Master Thesis

Analysis of Hybrid Parallelization Strategies: Simulation of Anderson Localization and Kalman Filter for LHCb Triggers

Author: Jimmy Aguilar Mena
Supervisor: Sebastiano Pilati
Co-supervisor: Ivan Girotto

Trieste, 2015

Acknowledgement

First and foremost I would like to thank God.

I am using this opportunity to express my gratitude to everyone who supported me throughout this Master. I am thankful for their aspiring guidance and invaluably constructive criticism.

This work and the permanence in Trieste were supported by the Abus Salam, International Centre for Theoretical Physics (ICTP).

I am also grateful to the professors of the Master for their assistance, specially to my supervisors Sebastiano Pilati and Ivan Girotto. I could not have imagined having a better mentors.

I would like to special thank to those professors that after this year I can identify more than professors, as friends. Specially Giuseppe Piero Brandino, and Stefano Cozzini. Without their precious support it would not be possible to arrive at this stage of the Master.

Thanks to my family for their support and love. And finally I would like to thank all students that shared with me this year.

“Gratitude is not only the greatest of virtues, but the parent of all others.”

Cicero

Abstract

This thesis presents two experiences of hybrid programming applied to condensed matter and high energy physics. The two projects differ in various aspects, but both of them aim to analyse the benefits of using accelerated hardware to speedup the calculations in current science-research scenarios.

The first project enables massively parallelism in a simulation of the Anderson localisation phenomenon in a disordered quantum system. The code represents a Hamiltonian in momentum space, then it executes a diagonalization of the corresponding matrix using linear algebra libraries, and finally it analyses the energy-levels spacing statistics averaged over several realisations of the disorder.

The implementation combines different parallelization approaches in an hybrid scheme. The averaging over the ensemble of disorder realisations exploits massively parallelism with a master-slave configuration based on both multi-threading and message passing interface (MPI). This framework is designed and implemented to easily interface similar application commonly adopted in scientific research, for example in Monte Carlo simulations. The diagonalization uses multi-core and GPU hardware interfacing with MAGMA, PLASMA or MKL libraries. The access to the libraries is modular to guarantee portability, maintainability and the extension in a near future.

The second project is the development of a Kalman Filter, including the porting on GPU architectures and autovectorization for online LHCb triggers. The developed codes provide information about the viability and advantages for the application of GPU technologies in the first triggering step for Large Hadron Collider beauty experiment (LHCb).

The optimisation introduced on both codes for CPU and GPU delivered a relevant speedup on the Kalman Filter. The two GPU versions in CUDA® and OpenCL™ have similar performances and are adequate to be considered in the upgrade and in the corresponding implementations of the Gaudi framework.

In both projects we implement optimisation techniques in the CPU code. This report presents extensive benchmark analyses of the correctness and of the performances for both projects.

Contents

Chapter 1

Introduction

General-purpose GPU computing (GPGPU) is the use of a graphics processing unit (GPU) together with a central processor unit (CPU) to do general purpose scientific and engineering computing. (SCAI 2012)(NVIDIA Corporation 2015)

This thesis analyses two experiences of hybrid CPU and GPGPU programming applied to condensed matter and high energy physics respectively. Both projects differ in application field, technology and complexity, but have in common the exploitation of GPU hardware and hybrid programming to reduce the processing time.

The projects approach GPU technologies in different ways. The first one interfaces libraries that exploits GPU internally while the other is implemented with GPU's parallel computing programming languages.

1.1 Project 1

The first project is: "Enablement of a massive parallelism to study the Anderson localisation phenomenon in disordered quantum systems". It was developed in collaboration with the Condensed Matter and Statistical (CMSP) Physics department of the Abdus Salam International Centre for Theoretical Physics (ICTP).

Anderson localisation is the complete suppression of wave diffusion due to destructive interference induced by sufficiently strong disorder.

To study this phenomenon we represent the Hamiltonian in momentum space and execute a diagonalization to get the eigenvalues. Then we perform an analysis of energy-levels spacing statistics and the results are averaged over many realisations of the disorder pattern.

The starting point for this project was previously implemented FORTRAN serial version. This original version had several limitations and it interfaced to PLASMA to make the diagonalization.

The main challenge was to implement the different levels of parallelization using hybrid programming techniques(Rabenseifner 2003). The top level of parallelism among different nodes employs distributed memory paradigm with Message Passing Interface (MPI). Withing the nodes each process employs shared memory multiprocessing programming with Open Multi-Processing API (OpenMP).

A master-slave system controls all processes sequentially, assigning a given workload on demand to each slave, and managing the fault tolerance system. This system is based on Portable Operating System Interface (POSIX) threads to make a more efficient use of hardware. This master-slave code is modular, and can be reused in other similar problems as a framework without MPI or multi-threading coding by the user. The implementation provides thread save routines hidden from outside the framework.

A simulation consists in the ensemble of many realisations of the disordered model. It can be solved sequentially or in parallel, accordingly to compilation and execution options. For each realisation of disorder the program executes an initial generation routine, a Fast Fourier Transform (FFT) and a diagonalization of an Hermitian Matrix. The final diagonalization can be performed using Intel Math

Kernel Library (MKL), Parallel Linear Algebra for Scalable Multi-core Architectures Library (PLASMA) or Matrix Algebra on GPU and Multi-core Architectures Library (MAGMA).

The MAGMA library works one or multiple GPUs. This is another level of parallelization in the hybrid programming scheme of the project and it is a good case of study due to the exploitation of GPU technologies through the use of libraries without explicit GPGPU programming.

MKL is the only mandatory dependency, because all the solvers and FFT routines use it. But all the other parameters and dependencies are optional and configurable. The solvers interfaces are also modular to allow not only code reuse, but also the implementation of new solvers with a standard format.

For this first project the general objective was:

> To implement a massive parallel code to study the Anderson localisation phenomenon.

To accomplish this goal we had the following list as specific objectives:

1. To develop a new modularized software in C/C++ to be used as stand-alone binary or included in a framework for exploiting distributed systems. C-style coding is also aimed to better interface modern libraries for high performance linear algebra (i.e., MAGMA, PLASMA, etc. . .)

2. To analyse different linear algebra libraries and functions available that can solve diagonalization and chose the best ones for our purposes.

3. To implement a centralized communication system to control and address the simulation, this should be transparent for the final user.

4. To implement a useful log system and defensive programming strategies to facilitate debugging and latter modifications.

5. To manage the initialisation of the pseudo random-number generators used to generate the disorder realisations in a centralised scheme.

1.2 Project 2

The second project was "Kalman Filter speedup using GPGPU and autovectorization for online LHCb triggers". It was performed as a testing code for the Gaudi framework (LHCb software architecture group 2015) (Clemencic 2015) of the Large Hadron Collider beauty experiment (LHCb) in European Organization for Nuclear Research (CERN).

The Kalman filter is a set of mathematical equations that provides an efficient computational (recursive) means to estimate the state of a process, in a way that minimizes the mean of the squared error. (Welch and Bishop 1995)

A small section of the serial version of Gaudi framework was the starting point for this project. To compare possible results and behaviours we developed four different versions of the Kalman filter. The first one was an optimised version of the original serial code for CPU. Other two versions made for GPU used Compute Unified Device Architecture (CUDA®) and Open Computing Language (OpenCL™). The last version was multi-threaded with OpenMP, it served as a comparison patterns for the parallel OpenCL™ version tested on CPU.

This project exploits GPU hardware through direct GPGPU programming with CUDA® and OpenCL™. This is the main feature that distinguishes it from the first project, despite using the same hardware.

For this second project the general objective was:

> To implement, optimise and benchmark some Kalman Filter codes using different optimisation techniques for CPU and GPU programming.

To accomplish this goal we had the following list as specific objectives:

1. To port the code of interest out of the Gaudi framework guaranteeing the compatibility with the results and consistency.

2. To implement and benchmark an improved version for CPU that can be interfaced easily with a GPU implementation.
3. To implement and benchmark a version of the Kalman Filter using OpenCL$^{\text{TM}}$ and CUDA®, compare performance and accuracy.
4. To improve or correct if possible the initial GPU implementations of the code and compare with CPU versions.

Chapter 2

Enabling massive parallelism to study the Anderson localisation phenomenon

2.1 Materials and Methods

Anderson localization is the complete suppression of wave diffusion due to destructive interference induced by sufficiently strong disorder (Lagendijk, Tiggelen, and Wiersma 2009). It was first discussed in (Anderson 1958) and observed much latter in various physical systems.

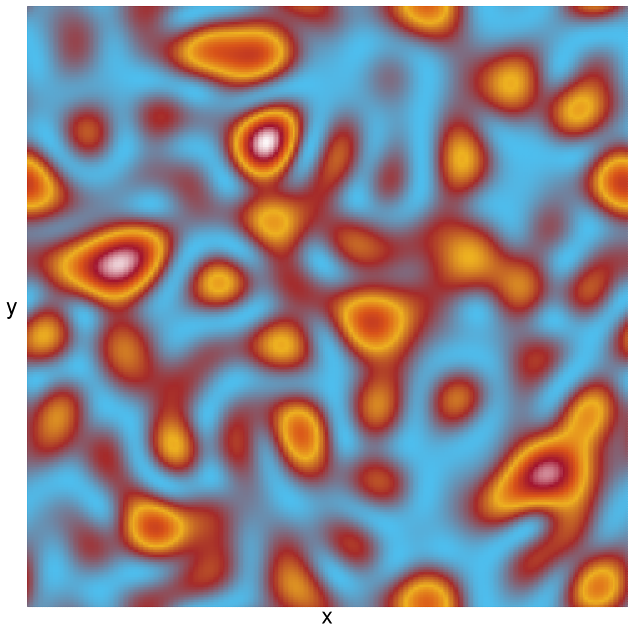


Figure 2.1: Speckle Image

Recently, transverse Anderson localization was realised in randomised optical fibres (Karbasi, Mirr, et al. 2012), paving the way to potential applications in biological and medical imaging (Karbasi, Frazier, et al. 2014).

The experiments performed with ultra-cold atoms are emerging as the ideal experimental setup to study Anderson localization (Aspect and Inguscio 2009; Shapiro 2012). Unlike other condensed-matter systems, atomic gases are not affected by absorption effects, and allow experimentalists to suppress the interactions.

Furthermore experimentalists can create three-dimensional disordered profiles (typically referred to as optical speckle patterns) with tunable intensity. Figure: 2.1 shows the intensity profile of a speckle pattern measured on a plane orthogonal to the beam propagation axis z.

The program we implemented is a tool to study the Anderson localization of atomic gases exposed to three-dimensional optical speckles by analysing the statistics of the energy-level spacing. This method allows us to consider realistic models of the speckle patterns, taking into account the possibly anisotropic correlations which are realised in concrete experimental configurations.

The main goal is to compute the mobility edge E_c of a speckle pattern, that is, the energy threshold that separates the localised from the extended eigenstates. In the Program we consider different models, one of them takes into account the case where two speckle patterns are superimposed, forming interference fringes.

2.1.1 Numerical procedure to determine the energy spectrum and the level-spacing statistics speckle patterns

The basic steps for the simulation are: first, the program uses one of the numerical algorithms described in (Fratini and Pilati 2015) to generate isotropic or anisotropic speckle patterns. Second, it calculates the Hamiltonian for the system; third it obtains the eigenvalues of the Hamiltonian. Finally the program processes those values to determine the statistical distribution of the spacing between consecutive energy levels to locate E_c.

The mathematical details related with the generator algorithms are not relevant in this report, but the other steps are due to their relation with the code.

The equation 2.1 gives the real-space Hamiltonian of a quantum particle moving inside a speckle pattern:

$$\hat{H} = -\frac{\hbar^2}{2m}\Delta + V(\mathbf{r}) \tag{2.1}$$

where:

$\hbar$ reduced Planck's constant.

m particle mass.

$V(\mathbf{r})$ external potential at the position $\mathbf{r}$ corresponding to the intensity of the optical speckle field.

The implemented models consider a box with periodic boundary conditions and linear dimensions L_x, L_y and L_z, in the three directions $\iota = x, y, z$. For numerical calculations it is convenient to represent the Hamiltonian operator in momentum space as a large finite matrix (equation: 2.2)

$$H_{\mathbf{k},\mathbf{k}'} = T_{\mathbf{k},\mathbf{k}'} + V_{\mathbf{k},\mathbf{k}'} \tag{2.2}$$

where:

$T_{\mathbf{k},\mathbf{k}'}$ kinetic energy operator

$V_{\mathbf{k},\mathbf{k}'}$ potential energy operator.

The wavevectors form a discrete three-dimensional grid: $\mathbf{k} = (k_x, k_y, k_z)$, with the three components $k_\iota = \frac{2\pi}{L_\iota} j_\iota$, where $j_\iota = -N_\iota/2, \ldots, N_\iota/2 - 1$, and the number N_ι determines the size of the grid in the ι direction and, hence, the corresponding maximum wavevector.

Therefore, when expanded in the square matrix format, the size of the Hamiltonian matrix $H_{\mathbf{k},\mathbf{k}'}$ is $N_{\text{tot}} \times N_{\text{tot}}$, where $N_{\text{tot}} = N_x N_y N_z$. This is an important factor to consider because it represents the fast Hamiltonian dimension growing relation with respect to the grid size.

In this basis the kinetic energy operator is diagonal: $T_{\mathbf{k},\mathbf{k}'} = -\frac{\hbar^2 k^2}{2m} \delta_{k_x,k'_x} \delta_{k_y,k'_y} \delta_{k_z,k'_z}$, where $\delta_{k_\iota,k'_\iota}$ is the Kronecker delta.

The element $V_{\mathbf{k},\mathbf{k}'}$ of the potential energy matrix can be computed as:

$$V_{\mathbf{k},\mathbf{k}'} = \tilde{v}_{\mathbf{k}'-\mathbf{k}} \tag{2.3}$$

where $\tilde{v}_{\mathbf{k}}$ is the discrete Fourier transform of the speckle pattern $V(\mathbf{r})$.

$$\tilde{v}_{k_x k_y k_z} = N_{\text{tot}}^{-1} \sum_{r_x} \sum_{r_y} \sum_{r_z} v_{r_x r_y r_z} \exp[-i(k_x r_x + k_y r_y + k_z r_z)] \tag{2.4}$$

where $v_{r_x r_y r_z} = V(\mathbf{r} = (r_x, r_y, r_z))$ is the value of the external potential on the N_{tot} nodes of a regular lattice defined by $r_\iota = L_\iota n_\iota / N_\iota$, with $n_\iota = 0, .., N_\iota - 1$.

Wavevectors differences are computed exploiting periodicity in wavevector space. Our code computes the Hamiltonian eigenvalues using a numerical linear algebra library. These represent the energy levels.

In a delocalized chaotic system the distribution of the level spacings $\delta_n = E_{n+1} - E_n$ should correspond to the statistics of random-matrix theory (in particular, to the Gaussian orthogonal ensemble), namely, the Wigner-Dyson distribution. Instead, in the "low-energy regime" the energy levels easily approach each other like independent random variables. This is a consequence of the localised character of the corresponding wave functions. In this regime the levels spacing follow a Poisson distribution.

