High-Performance Computing Techniques in Theoretical Physics

Julio Barrientos* and Kenneth Roche (Capstone Advisor)
University of Washington. Seattle, WA

(2024 PHYS 600 Capstone)
(Dated: August 4, 2024)

In the last three decades, we have seen rapid advancement in what has come to be known as High Performance Computing (HPC). This has been achieved, among other things, thanks to the development of multiple cores in the same CPU chipset, clusters of computers connected by high-speed fiber optics, and general-purpose Graphic Processing Units (GPGPU). To support the efficient management of these hardware innovations, software frameworks such as the Message Passing Interface (MPI), thread programming models, and proprietary languages like CUDA have also been developed. HPC advances have been fundamental to the progress of many fields, such as machine learning through neural networks, big data analysis, and scientific simulation of physical systems.

In this capstone, we summarize several of these software and hardware advances that enable scientific HPC in particular. Finally, we give a brief introduction to what will be a paradigm shift from classical computing to quantum computing, based on different foundations than boolean logic, which is the basis of today's computing. C++ and Python code[1] can be found in the public GitHub repository: `https://github.com/ulitoo/HPC583`.

I. INTRODUCTION

With the advent of quantum computing, there is a promise to deliver a yield of computing power that our classical computers cannot provide. As more and more high-performance computing applications, such as physics simulations or machine learning model training, emerge, the limits of classical computation are put to the test. These limits include data storage, computation speed, and energy consumption within a given mass or volume of specific hardware. But where exactly do these limits lie for classical computation today?

This document attempts to give an overview of the main hardware and software tools used today in classical computation applied to scientific computing and simulation. We will learn that a significant amount of resources and tools are focused on basic linear algebra and the discrete Fourier transform, which are the mathematical foundations for physics simulations. Building on these two pillars, we will explore other computational software developments to scale the use of multiple distributed hardware systems. In the course of this exploration, we will also describe how to use these tools in a well-defined development environment, specifically Linux and C++.

II. PHYSICS PROBLEMS AND LINEAR ALGEBRA

Differential equations appear in all fields of physics, such as fluid dynamics, electromagnetism, heat conduction, and both classical and quantum physics. In this section, we explore how we can reduce both linear and non-linear differential equations (including both ordinary and partial types) to ultimately form a system of linear equations that can be numerically solved as a matrix linear algebra problem.

A. Explicit vs Implicit methods

There are a couple of approaches to solving physical systems. The first one is the explicit method, which calculates the future state of the system based only on the current state. This method is relatively easy to implement compared to implicit methods and usually does not require matrix algebra. A good example is seen in section VIII, where we explore explicit methods for a gravitational system. This method can be used when we can get a differential equation into an explicit form, but it is not always possible to do so. For example, when you have non-linear terms in the equation (example equation 1) or if you have higher order differential equations (example in equation 2) as these:

$$\frac{dy}{dt} = y^2 - t \tag{1}$$

$$\frac{d^3y}{dt^3} - 3\frac{d^2y}{dt^2} + 2\frac{dy}{dt} = \sin(t) \tag{2}$$

In these cases, we are dealing with differential equations in implicit form. Implicit computational methods utilize a wide range of system states, both past and future, and are typically addressed with matrix linear algebra. These methods are usually applied to 'stiff' differential equations, which exhibit low stability, especially when the time discretization is coarse. While stability can be improved with smaller time steps, implicit methods are more robust for these problems. This section will succinctly explore the steps needed to transform a

* barrient@uw.edu

physics problem described by a differential equation into a matrix linear algebra problem to be solved with implicit methods. These methods are typically iterative and well-suited for computational solutions.

B. Finite Difference Method

The finite difference method is a simple method that approximates every derivative in the differential equation to the finite difference representation. For example, for time steps and space discretization that are small enough:

Time derivative:

$$\frac{\partial u}{\partial t} \approx \frac{u_i^{n+1} - u_i^n}{\Delta t} \tag{3}$$

Second spatial derivative:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{(\Delta x)^2} \tag{4}$$

In order to follow this method, the first thing that has to happen is to decide what is the discretization criteria to be applied. The size of the time step Δt or spatial distance between samples Δx will depend on several factors, including the stability criteria for the given differential equation (we will expand on the stability criteria in section VIII A), the convergence of the solution when dealing with iterative methods and on the accuracy desired for our outcome.

Once we substitute the finite approximations to the differential equation, we will end up with a matrix equation depending on the amount of data we are considering. For example, for the heat equation:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} \tag{5}$$

it is easy to demonstrate we will end up in an equation such as:

$$u_i^{n+1} = u_i^n + \frac{\alpha \Delta t}{(\Delta x)^2} \left(u_{i+1}^n - 2u_i^n + u_{i-1}^n \right) \tag{6}$$

that can also be represented in matrix form like:

$$\mathbf{u}^{n+1} = (\mathbf{I} + \mathbf{A})\mathbf{u}^n \tag{7}$$

where:

$$\mathbf{u}^n = \begin{pmatrix} u_1^n \\ u_2^n \\ \vdots \\ u_{N-1}^n \end{pmatrix} \tag{8}$$

$$\mathbf{A} = \frac{\alpha \Delta t}{(\Delta x)^2} \begin{pmatrix} -2 & 1 & 0 & \cdots & 0 & 0 \\ 1 & -2 & 1 & \cdots & 0 & 0 \\ 0 & 1 & -2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -2 & 1 \\ 0 & 0 & 0 & \cdots & 1 & -2 \end{pmatrix} \tag{9}$$

At this stage, we have a system of linear equations that can be solved numerically through the methods we will see in this document.

C. Finite Element Method

It is worth mentioning the Finite Element Method (FEM) to avoid confusion with the earlier Finite Difference Method (FDM), which has a much longer history, dating back to Carl Friedrich Gauss in the 19th century. The Finite Element Method was developed in conjunction with the advent of computing machines around the forties, fifties, and sixties. In 1943, Richard Courant[2] (whom we will mention later regarding the study of the stability of computational problems) wrote a paper that laid the foundations of the finite element method applied to torsion problems. The main idea of this method is to take a highly complex problem in 2D or 3D and split it into much smaller, contiguous problems. For example, it is common to use triangles to solve 2D surface problems and tetrahedrons to solve 3D volume-related problems. These 'elements' produce a set of equations with boundary conditions that collectively create a much larger set of equations. The complexity arises not only in finding the optimal element shape but also in assembling the system of equations to get the final global system that models the entire problem. Once this is done, the next step is to apply known mathematical and computational techniques for large-scale matrices in linear algebra. This document will focus on these computational techniques but will not go into detail on how FEM consolidates the partial equations into the final system of equations.

D. Boundary Conditions and Boundary Value Problem

One essential step that we have not discussed so far, but is crucial when framing the physics problem and converting differential equations into a linear algebra problem, is defining the boundary conditions. As we know, differential equations are ill-defined without boundary conditions, which provide a specific solution instead of a range of infinite solutions. Boundary conditions ensure the uniqueness and even the existence of the solution in a given domain. A boundary-value problem is nothing more than a differential equation problem that is subject to constraints or boundary conditions. There are three types of boundary conditions, depending on the limitations they impose on the problem:

- **Dirichlet** Boundary Conditions. They define the value of the function at the limits or boundaries of the domain.
- **Neumann** Boundary Conditions. They define the value of the first derivative of the function at the boundaries.

- **Robin** Boundary Conditions. They define a more complex functional relationship between the values of the function and its derivative at the boundary, for example: $\alpha \cdot y(x_0) + \beta \cdot y(x_0) = \gamma$.

The application of Dirichlet conditions will reduce the size of the system of equations (order of the problem), thereby reducing the size of the matrices we need to handle in our computations. In contrast, applying Neumann conditions will not reduce the matrix size but will change the entries in the matrix and the result vector in the equation $\mathbf{A} \cdot x = b$.

Note that common boundary-value problems include wave equations, electromagnetism problems, heat conduction and many more.

E. Non-linear differential Equations

Some common problems we will have to face in physics are non-linear problems, in which we will see non-linear differential equations. Examples of these are the Navier-Stokes equations in fluid dynamics, the Einstein general relativity field equations or the non-linear Schrödinger equation under non-linear potentials or non-linear particle interactions. These problems will end up in a system of equations that are non-linear, for example with terms like e^x, x^2 or $sin(x)$.

There are several iterative methods to solve these problems, but one of the most used is the **Newton-Raphson** method, first described by Newton in 1669 only for polynomials and later extended by Thomas Simpson in 1740 for its applicability to any non-linear system of equations.

Let's briefly describe what this iterative method does and how can it transform a non-linear system of equations into a succession of system of linear equations that will converge into a final solution. Regardless of the linearity, you will need as many equations as the number of unknown parameters we have in the system of equations. $\mathbf{F}(\mathbf{x}) = 0$ will be the n equations, with $\mathbf{x}$ being the vector with the n unknowns. We will need two things now:

1. An initial good guess that can drive you near the solution and can ensure the convergence: $\mathbf{x}^{(0)}$
2. The formulation of the **Jacobian** Matrix of the system of equations

The Jacobian will be formulated as follows:

$$\mathbf{J}(\mathbf{x}) = \frac{\partial \mathbf{F}}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_1}{\partial x_2} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} & \cdots & \frac{\partial F_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial x_1} & \frac{\partial F_n}{\partial x_2} & \cdots & \frac{\partial F_n}{\partial x_n} \end{pmatrix} \tag{10}$$

Now with a known $\mathbf{x}^{(0)}$ and a known Jacobian $\mathbf{J}(\mathbf{x}^{(0)})$, we can formulate the following linear equation:

$$\begin{aligned} &\mathbf{F}(\mathbf{x}^{(0)}) + \mathbf{J}(\mathbf{x}^{(0)})(\mathbf{x}^{(1)} - \mathbf{x}^{(0)}) = 0 \\ &\Delta\mathbf{x}^{(0)} = (\mathbf{x}^{(1)} - \mathbf{x}^{(0)}) \\ &\mathbf{J}(\mathbf{x}^{(0)})\Delta\mathbf{x}^{(0)} = -\mathbf{F}(\mathbf{x}^{(0)}) \end{aligned} \tag{11}$$

Which is in the format of a system of linear equations like $\mathbf{A} \cdot x = b$, and can be solved using the methods we will explore later in this document, namely an LU decomposition and a triangular solver.

Once we get the solution $\Delta\mathbf{x}^{(0)} = (\mathbf{x}^{(1)} - \mathbf{x}^{(0)})$, we are now in possession of the next iteration $\mathbf{x}^{(1)}$ and we are on our way to the final solution. The number of iterations will depend on the stability of the problem and the needed accuracy we are looking for. To control the accuracy and the stability of the problem, in each iteration we have to make sure that the norm of the current iteration solution $\|\Delta\mathbf{x}^{(i)}\|$ does not grow with respect to the previous $\|\Delta\mathbf{x}^{(i-1)}\|$. Otherwise, we will be under a case of lack of stability, and we will probably have to review the problem, the boundary conditions or the initial guess $\mathbf{x}^{(0)}$.

F. Eigenvalue problems

In physics, there are subsets of problems that require gathering a limited set of information rather than the full state evolution of the system. For example, consider a vibrational system of a string where we need to determine the natural frequencies, or a quantum mechanical system like a particle in a box where we need to find the ground energy and the energy of possible excited states. These questions are answered by intrinsic properties of the system, independent of its detailed evolution.

These problems are known as eigenvalue problems because the solutions are derived from the eigenvalues of a matrix describing the system. A common example is finding the allowed energy values of a system given its Hamiltonian matrix. This applies to both classical and quantum mechanics. Let's explore this in quantum mechanics using a simple particle in a box with infinite potential as boundary conditions, also known as a quantum well. The Schrödinger equation for this system is straightforward, as the potential is zero inside the box, and the Hamiltonian includes only the kinetic energy term:

$$\hat{H}\psi = E\psi = -\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} \tag{12}$$

with boundary conditions of $\psi(0) = 0$ and $\psi(L) = 0$. When using the finite differences method we saw before, we will come up with the following matrix equation

$$\mathbf{H} = -\frac{\hbar^2}{2m\Delta x^2}\begin{pmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ 0 & 1 & -2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & -2 \end{pmatrix} \tag{13}$$

with Δx being the spatial discretization parameter for the width of the well L. So if we decide to have $N \times N$ matrices, we will have that $\Delta x = L/(N+1)$. When we solve for the eigenvalues of the matrix, $\mathbf{H} - E\mathbf{I}$ we will have numerical values that will get closer to the analytical solution as N increases:

$$E_n = \frac{n^2\pi^2\hbar^2}{2mL^2}, \quad n = 1, 2, 3, \ldots \tag{14}$$

For the eigenvalue problem, there will be also matrix linear algebra solutions that will help us automate this computations without the need or recurring to analytical solutions.

III. SOFTWARE DEVELOPMENT ENVIRONMENT

In the coming sections we will explore the different tools available to solve system of linear equations of different sizes thought matrix algebra and computational methods. We will also develop our own tools and benchmark it with the most used tools in the HPC community. We have chosen Linux as the operating system to do our exploration of all the high performance computing tools. In particular, we base our instructions and guidelines on Ubuntu 22.04. Other development tools include Visual Studio Code and its remote ssh plug-in. With that, we can run a C++ development environment on any PC with Visual Studio Code (version 1.89.0) that is connected to the network of the Linux system. We have also ensured that our Linux System has the following hardware requirements to do a minimum set of the exploration we intend to walk through.

- At the very least, a CPU with 4 cores to test the multiprocessing capabilities.
- One or more GPU that will allow offloading some of the computing out of the CPU. In our case, we have been doing some testing with Nvidia Quadro P600 and Quadro K1200.

It is very important to install of the Nvidia drivers (version 550.54.15) and CUDA Toolkit (version 12.4). These are the basic components, but for each of the sections below where I explore different tools and software packages, I will also give some guidance on the libraries needed, the procedure of installation, and the version that I used for this capstone.

IV. BASIC LINEAR ALGEBRA. BLAS AND LAPACK

Across all scientific computing and physics simulations, we constantly encounter system of linear equations and linear algebra in general. This is not exclusive to classical physics, quantum physics problems and scientific computing, but linear algebra is also in the core basics of machine learning and neural networks weight calculation, image processing, chemistry, statistics and many other fields where computing is applied. In summary, any scientific or non-scientific problem that can be reduced to a matrix problem, will be a great candidate for computers to solve them in a very efficient manner. That is why, since the eighties, several software packages have been developed to optimize these linear algebra operations.

