and given the corresponding scalar (*v, vv, hv* and *sv*). As it has been mentioned, the current implementation of NEMO employs an MPI domain decomposition. Every MPI process computes the 3D resulting arrays and the reduction for its local domain. Finally, a global reduction is done using the MPI_All_Reduce() collective call.

For the reduction, a new problem arises. Floating point operations are not totally precise, especially when adding numbers with huge orders of magnitude of difference, as when in a reduction procedure. Consequently, the Knuth's trick [5] is exploited to handle these operations as described in Section 3.4 to ensure the bit to bit reproducibility of the NEMO execution.

## 3.3 Selecting a diagnostic

As shown in Figure 1, Meridional transports and zonal means (dia_ptr) and heat and salt budgets (dia_hsb) are the diagnostics occupying more time. Therefore, we have analyzed more in depth both. In this section, we evaluate

which one can take more benefit of a GPU. Below there are some general considerations to have in mind for GPU implementations[6]:

- SIMD instructions are the ones that take more advantage of this type of architecture
- Moving data from the main memory (Host) to the GPGPU memory (Device) adds an overhead to the computation
- Initialize the buffers for the data transfers and the kernel launch has an overhead (latency)
- A kernel call is non-blocking from a CPU view. Thus, it is possible to overlap CPU and GPU computation
- Moving more data between Host and Device at once results in a better usage of the effective bandwidth (Appendix A.2)
- Warp divergence serialize the execution of threads
- Memory coalescing [7](i.e. having adjacent threads accessing consecutive memory addresses) helps threads to perform read and write operations to memory more effectively and reduce memory bank conflicts[8]

At the beginning of this section, we have demonstrated that the diagnostics taking more execution time are dia_ptr and dia_hsb. In section 2.1 an introduction to dia_ptr was done and showed that despite of being called twice every time step, its execution path is different (if-else statement). What's more, as shown in Figure 3, the time spent in this diagnostic is divided mainly into 3 different functions:

- dia_ptr_out: Mainly to output variables
- ptr_sjk: Function to compute the sum of a "meridional" flux array, called 40 times every time step giving a total of 26880 calls at 672 time steps
- ptr_sj_3d: Wrapper for heat and salt transport calculations. Called 50 times every time step giving a total of 33600 calls at 672 time steps

In section 2.2, the analysis for the diagnostic Heat and salt budgets (dia_hsb) illustrated that the computation for the volume, the heat and the salinity are very close. Moreover, a global reduction across all the subdomain is done to produce a scalar output.

For this project, dia_hsb have been selected. It mainly consists of SIMD instructions and a global reduction is needed. Both have been proved to take advantage of a GPU architecture [9]. Whatsmore, the kernel can reuse the data for the reduction and imply fewer data transfers between the Host and the Device.

On the other hand, for dia_ptr, the shared functions ptr_sjk and ptr_sj_3d are taking time because of the number of calls (26880 and 33600 respectively). Each one only takes about 300$\mu$s per call. For the dia_ptr_out function, it mainly consists of calls to the XIOS server to output multiple independent variables after some computation. Developing a kernel for each output require more effort than for dia_ptr. Nonetheless, it might be taken into consideration in future works, especially since figure 3 shows that it is also taking an important part of the total execution time.

# 4 GPU IMPLEMENTATION

This section contains the GPU implementation for Heat and salt budgets diagnostic in CUDA. Furthermore, optimizations for the CUDA kernels and data transmissions will be commented. All results have been taken in CTE IBM POWER9 machine with the configurations defined on Table 1. For this first approach, we consider that every MPI process is responsable to call a kernel with its data domain ($x = 87, y = 105 and z = 75$). To get the performance results of the original MPI version, the example model has been used to run both, the CPU and the GPU equivalent codes. CPU timers and The Nvidia profiler nvproof[5] has been used to get the performance metrics and compare the execution of the equivalent CPU and GPU versions respectively.

## 4.1 Heat and salt budgeds

During the section 2.2 the mathematic procedures for each calculation and the pseudocode were exposed. In this section, the implementation and results are presented.