In order to identify the two statistical distributions and determine the energy threshold E_c which separates them, we compute the ratio of consecutive level spacings:

$$r = \frac{\min\{\delta_n, \delta_{n-1}\}}{\max\{\delta_n, \delta_{n-1}\}} \tag{2.5}$$

The average over disorder realisations is known to be $\langle r \rangle_{WD} \simeq 0.5307$ for the Wigner-Dyson distribution, and $\langle r \rangle_P \simeq 0.38629$ for the Poisson distribution. This statistical parameter was first introduced in the context of many-body localization.

2.1.2 Serial program functions

The program can use two different strategies to collect the eigenvalues. The first option is to consider a large system but this method involves two problems.

On one hand the Hamiltonian size increases as a power of 6 with respect to the grid dimension. Such array requires a large amount of memory that is not available for our researchers. On the other hand the eigenvalues calculation is a $\sim O(n^3)$ problem remaining a $\sim O(npmax^9)$ problem with respect to the grid dimension. This non lineal relations make this strategy inefficient.

The second option is to solve many systems if intermediate size. The only consideration in this case is that all the systems must be different from each other and the results are accumulative contributions to the same statistics. This consideration is easy to solve.

The first step in this project was to implement the main FORTRAN routines porting them to C/C++. We elected this language because all the possible libraries to use are implemented in C. Programming language mixture was undesirable because it makes the code harder to maintain and adds extra complexities. From the beginning we wanted to create a modular code, and C++ Object Oriented Programming (OOP) functionalities were also desired.

We classified the functions in the serial code according to their functionality. First we implemented the essential routines for the calculations. We ported also some other functions used exclusively for development.

Mandatory functions

The initial software had a five steps sequence that we reproduced in the parallel version. At first, we ported this functions as they were. After some tests, we modified them to fit in the parallel version and to prevent potential errors.

The main steps were:

1. Initial disorder system generation.

   The first step is the generation of a disorder system using an initial seed in the random number generator. Using a different seed in the random number generator it is possible to generate several statistically equivalent disorder realisations of disorder with the same distribution of intensities and two points spatial correlation function.

In the FORTRAN code only one function implemented this, it guaranteed the diversity between different systems providing different seeds to the random number generator in every call. A loop provided the sequence in the serial code but a seed range change implied a source code modification.

To distribute the disorder realisations, the serial code implemented a loop, but our parallel version uses MPI to distribute the different seeds among the processes.

The initial function implemented four different models of speckle patterns, we call them: Plain, Shell, Sum2 and Spherical. Details are described in (Fratini and Pilati 2015). A hard coded global variable determined the choice between the models and the function contained several conditional instructions combined with loops. This made the code unreadable and any modification could easily create bugs.

In our implementation we split this routine in four different functions, one for each model. This choice increased the number of lines of code for all the program, but it made the code much more readable and changeable. We used a pointer to functions and a helper function to avoid conditional instructions and to guarantee that new models can be added easily. A better memory management was an extra advantages of this implementation.

For the final parallel version we inserted all this functions in an OOP schema to eliminate data replication and global variables.

2. Fast Fourier Transform.

   The second step after the generation of the speckle pattern is the Fast Fourier Transform function (FFT) in three dimensions. This step is needed to represent the Hamiltonian operator in momentum space as explained in equation 2.2 section: 2.1.1 page: 6. The original code used dfti MKL interface to FORTRAN Fast Fourier transform. Each function call needs an error checking after it and the code became harder to read.

   Our code implements this function using the same dfti interface as in the original code. The impact in performance for this function is negligible comparing with the diagonalization. The only modification included for this function was to port the code and interface with MKL from FORTRAN to C, and the addition of some macros for error checking.

   The macros were to make the code more uniform and readable considering that all the routines would have the same behaviour in case of failure. We inserted extra error checking strategies due to the complexity of the MKL interface.

3. Hamiltonian calculation.

   The Hamiltonian has to be defined before the diagonalization. It is the biggest array to be allocated in memory and its eigenvalues are the data we need to perform the statistical analysis, which allows us to extract E_c.

   Our implementation for this function follows exactly the same steps than the original one. We added memory bound checks and extra error checking associated with the memory management functions like malloc and free. We guaranteed to free all the unneeded arrays after this function to provide extra space for the diagonalization.

   The Hamiltonian is an array of complex numbers. The C99 standard added a complex type to C programming language; C++ provides a template type for the same purpose. The libraries made in C should use internally the standard complex type from C99. Any modification in the standard will results in equivalent modifications to the library. We took this into account to make our code more portable choosing the C99 complex type instead of the C++ template.

4. Some eigensolver algorithms.

   The FORTRAN code solved the eigenvalues problem calling the PLASMA zheev routine. This function computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A.

   In our code we considered also some other functions and libraries combinations. Subsection 2.1.3 contains more details about solver algorithms.

5. Processing results.

   The last step in the simplest serial code is data processing.

   The program constructs an accumulative histograms with the eigenvalues from all the systems. The process functions is called in the same loop after the other functions.

   In the parallel version the processing function should receive the results from all the remote processes, because the final results are additive. We isolated this function from the previous four to take advantage of the master-slave system in the parallel version. Subsection 2.1.5 explains the master-slave system details.

Figure 2.2 illustrates the flowcharts for the serial implementations. The FORTRAN code's flowchart (Figure 2.2a) shows all the steps in subsection: 2.1.2. We grouped all the sequential steps using only a "Calculate" step to simplify the C++ diagram (Figure 2.2b).

Our code's design has a higher complexity because such a version already includes error handling mechanisms. An extra function provides the seeds so that we can use different random seed sequences. The exit condition is when any function returns -1 to provide a uniform method to check it.

We considered to provide an option to specify the interesting interval for eigenvalues. This feature implies that some diagonalization can return zero values in the interesting region of the eigenspace. In this situation the code checks the existence of results before any action to avoid useless MPI calls or blocking thread save executions.

Optional functions

After porting the functions needed to make calculations, we ported some functions needed by developers. This functions were in the original program and are useful to detect errors in some of the previous steps. The original FORTRAN code needed to be modified and recompiled to call any special functions during development-testing time.

Those functions are used to make tests only during development, like debug options. The researcher only makes serial tests therefor the implementations of this functions is consequent with that. Out implementation includes new command line options to avoid extra code modifications and involuntary bugs creation. Subsection 2.1.8 provides more details about command line options.

In this category of testing routines we included only two functions called: writespeckle and correlation.

1. writespeckle creates text files to plot the speckle and detect errors in the initial disorder system generation.

2. correlation uses statistical criteria to check the consistency of the speckle model. This statistics needs iteration over very big data and the function can be slow. We improved this function in the final version using OpenMP.

2.1.3 Eigensolver algorithms

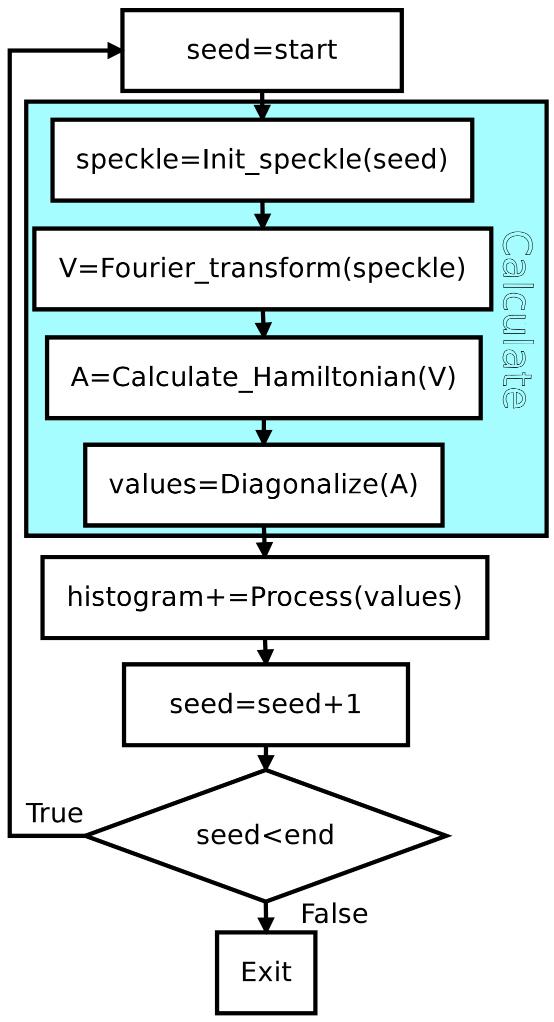
At this point we face the most time consuming part of the code. The program needed to calculate the eigenvalues and optionally eigenvectors of a complex Hermitian matrix. The possible functions that solves this problem are named with the pattern zheev* in all the libraries that use the Linear Algebra PACKage (LAPACK) library naming scheme. Each library adds some extra suffixes.

The two standard functions zheev and zheevd can solve this kind of problems. This functions are available in almost any Linear Algebra Library, usually with small variations. A third function zheevr is frequently available in some libraries.

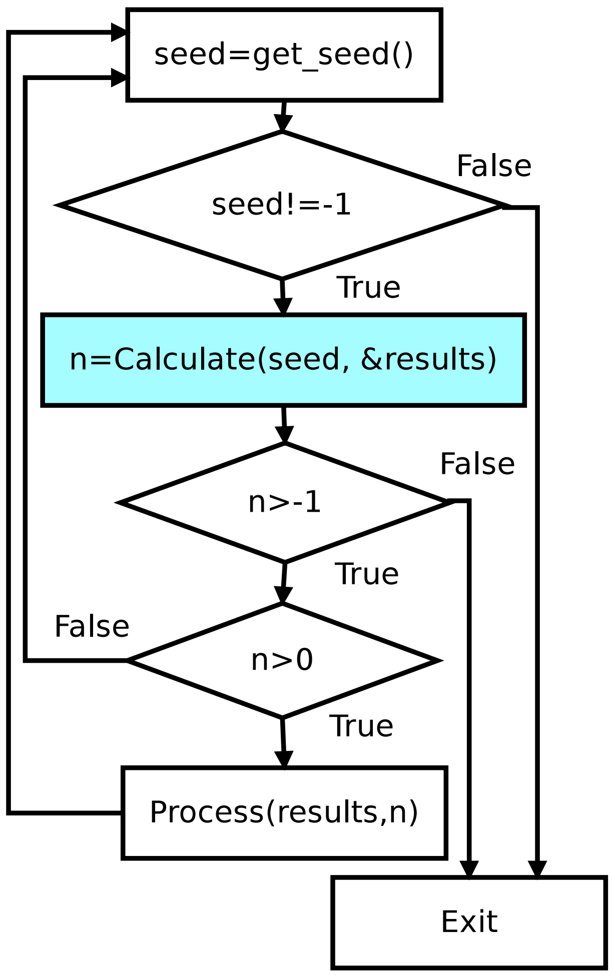
This three functions differs in functionality or the algorithm used.

zheev Computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A. In this eigensolver the matrix is preliminary reduced to tridiagonal form.

There are two different versions for this eigensolver:


seed=start
speckle=Init_speckle(seed)
V=Fourier_transform(speckle)
A=Calculate_Hamiltonian(V)
values=Diagonalize(A)
Calculate
histogram+=Process(values)
seed=seed+1
True
seed<end
False
Exit

(a) Flowchart serial Fortran code



(b) Workchart serial C++ code

Figure 2.2: Serial flowchart of the original FORTRAN (2.2a) and the ported C++ (2.2b) implementations. The routines inside the "calculation" function are the same in both implementations, but we grouped them in the C++ representation to simplify the diagram

One-stage eigensolver The one-stage tridiagonalisation is the approach used in the LAPACK eigensolvers. The modern versions are partially improved including an hybrid algorithm of the tridiagonal eigensolver. The tridiagonalisation's performances are limited, since 50% of the operations are level 2 BLAS operations. (Solca et al. 2012)

Two-stage eigensolver This version solves the problem of the limited performance of the tridiagonalisation step in the previous routine. In the first step the matrix is reduced to a band matrix, while in the second step the band matrix is reduced to tridiagonal one using a bulge chasing technique. (Haidar, Ltaief, and Dongarra 2011).

For the second step it uses a bulge chasing algorithm, very similar to the previous one. They differ by using a column-wise elimination, which allows to have better locality for computing or applying the orthogonal matrix resulting from this phase. The drawback of this approach lies by the eigenvectors backtransformation, since the tridiagonalisation is performed in two steps, the backtransformation requires also two steps.

zheevd The main difference with zheev is that zheevd uses a divide-and-conquer (D&C) eigenvalue algorithm.

This algorithm was introduced in (Cupper 1981) the approach can be expressed in three phases:

1. Partition the tridiagonal matrix T into several sub-problems in a recursive manner.
2. compute the eigen decomposition of each sub-problem, which are defined by a rank-one modification of a diagonal matrix.
3. Merge the sub-problems and proceed to the next level with a bottom-up fashion.

(Haidar, Ltaief, and Dongarra 2012)

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits

zheevr This routine is a one-stage or two-stage eigensolver like zheev but includes the possibility to compute only a fraction of the eigenspace.

The main reason to use this function is the reduction of computational time when only a small portion of eigenvectors is needed.

2.1.4 Solver interface

The most important aspects in our code were the solvers and the parallelization framework. The first challenge we got was designing a common interface for all the libraries.

We checked the state of the art and MKL, PLASMA, ScaLAPACK and MAGMA are extensively used in linear algebras high performance applications. All this libraries contain variants of the eigensolvers described in section 2.1.3

From now the main limitation for the program is the systems size and number of this realisations. In our parallel version we considered that every individual problem to solve will fit in memory on one node. And the parallel schema will try to solve simultaneously as many systems as possible. For this reason the use of ScaLAPACK was neglected.

In the initial benchmarks we obtained better results with PLASMA and MAGMA. We also implemented a solver interface to MKL because it is a dependency anyway. This solver can be useful for developers and to make tests in personal computers.

To provide a common interface with all the libraries, we took advantage of OOP techniques. We created an abstract class **solver** that exposes only the needed functions to the main program. All the solver objects will derive from this class and provides a common interface to the program. The solver classes manage all the different types names for complex numbers in the different libraries exposing always the same types to the client side.

In general, all the solvers provide the same options and internally they call the right functions according con the initialisation options. The user can not change the options ones the solver is constructed. The serial code only calculated eigenvalues, but in our version we included an option to calculate also eigenvectors. So far we have not implemented any functionality for eigenvectors, but the solver should provide the option for a near future.

The solvers are independent of each other and the rest of the program. This modular design allows us not only to reuse any solver in any other similar application, but also to select the desired solver taking into account the available hardware and software. In our implementation the user can chose the solvers in compilation or run times.

The function needs more time if also eigenvectors are computed and in the centralised schema the amount of data to transmit between processes is also larger. We gave some optional solutions to this problems within the interfaces but the developer should consider them to add new functionalities with eigenvectors.