A. BLAS

BLAS stands for Basic Linear Algebra Subprograms and was created in 1979 based on FORTRAN programming language. Even if it is a relatively old language, still today, the BLAS reference implementation is maintained in FORTRAN. The code is maintained in www.netlib.org/blas. There are three level of BLAS routines:

- Level 1 BLAS Routines deal only with vector and vector operations like multiplication, sum or rotation.
- Level 2 BLAS Routines deal with Matrix - Vector operations like multiplications or triangular solvers.
- Level 3 BLAS Routines deal with Matrix - Matrix multiplications or solvers.

It is worth mentioning that the nomenclature of the routines in BLAS set a standard that follows for other libraries. That is why we will detail them here: If the functions have the name 'XYYZZZ' then:

- X is the format data type of the values inside the vector or matrix. S for real single precision, D for double precision, C for complex and Z for double precision complex numbers.
- YY indicates the format of the matrix. For example, GE means a general matrix and DI means a diagonal matrix, TR triangular and HE is complex hermitian.
- ZZZ will indicate the actual operation. MM for matrix multiplication, MV for matrix vector multiplication, SV for solver and so on.

B. LAPACK

LAPACK is also a library of code and routines that stands for Linear Algebra Package and, like BLAS, the

reference code is maintained in FORTRAN. The goal of LAPACK is to solve more advanced linear algebra problems like the eigenvalue problem, the singular value decomposition or the linear least squares problem. The code is maintained in `www.netlib.org/lapack` and the routines follow a similar nomenclature that the one described above for BLAS. In fact, the more complex routines that LAPACK is focused on, are built on top of BLAS.

C. OpenBLAS

OpenBLAS is the open source implementation of BLAS and LAPACK both in C and FORTRAN which is optimized to many of current today's hardware architectures like x86 or ARM. The code is based on the also open source implementation GotoBLAS[3] under BSD License. Code releases and information are maintained in `openblas.net` and repository is in GitHub[4].

D. Installation of OpenBLAS Libraries

In this section, I will give a step-by-step guidance on how to install the libraries in the Linux environment.

- Download the latest tar.gz from `openblas.net`
- Run the `gunzip` and `untar` commands for the file you downloaded into a directory in your Linux computer.
- Follow the `make` instructions also present in `openblas.net`. You will need to 'make' the whole package so it will compile for your specific hardware.
- You might want to give some parameters in the make command prompt:
  - If you are using `gfortran` as a compiler for FORTRAN, you might want to specify it as `make FC=gfortran`
  - Also, you might want to install your libraries in a specific path. e.g. `make PREFIX=/usr/local`
- Please avoid installing it as a Linux package with `apt install` commands. Instead, compile it from source code so it can run smoothly in the specific architecture of your computer.

This will set up the development environment so you can access the libraries when you code in C++. Make sure your libraries appear in your desired path that is aligned with your environment in Visual Studio Code. (e.g.:`/usr/local/lib/libopenblas.a`). You have to make sure your include path in Visual Studio points to this directory. The configuration file is called `./.vscode/c_cpp_properties.json` and the parameter is `"includePath"`.

Now you should be setup to start using the BLAS And LAPACK functions that are included in the OpenBLAS package.

E. Column Major and Row Major representations

When storing matrices in the memory of a computer, we will have to decide how to store the information. The memory of any computer can be thought as a linear one dimensional array of elements that can be of different size in bytes depending on the data type we are storing. When storing a matrix, we have to decide in which order will we place the element of the matrix contiguous to each other. There are two main choices to do that, which are called Column-major and Row-major order.

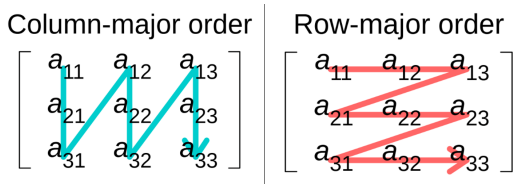

FIG. 1. Matrix Column-major and Row-major Order

Looking at figure 1 we see that the 9 elements of the matrix in Column order will be stored in Column-major like: $a_{11}, a_{21}, a_{31}, a_{12}, a_{22}...$ and in Row-major order like: $a_{11}, a_{12}, a_{13}, a_{21}, a_{22}....$ This ordering will have implications in memory access efficiency, but mostly we need to be consistent on our approach to read and store the matrices so we don't have any confusion. Typically, even if there is flexibility to select one or the other, the default method of storage is Column-major order and that will be the default in all the code presented in the GitHub repository

F. Recursive LU Decomposition without Pivoting

In this section, we are going to explore how to develop a recursive LU decomposition algorithm for a given Matrix. LU decomposition is the first step in order to do solve a given linear algebra system defined by a matrix A. We will focus or efforts on square matrix and leave the rectangular matrices out of the scope for this exercise. The LU decomposition was first proposed in 1938 and is based on the well known Gaussian elimination method to reduce and solve a system of linear equations. Given a Matrix A we have to find the lower triangular matrix L and the upper triangular matrix U such as:

$$A = L \cdot U \tag{15}$$

For example, for a 3x3 matrix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} \ell_{11} & 0 & 0 \\ \ell_{21} & \ell_{22} & 0 \\ \ell_{31} & \ell_{32} & \ell_{33} \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \quad (16)$$

In fact, we can choose L or U to have ones in the diagonal. Let's do the calculations with L having 1 on the diagonal:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \ell_{21} & 1 & 0 \\ \ell_{31} & \ell_{32} & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \quad (17)$$

In a general case of matrices of size $n \times n$ we will have the following decomposition of the matrices where a_{11} represents a scalar, $\mathbf{a_{21}}$ represents a vector of size $(n-1)$ and $\mathbf{A_{22}}$ represents a matrix of size $(n-1) \times (n-1)$.

$$\begin{aligned} \begin{bmatrix} a_{11} & \boldsymbol{a}_{12} \\ \boldsymbol{a}_{21} & \mathbf{A}_{22} \end{bmatrix} &= \begin{bmatrix} 1 & \mathbf{0} \\ \boldsymbol{\ell}_{21} & \mathbf{L}_{22} \end{bmatrix} \cdot \begin{bmatrix} u_{11} & \boldsymbol{u}_{12} \\ \mathbf{0} & \mathbf{U}_{22} \end{bmatrix} \\ &= \begin{bmatrix} u_{11} & \boldsymbol{u}_{12} \\ u_{11}\boldsymbol{\ell}_{21} & (\boldsymbol{\ell}_{21}\boldsymbol{u}_{12} + \mathbf{L}_{22}\mathbf{U}_{22}) \end{bmatrix} \end{aligned} \quad (18)$$

Note that we are dividing the matrices into four pieces, one scalar, one row vector of size $(n-1)$, one column vector of size $(n-1)$ and one matrix of size $(n-1)\times(n-1)$. When we equate term by term of equation 18, we will have the following equations:

$$\begin{aligned} a_{11} &= u_{11} \\ \boldsymbol{a}_{12} &= \boldsymbol{u}_{12} \\ \boldsymbol{a}_{21} &= u_{11}\boldsymbol{\ell}_{21} \rightarrow \boldsymbol{\ell}_{21} = \frac{1}{u_{11}}\boldsymbol{a}_{21} \\ \mathbf{A_{22}} &= \boldsymbol{\ell}_{21}\boldsymbol{u}_{12} + \mathbf{L}_{22}\mathbf{U}_{22}. \end{aligned} \quad (19)$$

Which can be simplified to:

$$\mathbf{S}_{22} = \mathbf{L}_{22}\mathbf{U}_{22} = \overbrace{\mathbf{A}_{22} - \boldsymbol{a}_{21}(a_{11})^{-1}\boldsymbol{a}_{12}}^{\text{Schur complement of } \mathbf{A}}. \quad (20)$$

The right-hand side of equation 20 is also known as the Schur complement[5] of the original matrix $\mathbf{A}$ of size $n \times n$. This Schur complement, that we call $\mathbf{S}_{22}$ is a matrix of size $(n-1) \times (n-1)$. We can see now how this can be made a recursive process until we stop the iteration. The recursive process will be reducing the size of the matrix by 1 in every step, reducing the complexity of the LU decomposition.

In the code `BLAS` directory of the GitHub repository, we have a code to write random values into a matrix, and the matrix into a file. The code is `3.WriteColumnMajorOrderRandom.cpp`. After that, the code in `BLAS\7.LUdecomposition.Recursive.cpp` will take the file and run this recursive algorithm until we find a 2x2 matrix, and we return the elements calculated in equation 19 as scalars.

G. Recursive LU Decomposition with Pivoting

The recursive process has an inherent problem of stability, which is the division by a_{11} in the equation 20. When this value is near to zero, we will encounter that the LU decomposition diverges from the expected solution. To solve for this stability inconvenience, there are methods like the *'pivoting'* that reorders the rows of the matrix in each iteration to make sure that the element a_{11}, is the biggest we can find in any row. In order to achieve this reordering, we must keep track of the reordering in the recursive algorithm to later 'undo' and order the solution again. The recursive pivoting algorithm goes as follows[6]:

$$\begin{aligned} \mathbf{Ax} &= \mathbf{b} \\ \mathbf{PAx} &= \mathbf{Pb} \\ \mathbf{LUx} &= \mathbf{Pb}. \end{aligned} \quad (21)$$

$\mathbf{P}$ is called the permutation matrix and is applied on both sides of the system of linear equations. And so we will do a LU decomposition of the $\mathbf{PA}$ matrix instead.

If we divide $\mathbf{P}$ into two permutations $\mathbf{P} = \mathbf{P}_2\mathbf{P}_1$, first one $(\mathbf{P}_1)$ getting the desired row in the top and leaving rest of the rows untouched and $(\mathbf{P}_2)$ getting the rest of the rows reordered, then we will have the following equations:

$$\begin{aligned} \mathbf{PA} &= \mathbf{P}_2\mathbf{P}_1\mathbf{A} = \mathbf{P}_2\bar{\mathbf{A}} \\ \mathbf{PA} &= \mathbf{P_2}\bar{\mathbf{A}} = \mathbf{LU} \end{aligned} \quad (22)$$

$$\begin{bmatrix} 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{P}_{22} \end{bmatrix} \begin{bmatrix} \bar{a}_{11} & \bar{\boldsymbol{a}}_{12} \\ \bar{\boldsymbol{a}}_{21} & \bar{\mathbf{A}}_{22} \end{bmatrix} = \begin{bmatrix} 1 & \mathbf{0} \\ \boldsymbol{\ell}_{21} & \mathbf{L}_{22} \end{bmatrix} \begin{bmatrix} u_{11} & \boldsymbol{u}_{12} \\ \mathbf{0} & \mathbf{U}_{22} \end{bmatrix} \quad (23)$$

$$\begin{bmatrix} \bar{a}_{11} & \bar{\boldsymbol{a}}_{12} \\ \mathbf{P}_{22}\bar{\boldsymbol{a}}_{21} & \mathbf{P}_{22}\bar{\mathbf{A}}_{22} \end{bmatrix} = \begin{bmatrix} u_{11} & \boldsymbol{u}_{12} \\ u_{11}\boldsymbol{\ell}_{21} & (\boldsymbol{\ell}_{21}\boldsymbol{u}_{12} + \mathbf{L}_{22}\mathbf{U}_{22}) \end{bmatrix} \quad (24)$$

As we see here we have done the permutation of the row with the smallest a_{11} and now we operate with the matrix $\bar{\mathbf{A}} = \mathbf{P}_1\mathbf{A}$. If we follow the same one to one mapping of elements in equation 24 as we did for equation 18, we will find the new recursive equation for the pivoting LU algorithm:

$$\mathbf{P}_{22}\overbrace{\left(\bar{A}_{22} - \bar{\boldsymbol{a}}_{21}(\bar{a}_{11})^{-1}\bar{\boldsymbol{a}}_{12}\right)}^{\text{Schur complement of } \bar{\mathbf{A}}} = \mathbf{L}_{22}\mathbf{U}_{22}. \quad (25)$$

Once we finish the recursive iteration, and we reach a 2x2 size matrix, we need to keep memory of the permutation matrix. It will be essential in order to solve the equation in the last step, which is the triangular solver or backward substitution. That is why for example in the LAPACK solver of `LAPACK_dgesv` we have a parameter `IPIV` that outputs the permutation sequence used to solve the linear system. The code for the pivoting algorithm is included in the file `BLAS\8.LUdecomposition.Recursive.Pivot.cpp`.

H. Matrix-Matrix Multiplication - A recursive implementation

Before exploring the Triangular solver, we are going to experiment with recursion methods to divide matrices in quarters. For that, we are going to experiment with the square matrix-matrix multiplication. In the code we can find in `BLAS\9.MMMultiplication.cpp` we can see different approaches for this recursion.

1. MMMultRecursive function

The idea of the recursion algorithm is to divide the square matrix in 4 sub-matrices and multiply those as if it was a 2x2 matrix multiplication. This divide and conquer approach is called block partitioning and works like this. If we have matrices $\mathbf{A}$ and $\mathbf{B}$ of size $n \times n$, we will generate 12 matrices $\mathbf{A_{ij}}$, $\mathbf{B_{ij}}$, $\mathbf{C_{ij}}$ (4+4+4) of size $(n/2) \times (n/2)$ in this manner :

$$\begin{bmatrix} \boldsymbol{A_{11}} & \boldsymbol{A_{12}} \\ \boldsymbol{A_{21}} & \boldsymbol{A_{22}} \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{B_{11}} & \boldsymbol{B_{12}} \\ \boldsymbol{B_{21}} & \boldsymbol{B_{22}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{C_{11}} & \boldsymbol{C_{12}} \\ \boldsymbol{C_{21}} & \boldsymbol{C_{22}} \end{bmatrix} \tag{26}$$

$$\begin{aligned} \boldsymbol{C_{11}} &= \boldsymbol{A_{11}} \cdot \boldsymbol{B_{11}} + \boldsymbol{A_{12}} \cdot \boldsymbol{B_{21}} \\ \boldsymbol{C_{12}} &= \boldsymbol{A_{11}} \cdot \boldsymbol{B_{12}} + \boldsymbol{A_{12}} \cdot \boldsymbol{B_{22}} \\ \boldsymbol{C_{21}} &= \boldsymbol{A_{21}} \cdot \boldsymbol{B_{11}} + \boldsymbol{A_{22}} \cdot \boldsymbol{B_{21}} \\ \boldsymbol{C_{22}} &= \boldsymbol{A_{21}} \cdot \boldsymbol{B_{12}} + \boldsymbol{A_{22}} \cdot \boldsymbol{B_{22}} \end{aligned} \tag{27}$$

We can decide what will be the depth of the recursion, meaning when can we stop the recursion and execute a naive matrix-matrix multiplication. The maximum depth will of course be when the matrices reach the size 2x2, but we can decide what the depth will be depending on our desires for efficiency that will highly depend on our existing hardware. For example, if we have a very efficient hardware managing matrix multiplication of size 8x8, that will be our recursion limit. Also, we have to take into consideration how we implement this recursion. This is the most naive version, in which we do allocate memory for the 12 sub-matrices that we create before calling the same function again. This will help us with code readability but will come with a cost in memory. At the end of the call, we deallocate memory, but that does not remove the need for a great deal of memory requirements for this implementation.

