Available on-line at www.prace-ri.eu

Partnership for Advanced Computing in Europe

GPU Implementation of the DP code

F. Sottile[a], C. Roedl[a], V. Slavnić[b], P. Jovanović[b], D. Stanković[b], P. Kestener[c], F. Houssen[c]

[a]Laboratoire des Solides Irradiés, Ecole Polytechnique, CNRS, CEA, UMR 7642, 91128 Palaiseau cedex, France

[b]Scientific Computing Laboratory, Institute of Physics Belgrade, University of Belgrade, Pregrevica 118, 11080 Belgrade, Serbia

[c]Maison de la Simulation, USR 3441, bat. 565, CEA Saclay, 91191 Gif-sur-Yvette cedex, France

Abstract

Main goal of this PRACE project was to evaluate how GPUs could speed up the DP code – a linear response TDDFT code. Profiling analysis of the code has been done to identify computational bottlenecks to be delegated to the GPU. In order to speed up this code using GPUs, two different strategies have been developed: a local one and a global one. Both strategies have been implemented with cuBLAS and/or CUDA C. Results showed that one can reasonably expect about 10 times speedup on the total execution time, depending on the structure of the input and the size of datasets used, and speedups up to 16 have been observed for some cases.

1. Presentation of the DP code

The DP code [1] is an ab initio TDDFT linear response code, working in reciprocal space, on a plane-waves basis, and in frequency domain. Its main purpose (not the only one) is to calculate the electronic polarizability of a wide range of materials, including 0D (like atoms or clusters), 1D (nanotubes and nanowires), 2D (graphene, surfaces, and layered systems) or bulk. The code is written in Fortran 90, with some insertion of C (essentially for parsing the input file and dealing with the operative system). The testing suite is written in PERL. The post-processing tools are written in Fortran and in Python.

The structure of the code is the following:

- Initialization part: reading of a ground state electronic structure file and creation of all possible energy transistions N_t(from valence/occupied to conduction/empty states).
- Creation of the polarizability

$$\chi^0(g,g',\omega) = \sum_{t=1}^{N_t} \tilde{\rho}_t(g) \times \tilde{\rho}_t^*(g') \times \mathrm{den}_t(\omega)$$

  where for any transition t there is $\tilde{\rho}_t(g)$, a vector of dimension N_g (the number of plane waves), and $\mathrm{den}_t(\omega)$, a vector of dimension N_ω (number of energies in which the polarizability is evaluated), to be calculated. The CPU times grows linearly with N_t and quadratically with N_g, while the memory occupancy goes like $N_g^2 \times N_\omega$ (the dimension of χ^0). This is the most cumbersome part of the calculation, especially for what concerns the CPU time, but also, in most cases, for what concerns the memory occupancy. The evaluation of χ^0 gives also the scaling of the whole code, which goes as N_{at}^4, with N_{at} = number of atoms.
- Creation of the macroscopic dielectric function via the formula

$$\epsilon^{-1} = 1 + v(g)\left(1 - \chi^0 v\right)^{-1}_{gg''} \chi^0(g'',g',\omega)$$

  which involves a matrix inversion. This term does not give particular problems: each energy ω can be treaten in an independent way and is easily and efficiently parallelized.

- Writing down the output and spectra.

After a preliminary analysis, it has been confirmed that the evaluation of the polarizability χ^0 and in particular the time spent in the matrix creation, through the library method CGERC, is the most time consuming part and will the object of the present parallelization project.

2. GPU strategies

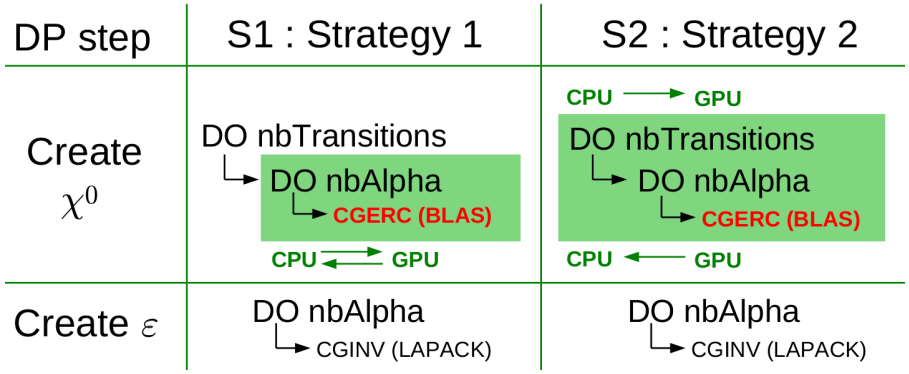


Fig. 1: GPU strategies for DP

The structure of the DP code is illustrated by the Figure 1. In DP, the typical parameter magnitudes are as follows: the number of transitions is about 1 000 000, and the number of alpha is about 500. The code is made of 2 major parts: the first one is "create χ^0", and the second one is "create ε". A profiling analysis made, first with GNU gprof [2], then with Scalasca [3], confirmed that there are two hot spots:

- The first hot spot is an intensive call to CGERC (BLAS method) in "create χ^0": CGERC is called about 500 000 000 times ($nbTransitions \times nbAlpha$ according to the notations in the Figure 1).
- The second hot spot is a repeated call to CGINV in "create ε": CGINV is called about 500 times ($nbAlpha$ according to the notations of the Figure 1). CGINV calls CGETRF and CGETRI (LAPACK methods).

From profiling results, it turns out that CGINV takes (in average) 1000 times more time than CGERC, roughly speaking. Finally, there is still a factor 1000 between CGERC and CGINV. On overall, the CGERC operation is costlier than CGINV: this is why CGERC is found to be the first hot spot. This is illustrated by the Figure 2 where one can clearly see that the χ^0 step is much more time consuming than the ε step. As a

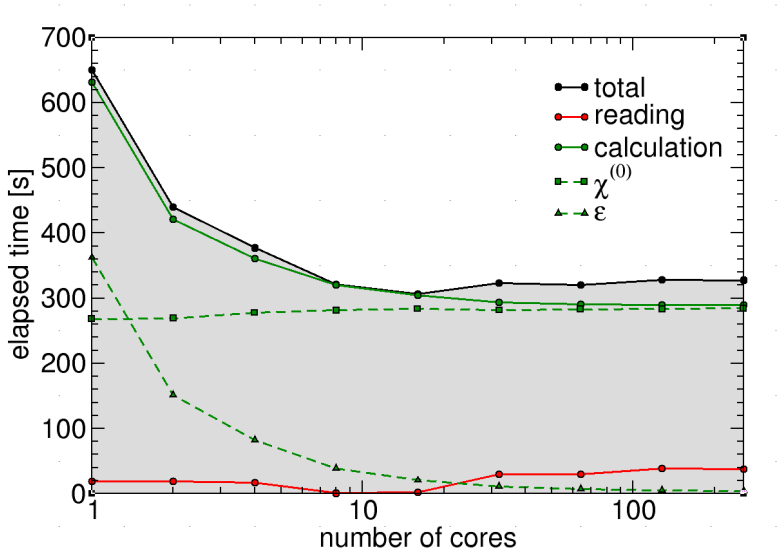


Fig. 2: Profiling of a typical DP run

consequence, speeding up "create ε" with a classical MPI approach works pretty well. Nevertheless, speeding up "create χ^0" with a classical MPI approach may not be that efficient: one could expect GPU to do things faster.