In the final code we implemented four solver classes. One for MKL, another for PLASMA and other two for MAGMA. All the solvers initialise the libraries in the constructor and provide an **int solver::solve(double*)** routine to calculate the eigenvalues. An internal array, allocated in the constructor, stores the results. All the routines have error prevention strategies. In case of failure, they print an error with information about the error location and return -1. We added a debug compilation option to the solvers that produces a verbose output to stderr. The output gives information about called functions and returned values. This strategies are consistent with the framework's design. (See subsection 2.1.6)

Interface to MKL Library

MKL library includes a one-stage variant of zheev. The divide and conquer (zheevd) and the range defined (zheevr) variants are also included.

The MKL library is usefull in computers without a PLASMA or MAGMA installations. In computers with a low number of cores MKL performance is better than PLASMA's and it is the simplest option for testing small systems.

As the FFT function uses MKL, the library became the only real dependency for our code and its eigensolver can help for development and testing purposes.

Interface to PLASMA Library

The name PLASMA is an acronym for Parallel Linear Algebra Software for Multi-core Architectures. It has been designed to be efficient on homogeneous multicore processors and multi-socket systems of multicore processors. (Dongarra, Kurzak, and Langou 2010)

PLASMA version 1.0 was released in November 2008 as a prototype software providing proof-of-concept implementation of a linear equations solver based on LU factorization, SPD linear equations solver based on Cholesky factorization and least squares problem solver based on QR and LQ factorizations, with support for real arithmetic in double precision only.

Plasma implements many important software engineering practices including: thread safety, support for Make and CMake build systems, extensive comments in the source code using the Doxygen system, support for multiple Unix OSes, as well as Microsoft Windows through a thin OS interaction layer, clear software stack built from standard components, such as BLAS, CBLAS, LAPACK and LAPACK C Wrapper.

zheev was a one-stage eigensolver before PLASMA 2.5, but the newest versions uses the two-stage algorithm. The tests with two different versions of plasma gave the same performance with one-stage and two-stage routines. But is mostly important most important that the same interface worked in both cases.

Plasma assumes the full running node allocation and only one plasma process running in each node. The default behaviour is to allocate the full node when there is not a thread number in the plasma initialisation routine, and allocate always starting on the first core. Plasma provides two more specific functions to manage the threads affinity, but we need to manage this using some environment information. A more detailed explication is in section: 2.1.6 pag: 17.

Interface to MAGMA Library

The MAGMA project aims to develop a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures, starting with current "Multicore+GPU" systems. It is based on the idea that optimal software solutions will have to hybridise to address the complex challenges of the emerging hybrid environments. (ICL Team 2015)

This library combines the strengths of different algorithms within a single framework. The algorithms and frameworks for hybrid many-core and GPU systems can enable applications that exploit the power offered by each of the hybrid components.

Many routines have the same base names and the same arguments as LAPACK, but sometimes MAGMA needs large workspaces or some additional arguments in order to implement an efficient algorithm. These are generally hybrid CPU/GPU algorithms and the suffix indicates if the matrices are initially allocated in the host or the device, not where the computation is done.(MAGMA development team 2015)

none hybrid CPU/GPU routine where the matrix is initially in CPU host memory.

_m hybrid CPU/multiple-GPU routine where the matrix is initially in CPU host memory.

_gpu hybrid CPU/GPU routine where the matrix is initially in GPU device memory.

_mgpu hybrid CPU/multiple-GPU routine where the matrix is distributed across multiple GPUs' device memories.

Magma provides the 1-stage and 2-stage eigensolvers as different functions. The 2 stage routine is available only for host storage and the divide and conquer function was merged with the range defined eigensolver adding the suffix “x”. In total there are 12 different variants of zheev* functions.

The users can implement many solvers with any combination of functions. We considered that the Hamiltonian is usually very big and does not fit in the GPU Memory. For this reason our solver do not includes GPU storage functions with the _gpu or _mgpu suffixes.

The one-stage solver included in our program uses a combination of 4 functions. The solver deals with the device number internally to call the routine for a single or multiple GPUs, It also considers if the range of interesting values is all the eigenspace or a section to call zheevd or zheevr.

The two-stage routine in Magma needs 2 times the memory than the one-stage algorithm. Our program considers the possibility of hardware improvement in a near future and includes a solver that uses a two-stages routines with similar characteristics that the other.

2.1.5 Master-slave thread based system

One of the simplest parallel programming paradigm is the “master-slave” approach. A master process generates many sub-problems, which are fired off to be executed by the slave. The only interaction between the master and slave computations is that the master starts the slave computation, and the slave computation returns the result to the master. There are no significant dependencies among the slave computations. (Sahni and Vairaktarakis 1996)

The traditional approach for this kind of problems implements the master-slave system on MPI processes. Traditionally, MPI programs used a fixed number of homogeneous processes but dynamic and heterogeneous resources are characteristic in modern architectures.

The master-slave strategy perfectly fits the ensemble averaging technique employed in our study of Anderson localization. This kind of statistical approach is employed in most theoretical studies of disordered systems in science and engineering. Therefore we expect that our implementation of the master-slave scheme will also find application in future studies.

In our application, the master only sends one integer value (the seed) to each slave and receives the results after the slaves ends the calculations. These integer values (the seeds) are used by the slaves to initialise the sequences of pseudo random-numbers that are employed in the algorithm that generates the disorder pattern. Therefore, the slaves can generate uniquely identified disorder realisations. The ensemble of the seeds is archived by the master providing to the users the advantage of a checkpoint/restart system, which is a fundamental requirement for scientists to access world-class grants on large-scale HPC systems.

The operations needed to process the results are simple and serial and the arrays storing the histograms for the compute of $< r >$ are negligible comparing with the Hamiltonian.

On the other hand the main solvers we implemented for intensive calculations used MAGMA or PLASMA. The first one requires some GPU cards in the node and the second gives a better performance with a bigger number of threads.

In this simulations only one problem fits in the memory node and the user needs to set only one process per node in the mpirun call. At that time the common way to use our program must be allocating one process in each node and to implement multi-threading programming inside the processes.

Considering all this we concluded that it is a waste of resources to use the conventional master-slave scheme with one node for the master process only. Even more if we use nodes with multiple GPUs or heaps cores.

Our proposal was to use a master slave system where the master will be a thread attached to one of the slaves. Using this schema all the nodes will be maximised, but the one running the master thread will receive some extra work. This decompensation should be taken into account because the master workflow impacts all the system. To solve this we modified the maximum number of threads in the slave that runs in the same node to provide some free cores for the master thread.

Thread Safety in MPI

Traditionally MPI implementations do not have highly tuned support for multithreaded MPI communication. In fact, many implementations do not even support thread safety. The issue of efficiently supporting multithreaded MPI communication has received only limited attention in the literature. (Balaji et al. 2008)

For performance reasons, MPI defines four monotonic "levels" of thread safety:

1. MPI_THREAD_SINGLE Each process has a single thread of execution.
2. MPI_THREAD_FUNNELED A process may be multithreaded, but only the thread that initialised MPI may make MPI calls.
3. MPI_THREAD_SERIALIZED A process may be multithreaded, but only one thread at a time may make MPI calls.
4. MPI_THREAD_MULTIPLE A process may be multithreaded, and multiple threads may simultaneously call MPI functions with a few restrictions.

MPI provides a function, MPI_Init_thread, by which the user can indicate the level of thread support desired, and the implementation will return the level supported. A portable program that does not call MPI_Init_thread should assume that only MPI_THREAD_SINGLE is supported.

The level of thread support that can be provided by MPI_INIT_THREAD will depend on the implementation and the MPI compilation options. It may also depend on information provided by the user before the program started to execute. The MPI_THREAD_MULTIPLE level is the most comfortable for multi-threading programming, but although it is implemented in most of the libraries, it is deactivated due to its performance impact.

The default thread support in most MPI implementations is MPI_THREAD_SERIALIZED or lower.

First Master-slave thread based implementation

Our first test for a master-slave system assumed the higher thread support level MPI_THREAD_MULTIPLE. The master was attached to the first process and all the slaves implementation were exactly the same. A simple system for tag values guaranteed the right communication between all the processes even between the master and the slave with the same process rank. No thread save strategies were applied to the seeds provider function or to process results because MPI blocking communication guaranteed a save master execution.

The problem in this implementation was that we needed to compile the MPI library, by default no MPI_THREAD_MULTIPLE was available anywhere. This implementation was simpler, but affected the portability.

Second Master-slave thread based implementation

In the second implementation we assumed a MPI_THREAD_FUNNELED thread support. The code was fully portable, but there were other problems:

1. The slave in the first process was executed as the attached thread because with this level only the thread that initialised MPI can make MPI calls.

   The first slave had a completely different workflow with respect to the other slaves. The code size increased 4 times, because the master and the remote slaves shared a common main function and code replication became hard to maintain.

2. The master and the slave in the same process communicated without MPI, but calling the same functions to get seeds and process results.

   This is a good choice because there are not MPI communications within the same process. The local array can be processed directly without an extra copy to a temporal buffer. The only special consideration is that shared routines needs thread save implementations to prevent data corruption.

Third Master-slave thread based implementation

The final implementation assumed MPI_THREAD_SERIALIZED thread support and it is a good middle point between the first and the second implementations. The master thread is executed as in the first implementation, but the local slave calls the functions directly without MPI communication. As the array with the results in the slave and the process function are in the same process, no local data copy are needed.

All the slaves had the same code with some simple conditional instructions, but the code size was closer to the first version. We needed to keep the thread save routines like in the second version for the same reasons.

To make thread save routines we used the simplest mutual exclusion strategies with mutex objects. This method serialises the access to shared data using mechanisms that ensure only one thread reads or writes to the shared data at any time. Incorporation of mutual exclusion needs to be well thought out, since improper usage can lead to side-effects like deadlocks, livelocks and resource starvation.

The C++11 standardises support for multithreaded programming, and we considered to use it to manage the threads easily. But for this kinds of operations the standard defines the threads and mutex as std members.

Considering this from another angle, the pthread.h header is extensively used and compatible with older compilers, it is useful in C and C++ and almost standard in Linux. We used the older version for portability and backward compatibility.

All the implementations use the on-demand master slave method where the master thread knows how many jobs will be assigned from the beginning. This master thread only takes care of the remote processes and when no remote processes are running the thread ends.

In the master thread some internal values contain the number of remote processes running, the last seed sent to each remote process, the number of systems already solved in each process and some other useful information. When all the planed jobs are submitted to the slaves, the master sends -1 to any new requester slave, the end condition is when the master has send -1 to all the remote processes.

2.1.6 Implementing a reusable module for the master-slave system

Figure:2.3 shows the simplified flowchart for the framework. Only one thread in each process makes MPI calls. The remote processes have a simpler scheme as the one on the right part of the picture. The attached thread in process zero collects all the results sent by the remote processes and executes the analysis. The slave thread in process zero does not use the master thread to process the results because such process have direct access to the same functions than the thread.

The arguments to implement the model with threads are applicable to any high performance simulation. Several applications can benefit from the model implemented in our project. For this reason we decided to implement the master-slave thread bases system like a simple reusable framework. In this way we can provide support to many applications and the improves will benefit more potential applications.

Current implementation takes advantage of the object oriented programming techniques and in the near future we will include generic programming techniques. We implemented the framework as an abstract class with three pure virtual functions. The functions should be implemented by the user following simple rules and using serial code. The final user do not need to know about MPI or multithreaded programming. The implementation turns the user defined function in thread save routines internally if needed. The three functions to implement are: the seed sequence generator, the calculate function and the function to process the results.

This three steps are generic and the rules to implement them are in the project's documentation.

We designed the module to support conditional compilation. The master-slave framework can compile as serial code without MPI library. This is useful to develop and test code in personal computers before testing it in parallel environments. This functionality is provided using macros, and can be combined with other compilation options.

It is different to run the serial code or the parallel version with only one process. The attached master thread is always created in the parallel version but it ends immediately if there is a single process in the communicator because it detects only remote processes. The first process modifies the maximum

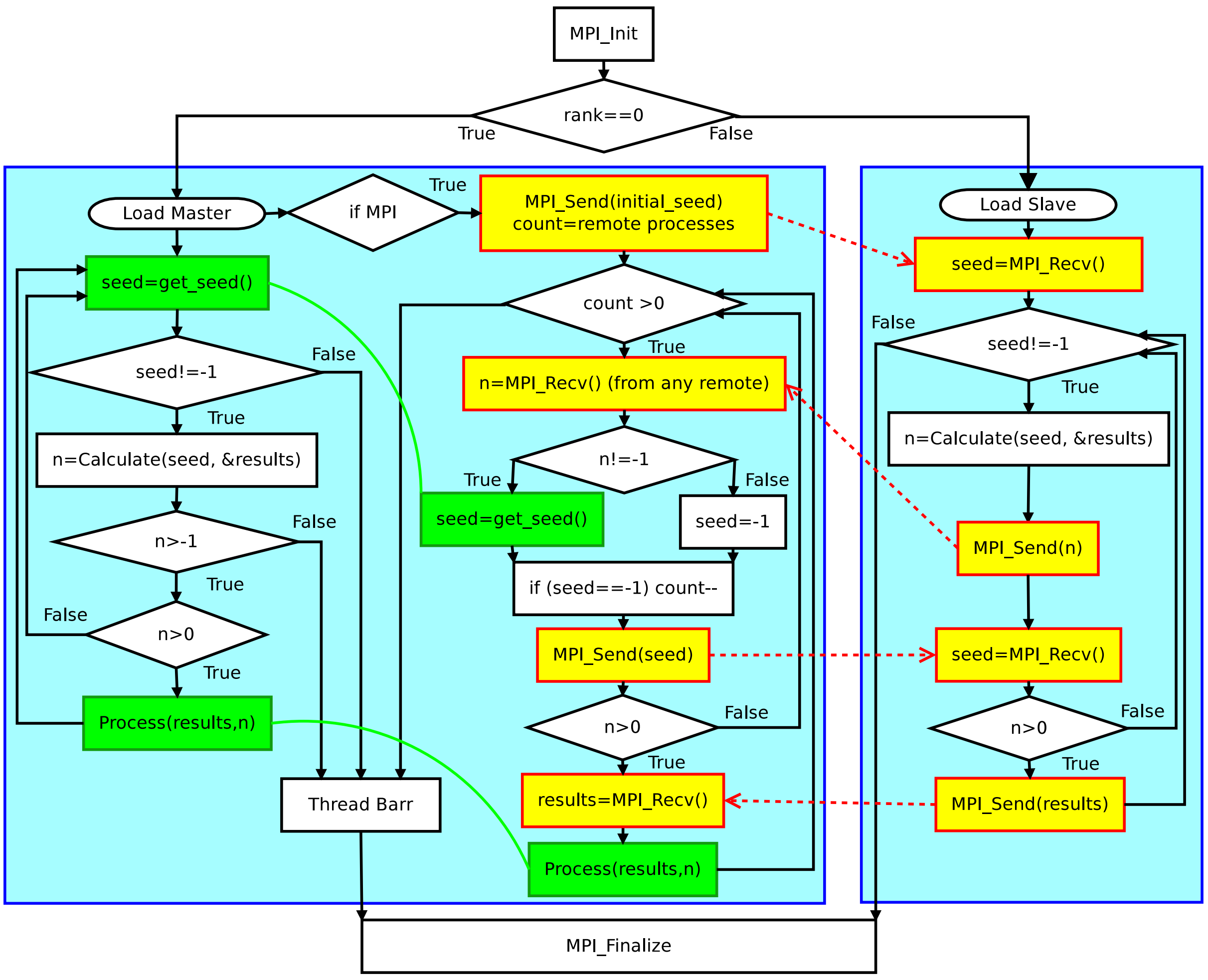


Figure 2.3: Thread based framework's flowchart.