2. MMMultRecursive2 function

In this more optimized version of the recursion, we will not allocate memory for the intermediate sub-matrices $\mathbf{A_{ij}}$ and $\mathbf{B_{ij}}$ but we will allocate memory for the 4 sub-matrices $\mathbf{C_{ij}}$. This is achieved looking for the specific positions of the elements of the original $n \times n$ matrices $\mathbf{A}$ and $\mathbf{B}$. Note that this will be more efficient in terms of memory allocation but will definitely have a cost when searching for matrix elements when N is big. Think that the cache hierarchy of the CPU is optimized to manage contiguous memory segment in a fast manner. When we have to move around a big data set, we will pay the cost when having to move memory segments in and out of the cache.

3. MMMultRecursive3 function

This final function will not even allocate intermediate memory for the sub-matrices $\mathbf{C_{ij}}$. As we said before, this will be more efficient in memory saving but at the same time will come with a cost as we will have to write the values of the matrix $\mathbf{C}$ in very different locations as we go through the recursion. There will always be a cost balance between memory allocation and how contiguous the memory elements will be in the memory segments we use.

I. Triangular Solver - Backward Substitution

Once we have a LU decomposition, we are ready to solve the system of linear equations in what typically is called backward or forward substitution. Note that under a lower triangular matrix multiplying the vector $\mathbf{x}$, we will have one of the equations with only one variable (x_1) and one value (b_1). The next equation will have two variables (x_1,x_2) and one value (b_2). In this second equation we will be able to derive the value of x_2 because we already know x_1, and that backward substitution will progressively continue until we reach the value of x_n, finalizing the process and getting the solution. We just described the process of forward under lower triangular matrices, and the same will apply starting from x_n and ending in x_1 for upper triangular matrices (backward substitution)

As a summary, the forward substitution in a $L \cdot x = b$ matrix equation can be expressed as:

$$\begin{aligned} \ell_{1,1}x_1 &= b_1 \\ \ell_{2,1}x_1 + \ell_{2,2}x_2 &= b_2 \\ &\vdots \\ \ell_{n,1}x_1 + \ell_{n,2}x_2 + \cdots + \ell_{n,n}x_n &= b_n \end{aligned} \tag{28}$$

$$x_1 = \frac{b_1}{\ell_{1,1}}; x_2 = \frac{b_2 - \ell_{2,1}x_1}{\ell_{2,2}}; \ \ldots \ ; x_n = \frac{b_n - \sum_{i=1}^{n-1} \ell_{n,i}x_i}{\ell_{n,n}} \tag{29}$$

These procedures of forward and backward substitution are implemented in the functions `LowerTriangularSolverNaiveReal` and `UpperTriangularSolverNaiveReal` in the code present in `BLAS\10.Triangular_Solver.cpp`.

Now we want to use this substitution in order to get the final values of the $\mathbf{x}$ vector. In a non-pivoting scenario, the equations will be the following:

$$\begin{aligned} \mathbf{A}\cdot\mathbf{x} &= \mathbf{b} \\ \mathbf{L}\cdot\mathbf{U}\cdot\mathbf{x} &= \mathbf{b} \\ \mathbf{U}\cdot\mathbf{x} &= \mathbf{x}' \\ \mathbf{L}\cdot\mathbf{x}' &= \mathbf{b} \end{aligned} \tag{30}$$

Assuming we have found L and U in the previous LU decomposition, we will now solve for the vector $\mathbf{x}$. If we read these equations from bottom up, we see that first we have to do a forward substitution with or $\mathbf{L}$ and the vector $\mathbf{b}$ to find $\mathbf{x}'$. Then, we will do a backward substitution with $\mathbf{U}$ and the known vector $\mathbf{x}'$ to find $\mathbf{x}$. Note that we can do this at scale if instead of one single system of equations we want to solve for several systems of equations. If instead of a vector $\mathbf{b}$ of length n we, instead, use a matrix $\mathbf{B}$ of size $n \times n$, we will solve for n systems of equations, for n different $\mathbf{b}$ vectors as columns of the matrix $\mathbf{B}$. This is exactly what we have done in the code, so we deal entirely with matrices. In this case, the solution $\mathbf{X}$ will also be a matrix of size $n \times n$.

$$\begin{aligned} \mathbf{A}\cdot\mathbf{X} &= \mathbf{B} \\ \mathbf{L}\cdot\mathbf{U}\cdot\mathbf{X} &= \mathbf{B} \\ \mathbf{U}\cdot\mathbf{X} &= \mathbf{X}' \\ \mathbf{L}\cdot\mathbf{X}' &= \mathbf{B} \end{aligned} \tag{31}$$

Moving on to the pivoting case, the solution will be achieved the following way:

$$\begin{aligned} \mathbf{A}\cdot\mathbf{X} &= \mathbf{B} \\ \mathbf{P}\cdot\mathbf{A}\cdot\mathbf{X} &= \mathbf{P}\cdot\mathbf{B} \\ \mathbf{L}\cdot\mathbf{U}\cdot\mathbf{X} &= \mathbf{P}\cdot\mathbf{B} \\ \mathbf{U}\cdot\mathbf{X} &= \mathbf{X}' \\ \mathbf{L}\cdot\mathbf{X}' &= \mathbf{P}\cdot\mathbf{B} \end{aligned} \tag{32}$$

As we can observe again going from bottom to top, we will need in this case to use the permutation matrix and apply it to the matrix $\mathbf{B}$ before starting the forward substitution. After this, it's easy to see that we can obtain the matrix $\mathbf{X}$ after the backward substitution. Remember, we are paying the price of keeping track of the pivoting order in order to get better stability and precision in the solving of the system of linear equations.

1. *Recursive version of Triangular Solver*

Finally, we are going to explore a recursive algorithm in order to have a divide and conquer approach for the Triangular solver. Let's start with the Lower Triangular Matrix Case, and let's divide the Matrices in half. For now, we will assume that the triangular matrix L is of size $n \times n$ and both the matrices X and B are rectangular of size $n \times p$. Also, we are going to do another simplification, assuming that both p and n are powers of 2. The fact that B and X are not vectors and are matrices, instead, will allow us to solve p different number of system of equations simultaneously. In the next figure 2, we see how we use the divide and conquer method to have 4 + 4 + 4 matrices of smaller size and get multiple smaller problems.

$$\overset{n}{\underset{n}{\begin{bmatrix} \boldsymbol{L_{11}} & \mathbf{0} \\ \boldsymbol{L_{21}} & \boldsymbol{L_{22}} \end{bmatrix}}} \cdot \overset{p}{\underset{n}{\begin{bmatrix} \boldsymbol{X_{11}} & \boldsymbol{X_{12}} \\ \boldsymbol{X_{21}} & \boldsymbol{X_{22}} \end{bmatrix}}} = \overset{p}{\underset{nn}{\begin{bmatrix} \boldsymbol{B_{11}} & \boldsymbol{B_{12}} \\ \boldsymbol{B_{21}} & \boldsymbol{B_{22}} \end{bmatrix}}}$$

FIG. 2. Recursive Algorithm for Triangular Solver

We know that $\boldsymbol{L_{11}}$ and $\boldsymbol{L_{22}}$ are also lower triangular matrices. We will find ways to leverage this fact and reduce the triangular problem of $n \times n$ to several $nn \times nn$ problems being $nn = n/2$. Let's see what are the two first immediate matrix equations (33) that we can derive:

$$\begin{aligned} \boldsymbol{L_{11}} &= \boldsymbol{X_{11}}\cdot\boldsymbol{B_{11}} \\ \boldsymbol{L_{11}} &= \boldsymbol{X_{12}}\cdot\boldsymbol{B_{12}} \end{aligned} \tag{33}$$

These two equations can enter into recursion because they are exactly the same problem we had before but with a smaller size. The question is now how to get to a recursion algorithm to solve X_{21} and X_{22}. If we continue with the mathematical derivation, we got:

$$\begin{aligned} &\boldsymbol{L_{21}}\cdot\boldsymbol{X_{11}} + \boldsymbol{L_{22}}\cdot\boldsymbol{X_{21}} = \boldsymbol{B_{21}} \\ &\boldsymbol{B'_{21}} = \boldsymbol{B_{21}} - \boldsymbol{L_{21}}\cdot\boldsymbol{X_{11}} \\ &\boxed{\boldsymbol{L_{22}}\cdot\boldsymbol{X_{21}} = \boldsymbol{B'_{21}}} \end{aligned} \tag{34}$$

Note that B'_{21} is known because we already solved for X_{11} in the previous step. Note that this breaks the parallelism in some way, as we have to wait for the first half of the problem to be solved in order to get to the second half. Now we do have a third recursive equation that we can continue solving in a recursive manner because, as we previously mentioned, L_{22} is also lower triangular. The final and fourth matrix X_{22} will be solved also following the same principles we just saw and based on the previous knowledge of X_{12} to deduce B'_{22}. Let's see how the equations look like and how the final recursion sub-matrix can be solved:

$$\begin{aligned} &\boldsymbol{L_{21}}\cdot\boldsymbol{X_{12}} + \boldsymbol{L_{22}}\cdot\boldsymbol{X_{22}} = \boldsymbol{B_{22}} \\ &\boldsymbol{B'_{22}} = \boldsymbol{B_{22}} - \boldsymbol{L_{21}}\cdot\boldsymbol{X_{12}} \\ &\boxed{\boldsymbol{L_{22}}\cdot\boldsymbol{X_{22}} = \boldsymbol{B'_{22}}} \end{aligned} \tag{35}$$

In summary, the algorithm will be divided in the following steps:

1. Recursively find X_{11} and X_{12}
2. Calculate B'_{21} and B'_{22} based on previous step

3. Recursively find X_{21} and X_{22}

The code for this recursion algorithm for triangular solver can be found in the GitHub repository in `BLAS\10.Triangular_Solver.cpp`. The different functions do the following:

`LowerTriangularSolverNaiveReal`. This is the naive algorithm that does the forward substitution one step at a time to get all the x_n values.

`LowerTriangularSolverRecursiveReal_0`. This is the recursive algorithm. It is the first draft, and it makes memory allocation in every iteration for all the four temporary matrices that are going to be used when splitting the main one. Note that this might make unnecessary allocation of new memory when calling recursively. When we reach sufficient big matrix sizes, we might run into issues consuming all the allocated space for that process memory stack. Let's not forget that in the UNIX programming model each process will have a heap and stack that will be growing and if we don't properly control their growth we might encounter those overwriting other memory spaces (if the operating system does not protect us from doing so first in what is called a 'stack overflow' sometimes raised as a 'segmentation fault')

`LowerTriangularSolverRecursiveReal`. This algorithm is the same as the previous, with the caveat of not making any memory allocation and instead referencing the original matrix position when extracting the elements out of it. As we discussed in the Recursive matrix-matrix multiplication this will mean a more efficient memory management (memory saving and avoiding segmentation faults), but at the same time it will pay the cost of not having data locality and hence not making an efficient use of the processor's cache hierarchy.

The same steps may be followed to deduce the backward substitution algorithm for an upper triangular matrix. The versions of such algorithms can also be found in the same 'cpp' file with 'Upper' instead of 'Lower' naming convention. In equations (36, 37) we see the mathematical foundation needed to implement those recursive algorithms in upper triangular matrix case:

$$\begin{bmatrix} \boldsymbol{U_{11}} & \boldsymbol{U_{12}} \\ \boldsymbol{0} & \boldsymbol{U_{22}} \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{X_{11}} & \boldsymbol{X_{12}} \\ \boldsymbol{X_{21}} & \boldsymbol{X_{22}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{B_{11}} & \boldsymbol{B_{12}} \\ \boldsymbol{B_{21}} & \boldsymbol{B_{22}} \end{bmatrix} \tag{36}$$

$$\begin{aligned} &\boxed{\boldsymbol{U_{22}} = \boldsymbol{X_{21}} \cdot \boldsymbol{B_{21}}} \\ &\boxed{\boldsymbol{U_{22}} = \boldsymbol{X_{22}} \cdot \boldsymbol{B_{22}}} \\ &\boldsymbol{U_{11}} \cdot \boldsymbol{X_{11}} + \boldsymbol{U_{12}} \cdot \boldsymbol{X_{21}} = \boldsymbol{B_{11}} \\ &\boldsymbol{B'_{11}} = \boldsymbol{B_{11}} - \boldsymbol{U_{12}} \cdot \boldsymbol{X_{21}} \\ &\boxed{\boldsymbol{U_{11}} \cdot \boldsymbol{X_{11}} = \boldsymbol{B'_{11}}} \\ &\boldsymbol{U_{11}} \cdot \boldsymbol{X_{12}} + \boldsymbol{U_{12}} \cdot \boldsymbol{X_{22}} = \boldsymbol{B_{12}} \\ &\boldsymbol{B'_{12}} = \boldsymbol{B_{12}} - \boldsymbol{U_{12}} \cdot \boldsymbol{X_{22}} \\ &\boxed{\boldsymbol{U_{11}} \cdot \boldsymbol{X_{12}} = \boldsymbol{B'_{12}}} \end{aligned} \tag{37}$$

As a last note, we have to decide where to stop the recursion. We already had a discussion in the matrix-matrix multiplication, and the same will apply in this case. We can always wait to reach the 2×2 matrix and do a simple calculation of the x_1 and x_2 values of that triangular system of equations, or we can find what is the optimal size that our hardware can manage efficiently, for example depending on how many bytes it can process in parallel. We can also reach a point with a certain size that we know our built-in libraries (LAPACK and/or BLAS) can handle efficiently.

J. Benchmark of Errors, Stability and Speed

In this section we are going to get all the pieces together and solve a large system of equations relying on the building blocks that we have been developing:

1. LU Decomposition with Pivoting.
2. Matrix-Matrix Multiplication.
3. Triangular Solver.

At the same time we are going to test the stability and precision of the different implementations we have been exploring, and we will be bench-marking those with the well known routines in BLAS and LAPACK. Specifically, we will be operating in column major order and with double-precision floating-point format (64 bits - 8 bytes). The gold-post we are going to be measuring against is the LAPACK routine 'dgesv' that does not have any optimization based on the type of input matrices format (dgesv = double general solver) and is called as `LAPACK_dgesv()`.

1. Definitions

Let's define some metrics to make sure we know what exactly we will be measuring against.