The pseudocode shows that the only dependence is between the result for the Volume, Volume variation, Heat content deviation and Salt content deviation and its global respective reduction. Thus, it is possible to compute the local result for the volume, heat and salanity at the beginning and, afterwards, call the global reduction. In section 2.3 we listed a general set of considerations to be aware to use a GPU properly and reordering the control flow[10] leads to the next benefits:

- Reuse of data: For the Volume, Heat and Salinity Variation and for the total Volume calculation, the variables **surf**, **e3t_n**, **mask** and **tsn** are needed. If everything is computed in one single kernel, resending these data can be avoided.
- The result of the previous computations are needed for the reduction. Thus, the data can stay in the GPU.

The variables needed for the conservation of the ocean heat content, salt content and volume conservation are sent as 1D arrays. Appendix A.1 explains the indexing conversion. The block dimension is set to 512 threads (i.e. dimBlock = dim3(512,1,1)). It is important to use a multiple of the warp size (32 threads) to minimize the warp divergence only on the edge of the data shape. The grid size is calculated dividing the number of points of the grid by the block dimension using the ceil function (i.e. gridDim = dim3(ceil($x \cdot y \cdot z$, dimBlock%x), 1, 1)).

Every point of the 1D resulting array is computed by one thread. A shared array is used to save the multiplication between the surface (surf(:,:) and the t-vertical scale factor (e3t_n(:, :, :)) variables. This result is needed for every calculation (heat, salinity and volume) so redundant work is avoided and less memory and faster read operations are achieved. Below it is shown the time comparison between the GPU and the CPU version.

[5]Nvidia profiler tools (nvproof): https://docs.nvidia.com/cuda/profiler-users-guide/index.html

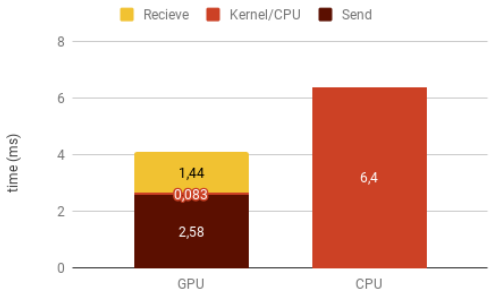


Fig. 5: dia_hsb_run GPU vs. CPU time comparison

In addition, Figure 6 illustrates the comparison between the CPU and the GPU version developed.

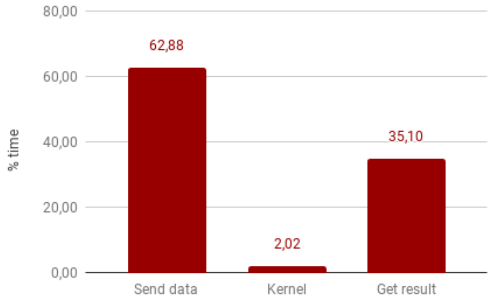


Fig. 6: Data transfers vs. kernel percentual time comparison

The total execution time (i.e. send data + kernel + receive results) for this approach is 4,1 ms (2,58 ms sending, 0,083 ms for the kernel and 1,44 ms receiving the result). It is clear that the data transfers are bottlenecking the execution and despite of the speedup on the kernel implementation (6,4/0,083 = 77x), the absolute time speedup is only 1,56x. The kernel only takes 2'02% of the total time, which represents a little proportion compared to the 62'88% and 35,10 % for the send and receive calls. The total data managed per MPI proccess (and for this kernel), considering a single MPI process with an ORCA025 configuration divided in 3 nodes (Table 1) using double precision floating-points (REAL(kind=8)) is:

**INPUT:**
2 x 2D arrays = $(2 \cdot 8B \cdot 87 \cdot 105) = 146KB$
5 x 3D arrays = $(5 \cdot 8B \cdot 87 \cdot 105 \cdot 75) = 27,4MB$
1 x 4D array = $(1 \cdot 8B \cdot 87 \cdot 105 \cdot 75 \cdot 2) = 11MB$

Giving a total of 38,5 MB and achieving a throughput of 0,038/0,0041 = 9,3 GB/s.

**OUTPUT:**
4 x 3D arrays = $(4 \cdot 8B \cdot 87 \cdot 105 \cdot 75) = 22MB$

Getting the result back shows the same problem. Four 3D arrays with a total size of 22 MB achieving a througput of 0,022/0,00144 = 15,3 GB/s.