- Blue squared sections represent a shared memory region (a process).
- The green lines represent that the functions joined are the same, but they are called from different execution threads in the same process. This routines are thread save.
- Yellow regions represent MPI calls.
- Red arrows are the messages between processes.

number of threads available for the libraries in the parallel version to provide capacity to the attached thread and the code contains useless conditional instructions, calls to MPI functions and thread save strategies. On the other hand, we optimised the serial code deleting all the unneeded calls and dependencies.

We made some tests to the serial and parallel version of the program and we could not get any differences in performance. Maybe the differences would be observed in applications with calculation time smaller than processing time; but our framework is not designed to be used in those conditions.

We added a logfile system to debug errors and to check the simulation status. The log file system provides information about the overall simulation and specific errors details. User defined routines can insert information in the logfile using a function similar to printf provided by the framework.

The framework includes a self consistent error managing system. All functions should return -1 in case of error to properly integrate in the framework. We included special macros to guarantee and facilitate an effective error checking and readability.

The process prints a message with useful information when the framework detects an error in a remote process. After that it sends -1 to the master thread to inform the problem, the master thread adds more information about the failure to the logfile and sends -1 back. When a process receives -1 it jumps automatically to the MPI finalise barrier and it starts spinning. The master thread sends -1 after detecting an error or to inform that the last seed was reached.

This workflow guarantees that a single failure does not kill all the program and prevents corrupted data insertion in results.

Affinity assignation

Processor affinity, or CPU pinning enables the binding and unbinding of a process or a thread to a CPU or a range of CPUs. So that the process or thread will execute only on the designated CPU or CPUs rather than any CPU. This is like a modification of the native central queue scheduling algorithm in a symmetric multiprocessing operating system. Each item in the queue has a tag indicating its king processor. At the time of resource allocation, each task is allocated to its kin processor in preference to others.

A thread's core affinity mask determines the set of cores on which it is eligible to run on a multiprocessor system. Setting the CPU affinity mask can be used to obtain performance benefits. By dedicating one CPU to a particular thread, it is possible to ensure maximum execution speed for that thread.

Restricting a thread to run on a single CPU also avoids the performance cost caused by the cache invalidation that occurs when a thread ceases to execute on one CPU and then recommences execution on a different CPU. The affinity mask is a per-thread attribute that can be adjusted independently for each of the threads in a thread group.

External libraries can assume a full node allocation, some libraries provides functionalities to set the number of threads to use, but default behaviour is not standard. When the application creates an external thread, conflicts are possible with the libraries and the cores affinity. This conflicts are more dangerous when running many processes in the same node. Plasma, for example, provides a method to set the number of threads but assumes the full node allocation. Magma have an affinity macro but the some control functions are unprovided.

We implemented some environment checks using POSIX for hardware and MPI for the processes information. The framework collects this information in its constructor and estimates the maximum number of threads in each process and the index of the first core to use. This information is useful for the interfaces to the solver classes and to decide the core where the slave thread should run in the first process.

Our master thread's performance can impact all the processes in our system therefore we assigned a dedicated core to it. The default behaviour in our implementation is to run the master thread in the last core available for the first process and assign the other $n-1$ cores to the libraries when affinity functionalities are available.

The current implementation does not handle multiple processes in the same node with a MAGMA solver. The reason for this is that analogous options are not available to manage the core affinity in MAGMA and GPU affinity should be considered.

2.1.7 Object oriented thread programming

Multi-thread programming using the pthread.h header is limited by the absence of object orientated implementation. This is expected considering that the header is useful not only in C++ but also with C. Thread creation routine receives a pointer to a function as an argument. Such a function can not be an object member because those are not defined in the standard. But in the framework all functions are a member of the base abstract class to provide a simpler transparent interface to the user. It is possible to create a thread with an object's static function, but static functions have not access to the “this” pointer and can modify only other static members.

We solved this using two functions: first a static function that receives a pointer to an object as the framework and second a private helper function. A non static member function creates the thread passing its this pointer as argument to the static one throw the pthread interface. At that time the static function calls the private one using the received “this” pointer and the dereference operator (->).

The real execution for the thread is in the second function and the static one works just as a helper to the interface.

2.1.8 Command line options

The implementation of command line options was a time consuming feature, but very useful for the benchmarks and the code usage. The framework sets any command line option aside the constructor. It is a framework design choice, but the user can implement this feature, or any other parameter modification function. The last configuration step and memory allocation is the first step before starting the calculations inside the framework. This step is independent from the constructor.

To prevent the extra code modification we implemented two different input options: a command line and an input file. The input file is a mandatory command line option and the last argument to be provided.

The command line options have the same names than the input file. The command line value is used if command line and input file provide different values for the same variable. We provided some extra options for the serial or the parallel versions that can be useful to debug or testing.

The provided options allows to call an optional function (see subsection: 2.1.2 page: 9) instead of the normal calculation sequence, or to build a dynamic graph with gnuplot for a small number of runs. We provided a full help command with all the possible options.

To implement the command line options parser we employ the GNU getopt routines. Some code is reused to parse also the input file making the program completely consistent and the new insertions easier to implement.

2.1.9 Functionalities in development process

The already implemented parallel program contains all the functionalities available in the original FORTRAN version. We are still improving it and automatising operations that used to be hand made. Some functionalities in development process deserve a comment.

Restart a previous calculation

The statistic tests are performed only after the program ends. The researcher starts a new calculation if needed and the results used to be combined manually. Now the program contains a restart option that allows to restart a previous calculation from the previous state. To use this function the user should provide a restart parameter with the output file from a previous run and provide a different seeds range. This option is an experimental feature and needs to be improved.

Documentation

Doxygen is the standard tool for generating documentation from annotated C++ sources. It also supports other popular programming languages such as C, Objective-C and *C#*.

We are developing a full documentation using Doxygen to facilitate the use of the program for new users. The completed parts in the documentation is already available using bitbucket hosting with useful information about the program in general and some results for comparing performance and accuracy.

The installation and the different alternative combinations are commented too and we provide some links to get the dependencies and their installation instructions. The class hierarchy is provided with comments about the most important variables and functions.

There is a special section for the master-slave framework. There the functionalities and usage instructions are explained. In case the master slave framework receives enough interest in the future, we are planing to give it a special support as an independent library.

Installation

A proper build and installation system is essential to use any software extensively. CMake is a cross-platform, open-source build system. It is used to control the software compilation process using simple platform and compiler independent configuration files. CMake is available almost any server because most of the projects are migrating from autotools. We considered to use CMake as the project's build system, but there is not any standard way to find the dependencies we need.

The portability is a priority, for this reason we made all the modules optional. The build system should create only the executable files to avoid the risk of new dependencies or complexities in different architectures. Simplicity is always beneficial to get usability in a project and we are careful with build automatising.

From now we decided to use an elaborated Makefile until a more complex build system could be implemented. The absolute paths for the libraries should be declared as environment variables with specific names. This is the simplest option at the moment to perform several tests with different compilers or libraries versions.

The Makefile checks the variables and compilers in the path and makes the best possible choice. It will provide always the serial version and will add the support for all the libraries it could find with the environment variables. It will create the parallel executable if there is some MPI compiler.

2.2 Results and Discussion

2.2.1 Ported code validation

When porting a code to a different programming language the first step is to check that the new version produces the same results than the old one. This validation should be made before anything else. Since we use a different random number generator, the comparison has to consider the statistical fluctuations. We could not check some code sections until the end.

Our validation required two steps. First we corrected some errors in the interfaces to the libraries and we started the validation for the mandatory routines after(see section: 2.1.2 page: 7). An initial benchmarks to the libraries functions provided the information about the useful functions and expected performance for final code.

The tests were made in Ulisse using the regular queue for PLASMA or MKL and the gpu queue when using MAGMA's. Regular nodes contain 20 Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz cores and at least 40 GB of ram memory. Each node has two GPUs NVidia Tesla K20m with 4 GB of memory.

Validating the interfaces with the libraries

The first step was to validate the interfaces with the libraries because we needed a solver class to test the rest of the ported code. Such test code will be included with the program because it is useful to develop new interfaces in the future.

The validation for the interfaces was quick because all the libraries produced the same results when working. We automatised the test with a bash script and a small C++ program. The program calculates the eigenvalues of several random Hermitian matrices using all the interfaces and saves the results in text files. The bash script compares all the files and prints any difference. We added some extra error handling strategies to the interfaces considering the errors we found in this section.

The only problems we faced were with memory allocations in MAGMA's two-stage routine. We reported the problem to the developers and they corrected it. Therefore we made all the benchmarks using the last MAGMA release, version 1.7.0.

Validating the ported functions

In this test both optional functions (see section: 2.1.2 page: 9) helped to validate the initialisation routines and correct all the bugs. Being save that all the interfaces to the libraries produce the same results we tested the accuracy using PLASMA interface only. We made this choice because it is faster to access some nodes in the regular queue than in the gpu one and the system's dimension is relatively small.

The system dimension and parameters were the same used to obtain some previous results with the serial code. Our software needed less that 2 hours to solve 20000 systems $npmax = 16$ using 16 nodes and one process per node.[1]

Figure: 2.4 illustrates the ensemble-averaged adjacent-gap ratio $\langle r \rangle$ as a function of the energy E/E_σ for an isotropic speckle pattern of intensity $V_0 = E_\sigma$ and two different system sizes. E_σ is the correlation energy. The horizontal line is the result for the Wigner-Dyson distribution. (Fratini and Pilati 2015)

Figure: 2.5 shows the relative difference between results from our code and the original one using three values for the grid. The difference is compatible with the standard deviation. The calculations for each grid size needed around 24 hours with the FORTRAN serial code. The difference between both data is lower than 0.5% and it is associated with statistical fluctuations. In the lower energy region the errors are bigger because there is a lower values accumulation and consequently worst statistics.

2.2.2 Scalability on CPU

Scalability refers to the capability of a system to increase its total output when resources are added. Testing the scalability in our code provides information about if it is suitably efficient and practical when applied to a large number of participating nodes.

[1] Please remember that the diagonalised matrix is the system's Hamiltonian in complex representation with dimension $npmax^3$

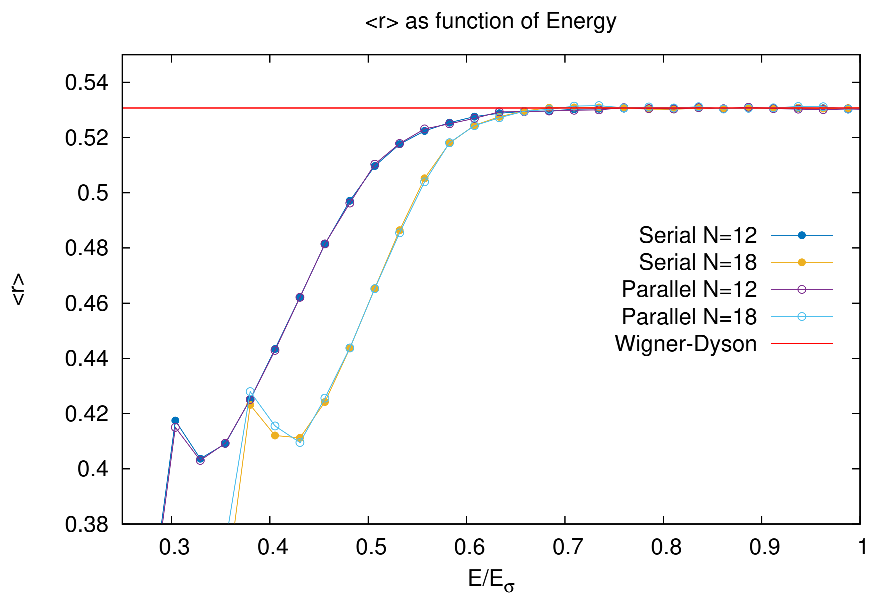


Figure 2.4: Ensemble-averaged adjacent-gap ratio $\langle r \rangle$ obtained with the original serial implementation and our parallel version. E_σ is the correlation energy.

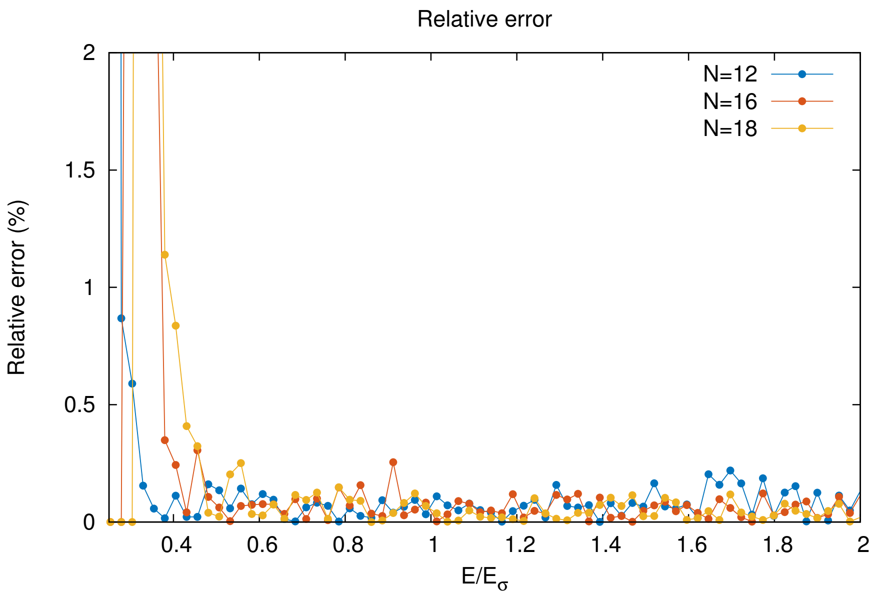


Figure 2.5: Percent of relative difference in results from our code and from the original FORTRAN code for grid size *npmax* = 12, 16 and 18

The scalability graph represents the calculation time with serial code divided by the time using parallel code. We calculated this value for 1,2,4,8,16 and 32 nodes to illustrate the system scalability respect to the number of nodes. All the nodes have the same number of cores therefore the local parallelism is considered constant in this benchmark.

To build the scalability graph the program solved a system as big as memory allowed. The limitation to solve bigger systems is the memory size available in individual nodes. With the hardware available in Ulisse we can solve systems with *npmax* = 36 using PLASMA or *npmax* = 34 with MAGMA's one-stage routine.