To port the DP code to GPU, one can use two strategies. The first strategy is local and has been implemented by the "Maison de la Simulation" (France): the idea is to delegate to GPU only a subpart of the "create χ^0" step. The second strategy is global and has been implemented by the Institute of Physics Belgrade (Serbia): the idea is to delegate to GPU the whole "create χ^0" step. Before activities on GPU implementation started, MPI version of DP, which was able to distribute the work over several MPI processes, was already available, and it was used as a starting point in porting activities.

3. Strategy 1

3.1. Presentation

The first strategy is local: basically, the idea is to delegate CGERC to GPU. This is the very first and most simple idea one can have. The main drawback of this approach will clearly be CPU/GPU transfers that are known to be bottlenecks. The Figure 3 illustrates the algorithm. For the sake of clarity, we recall that CGERC (BLAS method) performs the following operation:

$$A(i,j) \mathrel{+}= \alpha \times X(i) \times \overline{X(j)} \tag{1}$$

where A is a complex 2D matrix, α is a scalar and X a complex vector. In DP, CGERC is called a lot of times so that α is actually a vector. Finally, the work to delegate on GPU looks like:

$$A(i,j,k) \mathrel{+}= \alpha(k) \times X(i) \times \overline{X(j)} \tag{2}$$

where A is a complex 3D matrix, α is a vector and X is a complex vector.
The 3 data structures to transfer from CPU to GPU are A, α and X. After the GPU computation is done, the only data to transfer back to CPU is A. α and X are updated by the CPU at each iteration, and must be transfered to the GPU at each iteration: according to notations of Figure 3, α and X are transfered $nbTransitions$ times (the typical magnitude of $nbTransitions$ is 1 000 000). A is not modified at CPU side so that, when possible, A may be only transfered once to the GPU (at CPU side, A is seen as a computational result needed to proceed to the following "create ε" step).

Clearly, one has two possible situations:

- A, α and X fit all at once on GPU memory. Here, A can be transfered once from CPU to GPU at the first iteration, and, A can be transfered back from GPU to CPU at the last iteration (A stays on GPU and is updated on GPU). One can expect significant speedup using GPU.
- A, α and X do **not** fit at once on GPU memory. Here, A will have to be transfered by chunk at each iteration. The GPU approach will be much slower than the initial full CPU code. In this case, the only way out is to use MPI to split the initial data into smaller pieces to distribute over several processes. A is a matrix whose elements are single precision complex (2 * 4 bytes = 8 bytes). According to Figure 3, the size of A is $dimX^2 \times nbAlpha$ where $dimX$ is the size of the X vector and $nbAlpha$ is the size of the

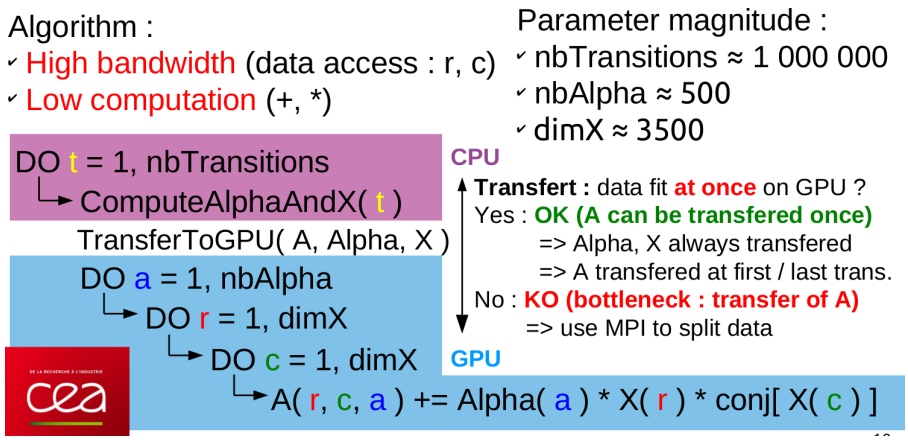


Fig. 3: Strategy 1 - Presentation

α vector. As a result, the rule of thumb to apply to be sure that data will fit on GPU, is to choose $dimX$ and $nbAlpha$ such that:

$$8 \times dimX^2 \times nbAlpha \approx 0.8 * GPUMem \quad (3)$$

where $GPUMem$ is the memory in bytes available on GPU (generally, $dimX$ is fixed and $nbAlpha$ is adjusted). Note that to know the value of $GPUMem$, the user has just to run DP during one iteration (one transition) using the verbose mode (to specify in the input file): $GPUMem$ will be printed in DP output log. This rule enables to target 80% occupancy on GPU (to get the best possible performance, GPU must be loaded enough).

In short, the first strategy can be wrapped-up this way: the (potentially large) data are split (in smaller pieces) and spread over several MPI processes (CPU), then each CPU is associated to a GPU, and finally each CPU delegates the job to his GPU. Once the GPU is done, the CPU will get results back and go over the "create ε" step. One can expect to overlap GPU computations with CPU to GPU transfer (X and α are updated at CPU side, so that they must be transfered to GPU at each iteration). Note that a verbose mode is available in the implementation. This verbose mode forces GPU synchronization (with CPU) to get reliable timing information (profiling information). As a result, the user should turn off the verbose mode for production runs (the verbose mode makes the CPU wait for the GPU at each iteration, so the transfer/computation overlap will not occur). Both CUDA and cuBLAS implementations have been done and compared (the cuBLAS version was used as a reference in terms of results and performance).

3.2. CUDA kernel validation

The CUDA kernel has been validated outside of any context of use (outside of DP). The Figure 4 illustrates the validation process. On GPU, floating point operations will not be run in the same order as on CPU. Moreover, the IEEE norm can not ensure commutativity. CPU results are taken to be the reference: to allow a fair comparison with GPU, some specific options have been set during compilation (nvcc options [4]: –ftz=false –prec-div=true –prec-sqrt=true). To validate the CUDA kernel, one had to use double precision complex and to perform every calculation with double precision: results show the relative error (compared to CPU results) is bounded by 10^{-14}. Note that using double data, we can observe here the transfer bottleneck that may occur if all data do **not** fit at once on GPU (Figure 4 for $dimX >= 800$): speedup drops as A must be transferred by chunk at each iteration (transition). At this point, a float version of the double kernel was derived: basically, double are replaced by float. The float kernel will be faster but the relative error magnitude will be about 10^{-5}. Hoping a better error/speed up trade-off, an additional test has been performed using float data, converting float to double, and computing with double precision: results show that the benefit is only a potential error decrease (10^{-7}), while the speedup drops (tests showed that one can expect error decrease when the dataset is "small" enough, otherwise the number of operations will increase so much that at least one of them will cut the error back to the previous full float kernel). Using float data and performing double computation is not a