The throughput achieved in both cases is really far from the 75 GB/s theoretical bandwidth illustrated in Figure 2. To increase it, it is possible to tell the compiler that a variable will always reside on the main memory using the PINNED[11] definition. Using locked memory allows the device to fetch the data without the help of the CPU (DMA, the device only needs the physical pages). In the other hand, not-locked memory can generate a page fault on access, and it is stored not only in memory (e.g. it can be in the swap partition), so the driver needs to access every page of non-locked memory, copy it into pinned buffer and pass it to DMA (Synchronious, page-by-page copy).

Figure 7 demonstrates the improves on data transfers achieved when using locked memory (Pinned) respect the original (Pageable). Having in mind the architecture shown in Figure 2, two different cases exist:

- Directly Attached (DA), that is, when a process is communicating to a physycally connected GPU (75 GB/s)
- Not Directly Attached (!DA), thus, the data goes thru the connection between Power9 process. (64 GB/s)

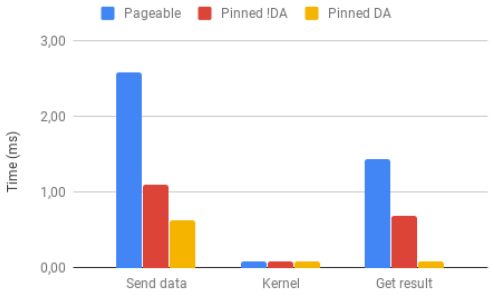


Fig. 7: Pageable vs. pinned memory transfers

The mean throughput (including Host to Device and Device to Host transfers) is 63 GB/s for DA, and 37 GB/s for !DA. It is clear that the connection between sockets (!DA) not only is slower but also adds an important overhead since it is only using $37/64 \cdot 100 = 58\%$ of the bandwidth. However, the H2D transfers are 3,16 times faster and 3,83x when the connections are D2H compared to the pageable version. For the Directly Attached (DA) connection, it almost reaches the theoretical bandwidth ( $63/75 \cdot 100 = 84\%$ ), achieving a 5'07 faster connection from H2D and 4,8x for the D2H. This difference must be taken into consideration, especially since the CUDA implementation bottleneck are still the data transfers. Appendix A.2 shows the differences between Pageable and Pinned memory depending on the size of the data in CTE IBM Power9 machine.

Even with the best configuration (i.e. Pinned and using a Directly Attached GPU), the percentual time spent on the kernel reprecents not more than 11%, still far from the 78% and 11% for the H2D and D2H transfers (Figure 7). Nonetheless, Figure 8 shows how the data transfer improvements impact on the GPU vs. CPU time comparison.

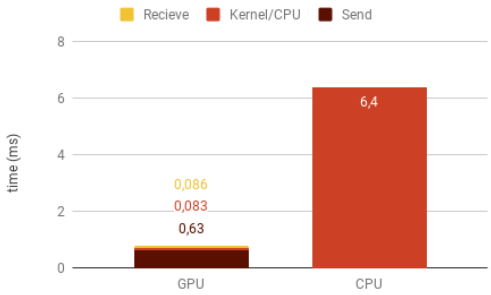


Fig. 8: pinned GPU dia_hsb_kernel vs. CPU time comparison

Again, although the impovements in the throughput achieved are huge, data transfers are bottlenecking the GPU implementation even that the computation speedup (kernel/CPU) is above 77x. However, this modification is 5 times faster than the previous one and the total speedup achieved (kernel + data transfers) is now 8x (6,4/0,8).

## 4.2 Reduction

Once the result is stored in the corresponding array, a reduction implying all the data is needed. The Knuth's trick is used to improve the numerical reproducibility and stability on parallel applications and the implementation is shown in Appendix A.4. Up to 6 different versions have been implemented for the reduction kernel [12, 13] itself and reducing the amount of data to transfer. Figure 9 shows the speedup achieved in different CUDA reduction implementations respect the first (*v1*) The size of the input array to reduce is equivalent to the data in a single MPI process. Thus, the number of elements per node is $(87 \cdot 105 \cdot 75) \approx 685125$. Only the total time spent on the reduction kernel is considered (i.e. data transfer times are omitted).

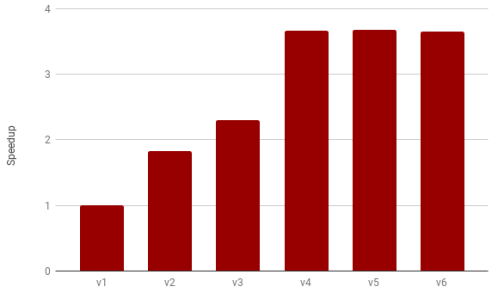


Fig. 9: CUDA reduction speedup per optimization version

Where the speedup of every new version is always calculated with respect to the first implementation (v1).