The two-stage routine in MAGMA needs more memory and we can allocate a system no bigger than *npmax* = 33. The formulation in our program only works for even dimensional systems, hence the bigger system we can solve with all the interfaces is *npmax* = 32.

To construct the scalability graph we need to solve the same problem using different number of nodes, starting with one. Plasma needs around 20 minutes to calculate the eigenvalues in a system with this dimension but walltime in the regular queue in Ulisse is 12 hours. With this in mind we estimated that around 32 systems could be solved within the walltime in serial code. We ran five times each configuration and we report the mean to make a precise measurement considering statistical fluctuations. To estimate the error we used the error of the mean as the standard deviation divided by $sqrt(N-1)$ where N is the total number of measurements.

We could not make this test using any GPU interface because in Ulisse we can get only two nodes per process in the GPU queue.

Scalability test for affinity implementation

Section: 2.1.6 page: 17 explains how we managed the affinity for PLASMA's interface and the extra master thread considerations. The first implementation did not consider the affinity, this consideration emerged from other tests and after the corrections we benchmarked the code again. We expected to find differences between the results when running multiple processes per node, but single process per node configurations also showed performance differences.

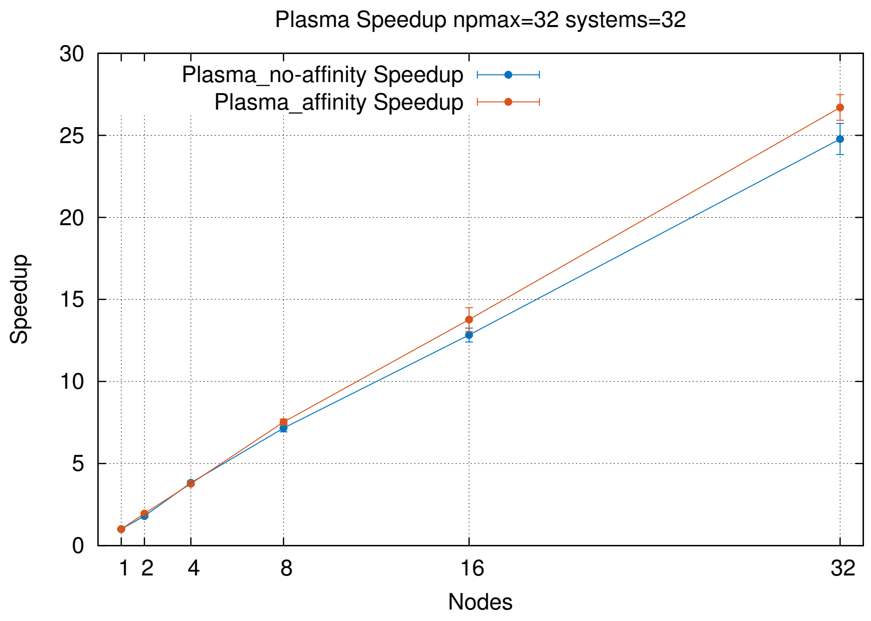


Figure 2.6: Scalability using Plasma with 1,2,4,8,16 and 32 nodes. The affinity line is always over the initial non-affinity implementation.

Figure 2.6 shows the scalability graph using PLASMA. Both lines shows an almost lineal scalability from 2 to 32 nodes, but the implementation that considers the affinity provides better values. The performance difference can be estimated throw the scalability, but we should consider that serial code was improved and the times fraction hides some information. The relative difference in time between both serial code versions is close to 10%.

This difference is due to a change in performance for PLASMA with affinity thread because that is the only difference between the two codes. The master thread affinity does not influence the serial version because that version never creates a master thread.

Table 2.1 shows the influence of affinity in calculation time. The times for the affinity code are always smaller in serial and parallel code. The reported values are the mean over 10 tests for each point in the table. We calculated the relative differences respect to the first implementation (non-affinity) using the formula $\sigma = \frac{|t_1 - t_2|}{t_1} \times 100$.

The parallel version evidences smaller relative time changes respect to serial code. This difference increases very slowly with the number of nodes and they are completely independent to PLASMA's number of threads. This can be a consequence of a more efficient master thread due to affinity assignation.

Nodes	affinity time (s)	no-affinity time (s)	Relative difference (%)
1	28916 ± 336	31332 ± 636	7.71 ± 2.94
2	14813 ± 180	17457 ± 116	15.14 ± 1.59
4	7667 ± 83	8196 ± 70	6.45 ± 1.82
8	3841 ± 49	4377 ± 52	12.24 ± 2.16
16	2099 ± 86	2442 ± 31	14.05 ± 4.62
32	1083 ± 19	1265 ± 23	14.35 ± 3.05

Table 2.1: Mean execution time and relative difference to solve 32 systems sized *npmax* = 32 using PLASMA with and without core affinity. The relative difference between times uses the non-affinity time as pattern time in the formula.

PLASMA vs MKL performance

An interesting test was the benchmark between MKL and PLASMA. The MKL version 11.1 is available in Ulisse, unfortunately such a version is old and the results using newer releases could be significantly different. MKL library is always a reference to solve this kind of problems and a comparison with PLASMA have sense considering that both have multi-thread support and will run on the same hardware.

We tried the same benchmarks we ran with PLASMA, but the calculation time typically exceeds the walltime in the regular queue. Because of this we could collect results only for 16 and 32 nodes. Although we compiled MKL solver with parallel support, runtime checks proved that regardless the number of threads available, zheev routine uses only 1 or 2 cores at the time.

Nodes	Plasma time (s)	MKL time (s)
16	2099	29693
32	1083	14879

Table 2.2: Mean execution time to solve 32 systems sized *npmax* = 32 using PLASMA and MKL.

Table 2.2 confirms the low parallelization in the zheev routine of MKL. The times using MKL are in the order of

Small Systems and Multiple Processes per Node

During the parallel solution of linear algebra problems with PLASMA, the scalability respect to the number of threads is not lineal. PLASMA's scalability with respect to the number of threads is not lineal and excessive parallelism is useless for small systems. Consequently the time to solve two systems sequentially in one process with 20 threads is bigger than in two process with ten threads each.

Considering this from another angle, an excessive number of processes in the master-slave scheme can impact performance. The master consumes time to process the results forcing any remote sending process to wait. The probability of this undesirable situation increases proportionally to the number of remote processes and inversely to the individual processing time.

We should consider to run multiple processes per node when many small systems fit in the memory of the individual nodes. As the number of cores in a node is fixed, increasing the number of processes per node implies a reduction of the cores available per process.

Plasma manages internally the affinity and MPI binding command line options did not produce the expected behaviour. To run multiple processes in the same node we managed the affinity internally not only for plasma, but also for the master thread. A brief description of our strategy is in section: 2.1.6 pag: 17.

In this benchmark we ran the parallel program on four nodes with 20 cores each using 1,2,4 and 5 processes per node. We selected a different number of total systems to solve according to their size to get better precision with smaller times. To build the graph we normalised the times respect to the number of solved systems in each tests for each size.

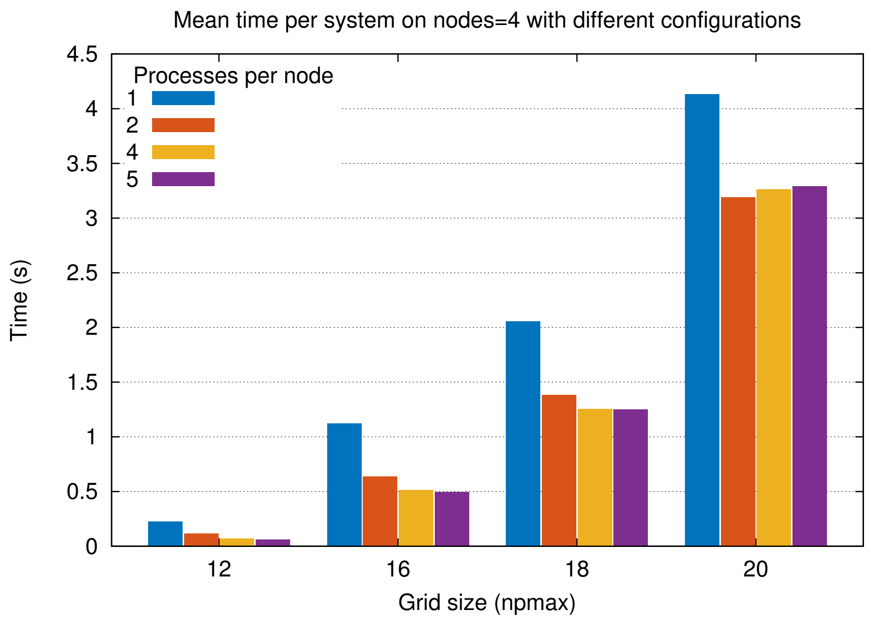


Figure 2.7: Mean solution time per system for different configurations with multiple processes per node for small systems.

Figure 2.7 shows significant differences when running two processes per node respect to one. In small systems the difference is around 50%. This relative difference decreases with system size because in bigger systems the scalability is closer to linear in PLASMA. The data to process in the master thread is a vector of the same dimension than the Hamiltonian (($npmax^3$)) and processing time for results grows proportionally. This can explain why performance is worst for bigger system sizes when we run four processes per node.

A reduction in the total number of threads per process increases the individual time to solve one system, but we should also consider the affinity impact if the processes are allocated correctly in the processors.

2.2.3 Performance with GPUs

We tested the system's behaviour using MAGMA solvers interface for one and two stage routines. All the benchmarks ran on Ulisse using the gpu queue and this limited the maximum number of nodes in use because that queue only allows two nodes per process.

MAGMA provides different function prototypes for one-stage and two-stages routines. Each function has a single and a multiple GPU's version with slightly different interfaces. We ran the benchmarks with the same system size used in subsection 2.2.2 to reuse PLASMA's results in the graph.

Solving this we found errors in the MAGMA one-stage routines for single and multiple GPUs. The error "info" variable returned a successful value but the results values were unset. The function returned after about a minute.

On the other hand the two-stage routines were executing for longer time than one-stage version and produced some results. But those did not match with the values from PLASMA. We tested memory allocation in the workspace in the host side and other possible errors in our code unsuccessfully.

This error occurred only for large systems and not in all the routines. In the Ulisse's GPU nodes the errors started for *npmax* = 26 in the one-stage version with one gpu. The others routines started to fail with slightly larger systems.

We informed the developer about this issues but we set *npmax* = 24 in our benchmarks because this is the bigger even value we could find where all the functions run correctly.

Figure 2.8 shows performance difference for one-stage and two-stage routines on single and multiple GPUs. The current implementation can not handle multiple processes per node with a MAGMA solver because of the reasons in subsection: 2.1.6 page: 17.

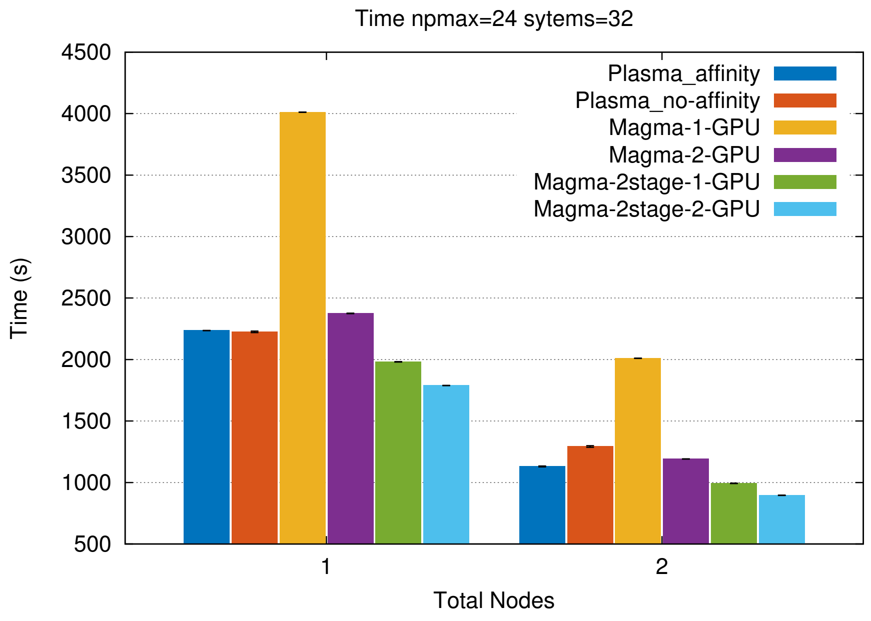


Figure 2.8: Total calculation time for 32 systems with *npmax* = 24 using MAGMA one-stage and two-stage routines on one and two nodes. The black lines on the top of each block are the error bars. The statistic error for this system sizes is negligible and it was calculated using the same procedure than in previous section: 2.2.2

The one-stage function is slower than two-stage routine as expected according to subsection: 2.1.3 page: 9. The PLASMA eigensolver uses a two-stage routine and this makes it faster than the one-stage routine in MAGMA regardless the hardware. The speedup is 1.7x when using two GPUs instead of one, but even with this hardware it is 5% slower than PLASMA.

The two stage routines are two times faster than the one-stage with the same number of GPUs. The main limitation for this function is the amount of memory it needs in the workspace. On the other hand the speedup when using two GPUs is just 1.1x.

In the tests with *npmax* = 26 we found that the two-stage routine was even slower with two GPUs than one. This is a problem in the MAGMA release that falls back some computation on CPU and need to be fixed by hand in the source code.

Comparing with PLASMA the two-stage routine in MAGMA is 10% faster with one GPU and 20% with two GPUs. This difference will increase when using pinned memory (Boyer 2015) because according to the developers it is always faster for these eigensolvers. But we did not implement this functionality due to the expected sizes of the matrices.

Scalability

We calculated the scalability respect to the total number of nodes. In this case it is perfectly lineal using MAGMA as a consequence of the small system size used in this test. We added some testing code in the framework to check the blocked time of processes and with this system size was always zero because the results are processed very fast in the master thread.

We made 20 tests for each function and the error was smaller than 1% for all the reported times. The statistic error for this system sizes is negligible and it was calculated using the same procedure than in previous section: 2.2.2.

Chapter 3

Kalman Filter improves using GPGPU and autovectorization for online LHCb triggers

3.1 Materials and Methods

3.1.1 Experiment description

The LHCb experiment is at one of the four points around CERN's Large Hadron Collider where beams of protons are smashed together, producing an array of different particles. It is a single arm forward spectrometer at the LHC collider, designed to do precision studies of beauty and charm decays, among others.

This experiment aims to record the decay of particles containing b and anti-b quarks, collectively known as "B mesons". The detector design guarantees to filter out these particles and the products of their decay. These decays feature a displaced vertex as signature. To isolate these interesting decay requires software triggers.

B mesons formed by the colliding proton beams stay close to the line of the beam pipe. Other LHC experiments surround the entire collision point with layers of sub-detectors, like an onion, but the LHCb detector stretches for 20 metres along the beam pipe, with its sub-detectors stacked behind each other like books on a shelf.