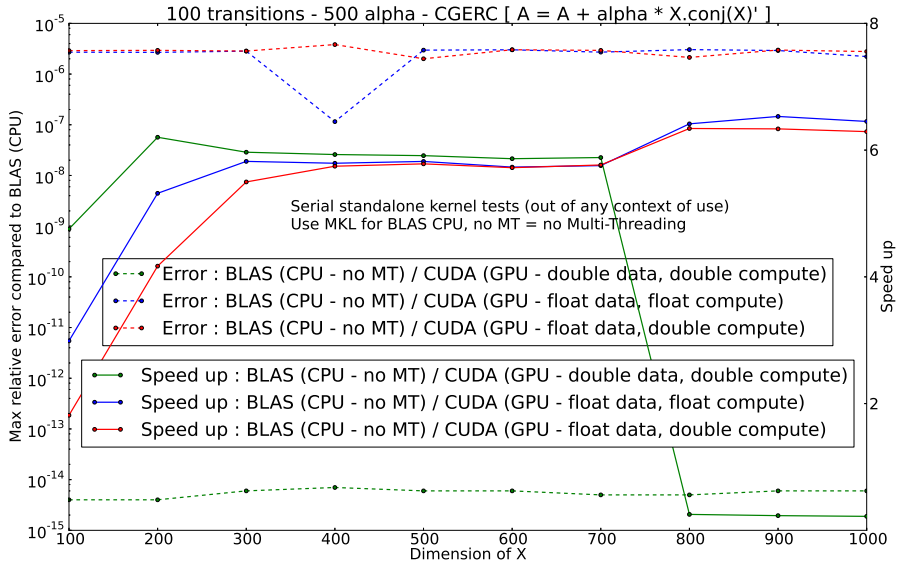


Fig. 4: Strategy 1 - Validation

good trade-off. Finally, the float kernel will be used to speed up DP as the A matrix handled by DP is a single precision complex matrix.

3.3. CUDA kernel optimization

The CUDA kernel has been optimized outside of any context of use (outside of DP). The Figure 5 and 6 illustrate the optimization process. The process is as follows: test several grid sizes to find the best one (here, 768 X 16), and optimize step by step. First, to get the best possible performance, coalescing [5, 6] has to be ensured. Then, as the algorithm is highly memory bandwidth intensive, use of shared memory has been added to the CUDA implementation, and improves significantly performances (the first coalesced kernel was naive, and shared memory has been added after some optimizations so that the gap between the coalesced and shared memory version is not only, but mostly, due to shared memory [5, 6]). Finally, computations have been overlapped with transfers. At this step, a profiling analysis (using nvvp, the Nvidia profiler) has shown there

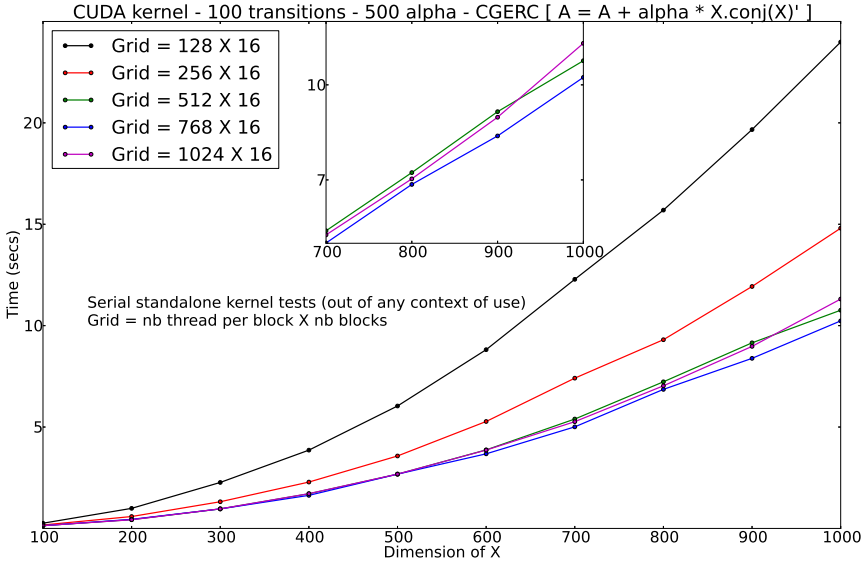


Fig. 5: Strategy 1 - Grid sizing

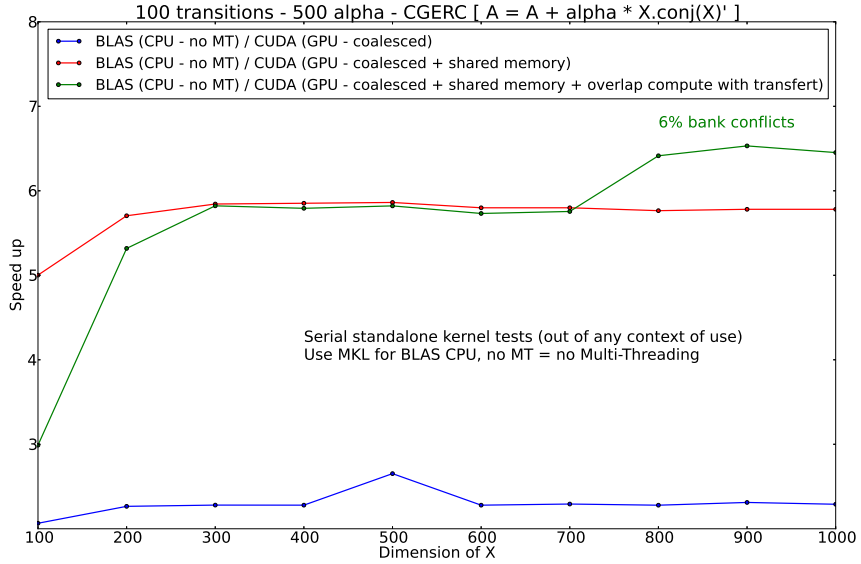


Fig. 6: Strategy 1 - Optimization

was only 6% bank conflict left so the optimization has not been pushed further (actually, a kernel has been written to reduce bank conflicts down to 1%, but to achieve this, data have to be stored and accessed in a complex way so that the speedup drops).

3.4. Results

First, the strategy 1 has been tested outside of any context of use (outside of DP). The Figure 7 illustrates the first results. The cuBLAS implementation has been done so that it should be possible to overlap computations with transfer: as the CGERC method from cuBLAS has to be called progressively at each iteration (one call for each α), results are pretty much the same when there is no overlap. The CUDA implementation allows to compute directly the whole dataset (one call for every α) and to overlap efficiently computations with transfer. The speedup is computed using a non multi-threaded CPU (1 thread, CGERC from MKL BLAS) as reference. For the sake of completeness, GPU approaches have been compared with multi-threaded CPU approaches (several threads, CGERC implemented with OpenMP, ensuring static scheduling with imposed chunk size and 1 core–1 thread affinity to get the best possible performances). It turns out that, as the strategy is local, threads have to be created and destroyed frequently so that multi-threaded CPU approaches are not efficient.