For versions *v1* and *v2*, interleaved access to shared memory are done. The difference is that the control loop to reduce every CUDA block in *v1* produces a high warp divergence. This is solved in *v2* creating an index and redefining the loop. However, in both cases bank conflicts are occurring since multiple addresses of a memory requests map to the same memory bank.

Figure 10 shows how threads access memory in *v1*:

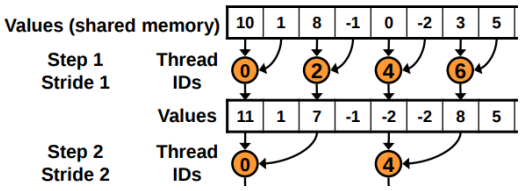


Fig. 10: CUDA reduction v1

And Figure 11 shows how after the warp divergence is solved in *v2*:

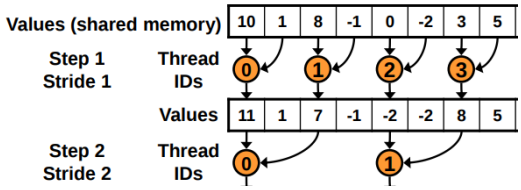


Fig. 11: CUDA reduction v2

Figure 12 illustrates the sequential addressing (i.e coalesced access) achieved in *v3* and *v4* when the loop is reversed and threadID-based indexing is used. It also solves the memory bank conflicts.

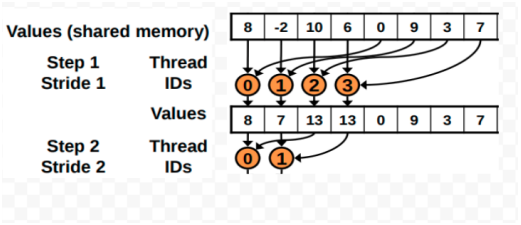


Fig. 12: CUDA reduction v3-v4

The *v4* implementation solves the problem that at the first iteration, half of the threads are IDLE. The number of blocks created is divided by 2 and stores the sum of two global values before entering the thread block loop.

Figure 9 shows that after *v4*, there is no improvement. Both, *v5* and *v6* optimizations are by un-rolling loops. We can infer that the compiler is already improving them.

Figure 13 illustrates the time per procedure (ms) for the CUDA reduction version *v4*. The time to send the data to reduce is null since after the dia_hsb CUDA kernel developed in Section 4.1, the data is already stored in the GPU.

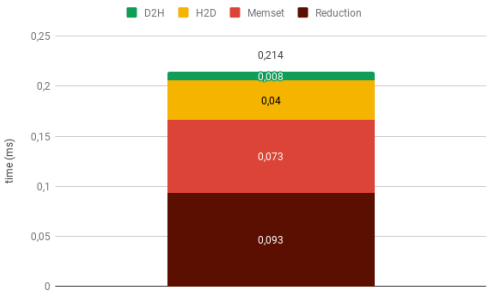


Fig. 13: CUDA reduction time per CUDA event

Finally, Figure 14 shows a comparison between the selected GPU reduction version *v4* (including the CUDA

memsets and data transfers) and the original CPU version for 385125 double floating point elements and using the Knuth's trick for both. Note that the reduction for the volume, volume variation and heat and salt content deviation are computed.

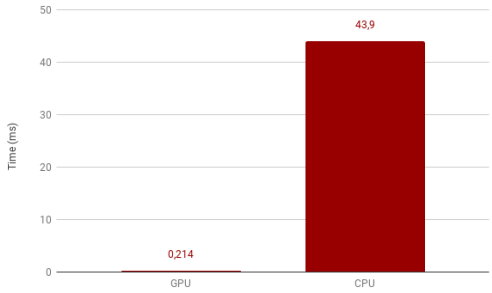


Fig. 14: GPU vs. CPU reduction time comparison

The speedup achieved with the GPU reduction with respect to the original CPU version is $43,9/0,214 = 205x$.

### 4.3 Final implementation

The final step is to gather the two kernels developed above into one single dia_hsb diagnostic GPU implementation. Figure 15 shows the control flow of the CUDA version.