Norm: In mathematics, a norm is a function that takes a vector in a vector space and maps it to a non-negative value, that usually represent a length or a distance. There is also a concept of norms that can be applied to matrices. The two most common matrices norms are the ***'one norm'*** of a matrix, A which is the maximum absolute column summation of the matrix:

$$\|A\|_1 = \max_{1 \le j \le n} \sum_{i=1}^{m} |a_{ij}| \tag{38}$$

and the ***'infinity norm'*** of a matrix, A which is maximum absolute row summation of the matrix:

$$\|A\|_\infty = \max_{1 \le i \le m} \sum_{j=1}^{n} |a_{ij}| \tag{39}$$

During this bench-marking, we will mostly make use of the infinity norm $\|A\|_\infty$ through the function `InfinityNorm()`. There is another norm that we will be using as well a couple of times called **Frobenius** norm or Hilbert–Schmidt norm. Taking the square root of the sum of the squares of all elements in a matrix gives the Frobenius norm of the matrix A. The Frobenius norm $||A||_F$ of a matrix A of size $m \times n$ is defined as:

$$||A||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2} \tag{40}$$

Where a_{ij} represents the element in the i^{th} row and j^{th} column of the matrix A.

This norm provides a measure of the "size" of the matrix, similar to the Euclidean norm for vectors. It is calculated by squaring each element in the matrix, summing all these squared values, and then taking the square root of that sum.

Condition number: In the context of numerical analysis, the condition number of a function measures how the output of the function will vary under the small changes in the input, in other words, how stable is the function to perturbations or errors in the input. We typically look for low condition numbers, in which case we can say that the function or the problem at hand is well-conditioned. In the context of matrices in the linear equation problem $A \cdot x = b$, we can calculate the condition number as:

$$\kappa(A) \simeq \left\|A^{-1}\right\|_\infty \|A\|_\infty \geq \left\|A^{-1}A\right\|_\infty = 1 \tag{41}$$

There will be a limit of condition number above which, we should avoid pursuing an analysis. We have to make sure that the analysis or benchmark we perform is under well-conditioned problems to avoid undesired stability issues. For that, we will configure a threshold of maximum condition number when randomly generating our matrices, above which, we will discard the matrix and randomly generate another. We can see the details of the random matrix generation and the condition number vetting (with the function `ConditionNumber()`) in the code `BLAS\13.WriteColumnMajorOrderRandom.cpp`.

Residual error: When solving the linear equation, we will have many ways to calculate the error, but there will be two main metrics we need to define. The first one is the residual. In the context of a system of linear equations, we will find the vector x in the equation $A \cdot x = b$. Once we do solve it, we will be able to calculate the error we made, $A \cdot x - b$ which will be the residual error of the solver. This residual will be a vector of size n. Now, note that during this bench-marking, we will be using a Matrix B and X of size $n \times n$ instead of a vector b and x of size n. The reason is that we will be able to solve n equations in one shot, and we will also handle matrices of size $n \times n$ instead of vectors. Hence, the residual error will be really a matrix $E = A \cdot X - B$ of size $n \times n$. In order to reduce this matrix to a single number, we will find the norm of the matrix, specifically the infinite norm. So finally, the residual norm will be:

$$\|E\|_\infty = \|A \cdot X - B\|_\infty \tag{42}$$

The **Normalized forward error**: To be more precise in our error estimation, it will be better to normalize our residual norm against the nature of the problem. That way we will be able to make a fair comparison of the error if we intend to compare errors among different problems. In order to do that, we will calculate the normalized forward error as:

$$\frac{\|A \cdot X - B\|_\infty}{\|A\|_\infty \cdot \|X\|_\infty \cdot \varepsilon_{machine}} \tag{43}$$

where $\varepsilon_{machine}$ is the upper bound error of rounding floating points for the given type (double float in this case) for the specific machine we are compiling the code on. A fairly direct method of calculating the machine epsilon of the given type is to divide the number 1 in half until the number is indistinguishable from zero in the machine. The code that does that for us can be found in the same file as the function `double_machine_epsilon()`. $\varepsilon_{machine}$ is around `2.22045e-16` for double type and `1.0842e-19` for long double type.

We can always find other methods of calculating errors, like the cumulative error or the sum of all the error elements of the residual matrix, but in this benchmark, we will focus on the normalized forward error we just described.

2. Improving stability with pivoting

First, we will analyze the improvement in stability we achieved with the pivoting method for the LU decomposition. Using the code found in `BLAS\18.Complete_Solver_with_Pivot_Clean.cpp` we find that

In the Table I, we see what is the value of the Frobenius Norm of the residual matrix for my implementation of **non-pivoted** solver, for my implementation of **pivoted** solver and for the LAPACK implementation of the solver `LAPACK_dgesv`. This is shown for two different sizes of matrices, N=512 and N=1024.

$\|\|A \cdot X - B\|\|_F$	Non-Pivoted	Pivoted	`LAPACK_dgesv`
512×512	$3.84453 \cdot 10^{-9}$	$1.6329 \cdot 10^{-11}$	$1.59339 \cdot 10^{-11}$
1024×1024	$6.32547 \cdot 10^{-08}$	$1.35544 \cdot 10^{-10}$	$1.27487 \cdot 10^{-10}$

TABLE I. Frobenius Norm of the residual matrix.

In a 512×512 matrix, my pivot implementation is around 200 times more precise than my non-pivot implementation and only 2% less precise than the LAPACK

solver. In the case of a 1024×1024 matrix, the pivot is around 500 times more precise than the non-pivot and 6% less precise than the LAPACK solver. With this we can see that the improvement of stability and accuracy that the pivoting brings is around 2, almost 3 orders of magnitude higher than a non-pivoting solution, which makes the pivoting algorithm a must in our implementation. From now on, all our calculations and bench-markings against `LAPACK_dgesv` will be done based on my pivot implementation of the LU decomposition.

3. *Metrics of the solver implementation vs LAPACK*

The code we will be exploring now is focused on the performance comparison between `LAPACK_dgesv` and my implementation for different matrix sizes and different matrix properties. The code will be found in `Basics\Errors\` directory and these are the files:

- `JBG_BLAS.h` Header file with the list of all the functions.
- `1.NormalizedFWDError_f.cpp` File with all the function definitions.
- `1.NormalizedFWDError_v0.cpp` Main code to do the preliminary bench-mark and print numerical results to screen.
- `1.NormalizedFWDError.cpp` Main code to create a file with all the data results for different matrix sizes.
- `1.PlotFwdError.py` Python code that take the data file as an input and plots the results.
- `Results_lapack` Data file with data points for LAPACK simulation results.
- `Results_mine` Data file with data points for my LU + triangular solver implementation results.

In the main part of the algorithm, we will progressively increase the size of the matrix (2^n) from $n = 1 \Rightarrow (2 \times 2)$ to $n = 14 \Rightarrow (16384 \times 16384)$. Because of time constraints we will not simulate our algorithm with 16384 and will stop at $(8192 = 2^{13})$. The matrix elements of a_{ij} of A, and b_{ij} of B in the equation we want to solve $A \cdot X = B$, will be double precision float values randomly generated from the range $[-0.5, 0.5]$.

After solving the equation, the seven main data points that we will be collecting are:

1. Matrix size
2. $||A \cdot X - B||_\infty$ Residual Infinity Norm
3. $||A||_\infty$ Infinity Norm
4. $||X||_\infty$ Infinity Norm
5. Machine Epsilon $\varepsilon_{machine}$
6. Normalized Forward Error
7. Elapsed Time of the solver.

All those data points are written in the files `Results_lapack`, `Results_mine` and will then be read by the python code and plotted with the `matplotlib` library. In figure 3 we can see the comparison between `LAPACK_dgesv` implementation and our implementation for the Residual Error. The graph is logarithmic in the y axis, but still we can see a close trend between both implementations, as we discovered in the previous analysis. The pivoting solution gets the stability well under control, aligned with the standard LAPACK implementation.

We can observe that as the size of the matrix increases, LAPACK implementation is more stable, probably due to the use of other techniques, for example scaling and normalization of rows and columns before starting the solver. Note that for some matrices, QR decomposition (using an orthogonal matrix) or SVD (Singular Value Decomposition) might be better suited than LU decomposition, and they might be optimizing for stability or for time to solve.

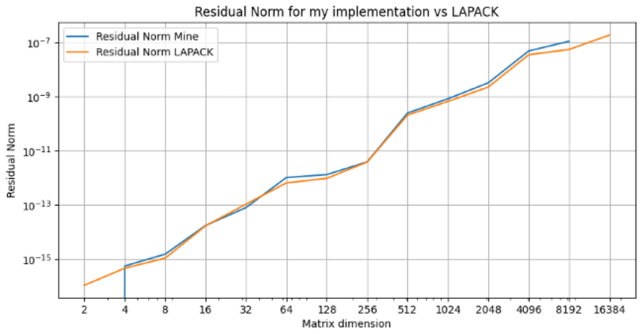

FIG. 3. Residual Norm of LAPACK vs this implementation

The next comparison we will do is the Normalized Forward Error according to the equation 43. As we can see in figure 4, the difference of both lines is now much more clear and also smaller than the residual norm on the basis of the normalization of the error.

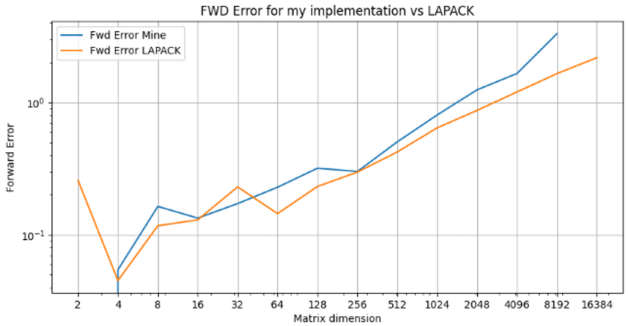

FIG. 4. Forward Error of LAPACK vs this implementation

We can observe that for small matrices, both LAPACK and the test implementation follow a similar trend, but for matrices larger than 512×512 we start to see a clear divergence in which the LAPACK solver shows a better behavior as the size of the matrix grows.

To end the analysis, we will now compare the CPU performance. Clearly, this is the point where we can see the major difference between both implementations. We see in figure 5 that, starting with matrices of 1024×1024 and beyond, `LAPACK_dgesv` is around two orders of magnitude faster than the implementation we developed. This is completely expected as BLAS and LAPACK libraries have been compiled and optimized for our current hardware and will take advantage of data structures and data blocking that will run more efficiently in the hardware, making efficient use specially of the CPU cache hierarchy. Also note that our code is single threaded, and we are not leveraging the existence of multicore CPUs in this example.

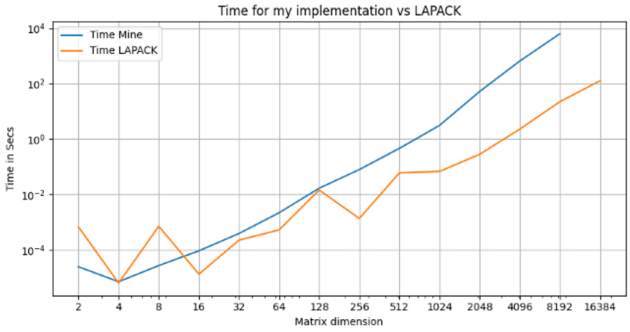

FIG. 5. Elapsed Time of LAPACK solver vs this implementation

One of the proposed next steps to get this time to execute closer to the BLAS/LAPACK libraries will be

1. Explore efficiencies in data structure and make sure the calculations leverage the CPU cache as much as possible to avoid paging and the excessive moving of blocks of memory from RAM to cache.

2. Explore a threaded approach to each of the different steps of the algorithm (LU decomposition and triangular solver) so we can get multiple blocks running in parallel. We will explore these kinds of approaches in section VII A.

Once we get closer to the time efficiency of the LAPACK and BLAS libraries, the next step will be optimizing the algorithm offloading some of the workloads into GPUs or making use of MPI and ScaLAPACK libraries in order to scale into multi CPU environments. We will see some of these techniques in sections VI, VII C and VII D.

4. *Behavior of Solver in Edge Cases*

In this section, we are going to input some edge cases matrices instead of the random matrices we have been working with. The purpose is to check the behavior of our solver and the LAPACK solver against these scenarios.

Two rows of the A matrix are similar. We will take the first row and copy it to the second row, summing a specific number of machine epsilons $\varepsilon_{machine}$. We expect the solution to diverge quickly as those rows get closer together, as soon as they are the same row we will not have a solution to this system of equations.

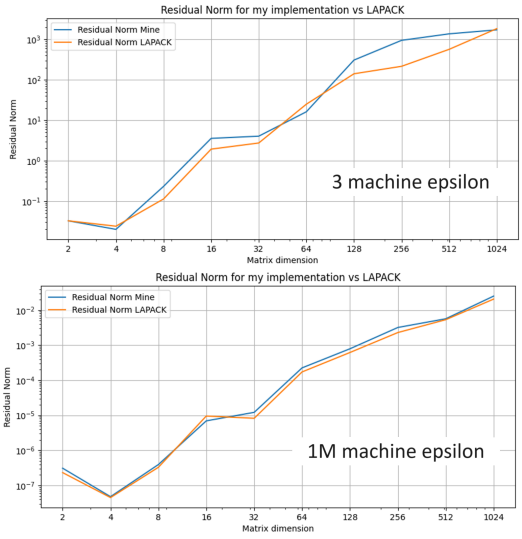

FIG. 6. Residual Error when row 1 and 2 diverge k·$\varepsilon_{machine}$

Note that we had residual errors of the order of 10^{-9} when we were dealing with 1024×1024 random matrices. the residual norm will increase to around 10^{-2} when the two rows are separated by a million machine epsilons, and it goes up to 10^3 when the rows are separated only by $3 \cdot \varepsilon_{machine}$. It is worth noting that the stability of both my solver and LAPACK implementation follow a very similar trend, with LAPACK implementation always having a minor difference of error to its favor.

Diagonal is scaled by N and N^2. In this case, we will multiply the diagonal by N or N^2 to see how the stability increases. It is well known that when the diagonal absolute value is higher than the rest of the matrix A, the error of the solution will be lowered. Let's see this in action in figure 7. As we noted before, we had residual errors of the order of 10^{-9} when we were dealing with 1024×1024 random matrices. Now with the random matrix diagonal scaled by N (multiplying each diagonal element by N) we will see that the residual error at that matrix size drops to 10^{-11} and when we scale further by N^2, the error drops to the orders of 10^{-13}. Again, as in the other edge case, we can see how the LAPACK implementation offers a better stability in this edge case, surpassing my implementation by at least one order of

magnitude when the matrix dimensions are higher than 1024×1024.