Each sub-detectors specialises in measuring a different characteristic of the particles produced by colliding protons. Collectively, the detector's components gather information about the identity, trajectory, momentum and energy of each particle generated, and it can single out individual particles from the billions that spray out from the collision point. (LHCb Experiment 2008)

Trigger description

The detector registers around ten million proton collisions per second. Due to limited storage capacity a trigger system filters the best of them reducing the frequency from 30 MHz to 15 kHz in two steps.

1. The first step uses information taken in real-time from the detector–specifically from the Vertex Locator (VELO), the calorimeter, and the muon system.

   This trigger level reduces the frequency from 30 MHz to 1 MHz. The first level trigger takes decision in 250 ns.

2. A very large amount of data still remains after the first level trigger.

This trigger level processes 35 GB of data every second into 2000 computers underground at the LHCb site. This second level trigger has more time to take the decision than the previous one and reduces the frequency from 1 MHz to 15 kHz

An upgrade to all the LHCb experiment will take place in the next years. Different software and hardware modification are tested constantly to improve performance. This tests and simulations take place inside the framework “Gaudi”.

Gaudi Framework

The Gaudi project is an open project for providing the necessary interfaces and services for building High Energy Physics HEP experiment frameworks in the domain of event data processing applications. The framework is experiment independent. It was developed for LHCb, but now it has been adopted and extended by ATLAS and several other experiments including GLAST and HARP. (Clemencic 2015)

Gaudi is being updated to use modern hardware more efficiently, while developing performance and compatibility tests. The upgrades include not only the usage of features like SSE4.2 and AVX instruction sets on the CPU, but also running on different architectures.

Some development versions of Gaudi, or projects implemented on it, are testing the advantages of using GPU hardware. Different routines and simulation projects can already run on this hardware but most of them use CUDA® as programming language. CUDA® is an excellent platform but it limits the software to NVIDIA hardware.

Considering the possibility of using hardware from other manufacturers like ATI’s GPU cards, or other parallel architectures like Intel Xeon Phi, several new tests are implemented with OpenCL™ to provide portability among hardware. (subsection: 3.1.6 page: 30 contains a brief description of CUDA® and OpenCL™)

3.1.2 Particle reconstruction on triggers.

A hit is the information collected by a detector. Each hit represents a particle position detected by the coordinates (x,y,z). Many particles are detected at the same time and the detectors bring only hits information.

A track is a set of hits that describes a path of a particle inside the detectors. With a track it is possible to determine several particle’s properties like mass, charge, energy, etc..

All the tracks associated to the same interaction are grouped in “events”. The common events contain around 400 tracks but the limit is 2500.

The main action in the first trigger step is the track reconstruction from hits. The triggers use different techniques for this task but they always implement some Kalman Filters for this purpose. This filters consumes about $10 - 25\%$ time in the first trigger step.

The more complex routines are in the second triggering step. More sophistication inside these triggers implies more sensitivity, but will require more time for the data flow processing. Improving the first trigger step will provide some extra time to the second one for making more complex analysis and get better resolution.

Different processing systems implemented on Gaudi simulates the triggering systems. This simulations look for the better options to be implemented in the next upgrade. Our goal in this project was to implement some Kalman filter prototypes for the track reconstruction routines for different architectures.

3.1.3 Kalman Filter

The Kalman Filter (Kalman 1960) is a linear quadratic estimation. This algorithm uses a series of measurements containing statistical noise and other inaccuracies, and produces estimates of unknown variables that tend to be more precise than those based on a single measurement alone.

This filter addresses the general problem of trying to estimate the state $x \in \Re^n$ of a discrete-time controlled process that is governed by the linear stochastic difference equation:

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1} \tag{3.1}$$

where:

A**:** the state transition matrix which applies the effect of each system state parameter at position x_{k-1} on the system state at position x_k.

B**:** the control input matrix which applies the effect of each control input parameter in the vector u.

u_k**:** the vector containing any control inputs.

w_k**:** the vector containing the process noise terms for each parameter in the state vector.

With a measurement $z \in \Re^m$ that is:

$$z_k = Hx_k + v_k \tag{3.2}$$

where:

H**:** the transformation matrix that maps the state vector parameters into the measurement domain

The random variables w_k and v_k represent the process and measurement noise (respectively). They are assumed to be independent (of each other) and with normal probability distributions.

$$\begin{aligned} p(w) &\sim N(0, Q) \\ p(v) &\sim N(0, R) \end{aligned} \tag{3.3}$$

In practice, the process noise covariance Q and measurement noise covariance R matrices might change with each time step or measurement, however here we assume they are constant. (Welch and Bishop 1995)

3.1.4 Detector and Software Characteristics

The VELO registers tracks in 42 silicon detector elements positioned close to the point where protons collide. This elements provide hits information with some noise in the (x, y) axis. Each detector element is in a well know position along the z axis, this is the value of z used for the hits the element detects. The error in z is negligible and the detectors are in consecutive positions each other. Due to the VELO geometry each track will have not more than 24 hits. (Experiment 2008)

VeloPixel is a track reconstruction code implemented on Gaudi to simulate the trigger system for the VELO. The current version of this code implements a PrPixel class that reconstructs tracks using a Kalman Filter. We elected this filter for our study case.

In this filter the measured values are $(x; y)$ and the positions z are x_k in equation: 3.1. This is a two dimensional Kalman filter where each axis is filtered independently but using the same equation.

In the simulations Gaudi generates random hits emulating some tracks and noise grouped in events. This algorithm only provides hits information as the detector do, but keeps the information needed to check the reconstruction effectiveness at the end.

The Gaudi framework relies heavily on Object Oriented Programming. The tracks are objects and the events are items containing arrays of tracks. The implementation of the PrTrack class consists basically on an std::vector of objects that contains the data of the position (x,y,z) and the errors (tx, ty) for the hit.

The code before the filter makes a partial reconstruction associating the hits belonging to the same track. The tracks store the hits in vectors because this distribution is similar to the hardware output. But it keeps this distribution for the triggering process.

The implementation in use calls a function two times for each hit in a loop over all the hits in the track. All the filter is embedded inside Gaudi and the operations are single precision.

3.1.5 Code isolation

Gaudi compilation is a slow process and the tests take a long time because all the hits generation and track reconstruction takes much longer than the filter. For this reason the first task was the isolation of the Kalman filter code to use it outside Gaudi.

First we modified the original code to create files with the filter's inputs and outputs. This is called serialisation.

The serialisation requires different detail levels because the default serialisation of objects available in C++ does not save the information inside arrays or vectors. We used text files for both outputs to make easiest accuracy tests at the end.

We reproduced the OOP based implementation, cloning the needed dependencies or deleting the unneeded ones. The methods were copied or re-implemented (cloned) to delete external useless dependencies.

To import the data from the Gaudi's output file we added an extra function, and the same serialisation than before for the output. This was the only modification that this code received. As both codes ran on the same hardware the accuracy should be the same. We tested the accuracy with diff.

All the benchmarks to the other implemented versions compare with this serial code because it is the original one.

3.1.6 Comments about CUDA® and OpenCL™

CUDA® and OpenCL™ are the standard languages for GPGPU programming. While there are more solutions, these have the most potential and are extensively used.

CUDA®

CUDA® is a parallel computing platform and API model created by NVIDIA. It allows software developers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing GPGPU. The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements.

The CUDA® platform is designed to work with programming languages such as C, C++ and FORTRAN. This accessibility makes it easier for specialists in parallel programming to utilise GPU resources, as opposed to previous API solutions like Direct3D and OpenGL. CUDA supports programming frameworks such as OpenACC and OpenCL.

Being developed by NVIDIA, CUDA® restricts its domain to NVIDIA GPU cards. This provides a highly optimised code for this architecture, but sacrifices portability among different architectures.

OpenCL™

OpenCL™ is a framework for writing programs that execute across heterogeneous platforms like CPUs, GPUs, digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators. OpenCL specifies a language based on C99 for programming and APIs to control the platform and execute programs on the compute devices. The platform provides a standard interface for parallel computing using task-based and data-based parallelism. It is an open standard maintained by the non-profit technology consortium Khronos Group.

A key feature of OpenCL is portability but the programmer is not able to directly use hardware-specific technologies. It is possible to run any OpenCL kernel on any implementation. However, performance of the kernel is not necessarily portable across platforms.

Existing implementations proved competitiveness when kernel code is properly tuned. Some studied has suggested possible solutions to the performance portability problem.(Fang, Varbanescu, and Sips 2011) yielding "acceptable levels of performance" in experimental linear algebra kernels.(Du et al. 2012)

Portability of entire applications containing multiple kernels with differing behaviours is available in literature.(Dolbeau, Bodin, and Verdiere 2013)

3.1.7 Implemented code

We implemented 10 different versions for the code: first a serial improved version for CPU, 4 versions for GPU in CUDA®, 4 in OpenCL™ and a last version with OpenMP for benchmark purposes only. The GPU code implementation had extra testing versions but this work reports only the most important ones.

Contiguous memory implementation on CPU

The first implementation we made was an improved CPU serial version for the code. This was also useful for the later GPU implementations.

The main performance hit in the original code was the storage of data using arrays of structures. The excessive use of getter/setter methods that can also have a significant impact because these functions some times can not be inlined by the compiler.

The generated code also depends of the optimisation level and encapsulation. Irrespective of the compiler used, autovectorization can not give good results, since the storage layout does not match the hardware capabilities. The code uses arrays of structures (AOS) on several different levels, then there are many internal pointer potentially aliasing the same memory location. This prevents the compiler to make autovectorization or reordering strategies.

This first implementation substituted the arrays of structures (AOS) with structures of arrays (SOA). The getter/setter functions were avoided declaring friendship relations between the most related objects. All the hits were rearranged in memory to be contiguous. This way the autovectorization and cache usage are improved simultaneously.

The AOS will benefit the GPU implementation because it reduces the number of copy operations to the device. This copy reduction is important taking into account the huge latency in copy operations host-device.

The new storage enables the code to process many events at the same time on the CPU or GPU. This makes sense from the GPGPU point of view if the data fit in the memory, because that will reduce kernel's load and initialisation operation, copy operations host-device and the associated latency.

Hybrid implementation

Far from the optimal previous conception we should consider that the data incoming from the detectors is ordered in portions more similar to AOS than to SOA. This is unavoidable from the hardware point of view and any reshape should be made in the software's side. Such a reshape can be expensive in terms of performance, but it is an option we should consider due to its simplicity.

Mixing both serial codes we implemented a mix serial algorithm to estimate the viability of data reshaping in the final code. This version imports the data to AOS as in the original code, but moves the information to the contiguous memory shape (SOA) before the filter execution.

The time counter in this case considers not only the filter application time, but also the time needed to reshape the data from AOS to SOA.

GPU implementations

In the GPU implementations we used the contiguous memory implementation to provide better parallel memory access to the threads.

In the serial code there are four nested loops iterating over the runs, the events, the tracks and the hits respectively. The filter runs the inner loop over hits for both axis $(x; y)$ simultaneously but filtering independently.

The Kalman filter is an iterative algorithm but every step depends of the previous one according to equation: 3.1. This kind of dependent algorithm is not parallelizable in this level.

To implement the filter for GPUs it is possible to parallelize up to the track level, but the inner loop (iteration over hits) should remain serial. As all the tracks and events are independent each other and our code processes the tracks as individual elements.

This ignores the event which the tracks belong to, and it is a design choice to maximise the number of threads in each kernel call. Some grouping procedure will enforce multiple kernel calls and will reduce the occupancy affecting performance. We implemented all the GPU codes using both CUDA® and OpenCL™.

1. Our first implementation used one thread to filter each track. The hits were copied to local memory with a loop and saved the results at the end of the execution.[1] The kernel in this implementation

[1] In this report we will use the CUDA names convention. Website (Advanced Micro Devices, Inc 2014) contains more information of OpenCL equivalences.

runs two loops: first to copy and a last one to run the filter over its track.

The loop followed the same sequence than serial code and applied the filter to both axis at the time in the same loop.

2. The second option was similar to the first one but we extended the level of parallelism. Considering that the hits in a track were dependent for the filter, but the axis in the hits were not, we added an extra dimension to the problem.

   In this version each thread will filter only the values of one axis x or y instead of both. This doubled the number of threads but reduced the amount of data to copy in each thread, the operations per thread and the global memory access to save results at the end.

3. The third and forth versions were the same than above but eliminating the copy loop with a pointer to global memory. As the copy to local memory used a loop the total amount of memory accesses were the same, but some calculations could overlap with memory read operations.

   This version needs frequent access to global memory during all the kernel execution instead of only on the beginning and the end. Considering this from another angle no local vectors are allocated inside the kernels and the local memory is free.

Parallel CPU implementation

This version is not useful for application due to the conditions the code should be executed. We made a modification in the contiguous memory implementation code for CPU to provide OpenMP parallelization. In the development tests the performance difference between versions was very big and evident from the beginning. As expected, the better performance was the OpenMP parallelization of the outer loop and we used it for the other benchmarks.

This was an academic experiment to compare the parallel performance for code generated with OpenMP and with OpenCL. As OpenCL executes dynamic compilation the benchmarks only measured the filter execution time over the events.

3.1.8 Other details

Code functionalities

In all the implementations we tried to follow the standards for the host code. CUDA programs were implemented with C++ but the OpenCL™ ones followed the C99 standard. All the programs used simple precision arithmetic because that is the kind of precision of the data incoming from the detectors.

To detect bugs we added a debug compilation option to the build system. This debug option activates not only the compiler debug flags but also extra macros to provide more verbose information. The extra macros exists in the kernels and the C/C++ codes and they provide information about the program status, the hardware and some time information.

Time measurement method for benchmarks

To measure the time we tried two methods: Inside and outside the kernel. CUDA® and OpenCL™ provides completely different methods to measure execution time on the GPU side.

OpenCL measuring method had more impact in performance than the kernel we implemented. The method needs debug compilation for the kernel and extra objects creation. CUDA®, on the other hand, provides a simpler method for this task, but some kernel modifications are needed for precise measurements. The compatibility between both methods is undocumented, and for this reason we rejected this results.

Another option is to use CPU timing measurement. For long kernels, like the ones we implemented, this method provides results with enough accuracy. The main advantage of this method is that it uses only standard C and all the programs can use it with few modifications.

OpenCL providers

Unlike CUDA, different companies provide OpenCL implementations. We tried our code with libraries from NVIDIA, ATI and Intel. Most of the code ran in all the architectures but we found some issues kernel functions like printf. We observed that some implementations produced warning messages in code sections than the others compiled without any problem.

Our initial benchmarks with a downloaded code showed that in similar hardware the ATI's library is about 10% slower than NVIDIA's. On the other hand ATI implementation provides some extra functionalities that the others lack.

In this project we only used the OpenCL library provided by NVIDIA. We chose this library to execute all the tests on the same hardware because CUDA code is limited to NVIDIA cards.