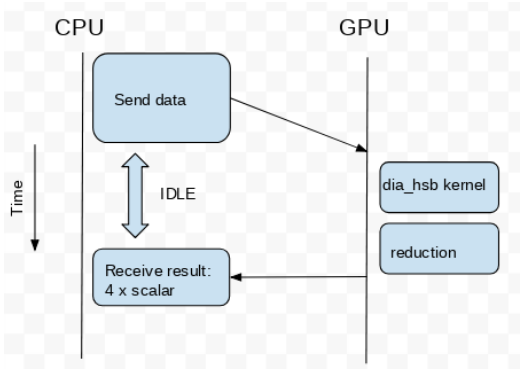


Fig. 15: Control flow for CUDA dia_hsb diagnostic implementation

Finally, Figure 16 shows a time comparison between the CUDA implementation for the diagnostic dia_hsb, with the reduction and Knuth's trick included, with respect to the same execution based on a single MPI process with the configuration descrived in Table 1.

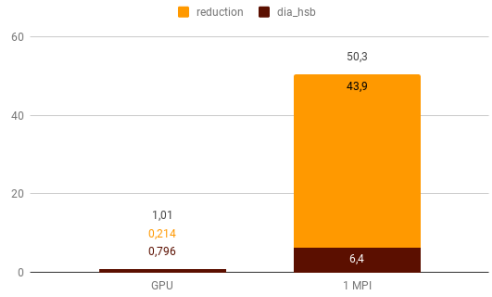


Fig. 16: dia_hsb diagnostic GPU vs. CPU time comparison

It demonstrates that the GPU implementation is 50 (50.3/1.01) times faster than a single MPI processes version.

## 5 Conclusions

This project has shown how big the impact of diagnostics can have on Nemo. Only some of them have been proved to take more than 10% of the total NEMO execution time. The most computing-hungry part of the Heat and Salt Budgets has been successfully ported to GPUs using CUDA. The calculation for the heat and salt conservation, volume and volume variation using CUDA is evaluated and different versions for the reduction procedure are explored. An example model has been created to compare the performance and validate the results of the GPU version while having in mind a future implementation on NEMO.

The results show that the data transfers were penalizing the dia_hsb_kernel GPU implementation. But allocating the memory as PINNED and selecting a GPU Directly Attached to the Power9 socket improved the new version (Figure 8), achieving a speedup of 8x compared to the CPU version. Afterward, several CUDA reduction optimizations have been discussed and the best one does achieve a speedup of 205x for 385125 double floating point elements. For the final implementation (Section 4.3), it has been demonstrated that since the data is already in the GPU after the dia_hsb_kernel, the CUDA reduction implementation does not require any data transfer from the Host to the Device. Whatsmore, the output is 4 scalars with the total volume, volume variation and heat and salt content deviation after the reduction. Thus, D2H data transfers overhead is completely hidden. The final implementation does achieve a total speedup of 50x compared to the performance of a single MPI process for a diagnostic that takes almost 4% of NEMO total execution time.

### 5.1 Future

The implementation of this optimization to the model is still pending and it is the next step of this work. As argued, the first goal remains to develop a stable version of NEMO where the dia_hsb CUDA diagnostic could run in any given experiment on the CTE IBM Power9 machine. However, this project is the first one on this direction (for NEMO at least) and can be as an example for any future similar work. Also, the results demonstrate that the implementation can be worth the effort. For now, only dia_hsb CUDA implementation has been studied in depth, although other diagnostics have also been mentioned (e.g., dia_ptr) and can also benefit from a GPU accelerator optimization while reducing the overall NEMO execution time. Especially since we can overlap the execution of the model (CPU) and the diagnostics (GPU). All diagnostics also imply I/O operations through XIOS. With the current MPI implementation, NEMO processes send the data to a XIOS process and then it is written. For now, Device variables are unique for every MPI process (as any other variable in a distributed memory environment). If it was possible that two processes had access to the same Device variables, it would be feasible to send the result of a CUDA kernel directly to the IO server (D2H copy). This would not only allow overlapping CPU

and GPU code (NEMO would not need to wait for the end of the kernel execution to get the result), but also reduce the number of MPI messages (NEMO would not need to send an MPI message to XIOS to write the diagnostics), resulting in less network usage.