Then, the CUDA and cuBLAS implementations have been plugged into DP. The Figure 8 and 9 illustrate the speedup one can expect. First, a small test case (Argon) has been ran from 1 to 8 MPI processes. Then, a bigger test case (HFO2O) has been ran from 16 to 256 processes (on "Curie", 288 GPUs are available so that using 256 processes was the biggest possible test). In both case, strong and weak scalings have been performed. On "Curie", 2 queues are available: a full CPU one (xlarge), and a CPU/GPU one (hybrid). The xlarge nodes are made of 8 (quadricore) CPUs, while the hybrid nodes are made of 2 (quadricore) CPUs and 2 GPUs. To allow a fair comparison, the CPU runs have been performed ensuring 2 MPI processes per node on the xlarge queue. Results show that, the more there is work to do, the more CUDA outperforms cuBLAS. To get the best possible performance, GPU must be fully loaded (target 80% GPU occupancy), and, for big runs (128 MPI processes or more) the user must make sure that enough RAM is available (otherwise the code swaps and the speedup drops). Depending on the case, one can expect a 10 to 14 times speedup (on the total elapsed time).

It should be stressed that for:

- The faster Argon CPU run (weak scaling over 8 procs), the initialization time (reading input files) was about 15 seconds while the computation time (χ^0 and ε steps) was about 19500 seconds. The ratio between initialization and computation time is about 1300.
- The faster HFO2O CPU run (weak scaling over 256 procs), the initialization time (reading input files) was about 170 seconds while the computation time (χ^0 and ε steps) was about 10150 seconds. The ratio between initialization and computation time is about 60.

As only the χ^0 step can benefit from GPU speedup, we can expect better speedup (than 10-14 times) in cases where the ratio between initialization and computation time is higher (ratio >10000) than the ones presented in this document.

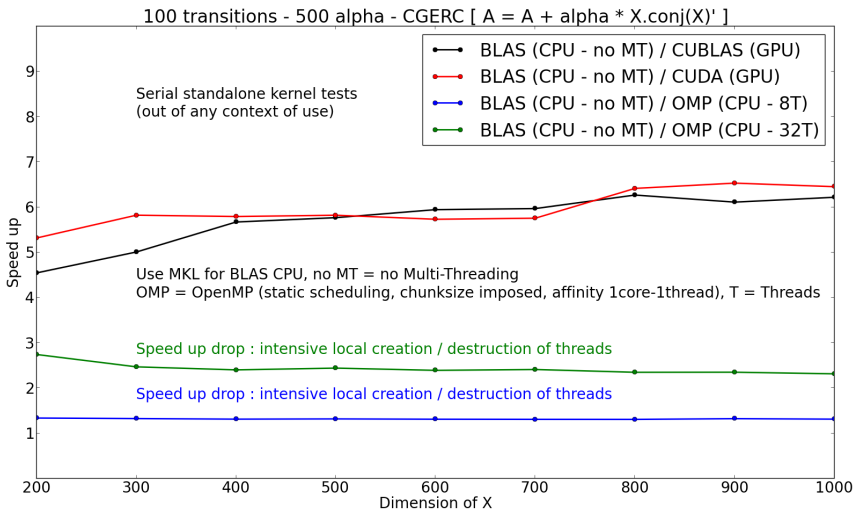


Fig. 7: Strategy 1 - Standalone results

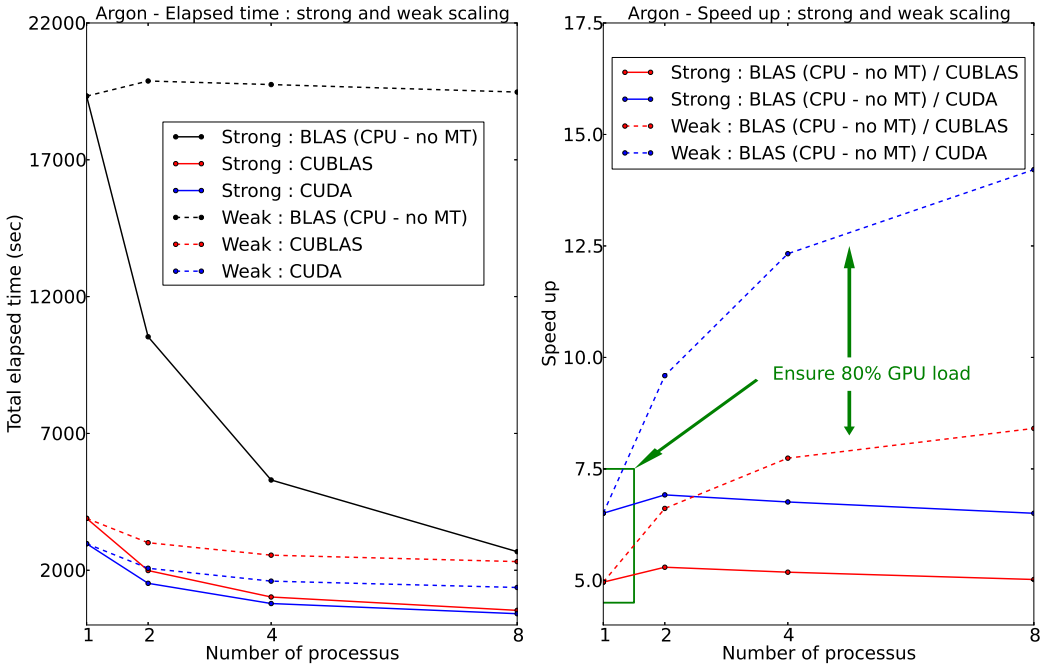


Fig. 8: Strategy 1 - Argon : small test case

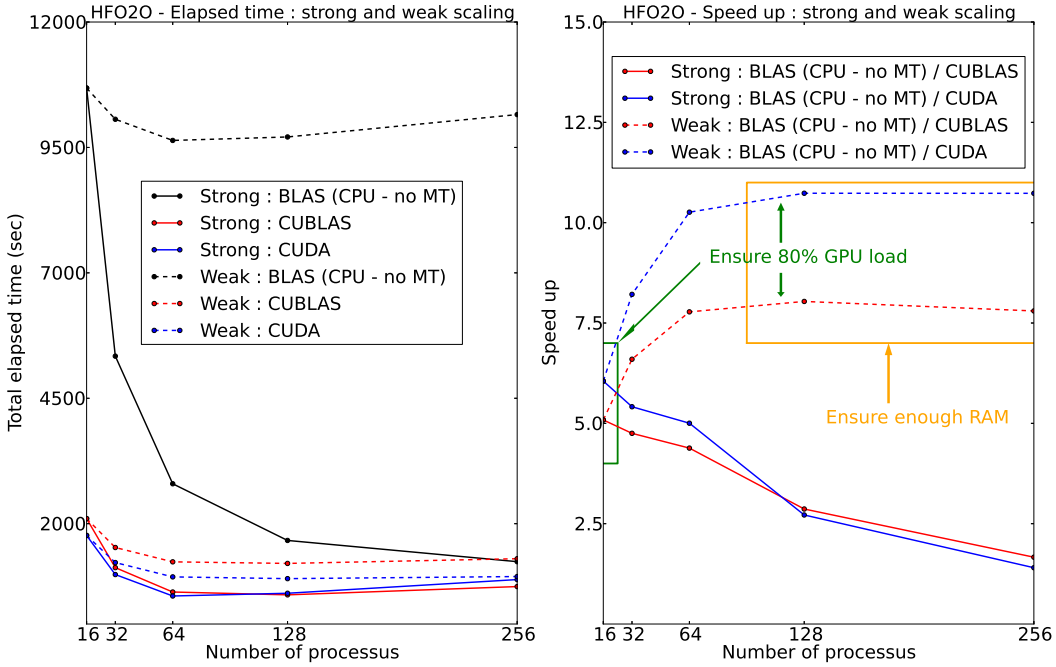


Fig. 9: Strategy 1 - HFO2O : typical test case

4. Strategy 2

4.1. Presentation

The main goal when designing strategy 2 of the GPU approach for the DP code was to minimize communication and data transfers between the CPU and the GPU. This communication has an impact on the performance, because in the algorithm presented in strategy 1 section, data has to be copied to the GPU for each transition (one iteration of the computational loop). When the number of transitions is large (which is often the case for some real inputs DP was designed for), it is expected that the memory transfer time can be a significant factor in the performance of the entire program.