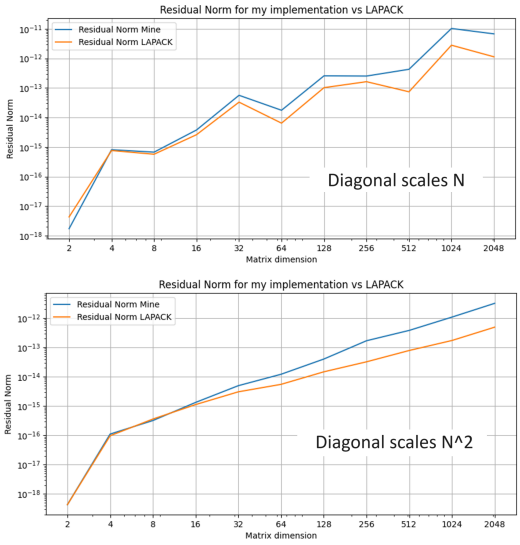

FIG. 7. Residual Error when diagonal scales as N and N^2

Poking holes into the matrix A diagonal. Also in the file `Basics\Errors\1.NormalizedFWDError.cpp` there is code to poke holes in the matrix diagonal. The results are not so dramatic in terms of stability difference as the other cases, but we can see that the residual error increases when we poke more and more holes into the diagonal. The code allows entering as parameter every how many elements we should poke a hole (write a zero) in the diagonal.

As a general conclusion we clearly see the expected trends on all the edge cases, and the LAPACK implementation wins against my implementation specially in the case of diagonal scaling for high matrix dimensions.

V. FOURIER TRANSFORM AND FFTW

In this section we are going to move away from the linear equation solving problem, and enter into a completely different problem which is the Fourier Transform.

The Fourier Transform is a mathematical tool that permeates the world of physics. In 1822 the French mathematician and physicist, Jean-Baptiste Joseph Fourier, declared in his book The Analytical Theory of Heat[7] that any function could be expanded as a sum of a series of sines. That means that if we have a representation of a signal though time, can also have a representation of the same information in a completely different form or domain, which will be the frequency domain. We can move from one domain (time) to another (frequency) and have different representations of the same signal without loss of information. This can also be applicable to space instead of time, we can have a representation of a signal in space to a signal in frequency. A good example of this is the how to capture the structure of a crystal (static in time) from space (e.g. how separated are the atoms) to a frequency domain.

Fourier transform continues to show up in many domains of physics and technology. One good example is the compression of data, nowadays all the video streaming and image compression is based on the same concept. It is more cost-effective to save information in the frequency domain than it is in the space domain. Take a picture and think every pixel like an atom in a crystal, we can do a representation in the frequency domain and only keep the frequency components that have a distinguishable power, so the eye can perceive them, disregarding the rest. Now we have to transmit or keep less information without losing the perceived quality of the image in question. The list of examples is too long to list, from audio processing, speech recognition, telecommunications to radio astronomy or quantum mechanics.

The main formula of a Fourier transform to move from time domain to frequency domain is:

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-j2\pi ft}\, dt \tag{44}$$

and the inverse from frequency to time is:

$$x(t) = \int_{-\infty}^{\infty} X(f) e^{j2\pi ft}\, df \tag{45}$$

It can be shown that both equations are reversible without any loss of information under some suitable conditions of the original signal. For example, the signal should be integrable in the first place. Let's think now of a discrete case like the atom crystal we spoke about before or the pixels in an image. How can we make this integration?, it is then when the Discrete Fourier Transform (DFT) comes to replace the integration:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi}{N}kn} \tag{46}$$

k in this case will be the discrete frequency domain, and N will be the size of the sample. This allows us to do Fourier transforms in a digital environment, first moving a continuous analog signal into a discrete digital signal and later applying the DFT, which is equivalent to the integration we saw in equation 44. Note that to calculate the Inverse Discrete Fourier Transform, we usually need to do a normalization based on the size of the sample. This detail will be critical when we heavily use the DFT because we can skip that normalization if we don't need it because we might only need to find qualitative details of the signal.

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{j\frac{2\pi}{N}kn} \tag{47}$$

Before we move into the exploration of actual software packages and implementations, we must mention the

Fast Fourier Transform (FFT) and the Discrete Cosine Transform (DCT). The Fast Fourier transform divides the problem of the Discrete Fourier Transform in smaller problems with a divide and conquer approach and moves from a complexity of $O(n^2)$ to a $O(n \cdot log(n))$. Cooley-Tukey[8] implementation of FFT published in 1965 is the most commonly used. It seems that Carl Friedrich Gauss's unpublished 1805 work on the orbits of two asteroids devised a method very similar to the FFT.

Discrete Cosine transform is a variant of the Discrete Fourier transform that uses only the cosine real components of the $e^{-j\frac{2\pi}{N}kn}$. The main advantage is obviously treating with real values instead of complex numbers, but the cosine component also have a stronger energy *compaction* properties[9], meaning that there will be less frequency components with high energy so we will be able to disregard more of the lower energy components, increasing the compression in applications for image and video compression very used in today's information society.

A. FFTW

FFTW is an open source implementation that stands for 'Fastest Fourier Transform in the West'[10], and was developed by the MIT in 1997. Latest implementation version is 3.3.10 which is also known as FFTW3. The source code and the documentation can be found in `fftw.org`. The main features that makes this implementation of the FFT very versatile and powerful are amongst others:

- Supports transforms with Complex and Real data formats.
- Manages even data for DCT or odd data for DST (Discrete Sine Transform).
- Works with one dimensional or multidimensional arrays for transforms, which makes it is easier to tackle problems like 3D spatial grids.
- Supports parallel computing implementations with MPI (Message Passing Interface) or thread models like OpenMP.

FFTW3 is making use of *'plans'* which are created ahead of the actual transform to get a sense of the hardware the code is running into, and the data types and sizes the transform will have to work with. That plan can be calculated once, and it's saved for running transforms more efficiently throughout the code.

1. Installation of Libraries

As in many other libraries, it is always more efficient and sometimes mandatory to download the source code and compile it in the target machine, instead of using pre-compiled libraries that are part of the operating system distribution. In case we need to make a quick exploration of the libraries, we could install them from the Ubuntu distribution using `sudo apt install libfftw3-bin libfftw3-dev`. Otherwise, when there is need for a more performing solution, it is always advised to download the source code from the home page and follow the usual installation steps of `make` and `make install`.

B. FFT implementations

In this chapter, we will be evaluating different implementations of the Discrete and Fast Fourier Transform. They can be found in `Basics\FFT\` directory of the GitHub repository, and these are the files:

- `4.fourier.cpp` DFT naive implementation.
- `5.FFT.cpp` Cooley–Tukey Fast Fourier Transform.
- `6.FFTw.cpp` FFTW3 implementation with Plan and plan execution.
- `7.FFTw3D.cpp` FFTW3 of a 3D Gaussian wave form and error calculations.

1. Naive DFT

The first implementation we will do is the naive discrete Fourier transform calculation based on the equation 46. In the file `4.fourier.cpp`, we take 128 points of the following example function and plot its transform:

$$y(\Delta t) = 10 \cdot \sin\left(\frac{2\pi\Delta t}{N}\right) + 3 \cdot \sin\left(5 \cdot \frac{2\pi\Delta t}{N}\right) + \\ + 2 \cdot \sin\left(10 \cdot \frac{2\pi\Delta t}{N}\right) \tag{48}$$

We can see in the figure 8 the example signal of the equation 48 and the module (absolute value) of its complex Fourier transform. As we will later see, this method of calculating the DFT is much more intensive in computing resources and consequentially slower. Let's remember that this DFT implementation has $O(n^2)$ complexity against the $O(n \cdot log(n))$ complexity of the FFT.

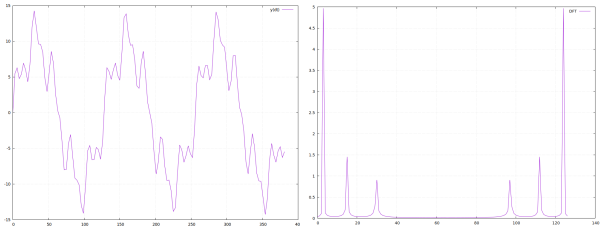

FIG. 8. Signal and its DFT

2. Cooley–Tukey FFT

This algorithm implementation of the fast Fourier transform is probably the most commonly used, and it was developed by J. W. Cooley and John Tukey in 1965. As we mentioned before, the complexity of this recursive algorithm is $O(n \cdot log(n))$. In the code shown in the file `5.FFT.cpp`, we can see how the complex signal is divided into even and odd terms and then the algorithm is recursively called for each one of the new signal with the function `fft()`.

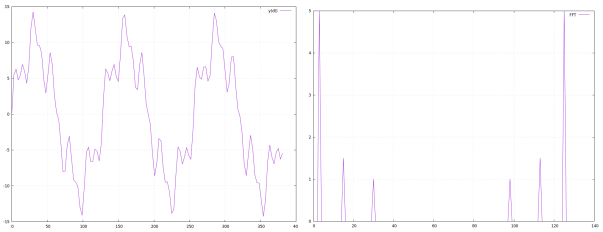

FIG. 9. Signal and its FFT

In the figure 9 we can see the result of the module of the FFT. The first difference we can notice from both DFT and FFT transform is the accuracy of the transform. In figure 10 it becomes more obvious how the FFT only has contribution to the exact frequencies where the signal is. To the right, the FFT transform shows abrupt peaks on that specific frequency in contrast to the DFT that we can see on the left, that shows some contributions related to frequencies that are not really there in the original signal. This is due to what is known as 'spectral leakage'.

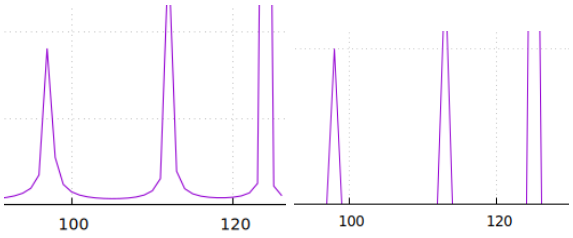

FIG. 10. DFT vs FFT and spectral accuracy

The DFT is expecting a periodic signal but note we only have 128 points and roughly 3 periods of the lower frequency. This abrupt stop of the signal (which the algorithms expect to be indefinite) produces the bleeding of some frequencies into the transform that should not be there if the signal were to be infinite. The solution to this is usually passing the signal through a window that removes the contribution at the extremes and enhances the middle part of it, or doing a conscious calculation of where to start and stop the signal so it falls exactly into multiples of the components.

3. FFTW3 Implementation and benchmark

To finish our exploration, we will do the same example signal transform with the FFTW3 algorithm that we talked about after proper installation of the libraries. In the code `6.FFTw.cpp` we create a plan and run the transform. We will see the accuracy is in line with the Cooley–Tukey FFT implementation, but we will see how the speed-up is the main gain in this specific case. The timing of the transform for the different implementations are:

- Naive DFT Time: 0.00111149 s.
- Cooley–Tukey FFT Time: 3.6402e-05 s. 30x faster than the naive DFT implementation.
- FFTW3 Time: 2.177e-06 s. 510x faster than the naive DFT implementation.
- Plan Creation in FFTW3 Time: 0.00492206 s.

As we can see, in FFTW3 we pay the one time price of a plan creation but once that is done, we see speed-ups of around 500x with respect to the naive implementation and around 15x speedup versus the Cooley–Tukey FFT recursive implementation. Also note that this benchmarking has been done with signals of 128 samples. With 8192 samples we see still the trend of 15x of FFTW3 vs Cooley–Tukey, but compared with the naive implementation we see speedups of 4000x as expected in the $O(n^2)$ vs $O(n \cdot log(n))$ complexity trend.

To finish the analysis of FFTW3, we will explore the implementation of a Fourier Transform of a 3D lattice. The file can be found in `Basics/FFT/7.FFTw3D.cpp`.We will create a plan with the function `fftw_plan_dft_3d()` of the FFTW3 library. The signal will be a three-dimensional Gaussian function. One of the properties of the Fourier transform is that a Gaussian function transforms into another Gaussian in frequency domain. Once calculated, we will do the inverse transform and calculate the error against the original function.

Note that we will need to do a normalization step after the inverse transform. This is done because the FFTW3 package by default does not apply any normalization, as we see in equation 47 with the $1/N$ term. In many applications the normalization is not needed, and we only need a qualitative analysis of the transform rather than an exact value of each of the frequency components. This saves resources and time in the FFT calculation. With the function `normalize()` we will divide every component by, N hence getting the original function. After that, we will calculate the error that has been carried over into the FFT and the inverse FFT operation.

VI. NVIDIA GPUS AND CUDA

A. Introduction to GPU and CUDA

Graphics Processing Units (GPUs) have evolved from their original purpose of rendering graphics in video games to become versatile computing tools, accelerating calculations in many areas such as physics simulations and artificial intelligence model training. GPUs have been offloading the burden of pure mathematical computation from CPUs in a more efficient and parallel manner. Today, an average GPU has hundreds of cores that can run computations simultaneously. As the applications of GPUs have expanded, their nomenclature has also evolved, now being referred to as GPGPUs or General-Purpose Graphics Processing Units. Additionally, there has been an evolution from GPUs to NPUs (Neural Processing Units), which feature hardware specifically tailored to the operations needed for machine learning in artificial intelligence computations.

We cannot forget the development of Digital Signal Processors (DSPs) almost at the same time as GPUs. DSPs were mainly focused on telecommunication applications, including signal filtering, Fourier transforms, and other matrix-based operations. GPUs saw a higher level of investment than DSPs with the advent of the video game industry. Additionally, due to the commonalities of instruction sets with the needs of scientific computing and AI, GPUs experienced stronger growth in the industry. Nowadays, the linear algebra operations that served ray-tracing applications in video games serve a different purpose in AI development, based on the same mathematical operations.

The main manufacturers of these GPUs are Nvidia, Advanced Micro Devices (AMD), and Intel Corporation. In this section, we will work with Nvidia GPUs, specifically the Quadro K1200, which has an architecture called "Maxwell." We have also conducted tests with the Nvidia Quadro P600, which features the "Pascal" architecture, yielding similar results.

The proprietary application programming interface developed by Nvidia for its GPUs is called CUDA (Compute Unified Device Architecture) and we will use its libraries and functions to access the capabilities of the cores of the GPU.