In this project, only the diagnostics that run during NEMO execution have been considered. However, once NEMO has finalized, other diagnostics are computed as post-processing. Thus, other diagnostics exist and could be done 'online', reducing the post-processing amount of work. Nevertheless, as it is shown during the optimization, data transfers can become the main bottleneck on GPU implementations easily. Thus, it is crucial to evaluate how data needed for a diagnostic that is already in the GPU can be reused to compute other diagnostics ('online' or as post-processing') compensating the overhead of CUDA memory copies. Furthermore, this could also lead to more GPU computation time and make overlapping of data transfers with kernel execution[6] worth.

Finally, in this project, only diagnostics have been discussed. In future implementations, other parts of the model could be taken into account. Lots of work still exist in this direction having in mind how technology advance and that MareNostrum 5 will come with GPU accelerators.

## ACKNOWLEDGMENTS

I wish to express my sincere thanks to Mario Acosta, Postdoctoral researcher in the Computational Earth Sciences Group (BSC), for providing me with all the necessary facilities for the research.

I place on record my sincere gratitude to Ramon Grau, UAB teacher (CAOS), for his consistent encouragement and his monitoring of the project.

I take this opportunity to express gratitude to all of the Earth model performance analysis team members for their help and support.

## REFERENCES

[1] "Nucleus for the European Modeling of the Ocean". https://www.cmcc.it/models/nemo

[2] "A global ocean mesh to overcome the North Pole singularity". Gurvan Madec and Maurice Imbard

[3] "IBM Power System AC922. Technical overview and Introduction". Ritesh Nohria and Gustavo Santos

[4] "Optimization of an Ocean model using performance tools". Oriol Tintó Prims, Miguel Castrillo, Kim Serradell, Oriol Mula-Valls, Francisco J. Doblas-Reyes

[5] "Using Accurate Arithmetics to Improve Numerical Reproducibility and Stability in Parallel Applications". Yun He and Chris H. Q. Ding

[6] "Parallel programming: Concepts and practice". Bertil Schmidt, Jorge Gonzalez-Dominguez, Moritz Schlarb. ISBN: 978-0-12-849890-3

[6] How to Overlap Data Transfers in CUDA Fortran. Greg Ruetsch. https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-fortran/

[7] "MCS 572: Introduction to Supercomputing. Lecture 35, Memory Coalescing Techniques". Jan Verschelde, 11 November 2016

[8] "CUDA Fortran for Scientists and Engineers, Best Practices for Efficient CUDA Fortran Programming". Gregory Ruetsch Massimiliano Fatica. 1st Edition. ISBN: 9780124169708

[9] Efficient Implementation of Reductions on GPU Architectures. Stephen W. Timcheck. The University of Akron

[10] "Writing Fast Programs: A Practical Guide for Scientists and Engineers". John Rile

[11] "How to Optimize Data Transfers in CUDA Fortran" Greg Ruetsch. https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-fortran/

[12] "Implementations of Parallel Reduction in CUDA Fortran". Degawa, Tomohiro. (2016). The Proceedings of The Computational Mechanics Conference

[13] "Optimizing parallel reduction in CUDA". Mark Harris.https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

# APPENDIX

## A.1 Appendix 1

The indexing conversion to 1D depends on the input array ($m$) dimension and on the method used to store them (Fortran is column-major). Considering $i$, $j$, $k$ and $u$ for the indexing for the 1st, 2nd 3rd and 4th dimension respectively, Nx where x = {i, j, k, u} depending on the shape of the input and, $a$ as the equivalent 1D array:

- Given any 2D array:
  $m(i,j) = a(i + (j-1) \cdot Ni$
- Given any 3D array:
  $m(i,j,k) = a(i + (j-1) \cdot Ni + (k-1) \cdot Nj \cdot Ni)$
- Given any 4D array:
  $m(i,j,k,u) = a(i + (j-1) \cdot Ni + (k-1) \cdot Nj \cdot Ni + (u-1) \cdot Nk \cdot Nj \cdot Ni)$
  Since the only 4D array in the model is **tsn** and uses the 4th dimension to switch between heat and salinity content (used for independent calculations), it is reshaped as two consecutive 3D arrays using 3D to 1D index conversion. Thus, we got memory coalescing.