To ensure that the data will be transferred only once to the GPU (and back to the CPU after all transitions have been processed), the whole loop which iterates through all transitions had to be ported to the GPU using CUDA technology. This can be seen on Figure 1, where the outline of the algorithm is presented.

However, because now both α and X vectors need to be computed directly on the GPU, all related data structures needed for their computation need to also reside in the GPU memory during all iterations of the transitions loop. To illustrate this, we will briefly present needed steps for computation of the X vector for one transition:

- We select 2 waveforms from the auxiliary data structure, which holds all waveforms for the input dataset, based on the index depending on the current transition.
- We perform FFT on them, multiply them together and perform inverse FFT on the resulting waveform vector. During the multiplication phase, elements of 2 waveforms to be multiplied in one step are selected using indexes stored in the table containing index mapping. Which column from the mapping table will be read, again depends on the current transition.
- We select a subset of the waveform calculated in the previous step to get the X vector which will be used together with α to update the A matrix.

Because the waveform structure and the mapping table described above also consume significant amount of memory, that means that the GPU memory now has to be shared between the A matrix and these structures. Also, the number of waveforms in the auxiliary structure grows linearly with the number of total transitions, so we can expect to reach the limits of the available GPU memory much faster than with strategy 1, as the size of the input test case grows (whether with more transitions or with larger A matrix).

Available memory limit can be overcame either implicitly with distributing computation among many MPI processes, or by explicitly managing data movement on the GPU card, so that the matrix is processed in parts. The MPI approach for division of the data is already present in the DP code, and when enough resources are not available, a modification of the strategy 2 for out-of-core solving of the problem was developed, and will be described later in this text.

Having in mind that now the entire body of the loop will be executed on the GPU, the strategy 2 approach has a risk of slowing down parts of the loop that are sequential or less parallel than updating of the matrix A, because in that case advantages of the GPU are lost or at least diminished (a single GPU core is much slower than the equivalent CPU core). However, since the FFT computation is now also moved to the GPU, it is expected that it will be faster than on the CPU. Additionally, for larger test cases, vectors that are being operated on inside the loop are larger and thus more suitable to be processed in parallel. Therefore, in overall, the sequential and less parallel parts of the computational loop are not expected to be a significant factor of slowing down of the algorithm when running on CUDA GPUs.

4.2. Implementation details

For the implementation of the strategy 2, CUDA C was used along with cuBLAS and cuFFT numerical libraries. CUDA code was split in multiple kernels, each corresponding to a single original FORTRAN routine called inside the loop. Each call to FFTW3 library routine was swapped with a call to cuFFT routine, and each MKL BLAS CGERC call was swapped with the corresponding cuBLAS call (cublasCgerc). One FORTRAN routine had to be split into 2 kernels, because there is a data dependency between the first and the second part of the routine, which couldn't be solved within a CUDA thread block. This initial GPU implementation works properly if the matrix A, all waveforms and index mapping tables can fit into GPU memory together.

Because kernels were developed in CUDA C, and could not be called directly from FORTRAN code, several wrapper C routines were created for initialization of the GPU memory, copying necessary data to the GPU, executing body of the computational loop, and copying the A matrix back to be further used in the create ε step. Custom kernel for updating the A matrix was not developed like it was done in strategy 1, because more focus had to be put on porting the entire loop on the GPU and on implementing modification for out-of-core execution when the input data sets are exceptionally large.

Even though it might look like performance-wise it could be better to put the entire loop in the single kernel, because some overhead related to starting multiple kernels from the CPU side could be alleviated, it was not done in this implementation. One of the reasons for this is that we wanted to keep using external FFT library (for simplicity, and better performance), and it was not possible to call cuFFT from inside the kernel. The other reason is that because waveforms are not always accessed in a linear fashion, but are instead sometimes indexed

indirectly using values from the mapping table, a situation where synchronization of all threads executing the kernel in necessary can arise. This cannot be done efficiently in CUDA, because it was designed in a way that only threads in same blocks can directly synchronize and share data.

There are two independent tasks that need to be performed before the A matrix is updated:

- Calculation of the X vector, and
- Calculation of the α vector

Because there are no data dependencies between these tasks, they can be executed concurrently on the GPU. To achieve this, separate CUDA streams are created, one for each task, and each kernel is configured to run in a stream associated with the task they are a part of. Because FFT is a part of the X vector calculation, cuFFT library was configured to run in a first stream. This overlapping of computation on the GPU can be useful because there is a varying degree of parallelism between kernels (some operate on long vectors, some on shorter ones, and some even run with only a single thread), and it is possible that GPU won't be fully utilized running just a single task at a time. CUDA runtime can then select which kernels can be overlapped and help us gain a bit of performance.

For updating of matrix A using cuBLAS *cgerc* call, streams are also used to help in gaining performance. Increase in performance when using streams can especially be significant when the size of a single transform is not very large. Since in DP code many matrices need to be updated in every loop iteration (meaning that we can view A as a 3D structure, see Equation 2 in Section 3.1.), and cuBLAS allows for overlapped calls, we have used 30 separate streams to perform matrix updates. This helped us gain approximately 2-3 times the performance compared to sequential updating.

We have managed to avoid host to device synchronization entirely in this case, save for the points of memory transfers, which only happen twice. Streams were synchronized directly on the GPU, so that as soon as the computation of X and α was done, updating of the matrix starts. Because synchronization with the host is much more expensive than the synchronization between streams on the GPU, a potential overhead was successfully eliminated.

4.3. Offloading computation to CPUs

Even though it is expected that moving main DP computational loop to GPU will result in significant performance gains, it shouldn't be forgotten that CPUs on modern HPC systems have multiple cores and can contribute to calculation tasks to some extent. The advantage of the algorithm employed in the DP code is that iterations of the loop are independent and can be executed in parallel or out-of-order, and this can be used to offload some subset of iterations to be executed on CPUs.

Because there is already a routine in the DP code which equally splits transitions to MPI processes when DP is used with MPI parallelization, it was easy to modify that routine to take into account that both CPUs and GPUs will participate in calculating transitions. There is difference in performance between a GPU card and a CPU core, so that routine was modified to perform a weighted distribution instead. Once a MPI process knows its set of transitions, it can begin with the computation, regardless of whether it will be performed on the GPU or on a CPU core.