A single thread is the minimum computation object in the CUDA programming model, a group of threads is called a **block**[11] (note that there will be a maximum of 1024 threads per block). CUDA blocks are grouped into a grid. The execution of the kernel happens over a grid of blocks of threads, as we can see in figure 11. In the same figure, we can see how each of the execution model objects map to a specific hardware. Each CUDA thread will run in one of the GPU cores, and each block will be executed by one streaming multiprocessor (SM). For example, our hardware Quadro K1200 will have 4 streaming multiprocessors and each of those SMs will have its own L1 dedicated and a L2 shared cache (of 64KB and 2MB respectively). All the 4 SMs will share a common GPU memory of 4GB. These specifications will change from one GPU to another, for example, the Nvidia H100 GPU based on the 'Hopper' architecture has 14.592 cores, 114 SMs, L2 shared cache of 50MB and a shared memory of 80 GB.

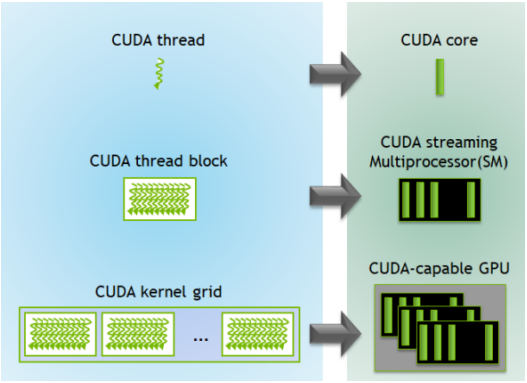

FIG. 11. CUDA Programming Model in GPU

B. Installation of Libraries and Tools

In this section, we will briefly describe how to install the libraries and the CUDA development kit of Nvidia into our target Linux operating system.

First, let's make sure that we have an NVIDIA GPU installed in our system with the command `lspci` or `lshw` piped with `grep NVIDIA`. We should see at least one entry for the GPU, in our case we can see:

```
"01:00.0 VGA compatible controller: NVIDIA Corporation GM107GL [Quadro K1200] (rev a2)"
```

Once we make sure that the device is discoverable, we will download and install the drivers following the instructions found in `https://developer.nvidia.com/cuda-downloads`. At the time of writing this capstone, the latest version is `CUDA 12.5` and the latest Nvidia driver for Ubuntu Linux platform is `555.42.06`. Once drivers are installed, we will be able to use the following commands to make sure we can detect and work with the GPU:

- `nvidia-detector` will show the version of the installed driver, in this case `nvidia-driver-555`.
- `nvidia-smi` will connect to the driver and show system information like the version of the driver, the CUDA sdk, the model of the GPU, the available and used memory, and others like the temperature of the GPU or the current processes making use of the GPU driver.
- `nvcc --version` will show the version of the CUDA compiler. `Cuda compilation tools, release 12.0, V12.0.140`.

- `nvtop` is the equivalent of `'top'` command for the CPU, showing the load of the GPU, the memory consumption and a graph on how those parameters evolve in time.

C. CUDA Examples and Benchmark

We are going to explore some examples of operations that will be offloaded to the GPU. The first one we explore is a matrix-matrix multiplication algorithm in `0.cudatest3._mm_mult.cu`. Right after that, we compare the CPU solver of a system of linear equations with LAPACK libraries against the CUDA libraries for the GPU. The code can be seen in `Basics\CUDA\4.cudavsCPUsingle.cpp`. In the following figure 12 we can see how at bigger matrix sizes, the GPU starts to outperform the CPU at least one order of magnitude in time. We cannot see visible differences in the accuracy, so the residual norm is very similar. We also repeat this experiment with double data types in `3.cudavsCPU.cpp`.

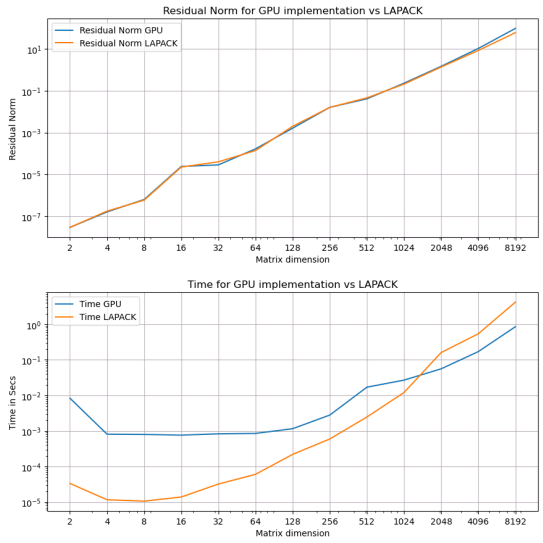

FIG. 12. GPU performance vs CPU with LAPACK solver

On the next figure 13 we can see the output of the `nvgtop` tool while we solve different sizes of system of linear equations in the GPU. In red we can see the GPU usage and in blue the GPU memory usage.

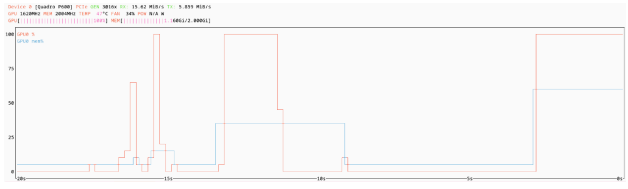

FIG. 13. nvtop output: GPU Solving system of linear equations

Finally, we tackle the problem of a vector reduction in a GPU. In particular, we are going to implement the summation operation over a large array of numbers to get one final scalar number. In the code shown in `6.Float.FiniteSum.cu` and `0.CUDA_functions.cu` we explore the progressive reduction of the sum of the vector to get a single number. We have used the sum of a finite number of terms of a well known sum (geometric series) to be able to compare the result of the reduction with the known sum.

$$\sum_{k=1}^{n} ar^{k-1} = \frac{a(1-r^n)}{1-r} \tag{49}$$

Executing the code we can plot the results of the speed of the GPU reduction vs the CPU reduction as the vector size, or number of addends grow. In the figure 14 we see how around vectors of size 2^{13} or 8192, the GPU starts to be more efficient than the CPU reaching around 8 times faster for 2^{20} elements in the sum and reaching a saturation at that point.

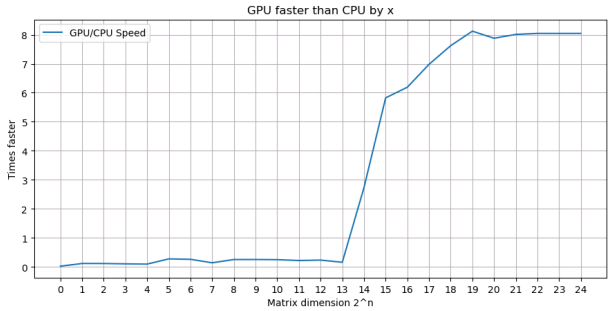

FIG. 14. GPU speed up of reduction vs CPU

VII. PARALLEL PROGRAMMING

When we have hardware platforms with several processors available, we have to start programming with some tools that will help us use those multiple processors. It might be the case of multicore processors or distributed systems. If we don't start using specific tools for parallel programming, we will be limited to a single processor, and we will not be able to scale our computation across all our available hardware.

A. Threads in C++

The most handy tool we can find at hand when we are programming in C++ is the `threads` library that is supported since C++11. These libraries at a very low level allows us to spawn or fork threads and control their execution with many tools like semaphores (that are a shared resource counters) or `mutex` directives that help us control our mutually exclusive code segments. We can

find some examples in the `LibTest` directory that work on the basics of this library.

This tools will allow the programmer the maximum level of control, but at the same time can become quite cumbersome to have to keep track of edge cases or racing conditions that can unexpectedly modify our code behavior.

In order to overcome this complexity, several other parallel programming frameworks have been developed throughout the years.

B. OpenMP

The OpenMP Architecture Review Board is a non-profit organization created in 1997 that is formed by multiples companies like AMD, Intel, IBM or Nvidia. They provide a portable and scalable model to develop parallel computing applications in a multiprocessor and shared memory environment. It is optimized to quickly move code to run in parallel architectures, but it sometimes lacks of the deep flexibility that threads library in C++ can offer. It is based on `pragma` directives that easily keeps the main single threaded code almost intact. The latest OpenMP application programming interface[12] is the version 5.2 and can be easily found in `https://www.openmp.org/`.

While the Threads library and OpenMP are more commonly used for single node servers with a multicore processor capabilities, there are other multiprocessor platforms that are used in larger scale HPC environments like networked clusters of several computing nodes or supercomputers. A clear example of this is the Message Passing Interface standard (MPI).

C. Message Passing Interface MPI

MPI is a protocol standard created to communicate between nodes in a parallel computing network. It is the predominant model in HPC architectures today and its first version of MPI1.0 was released in 1994 after a draft proposal[13] created by Jack Dongarra, Tony Hey, and David W. Walker. MPI standardization effort is today a global effort governed by the MPI forum and backed up by private corporations like Intel, IBM or Nvidia and also public institutions like the National Science Foundation (NSF) in US and the European Commission. MPI is the main communication protocol in all exascale computing projects today in US National Laboratories as well as European Supercomputing initiatives.

The main objects and functions in MPI are the following:

- **Communicators**. These are the 'channels' that MPI will use to send messages from one processor or node to another. These channels or Communicators usually map to different physical networks but can be used as a logical abstraction of different networks that run on the same underlying physical network (for the purpose of speed and heat reduction, these networks usually are optical networks in the context of the supercomputing clusters). The default communicator is called `MPI_COMM_WORLD`.
- **Rank**. Is the identifier of each of the processes. Rank 0 is usually the master or root process, and in most of MPI applications, its role is to coordinate the tasks, initiate broadcast messages or gathering results from other processes to consolidate final results.
- **Point-to-Point** Communication functions:
  - `MPI_Send`: Sends a message.
  - `MPI_Recv`: Receives a message.
- **Broadcast** Communication functions:
  - `MPI_Scatter`: Distributes data from one process to all the rest.
  - `MPI_Gather`: Gathers/Collects data from all processes towards one.
  - `MPI_Bcast`: Broadcasts a message from one process to all the rest.

Note that this model works with the assumption of dedicated memory segments per each process (distributed memory model versus a shared memory model used by other systems like OpenMP). The communication functions will move information from one memory segment to another, avoiding conflicts on a shared memory. They way this memory is implemented (dedicated or shared) will be up to the hardware implementation, and MPI will make sure there is an allocation reserved per each process. Nevertheless, MPI-3 introduces the explicit shared memory programming model.

There are several implementations of MPI, but the most used are:

- **OpenMPI**: An open source library used by many of the TOP500 supercomputers and was the merger of three MPI implementations from National Laboratories and Universities in US and Europe.
- **MPICH**: Also an open source implementation that serves as basis for other implementations like IBM MPI, Cray MPI or Microsoft MPI.

Both distributions can easily be found on Linux and other operating systems. During this Capstone, we have been working with the OpenMPI implementation. As we commented in previous chapters we can install precompiled distributions but in this case it is much better to download and the procedure for installation is the following. First, download the latest stable distribution from `https://www.open-mpi.org`. When writing this document, the latest version is 5.0.4. In the same URL you will find documentation that explain the steps to installation, but as a summary they are:

- Un-compress and un-tar the downloaded file. `tar xf openmpi-<version>.tar.bz2`.
- `cd openmpi-<version>`.
- Configure the packages with: `sudo ./configure --prefix=/usr/local 2>&1 | tee config.out`.
- Make the installation with: `sudo make all 2>&1 | tee make.out`.

Now OpenMPI should be installed in your system and to do a sanity check of the installation, follow the commands.

- `mpirun --version`: **mpirun** is the command that invokes the different processes in different computer nodes with the code that was compiled with mpicc. It should output the version of the installation, like: `mpirun (Open MPI) 5.0.2`.
- `mpicc --version`: should output the version of the compiler. mpicc is a link to the current compiler, in this case gcc, and the output should look like: `gcc (Ubuntu 13.2.0-23ubuntu4) 13.2.0`.

Under the `Basics\LibTest` directory we will find several simple examples of MPI to distribute, broadcast, gather, send and receive messages from processes, in both point-to-point and broadcast messaging. The C++ code will make use of the main functions of MPI: `MPI_Send, MPI_Recv, MPI_Bcast, MPI_Reduce, MPI_Scatter, MPI_Gather,` and give an example of each use case. For example, the Scattering and Gathering of data for the rest of processes can be found `7.MPIScatter.gather.cpp`.

D. SCALAPACK

ScaLAPACK, which stands for Scalable Linear Algebra PACKage[14], is a library of linear algebra routines based on BLAS and LAPACK that is designed for scaling in parallel distributed-memory cluster of computing nodes.

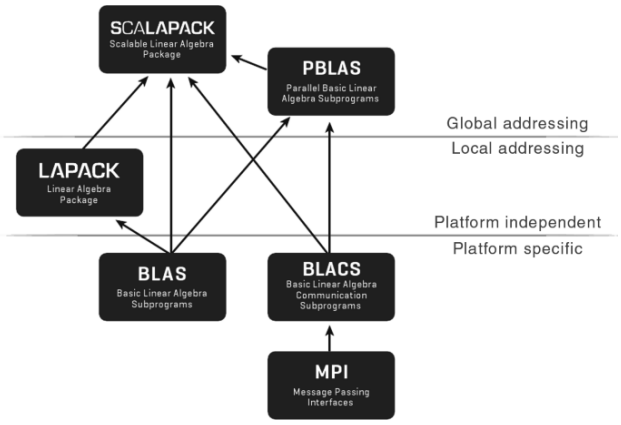

FIG. 15. ScaLAPACK relationship with BLAS, LAPACK, MPI and other libraries.

It is a freely-available software package that was first released as version 1.0 in February 1995. The latest version is 2.2.0, and can be found with its documentation in `https://www.netlib.org/scalapack/`. In the figure 15 we can see how ScaLAPACK relies on several libraries to operate:

- **MPI**: Message Passing Interface described in section VII C.
- **BLAS**: Basic Linear Algebra Subroutines described in section IV A.
- **BLACS**: BLACS (Basic Linear Algebra Communication Subprograms) library creates a common layer that will interface with MPI or any other message passing interface library that will be platform dependent. BLACS provides a layer of portability to ScaLAPACK.
- **LAPACK**: Linear Algebra Package described in section IV B.
- **PBLAS**: Parallel Basic Linear Algebra Subprograms (PBLAS[15]) is an implementation of Level 2 and 3 BLAS for distributed memory architectures. It is the computational basis in ScaLAPACK.

1. *Block Cyclic Data Distribution*

Two of the main ideas incorporated into ScaLAPACK are the use of **block cyclic** data distribution for dense matrices and the **block-partitioned** algorithms. When dealing with distributed systems and very large matrices in linear algebra, we need to think how to distribute those matrices efficiently, so each one of the processors can independently work on a segment of data of the whole problem. This has to be done efficiently to reduce the back and forth memory movement. The block cyclic distribution solves for this problem, and ScaLAPACK will work with the assumption that the main matrices have been distributed in this manner. The algorithms and routines of ScaLAPACK are in fact developed to operate with this distribution. Let's see first how this data distribution looks like.