## A.2 Pageable vs Pinned throughput

A practical throughput test has been done to know how the data transfers between the CPU and the GPU differ depending on:

- The allocation of the data (pageable or pinned)
- The physical path it takes. If it goes thru the 75 GB/s connection to a Directly Attached GPU (DA) or it goes thru the 64 GB/s connection between Power9 processors (!DA) as shown in Figure 2
- The size of the data

In Figure 11 below, the throughput between the H2D and D2H has been averaged.

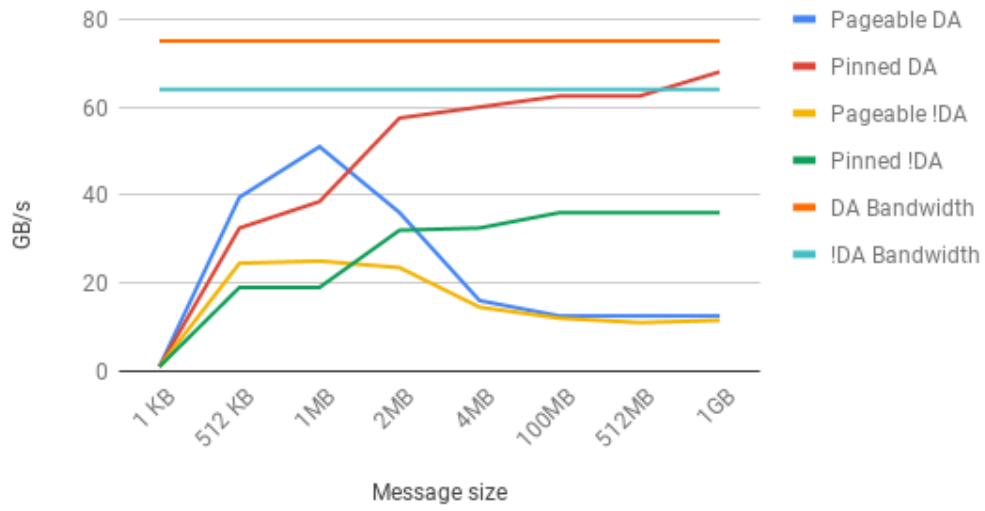


Fig. 17: Power9 Throughput comparison

## A.3 Heat and salt budgets kernel

We consider that all the data is already in the GPU. The kernel is launched with 512 threads per block and as many blocks as ceil(globsize, dimblock%x).

```
ATTRIBUTES(global) SUBROUTINE dia_hsb_kernel(surf, e3t_n, surf_ini, e3t_ini, &
    & tsn, hc_ini, sc_ini, tmask, Volume_variation, Heat_content, &
    & Salt_content, Volume, jpiglo, jpjglo, jpkglo)
    IMPLICIT NONE
    REAL(kind=8), DIMENSION(:) :: surf, e3t_n, surf_ini, e3t_ini, tsn, &
        &hc_loc_ini, sc_loc_ini, tmask, zwrkv, zwrkh, zwrks, zwrk
```

```
    REAL(kind=8), SHARED :: sdata(*)
    INTEGER, VALUE :: jpiglo, jpjglo, jpkglo
    INTEGER :: i, si, ti, globsize

    globsize = jpiglo*jpjglo*jpkglo

    !Thread indexing
    i = blockDim%x * (blockIdx%x-1) + threadIdx%x
    ti = threadIdx%x

    IF ( i .le. globsize ) THEN !Control out of bound accesses
        si = MOD(i, jpkglo)+1 !Access 2D array
        sdata(ti) = surf(si) * e3t_n(i) !Shared memory to avoid redundant code
        Volume_variation(i) = ( sdata(ti) - surf_ini(si) * e3t_ini(i) ) * &
            & tmask(i) * surf(si)
        Heat_content(i) = ( sdata(ti) * tsn(i) - &
            & surf_ini(si) * hc_ini(i) ) * tmask(i) * surf(si)
        Salinity_content(i) = ( sdata(ti) * tsn(globsize + i) - &
            & surf_ini(si) * sc_ini(i) ) * tmask(i) * surf(si)
        Volume(i) = sdata(ti) * tmask(i) * surf(si)
    END IF
END SUBROUTINE dia\_hsb\_kernel
```

## A.4 CUDA reduction implementation.

To improve the numerical reproducibility and stability on parallel applications the Knuth's trick is used. It employs 16 bytes complex numbers (8 for the real and 8 for the imaginary part) to increase the precision when adding multiple floating points. It is called from the Global reduction and improves the precision when adding each point of the array (a scalar) to a sum (current accumulated reduction value). Consecuentlly, it has been also ported to GPU (CUDA_Knuth_trick) and used in the reduce kernel (CUDA_reduction). The modifications made kernel can be found in the Appendix A.4.