Each MPI process can either use a GPU or a CPU core for the execution of the computation loop. In this implementation, first 2 processes on each node were selected to use GPU for computations, because a CURIE hybrid node contains 2 GPU cards. DP already supports running performance tests with only a few iterations of the loop (used to estimate total time needed for simulation, *testrun* input parameter controls this), so this test was extended to run on both the CPU and the GPU. After a short test run has been performed, reported ratio between execution times on the CPU and the GPU can be used as an input to routine where transitions are split. In this way, it is expected that processes that run loop on the GPU and those who run it on the CPU will complete calculation for transitions assigned to them in roughly the same time.

4.4. Modifying strategy 2 for out-of-core approach

As previously described in the presentation of this strategy for GPU code implementation, instead of only storing the A matrix on the GPU during loop execution, some auxiliary structures also need to be stored, so that vectors α and X can be calculated directly on the GPU. A problem arises when combined size of these structures exceeds available memory on the GPU, because then it is no longer possible to run DP for this input case. To solve this problem, we had to manipulate data movement between CPU and GPU so that data is transferred only when absolutely needed, and kept and reused on the GPU for as long as possible, in order to reduce the penalty for needing to synchronize data on the main memory and on the GPU. Although the performance of the code for the out-of-core approach won't be optimal (because now there will be more communication than initially planned for the strategy 2 approach), we wanted to show that it can be used to solve problems that couldn't even be solved without it when not enough memory is available, and that the introduced overhead will still be within acceptable bounds.

Analysis of the access patterns of the structure holding all waveforms and the table containing index mapping showed that they exhibit very good locality. Waveform structure was accessed in a way that only a small part of it is used for some subset of N transitions (where N is significantly larger than the number of used waveforms), while mapping table was accessed so that a single column is reused multiple times, and then wasn't used again

for a long time. Because of that, if we had enough space to hold the working set of waveforms on the GPU, and just a single column of the mapping table, we could ensure that data won't be copied between CPU and GPU more than necessary. In fact, for test cases we have used during development, after a part of the waveform structure was used, it was not needed again. If that is the case, then the total amount of data transferred to the GPU will be the same as when everything fits at once (only that now data transferring has to be done using multiple copy operations).

We implemented software caching of the waveform structure and the mapping table in the GPU memory, using FIFO replacement policy, so that data loaded the earliest is swapped first. One waveform or a mapping table column is transferred to the GPU when it is needed, and reused in subsequent transitions. That allowed us to have the number of "cache misses" equal to the total number of waveforms (meaning each one is loaded only once because of compulsory misses). Size of the waveforms cache was an order of magnitude (or more) smaller than the initial size of the waveform structure for our test cases, and only a single line was needed for caching mapping table entries, which allowed us to have more space for storing of the A matrix.

The situation is a bit different with the A matrix, because entire matrix has to be updated for each transition. That means that we can't make any use of caching, but instead have to resort to updating the matrix part by part. First, a part of the matrix is transferred to the GPU, then it is updated using cuBLAS calls, and it is returned to the main memory. This is repeated until all parts have been processed. Downside of this is that the entire A matrix has to be transferred back and forth for each update. Because memory transfers take more time than the computation itself, this can slow down the algorithm by at least an order of magnitude.

To overcome this problem, internal structure of the loop had to be slightly modified. Main change is that now instead of computing one α and X vector and then using them to update the A matrix, many α and X vectors that correspond to subsequent transitions are computed in a batch, and then used to update parts of the A matrix, for many transitions at the time. These transformation of the computational loop don't affect algorithm correctness, because updates on the A matrix are linear and can be executed in any order. This is better illustrated by the following pseudo-code:

- Initial version of the loop:

```
do i=1, num_trans
  calculate X
  calculate alpha
  update_matrix(A, X, alpha)
end do
```

- Modified version of the loop for out-of-core execution:

```
batch_size = K
num_batches = num_trans / K
do i=1, num_batches
  do j=1, batch_size
    calculate X(j)
    calculate alpha(j)
  end do
  for each A part
    copy A_part to GPU
    do j=1, batch_size
      update_matrix_part(A_part, X(j), alpha(j))
    end do
    copy back A_part to CPU
  end for each
end do
```

As a consequence, now the entire matrix A doesn't have to be moved for each transition, but can instead be transferred only once for a batch of K transitions. This can greatly reduce time spent in copying data, and can help us get close to performance of the original algorithm if the batch size is large enough. At the same time, with large batch size, significant space on the GPU card has to be dedicated to storing of intermediate α and X vectors, and that space also plays a part in the memory management.

Managing memory space for all these structures might not be easy for each possible problem DP is used to solve, but for our test cases it could be done with very good efficiency. It was done with assumption that waveform structures and mapping vectors exhibit good access locality for input cases used with DP (this was true for all the cases we tested during development). Our approach to divide available memory between all these necessary structures can be summarized as:

- First analyze access patterns for waveform structure and mapping table, and allocate just enough space for their software caches, so that entire working set can fit at once. This ensures that no unnecessary swapping of some waveform will happen, even when it is still needed in the near future.

- Then we allocate space for the number of A matrices equal to the number of CUDA streams used for cuBLAS calls. This was done to ensure that there will be enough data to fully utilize computational capabilities of the GPU, and was selected more as an educated guess than a strict rule.
- The remaining space can then be used to store K pairs of temporary vectors.

Theoretically, if we can manage to fit temporary vectors for all transitions on the GPU, we should be able to match the performance of the original algorithm, even when using much less space on the GPU card. The only difference will then be that the large amount of data would be transferred using many separate calls to copying routines, instead in a single call, but assuming that the modification for out-of-core execution will be used for very large data sets, overhead associated with a routine call will be insignificant compared to the time spent in actually transferring the data.

4.5. Results

Strategy 2 for porting DP code to the GPU has been tested using the DP code already supporting MPI parallelization, with the computational loop moved to the GPU. Also, a modification of the strategy 2 where a part of the computation is offloaded to the CPU was tested. Separate testing was done using modified input (to create larger data structures) with the out-of-core modification, using only one GPU card, to show what kind of a performance drop we can expect if there is not enough memory to solve the problem in-core (where there are only 2 points of data movement – before and after the computation).

All development, verification and performance testing of the GPU strategy 2 of the DP code was done using Curie supercomputer (specifically, Curie hybrid nodes). Each hybrid node on Curie consists of 2 Intel Westmere processors, each having 4 processing cores running on 2.66GHz, and 2 NVIDIA Tesla M2090 graphic processing units (GPUs).

First series of tests was performed with the MPI+GPU approach for the DP code, when the main computational loop was executed only on the GPU cards. Because each Curie node has 2 Tesla cards, when running on more than one node, processes were spread over all nodes so that there are exactly 2 MPI processes executing on a single node. Also because there are 2 CPU sockets, and each GPU is connected to a bus local to the socket, each MPI process was bound to a separate socket. More details on this, and the explanation on how to control process scheduling can be found in Curie Best Practice Guide [7].

On Figure 10 we show execution times and speedups for test cases of Argon (small test case), and Hafnium oxide (larger production-ready input). We would like to stress that these inputs are not exactly the same as the ones reported with strategy 1 (see Section 3.1.), because there the input was modified in order to reach at least 80% memory occupancy on the GPU. This was not done here, because the emphasis when designing strategy 2 was not just to showcase the maximum performance of the "updating A" step, but instead to address the overall performance of the loop. Tests were conducted for configurations having from 1 to 256 MPI processes (on 1 to 128 nodes), with 1 GPU card per process. Execution time is split into initialization time (mostly reading input files from disk, completely unrelated to GPU implementation) and computation time for create χ^0 step, which was the focus of the GPU algorithm.