Let's assume that we have a Matrix A of size 9×11 $(m = 9, n = 11)$ as the following that can be stored as column-major order in memory:

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} & a_{0,6} & a_{0,7} & a_{0,8} & a_{0,9} & a_{0,10} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} & a_{1,8} & a_{1,9} & a_{1,10} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} & a_{2,8} & a_{2,9} & a_{2,10} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} & a_{3,8} & a_{3,9} & a_{3,10} \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} & a_{4,8} & a_{4,9} & a_{4,10} \\ a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} & a_{5,8} & a_{5,9} & a_{5,10} \\ a_{6,0} & a_{6,1} & a_{6,2} & a_{6,3} & a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} & a_{6,8} & a_{6,9} & a_{6,10} \\ a_{7,0} & a_{7,1} & a_{7,2} & a_{7,3} & a_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} & a_{7,8} & a_{7,9} & a_{7,10} \\ a_{8,0} & a_{8,1} & a_{8,2} & a_{8,3} & a_{8,4} & a_{8,5} & a_{8,6} & a_{8,7} & a_{8,8} & a_{8,9} & a_{8,10} \end{pmatrix}$$

$$= \begin{pmatrix} a_0 & a_9 & a_{18} & a_{27} & a_{36} & a_{45} & a_{54} & a_{63} & a_{72} & a_{81} & a_{90} \\ a_1 & a_{10} & a_{19} & a_{28} & a_{37} & a_{46} & a_{55} & a_{64} & a_{73} & a_{82} & a_{91} \\ a_2 & a_{11} & a_{20} & a_{29} & a_{38} & a_{47} & a_{56} & a_{65} & a_{74} & a_{83} & a_{92} \\ a_3 & a_{12} & a_{21} & a_{30} & a_{39} & a_{48} & a_{57} & a_{66} & a_{75} & a_{84} & a_{93} \\ a_4 & a_{13} & a_{22} & a_{31} & a_{40} & a_{49} & a_{58} & a_{67} & a_{76} & a_{85} & a_{94} \\ a_5 & a_{14} & a_{23} & a_{32} & a_{41} & a_{50} & a_{59} & a_{68} & a_{77} & a_{86} & a_{95} \\ a_6 & a_{15} & a_{24} & a_{33} & a_{42} & a_{51} & a_{60} & a_{69} & a_{78} & a_{87} & a_{96} \\ a_7 & a_{16} & a_{25} & a_{34} & a_{43} & a_{52} & a_{61} & a_{70} & a_{79} & a_{88} & a_{97} \\ a_8 & a_{17} & a_{26} & a_{35} & a_{44} & a_{53} & a_{62} & a_{71} & a_{80} & a_{89} & a_{98} \end{pmatrix}$$

Now also let's assume we have 6 computing nodes each running a process, those nodes can be thought of being distributed in a 2×3 bi-dimensional setup called **process grid**. The process grid dimensions will be named as $(p = 2, q = 3)$. How would we distribute the matrix A among these 6 processes? First we have to define what is the size of the minimum continuous data block we will be working with, we call that size as $(mb \times nb)$ and in this case we will take the example of 3×2 so the dimensions of the block will be 6 elements $(mb = 3, nb = 2)$. We will be filling blocks of data in each process memory until we reach a limit, and we have to have contiguous data blocks smaller than $(mb \times nb)$. Let's see how this works and how we fill the processes that are named in the following matrix as (ip, iq) ip and iq being running in the range of the process grid p, q.

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,6} & a_{0,7} \\ a_{1,0} & a_{1,1} & a_{1,6} & a_{1,7} \\ a_{2,0} & a_{2,1} & a_{2,6} & a_{2,7} \\ a_{6,0} & a_{6,1} & a_{6,6} & a_{6,7} \\ a_{7,0} & a_{7,1} & a_{7,6} & a_{7,7} \\ a_{8,0} & a_{8,1} & a_{8,6} & a_{8,7} \end{pmatrix}_{0,0} \begin{pmatrix} a_{0,2} & a_{0,3} & a_{0,8} & a_{0,9} \\ a_{1,2} & a_{1,3} & a_{1,8} & a_{1,9} \\ a_{2,2} & a_{2,3} & a_{2,8} & a_{2,9} \\ a_{6,2} & a_{6,3} & a_{6,8} & a_{6,9} \\ a_{7,2} & a_{7,3} & a_{7,8} & a_{7,9} \\ a_{8,2} & a_{8,3} & a_{8,8} & a_{8,9} \end{pmatrix}_{0,1} \begin{pmatrix} a_{0,4} & a_{0,5} & a_{0,10} \\ a_{1,4} & a_{1,5} & a_{1,10} \\ a_{2,4} & a_{2,5} & a_{2,10} \\ a_{6,4} & a_{6,5} & a_{6,10} \\ a_{7,4} & a_{7,5} & a_{7,10} \\ a_{8,4} & a_{8,5} & a_{8,10} \end{pmatrix}_{0,2}$$
$$\begin{pmatrix} a_{3,0} & a_{3,1} & a_{3,6} & a_{3,7} \\ a_{4,0} & a_{4,1} & a_{4,6} & a_{4,7} \\ a_{5,0} & a_{5,1} & a_{5,6} & a_{5,7} \end{pmatrix}_{1,0} \begin{pmatrix} a_{3,2} & a_{3,3} & a_{3,8} & a_{3,9} \\ a_{4,2} & a_{4,3} & a_{4,8} & a_{4,9} \\ a_{5,2} & a_{5,3} & a_{5,8} & a_{5,9} \end{pmatrix}_{1,1} \begin{pmatrix} a_{3,4} & a_{3,5} & a_{3,10} \\ a_{4,4} & a_{4,5} & a_{4,10} \\ a_{5,4} & a_{5,5} & a_{5,10} \end{pmatrix}_{1,2}$$

The equations to go from (i, j) in the original matrix to (i_{ip}, j_{iq}) in the specific process sub-matrix will be the following:

$$\texttt{if } i_p = \left(\frac{i}{mb}\right) \mod p \texttt{ then } i_{ip} = \left(\frac{i}{p \cdot mb}\right) \cdot mb + i \mod mb$$
$$\texttt{if } j_q = \left(\frac{j}{nb}\right) \mod q \texttt{ then } j_{iq} = \left(\frac{j}{q \cdot nb}\right) \cdot nb + j \mod nb.$$

It is important to have a good handle on how to go back and forth from the original matrix to the distributed Block Cyclic data and backwards in order to be able to make use of ScaLAPACK in a distributed system. We will combine the MPI libraries in ScaLAPACK to send each of these matrices to the owning process. This will become critical when we are faced with problems involving very big matrices that need to be processed with linear algebra.

2. Installation of Libraries

As we can see in figure 15, it is extremely important to have the BLAS and MPI libraries compiled for your specific platform. In order to do that, we will follow the installation sections of OpenMPI in section VII C and the installations instructions of OpenBLAS that we saw in the section IV D. Once that is done, we will proceed to the following steps:

- Download the latest ScaLAPACK distribution (at the time of this writing is 2.2.0) from the URL: `https://www.netlib.org/scalapack`.
- Then run the `gunzip` and `untar` commands for the file you downloaded into a directory.
- Make sure you have the FORTRAN compiler, if not, install it with `sudo apt install gfortran-14`.
- Enter the directory where the source code of ScaLAPACK is and edit the file `SLmake.inc`. Make sure `FC` and `CC` point to the compilers in your machine. If you already installed OpenMPI you should be able to point those to `mpif90` and `mpicc` respectively.
- Add the `legacy` flag to the `FCFLAGS`. It should look like: `FCFLAGS = -O3 -std=legacy`.
- Finally point `BLASLIB` and `LAPACKLIB` to the OpenBLAS library, in my case I did install those on `/usr/local` so the path looks like `BLASLIB = /usr/local/lib/libopenblas.a`.
- You should be ready to compile the ScaLAPACK library now running `make lib`.
- That will output a file named `libscalapack.a` in your working directory, its size should be around 10 MB.
- To finish, you should link the library you created to the path where your libraries reside. In the case of my installation the command is: `sudo ln -s ~/scalapack-2.2.0/libscalapack.a /usr/lib/x86_64-linux-gnu/libscalapack.a`

It is very important to note the need of compiling both OpenMPI and OpenBLAS from the source code to your specific hardware in order for ScaLAPACK to compile correctly. Avoid using pre-compiled libraries with `sudo apt install`.

3. ScaLAPACK Examples and Benchmark

In the directory `Basics\PBLAS` we find examples of ScaLAPACK. One of the main examples is the scattering and collection code `4.Scatter_Collect.cpp` that takes n, m, nb and mb as inputs which are the size of the original matrix and the blocks. The size of the process grid will be given by the number of processors used for the calls with MPI. Note that we will call this code with `mpirun -np 4 ./4.Scatter_Collect <n> <m> <nb> <mb> 1`, in this case the grid will be composed by 4 processors which are the 4 cores available in my multiprocessor CPU, and we will have a process grid of 2×2.

The other interesting example that will help us to compare performances will be `5.Multiplication.cpp` that will do a scattering of two matrices into the four processes and then will do the multiplication in each process and collect the results. The code will be comparing the results and the timing with a non-distributed multiplication of the original matrix with `dgemm` in BLAS.

We will be able to try this multiplication with ScaLAPACK for different matrix sizes and different block sizes. For example:

```
mpirun -np 4 ./5.Multiplication 1024 32 2
```

Will do the multiplication of two matrices of size 1024×1024 with a block size of $nb \times mb = 32 \times 32$. The result is the following:

```
Cblas dgemm time:0.0490477 sec.
PDGEMM time:0.024514 sec.
Collect time:0.0695604 sec.
Speed-UP:2.00081x.
Collected FWD Error:  0.00111334
Collected residual Error:  1.4135e-12
```

As we change the size of the original matrix and the size of the block, we will notice how the speed-up changes with respect to the BLAS implementations. It is also worth noting that there is a cost of memory transfer when we do the distributed Block Cyclic split of the matrix for different processes, this cost will be minimized when we perform multiple operations on the same set of data in the process. The more processing we do on those data segments in a separate process before collecting, the more efficient ScaLAPACK will be compared to a BLAS implementation that runs directly in the initial set of data of the memory.

As a next step, we should be aiming to run this programs in multiple systems instead of one single node with a multicore processor. The real power of ScaLAPACK and MPI lies on the communication of different computing nodes in a distributed networked environment.

E. SLATE as replacement for ScaLAPACK

SLATE (Software for Linear Algebra Targeting Exascale)[16] is a joint project between the U.S. Department of Energy's Office of Science and National Nuclear Security Administration (NNSA). It started as part of the Exascale Computing Project (ECP) which was launched in 2016, and the first release SLATE 1.0 happened in 2019. It is written in C++ in contrast with ScaLAPACK (written in FORTRAN following LAPACK), is set to replace ScaLAPACK for exascale systems. The main reason of the development of SLATE is the difficulty of ScaLAPACK adapting to the modern GPU-accelerated architectures. More information can be found in its main web page: `https://icl.utk.edu/slate/`.

VIII. SIMULATION OF A CLASSICAL SYSTEM: GRAVITATIONAL SYSTEM

In this section, we will explore the computational stability of a simple classical physics problem. The two body gravitational orbit, and specifically we will use the parameters of the Earth and the Sun. Even if we should be considering this a three-dimensional problem, according to Bertrand's theorem[17], the inverse-square central forces (as the gravitational or electrostatic) are one of the two kind of forces (the other one is the radial harmonic oscillator potential) that can derive into closed orbits. The fact that we will have closed circular or elliptical orbits means that we can move the problem from 3 dimensions to 2 dimensions, which will simplify the simulation even further. We will only have to simulate the coordinates x and y and forget about z.

We will take the parameters of the earth orbit around the sun, and we will decrease the initial velocity of the earth so it can fall closer to the sun with higher speed and acceleration. As the derivatives of the equations of motion have larger slopes and as the slopes change faster with respect with the time intervals, we will see how the accuracy of the results start to decrease, giving out erroneous results. The code can be seen in the python [1] simulation named `Python/5.3.EarthSun.Stability.py`.

In each step of the problem, we will be calculating the acceleration based on the Gravitational equation:

$$F = m_1 \cdot a_i = G \cdot \frac{m_1 m_2}{r^2} \tag{50}$$

We will be calculating step by step what the acceleration should be in the new condition and deriving the velocity from that calculated acceleration.

$$v = \frac{dx}{dt} \quad ; \quad a = \frac{dv}{dt} \tag{51}$$

$$v_{i+1} = v_i + a_i \cdot dt \quad ; \quad x_{i+1} = x_i + v_i \cdot dt \tag{52}$$

This is the known and simplest first order Euler method to solve differential equations. But the first question we have to answer in this kind of first order approximation is what is the size of the step or the time interval to ensure our calculated solution does not diverge from the real solution. We need to ensure the stability of the calculation of the problem.

A. Propagation of errors and problem stability

In 1928, Richard Courant, Kurt Friedrichs, and Hans Lewy wrote a paper [18] discussing the numerical stability of the differential equation solutions in the context of computation. This mathematical convergence is known as the CFL stability condition and states how small the step or the interval in the calculation should be. The upper bound to this time step is called Courant number, and it is given by:

$$C = \frac{v\,\Delta t}{\Delta x} \leq C_{\max} \tag{53}$$

With this equation we can decide what the time step will be for our problem to optimize our computing resources without moving too far from the expected results. This problem can also be stated as the boundary value problem for the 2 body differential equation, also known as the Lambert's problem. There is an extensive literature [19] on the matter, but we won't get in depth in this document. Going to the python directory in the GitHub repository[1] `Python/5.3.EarthSun.Stability.py` we will find the simulation and see how the initial conditions of the velocity and the proximity to the sun, will condition the stability of the problem. For example, with an initial velocity of 29.7 km/s, the simulation of the earth will continue an orbit very similar to the real orbit without apparent lost of stability for a long simulation time. When the initial velocity starts to be reduced to 12 km/s we start to see how the simulation of the more elliptical orbit starts to see a precession that is not expected under the classical differential equation.