Code on CPU:

```
    !d_array: 1D device array with the input data and used
    !to store the partial and final result.
    !d_imag: 1D device array to store the imaginary part
    !during the knuth's trick
    !sum: host scalar to save the final result

    d_imag = 0
    iterator = globsize !i*j*k
    piterator = iterator

    !loop util one element is left
    DO WHILE ( iterator .gt. 1 )
        dimBlock = dim3(512, 1, 1) !512 threads per block
        !half the number of blocks
        dimGrid = dim3(ceiling(real(ceiling(real(iterator)/dimBlock%x))/2), 1, 1)
        !call kernel and save shared memory per block for
        !double precision floating point elements (8 Bytes)
        CALL CUDA_reduction<<< dimGrid, dimBlock >>>(d_array, d_imag, globsize)
        piterator = iterator !save the number of elements done this iteration
        iterator = dimGrid%x !update the number of remaining elements
        !cudaMemset to 0 elements used in this loop
        d_zwrk((iterator + 1):piterator) = 0
    END DO
    !Copy result (1rst element) back to the Host
    i = cudaMemcpy(sum, d_array, 1)
```

Code on GPU:

```
    ATTRIBUTES(global) SUBROUTINE CUDA_reduction(invec, iin, jpi)
      IMPLICIT NONE
      REAL(kind=8), DIMENSION(:) :: invec, iin !input with the values and error accumula
```

```
    REAL(kind=8), SHARED :: sdata(512), idata(512) !shared arrays to compute the reduc
    INTEGER, VALUE :: jpi
    INTEGER :: tid, i, stride, idx
    COMPLEX(kind=8) :: stmp, atmp

    tid = threadIdx%x
    i = blockDim%x*2 * (blockIdx%x - 1) + threadIdx%x

    !Initialize shared memory
    IF (i .le. (jpi-blockDim%x)) THEN
        stmp = CMPLX(invec(i), iin(i), 8)
        atmp = CMPLX(invec(i+blockDim%x), iin(i+blockDim%x), 8)
        CALL CUDA_Knuth_trick(stmp, atmp)
        sdata(tid) = REAL(atmp)
        idata(tid) = AIMAG(atmp)
    ELSE IF (i .le. jpi) THEN
        sdata(tid) = invec(i)
        idata(tid) = iin(i)
    END IF
    CALL syncthreads()

    stride = min(blockDim%x/2, jpi)
    !Reduction of a cuda Block
    DO WHILE (stride > 0)
        IF( tid <= stride) THEN
            atmp = CMPLX(sdata(tid + stride), idata(tid+stride))
            CALL CUDA_Knuth_trick(CMPLX(sdata(tid), idata(tid), 8), atmp)
            sdata(tid) = REAL(atmp)
            idata(tid) = AIMAG(atmp)
        END IF
        stride = rshift(stride, 1)
        CALL syncthreads()
    END DO

    !Save the result (value and error) of every block to global memory
    IF ( tid .eq. 1 ) THEN
        invec(blockIdx%x) = sdata(tid)
        iin(blockIdx%x) = idata(tid)
    END IF
END SUBROUTINE CUDA_reduction

ATTRIBUTES(device) SUBROUTINE CUDA_Knuth_trick(ydda, yddb)
    IMPLICIT NONE
    COMPLEX(kind=8), INTENT(in   ) :: ydda
    COMPLEX(kind=8), INTENT(inout) :: yddb
    REAL(kind=8) :: zerr, zt1, zt2

    zt1  = REAL(ydda) + REAL(yddb)
    zerr = zt1 - REAL(ydda)
    zt2  = ( (REAL(yddb) - zerr) + (REAL(ydda) - (zt1 - zerr)) ) &
          & + AIMAG(ydda) + AIMAG(yddb)
    !The result is t1 + t2 after normalization
    yddb = CMPLX( zt1 + zt2, zt2 - ((zt1 + zt2) - zt1), 8 )
END SUBROUTINE CUDA_Knuth_trick
```