3.2 Results and Discussion

In the benchmarks for GPU implementations we should consider that in practical systems the data will be already on the device memory. The time needed to copy from the host to the device should be ignored by the timers. To guarantee the copy time exclusion we inserted the time functions inside the code around the filter loops or kernel.

Even when our serial code maintains the tracks grouped in events and the events in runs, our graphs show processing time associated to the total number of tracks because the filters run over tracks. Therefore the number of tracks also means total number of Kalman Filters applied.

The only error to consider in this measurement is that the tracks could have from 3 to 24 hits. For a big number of tracks this is not important because more than 90% of tracks contains from 3 to 5 hits and less than 1% more than 15.

The time needed to apply the Kalman Filter to a known number of tracks is a good measurement of the efficiency of the code. The number of tracks in each event is limited, and on GPUs it is better to process as many tracks as possible. To process all the events at the time more sense on the GPU makes and it is more efficient, but in the serial code it does not matter.

For the CPU benchmarks we ran the codes on a node with two Intel®Xeon®CPU E5-2670 v2 @ 2.50GHz. This CPUs contain 10 multithreaded cores for a total amount of 40 threads per node.

For the GPU codes we used an nVidia®GeForce GTX 690 graphics card on the same node.

3.2.1 Results for serial code

The first benchmarks we did were to the serial code versions without any parallelization. The objective of this is to measure the effectiveness of the proposed changes to allow autovectorization and contiguous memory access.

For serial code we built two scalability graphs. In this point we benchmarked both contiguous memory implementations that applies the filter on SOA to compare with the initial implementation that applies the filter on AOS. This allows us to get some idea of the benefits of this implementation and the cost of data reshaping in terms of performance.

First we measured the time to apply the filters respecting to the total number of tracks in our clone of the original code with AOS for 1 to 300 events. This time values were used as pattern to get the scalability graph, the other times were compared with this.

Then we followed the same procedure with the version that applies the filter on the contiguous memory implementation explained in section: 3.1.7 page: 31. This implementation is expected to have the better performance due to autovectorization and and contiguous memory access.

Finally we compared the version that moves the data from AOS to SOA before applying the filter described in subsection: 3.1.7 page: 31.

Figure: 3.1 shows the time graph vs the total number of tracks for the three serial algorithms. The initial implementation (array of structs) holds the higher times in all the range of our study and increasing the time presents jumps every 40000 tracks. On the other hand the time in the contiguous memory algorithm (struct of arrays) grows lineally with the number of tracks but with a less vertical slope than the other.

The algorithm that includes the internal data reshape shows a similar behaviour than the original implementation, but not worst. This algorithm needs to access the non contiguous data to reshape it before the filter. From this follows that most of the difference in performance between the first and second algorithms is associated with the memory access, but autovectorization and compiler optimisations also influences, but less.

Figure: 3.2 contains the speedup of contiguous memory algorithm respect to the original code. For the pure contiguous memory version the speedup is close to 2 with a precision around to 10%.

On the other hand, the implementation with data reshape have a speedup close to one in all the range, but always a higher up to 20% for some values. However, although the speedup values for this algorithm are not surprisingly high, it proved viability not being worst than the original one.

According with this graphs the only exceptions to the previous affirmation is for 3 values with a small number of tracks. But only few points shows this behaviour and the difference is small to consider it relevant. Moreover this small number of tracks is very unlikely in real conditions.

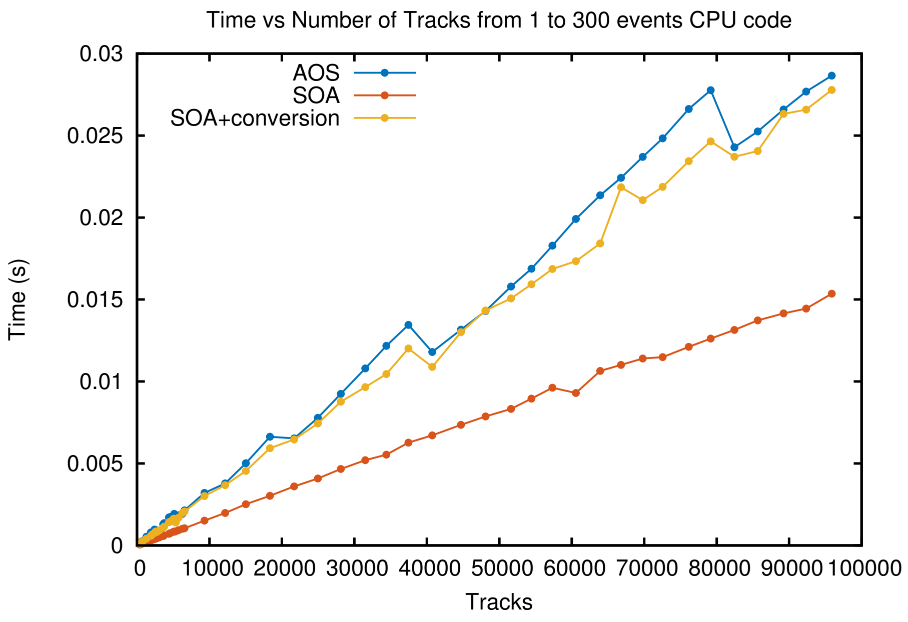


Figure 3.1: Time vs number of tracks in serial code for three different serial implementations:

AOS Array of structures.

SOA Structure of arrays.

SOA+conversion reshape the data in memory from AOS to SOA and then applies the Kalman Filter.

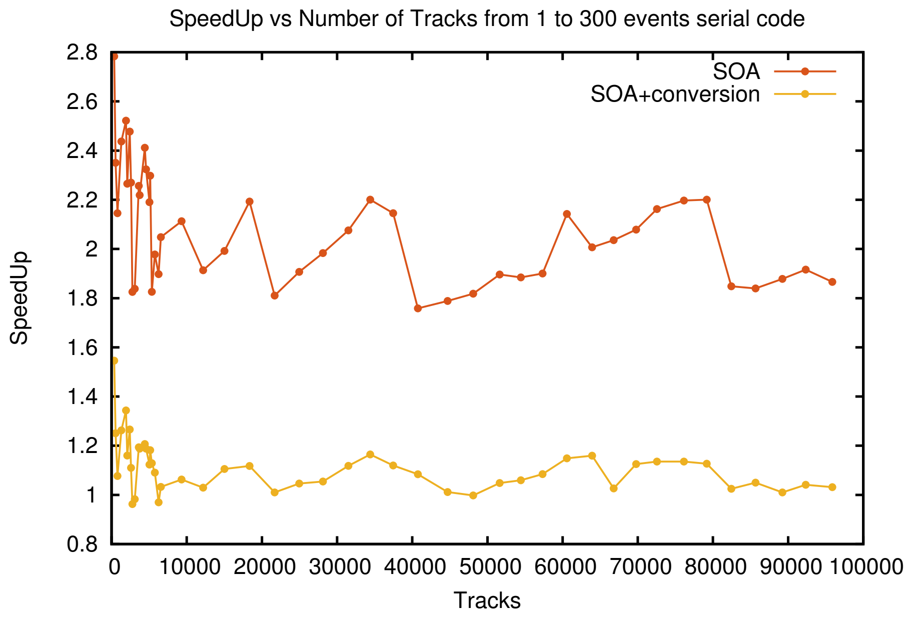


Figure 3.2: Speedup with respect to the original code using serial implementations.

3.2.2 Benchmarks to the GPU versions with data copy

This section presents the benchmark results of the first two implementations for GPU as given in section: 3.1.7 page: 31. Both versions are implemented in CUDA® and OpenCL^{TM}. The second implements the higher parallelism with smaller kernels.

This implementations copy the track data in the local memory with a for loop before starting the calculations. This was implemented assuming that the compiler could optimise the loop to reduce global memory access.

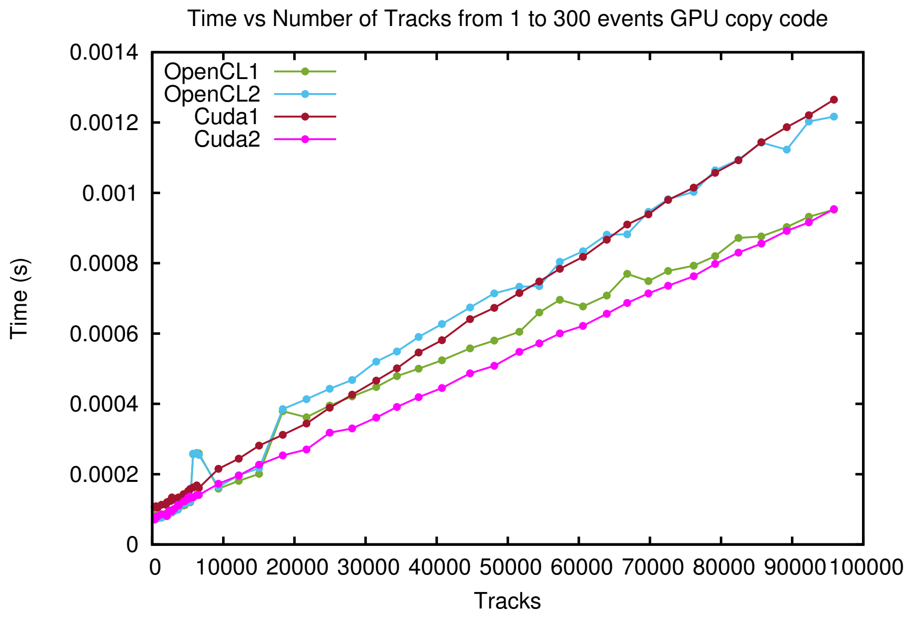


Figure 3.3: CPU time measurement for the Kalman Filter implementation on GPU.

Figure: 3.3 illustrates the filter execution time respect to the total number of tracks.

In the programs made with CUDA the time increases linearly with the tracks.[2]. The slope is stepper in the simple parallel version and OpenCL runs around 10% slower than CUDA. This is the expected behaviour, the higher parallelism and the library optimised for the architecture should provide better performance.

The most parallel version provides always a better performance in CUDA®. The relative rate between the first and the second implementations in CUDA is about 25% for the higher amount of tracks. This is a big rate taking into account that the GPU execution with simple parallelism is already 20 times faster than our best serial code processing 300 events.

There is a basis displacement for time when the track number is zero. This is a consequence the time measurement on the CPU side and it represents the kernel loading time as seen by the HOST side and the execution of at least one block of threads.

The performance with a small number of tracks is slightly better in OpenCL but for more tracks OpenCL becomes worst than CUDA. In the simple parallelization OpenCL brings a better performance than CUDA® over 30000 tracks. With smaller values the behaviour is unstable in OpenCL, opposite to the straight linear time grow observed in CUDA.

The second implementation in CUDA have the best performance of all, and the simple parallelization with OpenCL is the next. With twice the number of threads OpenCL^{TM} holds the worst performance of all as CUDA® have the best one in the same conditions.

This unexpected behaviour suggests that some resources in the GPU are worse managed with OpenCL^{TM} than with CUDA®. The arrays storing the hits information in private memory can be limiting the execution in OpenCL reducing the registers availability when twice the number of threads per block are created. For this reason a longer kernel and less number of threads is a better choice in OpenCL^{TM} opposite to CUDA®.

The other implementation for GPU confirms all this claims.

[2]remember that track number is also the total threads number and the total number of jobs.

3.2.3 Benchmarks to the GPU versions with a pointer to global memory

We tried different strategies to improve the first implementations. The global memory access limited time execution in all the tests according to the debugger tools. We tried the function memcpy from inside the CUDA kernel, but that maintained the same performance and there is not equivalent function in OpenCL.

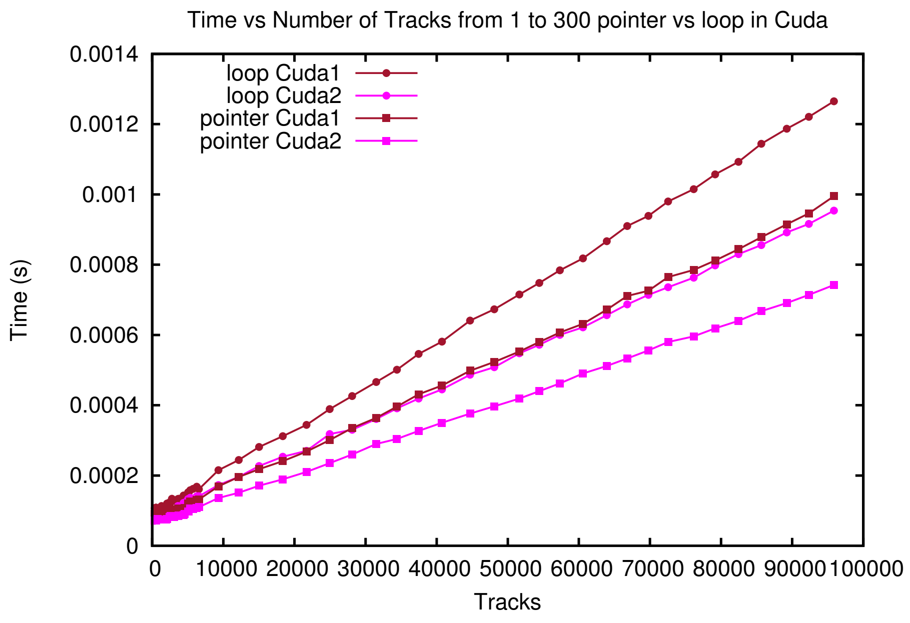


Figure 3.4: CUDA kernel runtime with a loop copy or a pointer to global memory.

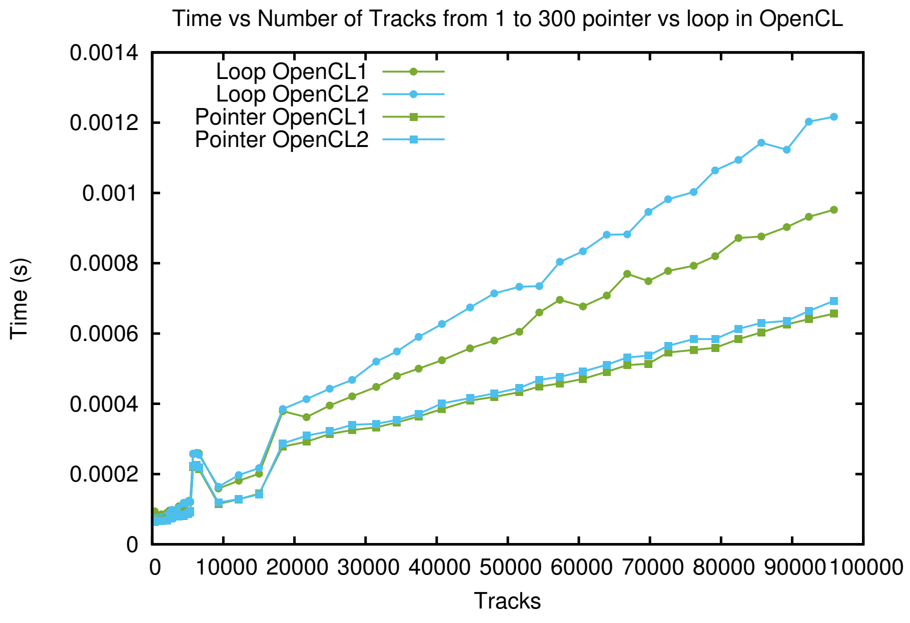


Figure 3.5: OpenCL kernel runtime with a loop copy or a pointer to global memory.