It can be seen that GPU implementation of strategy 2 for DP code scales reasonably well when a larger number of GPU cards is used in execution. For the small test case, it would be optimal to run with 16 MPI processes (using 16 GPUs), while for the larger input, scaling is very good for up to 64 processes. Since

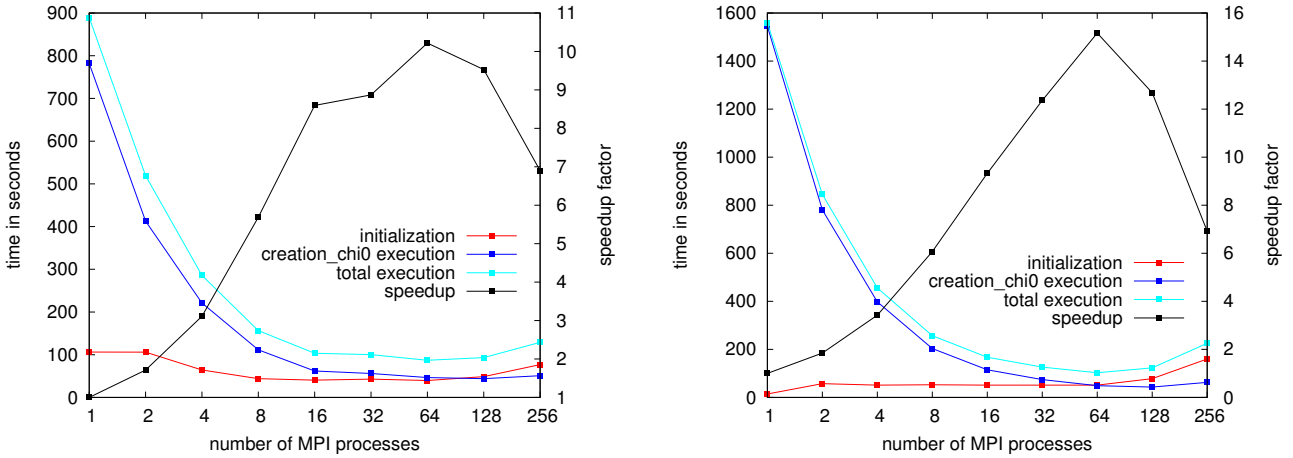


Fig. 10: Performance and scaling of the GPU strategy 2 for the DP code, for 1 to 256 total GPU cards with 2 GPUs per CURIE node (time when running with 1 GPU used as a baseline for speedup): (a) Argon (small) test case; (b) Hafnium oxide (moderate) test case.

computing one transition is independent from others, when the total number of transitions is divided to more processes, time for the computation itself decreases in a linear fashion. Because of that, reasons for limitations in scaling DP CUDA code are in other parts of the code, that are either constant or even increase as the number of processes increase.

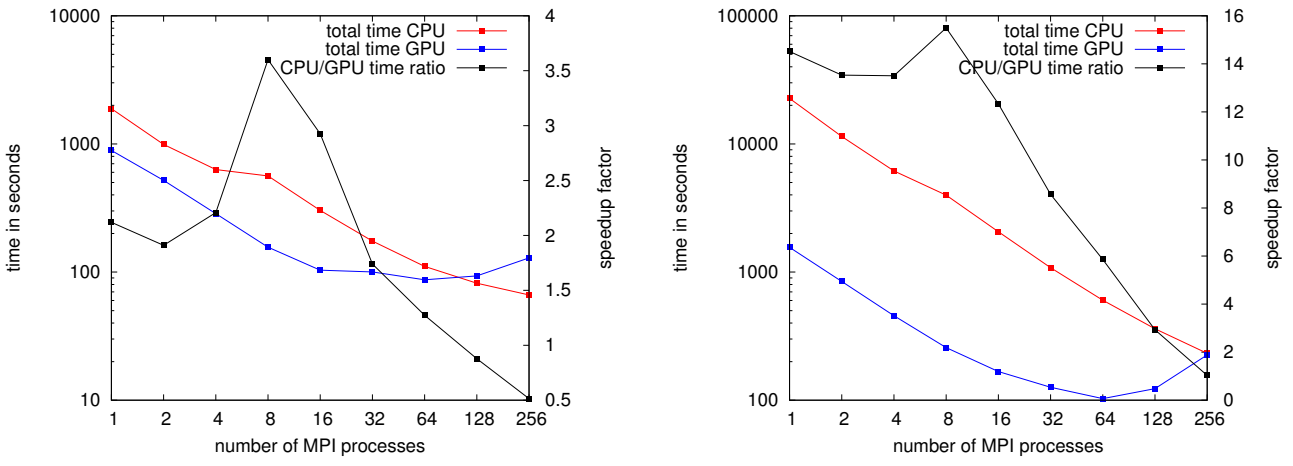
Time spent in DP initialization is unrelated to CUDA, and is roughly constant over the range of processes used for testing (with the exception for 256 processes, when its increase can be easily observed), so once that time becomes close to, or even surpasses the calculation time on the GPU cards, adding more and more card actually does very little for the performance. Figure 10 also shows that the execution time actually stops decreasing, as would be expected, and even increases at the end of the range. Reason for this is that the CUDA runtime initialization time is not constant, but instead starts to increase rapidly when the number of GPU cards used grows. This cannot be explained easily, and is probably related to the implementation of CUDA drivers. This time is spent at the first call to any CUDA routine during the execution, and we chose to present it here as a part of calculation time, because, as it can be seen, it is a realistic problem that can limit performance, and should be considered when running DP over a large set of GPU cards.

To compare GPU implementation execution times with CPU execution times, we tested the same cases for configurations with 1 to 256 MPI processes, but now without using GPU cards at all. MPI processes were distributed using by-socket discipline, so that inside of a node, processes were bound to CPU cores alternating between 2 sockets. On Figure 11 we report total execution times for all tests done using GPUs for computation, total execution times for all tests done using CPUs for computation, and their ratios (CPU/GPU time).

Here we see that the expected speedup of the DP code when GPUs are used can be up to 3.5 for the smaller test, and up to 16 for the larger one. Actual numbers are even larger when comparing execution times of the single iteration, but since the emphasis here is on the real use-case presentation, execution times for the entire simulation are presented. In both cases, performance ratio peaks for the case with 8 MPI processes.

The first reason for this is that when code is executed only on CPUs, 8 processes saturate the entire node. Because CURIE node has NUMA architecture, by-socket process scheduling is used to ensure equal load-balancing of both memory sockets. But when a node is full, memory bandwidth starts to bottleneck execution performance a bit. Because of that, when code runs on 8 processes, it cannot be expected to be twice as fast as in the case with 4 processes. Since GPUs do not share memory, this situation doesn't show up when everything is calculated on the GPU. The second reason is in the above mentioned issue with CUDA initialization time, which reduces effectiveness of GPUs as the number of processes grows. Because this doesn't affect CPUs, it influences the drop in GPU code speedup when compared to the CPU.