The only modification that made a real difference was the change of the copy loop. We substituted the loop and the local copy with a pointer to the global memory because every value is accessed only once. This improvement allows some calculations to overlapped with global memory access during the kernel execution. It is important to remark that the memory access model used in the implementation is not the most efficient one, but the most practical in the possible real application.

With this change the total accesses to global memory keeps constant, but all the kernels do not try access at the same time. Moreover the local array suppression provides more memory to the threads and reduced the amount of operations inside the kernel.

Figures: 3.4 and 3.5 shows the difference of copy data locally or access them directly in global memory. This change in performance confirms the claims we made in the previous subsection respecting to the resources and the kernel size.

Both CUDA codes received a performance improvements close to 1.35*x* for the higher number of tracks, but the bigger change was in the OpenCL double parallelization. The speedup for this code was 1.7*x*.

Both implementations with the pointer have almost the same performance in OpenCL. This confirms that the internal array allocation affected the performance in the double parallel implementation with OpenCL. Otherwise a bigger difference should remain.

3.2.4 Parallel versions on CPU

As many devices can have OpenCL support, it is possible to run the implemented OpenCL code in the same CPU than the serial code and compare both performances. The comparison between the GPU and CPU executions makes no sense, because those are quite different hardware.

The last benchmark compared the parallel version from OpenCL^{TM} running on the CPU and a parallel multithreaded version implemented with OpenMP.

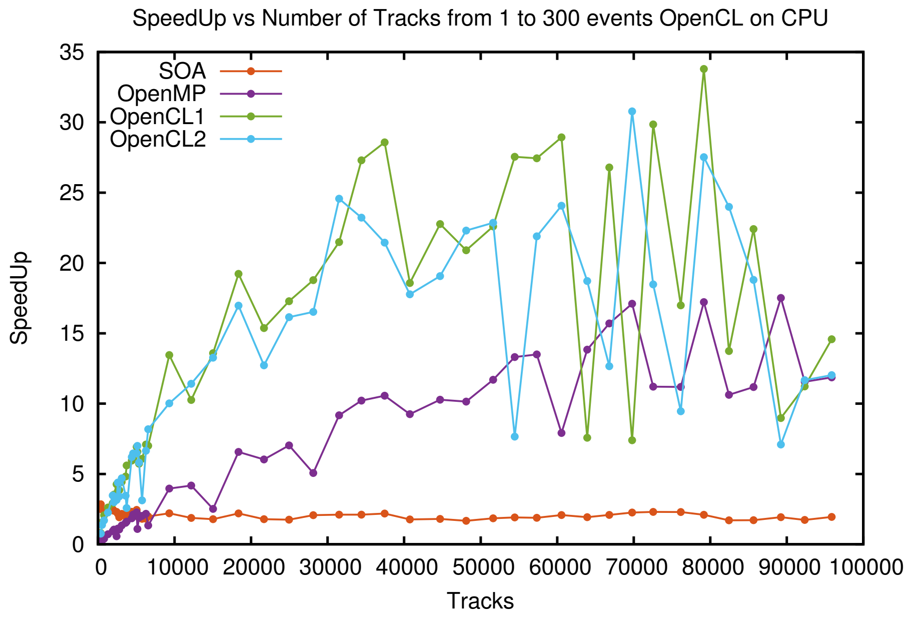


Figure 3.6: Speedup graph for parallel code using OpenCL^{TM} with simple and double parallelism and OpenMP respect to original serial code (AOS). This graph shows also the contiguous memory version with SOA

Figure: 3.6 shows the speedup for the parallel versions of the code using OpenCL^{TM} and OpenMP. This results were in a hardware with 20 cores and activated multi-thread (a total of 40 threads). In this system the speedup can reach around $20 - 30x$ with OpenCL and $15 - 20x$ with OpenMP.

Both parallel versions show the same speedup increase with the tracks on the beginning, but OpenCL grows faster. For a small number of tracks the serial version with contiguous memory access provides better speedup respect to the original code than any of both parallel versions. This only happens with less than 5000 tracks.

For OpenCL^{TM} the graph shows instability with a large number of tracks. We measured the mean over multiple executions but this behaviour always persisted for more than 50000 tracks.

On the other hand for the same system with OpenMP the speedup is lower than 20 and we can appreciate also the unstable behaviour but less intense. For a very large number of tracks

Conclusions

Our massively parallel code to study the Anderson localisation phenomenon was implemented and tested. This code is designed to run in different configurations on different architectures and hardware.

It combines different parallelization approaches in an hybrid scheme. The averaging over the ensemble of realisations of the disorder exploits massively parallelism using MPI in a master-slave configuration.

The calculation of the energy levels (Hamiltonian matrix diagonalization) exploits multi-core and GPU hardware via MAGMA, PLASMA or MKL libraries. The code is flexible and it allows the user to improve the efficiency by tuning the number of disorder realisations solved simultaneously in the same node.

This flexibility permits scientists to tackle the study of disordered systems in the computational optimal configuration, finding the compromise between system size and number of realisations. This provides them the most statistically significant data set.This is a very common situation in science and engineering. We expect that our hybrid scheme will find application in other future research projects.

In our application, the master only sends one integer value (the seed) to each slave and receives the results after the slaves ends the calculations. These integer values (the seeds) are used by the slaves to initialise the sequences of pseudo random-numbers that are employed in the algorithm that generates the disorder pattern. Therefore, the slaves can generate uniquely identified disorder realisations. The ensemble of the seeds is archived by the master providing to the users the advantage of a checkpoint/restart system, which is a fundamental requirement for scientists to access world-class grants on large-scale HPC systems.

The main achievements of the project are:

- We developed a new modularized software in C/C++ for exploiting distributed systems.

  The employed techniques guarantee maintainability and portability of our code.

  The final interface is transparent for the user and the developers. Most of the code is generic and reusable.

- PLASMA is the best option to solve this problems in our architecture.

  MAGMA is a promising library to exploit GPU, but at the moment the benefits are less relevant than the limitations and issues for our application.

  The MKL version we tested does not exploit parallelism efficiently.

- The implemented communication system with a master-slave thread based scheme uses hardware more efficiently.

  The parallelization among different nodes with our framework provides a transparent interface for the final user and exempt him/her from using parallel programming techniques.

  The affinity plays an important role in this master-slave scheme specially when multiple processes share a node.

- The error handler makes the code stable and save computing time.

  The log system within the error handler and defensive programming techniques facilitate to modify and debug the application.

- The master-slave framework manages the initialisation of the sequences of pseudo random-numbers and it archives the seeds used for the initialisation aiding a consistent checkpoint/restart.

The optimisation introduced on both codes for CPU and GPU delivered a relevant speedup on the Kalman Filter. Benchmark results of the various implementations are used at CERN to identify best strategies to improve the Kalman filters in the LHCb triggers. The first trigger execution is then reduced while increasing the the time frame available for the second step algorithm. With longer time to execute the second step, higher resolution will be achieved.

Both GPU versions in CUDA® and OpenCL™ have similar performance and can be considered to be included in the upgrade and in the corresponding implementations of Gaudi.

The main achievements of the project are:

- To isolate the code required simple techniques and makes sense when developing small portions of code from a large framework like Gaudi.

  The isolated code is easier to test, develop and benchmark.

  The invested time to extract the target codes from Gaudi is amply rewarded in the subsequent development time, quality of the code and benchmark's accuracy.

- The contiguous memory implementation substituting AOS with SOA improves our the serial code close to 50%.

  This implementation match the hardware capabilities and can be easily interfaced with other implementations.

  The performance for the SOA implementation is better even when including data reshape from AOS on runtime.

- Both implementations in CUDA® and OpenCL™ provide better performance comparing with the initial serial code.

  OpenCL™ and CUDA® codes behave in the opposite way when the parallelization level increases.

  In our architecture the OpenCL™ code on CPU gives the best performance comparing with the serial code and OpenMP implementations.

- Different memory access optimisations impact the performance much more than the operations within the kernel.

  The array copy substitution with a pointer to global memory improved the implementations for GPU due to memory access and resource availability.

Glossary

AOS Array of structures.

API Application programming interface.

CERN European Organization for Nuclear Research.

CPU Central processor unit.

CUDA® Compute Unified Device Architecture.

FFT Fast Fourier Transform.

GPGPU General purpose GPU programming.

GPU Graphical processor unit.

HEP High Energy Physics.

LAPACK Linear Algebra PACKage.

LHCb Large Hadron Collider “beauty” experiment.

MAGMA Matrix Algebra on GPU and Multicore Architectures Library.

MKL Intel Math Kernel Library.

MPI Message Passing Interface.

OOP Object Oriented Programming.

OpenCL$^{\text{TM}}$ Open Computing Language.

OpenMP Open Multi-Processing API.

PLASMA Parallel Linear Algebra for Scalable Multi-core Architectures Library.

POSIX Portable Operating System Interface.

SOA Structure of arrays..

VELO Vertex Locator detector of LHCb.

Bibliography

Advanced Micro Devices, Inc (2014). *OpenCL and the ATI Stream SDK v2.0*. URL: http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-and-the-amd-app-sdk/ (visited on 08/11/2015).

Anderson, P. W. (1958). "Absence of Diffusion in Certain Random Lattices". In: *Phys. Rev.* 109 (5), pp. 1492–1505.

Aspect, Alain and Massimo Inguscio (2009). "Anderson localization of ultracold atoms". In: *Physics Today* 62.8, pp. 30–35.

Balaji, Pavan et al. (2008). "Toward Efficient Support for Multithreaded MPI Communication". English. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra. Vol. 5205. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 120–129. ISBN: 9783540874744. DOI: 10.1007/978-3-540-87475-1_20.

Boyer, Michael (2015). *Choosing Between Pinned and Non-Pinned Memory*. URL: https://www.cs.virginia.edu/~mwb7w/cuda_support/pinned_tradeoff.html (visited on 12/01/2015).

Clemencic, Marco (2015). *LHCb Software Tutorials*. URL: https://twiki.cern.ch/twiki/bin/view/LHCb/LHCbSoftwareTutorials (visited on 07/10/2015).

Cupper, J.J.M. (1981). "A divide and conquer method for the symmetric eigenproblem". In: *Numerische Mathematik* 36, pp. 177–195.

Dolbeau, R., F. Bodin, and G.C. de Verdiere (2013). "One OpenCL to rule them all?" In: *Multi-/Many-core Computing Systems (MuCoCoS), 2013 IEEE 6th International Workshop on*, pp. 1–6. DOI: 10.1109/MuCoCoS.2013.6633603.

Dongarra, Jack, Jakub Kurzak, and Julien Langou (2010). *PLASMA Users Guide*. University of Tennessee.

Du, Peng et al. (2012). "From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming". In: *Parallel Comput.* 38.8, pp. 391–407. ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.10.002.

Experiment, LHCb (2008). *VErtex LOcator (VELO)*. URL: http://lhcb-public.web.cern.ch/lhcb-public/en/detector/VELO-en.html (visited on 11/05/2015).

Fang, Jianbin, Ana Lucia Varbanescu, and Henk Sips (2011). "A Comprehensive Performance Comparison of CUDA and OpenCL". In: *Proceedings of the 2011 International Conference on Parallel Processing*. ICPP '11. Washington, DC, USA: IEEE Computer Society, pp. 216–225. ISBN: 9780769545103. DOI: 10.1109/ICPP.2011.45.

Fratini, E. and S. Pilati (2015). "Anderson localization of matter waves in quantum-chaos theory". In: *Phys. Rev. A* 91 (6), p. 061601.

Haidar, Azzam, Hatem Ltaief, and Jack Dongarra (2011). "Parallel Reduction to Condensed Forms for Symmetric Eigenvalue Problems using Aggregated Fine-Grained and Memory-Aware Kernels". In: *Published in the proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Seattle, WA, USA.

— (2012). "Toward a High Performance Tile Divide and Conquer Algorithm for the Dense Symmetric Eigenvalue Problem". In: *SIAM Journal on Scientific Computing*.

ICL Team (2015). *MAGMA Main page*. URL: http://icl.cs.utk.edu/magma/index.html (visited on 11/10/2015).

Kalman, Rudolph Emil (1960). “A New Approach to Linear Filtering and Prediction Problems”. In: *Transactions of the ASME–Journal of Basic Engineering* 82.Series D, pp. 35–45.

Karbasi, Salman, Ryan J. Frazier, et al. (2014). “Image transport through a disordered optical fiber mediated by transverse Anderson localization”. In: *Nature Communications*.

Karbasi, Salman, Craig R Mirr, et al. (2012). “Observation of Transverse Anderson Localization in an Optical Fiber”. In: *Optics Letters* 37.12, pp. 2304–2306.

Lagendijk, Ad, Bart van Tiggelen, and Diederik S Wiersma (2009). “Fifty years of Anderson localization”. In: *Phys. Today* 62.8, pp. 24–29.

LHCb Experiment (2008). *The LHCb Detector*. URL: http://lhcb-public.web.cern.ch/lhcb-public/en/detector/Detector-en.html (visited on 08/11/2015).

LHCb software architecture group (2015). *The Gaudi Project*. URL: http://proj-gaudi.web.cern.ch/proj-gaudi/ (visited on 10/10/2015).

MAGMA development team (2015). *MAGMA User Guide*. URL: http://icl.cs.utk.edu/projectsfiles/magma/doxygen/index.html (visited on 11/10/2015).

NVIDIA Corporation (2015). *What is GPU Computing?* URL: http://www.nvidia.com/object/what-is-gpu-computing.html (visited on 11/05/2015).

Rabenseifner, Rolf (2003). “Hybrid Parallel Programming on HPC Platforms”. In: *Published in the proceedings of the Fifth European Workshop on OpenMP, (EWOMP 03), Aachen, Germany, 22-26 Sept 2003*. Aachen, Germany.

Sahni, Sartaj and George Vairaktarakis (1996). “The master-slave paradigm in parallel computer and industrial settings”. In: *Journal of Global Optimization* 9.3-4, pp. 357–377.

SCAI, CINECA (2012). *GPGPU (General Purpose Graphics Processing Unit)*. URL: http://www.hpc.cineca.it/content/gpgpu-general-purpose-graphics-processing-unit (visited on 11/05/2015).

Shapiro, Boris (2012). “Cold atoms in the presence of disorder”. In: *Journal of Physics A: Mathematical and Theoretical* 45.14, p. 143001.

Solca, Raffaele et al. (2012). “A hybrid Hermitian general eigenvalue solver”. In: *Partnership for Advanced Computing in Europe*.

Welch, Greg and Gary Bishop (1995). *An Introduction to the Kalman Filter*. Tech. rep. Chapel Hill, NC, USA.