To showcase the performance of our approach with offloading parts of the computation to CPUs, we selected the Argon test case. It was done because the GPU performance gain ratio on that test was lower than in the other case, which means that the performance gap between the GPU and a CPU core is not as wide. In cases like this, it is expected that the CPU will be able to contribute more to the computation, and that load-balancing will be closer to equal. To test the offloading, we constricted test configurations to multiples of 8 MPI processes (to fully saturate each CURIE node). 2 CPU cores were dedicated to controlling one GPU each, and the remaining 6 were used for computation. Weight parameter for load-balancing was selected based on previously presented tests, and was equal to 3.6. Results of offloading performance tests are shown in Figure 12, and it can be seen that using CPU to handle some subsets of transitions can give almost twice the computational performance as when using only GPUs, for suitable test cases. Although, because of the already described issues with DP initialization (IO related), and CUDA runtime initialization, overall performance gains are in



(a) (b)

Fig. 11: Performance of the GPU strategy 2 for DP code compared with the CPU only version, for 1 to 256 GPU cards (each MPI process controlling one GPU), and 1 to 256 CPU cores (each MPI process bound to a CPU core): (a) Argon (small) test case; (b) Hafnium oxide (large) test case.

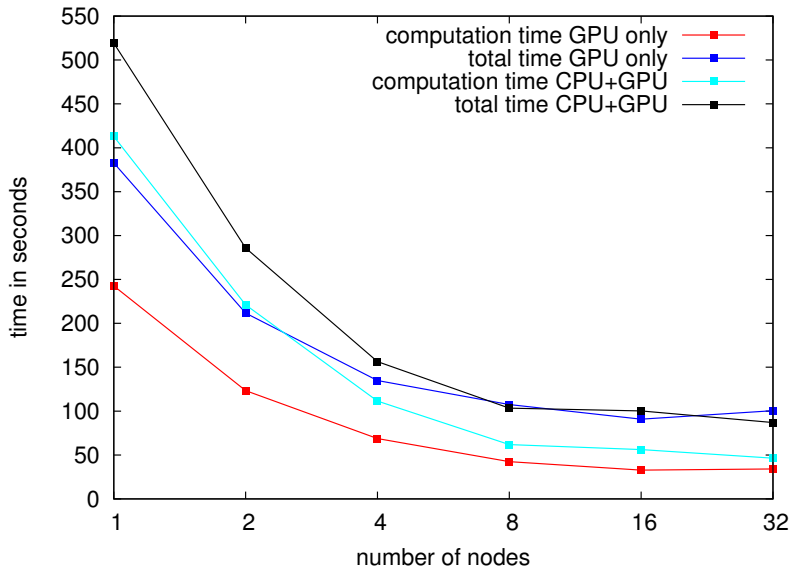


Fig. 12: Performance of the CPU offload modification of the GPU strategy 2 for DP compared to initial GPU implementation, for 1 to 32 CURIE nodes (6 CPU cores + 2 GPU cards per node for offload modification, 2 GPU cards per node for initial GPU implementation), for Argon test case.

fact not that large.

Out-of-core modification of the strategy 2 for DP code GPU implementation was tested on a single GPU card, using slightly modified Hafnium oxide test case. Some input parameters were modified, which increased the size of the matrix A to about 2.8 GB. About 500 MB more was needed for storing of all waveforms and mapping tables on the GPU, so the combined size of the data structures needed slightly exceeded 3 GB. Although this is still less than the available memory on the Tesla M2090 GPU card used for testing, this case was used because it allowed us to compare to the original strategy 2 approach.

The amount of available GPU memory visible to the program was gradually lowered. This was done to show how variable memory capacity available for the A matrix, and for K pairs of X and α temporary vectors used to update A for a batch of transitions can impact the performance. Total number of transitions for this test was 107008. Caches for the waveform structures and mapping tables used only 5 MB of GPU memory for all tested configurations. Table 1 shows execution times for various test cases along with the memory consumption given in total, as well as separately for the part of the A matrix and K pairs of temporary vectors:

Total GPU mem used	**A part mem**	**K**	**X and α mem**	**Execution time (s)**
3.3 GB (in-core, no caching)	2.8 GB	N/A	N/A	6187
4.25 GB	2.8 GB	107008	1.44 GB	6181
1.8 GB	0.4 GB	107008	1.44 GB	6184
0.92 GB	0.2 GB	53504	0.72 GB	6187
0.47 GB	0.2 GB	20000	0.27 GB	6183
0.24 GB	0.2 GB	3000	0.04 GB	6241
0.2 GB	0.2 GB	100	$\sim$ 0 GB	8395

Table 1: Performance of the out-of-core modification of the GPU strategy 2 for DP for modified Hafnium oxide test case.

We have managed to come within 1% of the original performance of the strategy 2 approach using only a fraction of the GPU memory (0.24 GB compared to 3.3 GB needed to solve the problem in-core). It can be seen that the size of the batch K has the most influence on the execution time, because copying of the entire A matrix happens once for each batch of transitions, and copying data to the GPU and back is much more expensive than computation for this algorithm.

5. Conclusion

We have successfully ported the DP code to GPUs, with two different approaches to implementation of the main computational loop. Using the first strategy, we have obtained a speedup of 10 to 14 times compared to the CPU code. Moreover, one can expect even better speedup when the ratio between initialization and computation time is high, since initialization time cannot be made to scale. From a practical point of view, the main drawback is that the user must make sure that all the data can fit at once on GPUs. To do that, a rule of thumb must be applied (see Section 3.1.). Moreover, to get the best possible performance, the user must ensure that the GPU is loaded enough (target 80% memory occupancy on GPU).

When the second strategy is used, speedups of up to 16 times were observed using production-ready inputs, and good scaling for up to 64 GPU cards was achieved. Some modifications to the original strategy were also implemented, allowing us to use CPUs to help with the computation, and to solve problems that cannot fully fit into GPU memory, using out-of-core approach. CPU offload for Argon case showed that on a CURIE node, CPUs can contribute to computation almost as much as GPUs. The use of the out-of-core approach allows solving of problems of much greater sizes, without severely impacting the performance.

Based on the analysis of these two strategies, it is recommended to use strategy 1 when we have inputs that satisfy conditions regarding GPU memory usage, because it contains custom written kernels that employ CUDA shared memory and give very good performance when *cgerc* BLAS operation is performed in a way used in the DP code. On the other hand, strategy 2 gives better relative performance when larger inputs are used, but its performance does not depend on the size of the data as much, and it can be used to efficiently solve very large problems when out-of-core modification is used.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-283493. The work is achieved using the PRACE Research Infrastructure resources [Curie, France].

References

1. V. Olevano, L.Reining, F.Sottile, *The DP code*, `http://etsf.polytechnique.fr/Software/DP`
2. GNU gprof profiler, `http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html`
3. Scalasca performance analysis tool, `http://www.scalasca.org/`
4. Precision and Performance Floating Point and IEEE 754 Compliance for NVIDIA GPUs, Whitehead N., Fit-Florea A., Nvidia, 2011
5. Optimizing Matrix Transpose in CUDA, Ruetsch G., Micikevicius P., Nvidia, 2009
6. CUDA C Best Practice Guide, Nvidia
7. Curie Best Practice Guide, `http://www.prace-ri.eu/Best-Practice-Guide-Curie-HTML`