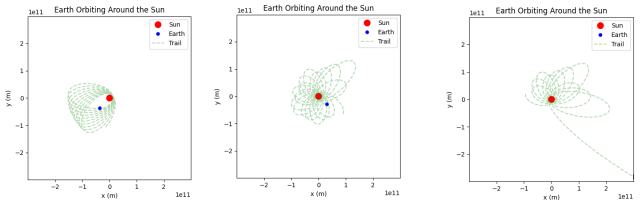

FIG. 16. Simulation of the Earth Orbit for 3 different initial velocities

The first of the three simulations in the figure 16 corresponds to 12 km/s as initial velocity, and we can clearly see the precession that is the result of the lack of precision of the calculations specially when the earth is near the sun (where the rate of change of velocity and position is the biggest). In the other two simulations the initial velocity is 10 km/s and as the earth gets nearer to the sun we can see how the orbit not only precesses but also does so in an irregular manner with different distances to the sun and at some point being expelled from the orbit.

B. 4th order Runge-Kutta method

In order to improve the stability of the problem without changing the size of the time interval, we are going to use the Runge-Kutta Method [20]. In a nutshell, this method will calculate the slope of the curve in three points instead of one. Runge-Kutta usually makes reference to the 4^{th} order approximation (aka RK4), but the method can be expanded to the n^{th} order, adding more points to slope calculation inside the interval.

The first point in Figure 17 is the point in question for the current step that we used in the first order Euler approximation. The second point is the midpoint of the interval, here we will calculate two different slopes k_2 and k_3. One of them (k_3) being calculated based on the slope we got in k_2. The last slope is calculated based on the end point of the interval. Having now the 4 slopes, we will have a weighted average, with the two slopes in the middle having double the weight of the slopes in the extremes of the interval h (also mentioned in the code as dt).

$$v_{n+1} = v_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{54}$$

The method is especially stable for what are called stiff differential equations[21] which are particularly prone to lack of stability and require an extremely small discretization criterion, significantly increasing the computing requirements.

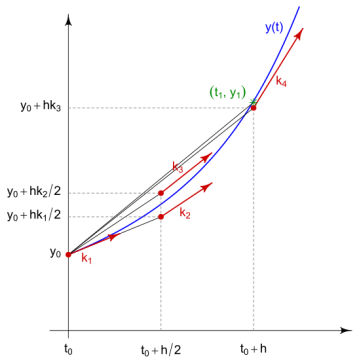

FIG. 17. Slopes of Runge-Kutta (RK4) Method

When we apply RK4 to the Earth-Sun orbital problem, we indeed eliminate the precession effect from the errors, but we find that the orbit distance is progressively reduced, especially for orbits very close to the Sun with high speeds. In figure 18 we can appreciate this result.

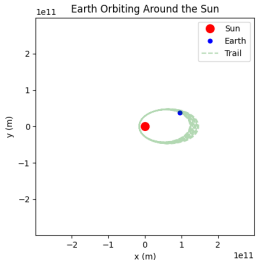

FIG. 18. Simulation of the Earth Orbit with RK4

C. Adaptive methods

Finally, we will resort to a basic adaptive method [22], in which we will modify the time interval depending on the rate of change of the variable we are willing to estimate. In this case and for this simulation, we will make

the step inversely proportional to the current speed of the Earth:

```
dt_new = dt_initial / (np.linalg.norm(v)/(v_earth))
```

This change will, firstly, affect the simulation computing resources that will not be well bounded now and will depend on the nature of the problem and the initial conditions. The great advantage of this kind of adaptive step is that the stability will be extremely high, and we will be running the simulation for hours without a visible degradation in the orbit. To test the stability, we took the simulation to very low initial velocities, like 4 km/s, which resulted in a very high orbit eccentricity.

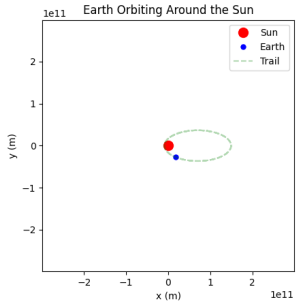

FIG. 19. Simulation with Adaptive RK4

In the figure 19 we can see the result of the simulation and how the simulation stays stable under many computation run cycles.

IX. QUANTUM COMPUTING

The journey of exploring scientific high-performance computing tools and hardware has taken us to the current limits of today's computing capabilities. Even Gordon Moore[23] estimates that Moore's Law might find its end by 2025. What is clear is that there are physical limits on how small the transistors can be made to implement Boolean logic gates. That physical limit is actually given by the size of atoms or molecules that are the building blocks of any electronic component. The technological advances will most likely try to reduce and optimize power consumption on the already very small scale. The so-called '2 nm' scale technology was achieved in 2024 by IBM. Note that, in fact, the smallest metal etching is 20 nm wide, so the 2 nm naming is just a marketing category. That being said, the silicon crystal lattice number is 0.54 nm, so we are getting near. The other possibility that we have at hand is breaking the Boolean logic paradigm and moving to a different mathematical model of calculation that could help speed up our computation capabilities.

A. Introduction to Quantum Computing

The idea of using the laws of quantum mechanics as the basis for computing is a very recent idea, starting in the 1980s. In fact, we find one of the first references to quantum computing in an article by Richard Feynman[24]. The term qubit appeared for the first time only in 1995[25]. Since then, there has been an explosion in the study of information theory, communication, and quantum computing that is still growing today.

Recent theoretical advances in quantum information over the past few decades have occurred in parallel with the revolution in classical computing. We are currently at a point where different promising technologies could become the basis for a future quantum computer capable of helping humanity solve computational problems that previously seemed out of our reach with the limits we already see in classical computing.

The greatest challenge that the scientific community faces is to create a system that can store a number of stable qubits (maintaining entanglement between them) for a sufficient amount of time to be useful in carrying out the operations intended to be performed on them. This time is limited by what is known as the coherence time of quantum information. Any external disturbance, from a single photon entering the quantum system, can ruin the information processing. Let us remember the double-slit experiment and how the observation of the particles (photons or electrons) destroyed the interference pattern on the screen. To avoid this decoherence effect, current systems that implement a qubit are usually extremely isolated from the environment and/or cooled to very low temperatures.

While hardware researchers continue to search for the holy grail of a qubit immune to external noise that allows for the implementation of a universal quantum computer, today we have to live with a few qubits that at the same time have a very low coherence time. This limitation has led the community to devise computing methods that are compatible with this uncomfortable situation. This is what is known as the NISQ era (noisy intermediate-scale quantum).

One of the most promising applications of quantum computing and this method is the simulation of quantum physics systems, including quantum chemistry. If we want to simulate the quantum state of the electrons in a molecule, we must take into account the amplitudes of each of the combinations of the number of electrons. These amplitudes are complex numbers, and the number of different amplitudes in a system scales like 2^n, with n being the number of electrons. To simulate 50 electrons, we would need 2^{50} or about 10^{15} bytes (a petabyte) assuming we store that amplitude with just one byte. We quickly realize how, with the current capabilities of a classical computer, it becomes extremely challenging just to store the data, let alone begin to think about the necessary data processing capabilities. We are talking about 50 electrons and a series of simplifications such as the

consideration of fixed point-like nuclei.

Quantum computing and the new computing paradigm can come to the rescue, since a qubit will allow us to directly simulate a particle or at least one property of a particle, so the problem will not scale as $O(2^n)$ but as $O(n)$. Other quantum systems that could be simulated include the interaction of quarks within the nucleus and quantum chromodynamics. However, quantum chemistry is receiving the most attention because of the number of applications that can be applied in the short term to the advance of material technologies. The most common examples are understanding chemical processes that we do not fully comprehend, such as the fixation of atmospheric nitrogen performed by bacteria through catalysts, the simulation of photosynthesis, or the simulation of superconducting materials at room temperature, to name a few. We must also not forget the impact that simulations of nuclear processes could have on the faster and more efficient development of nuclear fusion reactors that seem to be the only future for a sustainable energy source on our planet. There is no doubt that once we have the ability to harness the computing power of the quantum world, humanity will be able to advance in the fields of energy, transportation, health, technology, and in other scenarios that we probably cannot imagine at present.

B. HHL Quantum Algorithm

In this section, we will introduce a proposed quantum algorithm to solve a system of linear equations. The algorithm was proposed in 2009[26] by Harrow, Hassidim, and Lloyd.

The algorithm will help estimate a scalar measurement over the solution vector, and will not give the solution of the vector itself. One of the first demonstrations of the algorithm appeared in 2018, applied to Machine Learning[27], further feeding the promise of the application of quantum computing in the Artificial Intelligence arena. The quantum algorithm promises a speedup of $O(\log(N)\kappa^2)$ compared to the $O(N\kappa)$ that we can see in the fastest classical algorithm, with κ being the condition number of the linear system given by the following equation.

$$\kappa(A) = \|A^{-1}\| \, \|A\| \geq \|A^{-1}A\| = 1 \tag{55}$$

The basis of the HHL algorithm[28] is extracting the values of the solution x based on the initial values of b. The problem assumes we have a Hermitian matrix A. If that is the case, we can follow:

$$A = \sum_{j=0}^{N-1} \lambda_j |u_j\rangle\langle u_j|, \quad \lambda_j \in \mathbb{R} \tag{56}$$

$$A^{-1} = \sum_{j=0}^{N-1} \lambda_j^{-1} |u_j\rangle\langle u_j| \tag{57}$$

$$|b\rangle = \sum_{j=0}^{N-1} b_j |u_j\rangle, \quad b_j \in \mathbb{C} \tag{58}$$

$$A|x\rangle = |b\rangle \rightarrow |x\rangle = A^{-1}|b\rangle = \sum_{j=0}^{N-1} \lambda_j^{-1} b_j |u_j\rangle \tag{59}$$

In equation 56 and 57 we can see the decomposition of the hermitian matrix A and its inverse. We see that $|x\rangle$ will be represented as a linear combination of the coefficients of $|b\rangle$.

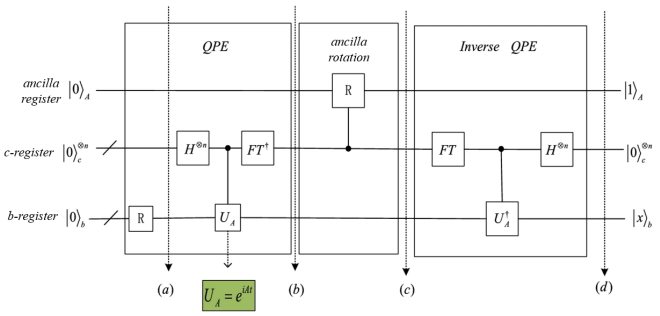

FIG. 20. Simplified version of HHL Quantum Circuit[29]

In order to achieve this transformation, the Algorithm takes 3 inputs:

- The b-register will have n_b qubits that will represent the $N_b = 2^{n_b}$ values of the vector $|b\rangle$.
- The c-register will have n qubits that will represent the $N = 2^n$ values of the vector $|b\rangle$. The number of qubits in this register will define the precision of the calculation.
- The a-register will be a single ancilla qubit that will rotate with the R_y gate.

Those 3 registers of qubits will be passed through a quantum circuit that will do the following:

- State preparation of b-register according to the input of the problem.
- Forward Quantum Phase Estimation Algorithm that will estimate the phase of the eigenvalues of e^{iAt} and is composed of 3 components:
  - Hadamard gates to enable the qubit superposition of the c-register
  - Controlled rotation of the b-register with the c-register as input. This is where data of the Hermitian Matrix lives.
  - Inverse Quantum Fourier Transform over the c-register.
- A Rotation of the ancilla qubit using the c-register as input.

- Inverse Quantum Phase Estimation (with the inverse circuits mentioned above)
- Measurement of all the registers.

Due to the mathematical nature of the algorithm, we will drop the results if the ancilla qubit measurement results in a '0' and will keep values as valid if the ancilla result is '1'. A reference implementation of the algorithm can be found on the IBM Qiskit GitHub site[30].

As a final note, this algorithm promises quantum speedup, but there are many conditions that have to be met in order to achieve it. Amongst these are the condition number we mentioned in the beginning, also, with a matrix that is not sparse and not 'well-conditioned', the speedup might not be as expected. Another cost that has to be accounted for is the state preparation of $|b\rangle$. All of these caveats are explored in detail in a paper authored by Scott Aaronson in 2015[31].

X. CONCLUSIONS

In this paper, we have discussed several software and hardware tools used in High Performance Computing and have been experimenting with the advantages and limitations of each one. We have seen how HPC can be applied to basic linear algebra and matrix algebra problems that form the basis of many of the mathematical foundations for solving physics problems in theoretical physics, such as quantum mechanics.

For example, the density functional theory (DFT) is a computational model for quantum mechanics that is extensively used to simulate many bodies in molecular or atomic physics and constructs the potential as a sum of the contributions of the different forces of the particles. For example, in simulating molecules, it gets the summation of the contributions of the effective potential of each electron in what is known as the Kohn-Sham[32] equations. This DFT model heavily relies again on linear algebra, as the Hamiltonian will ultimately be a matrix and there will be a need to get a diagonalization. Also, the density matrix will be processed via linear algebra and will represent the state of the broader quantum system.

The field of HPC is always evolving, and even when we are reaching the limits of Moore's law, more and more advances are made in distributed systems that can quickly contribute to scaling the computing power thanks to the addition of fast optical networks and dedicated hardware like the GPGPUs. Every year, Top500[33] publishes the list of the world's top 500 non-distributed supercomputers. This June 2024, we are starting to see private companies like Microsoft and Nvidia entering the top 10 supercomputers that are reaching exascale computing.

Exascale computers are the ones that can achieve exaFLOPs or 10^{18} double-precision (64-bit) operations (multiplications and/or additions) per second. Exascale computing will enable even more powerful physics simulations, allowing the science community to tackle previously intractable problems.

It has been more than 70 years since the first simulations achieved in 1953 by Enrico Fermi, John Pasta, Stanislaw Ulam, and Mary Tsingou[34] of vibrating string that included a non-linear term. That computer was called MANIAC I (Mathematical Analyzer Numerical Integrator and Automatic Computer Model I) in Los Alamos National Laboratory, and it is estimated that it could achieve from 500 to 1000 Floating Point Operations per second (FLOPS). We clearly have come a long way with computers that can reach 10^{18} FLOPS, but the development and research to increase computing power continues. The next qualitative jump might be hidden in the same quantum laws of nature, and it might be that, as Richard Feynman proposed, the simulation of a quantum world will be better done with a computer whose components are quantum and the basic logical and mathematical basis are not the bits and the Boolean algebra but the qubits and the linear algebra rules that guide the superposition and entanglement principles.

In conclusion, both HPC in classical computing and the advancement of quantum computing will grow in parallel and possibly complement each other to offer humanity computing power that we could not imagine some decades ago. This progress will undoubtedly boost the advance in scientific discovery in theoretical physics, enabling simulations when the experiments are too expensive or impossible to run with current technology.

Appendix A: List of Directories in GitHub

In the URL `https://github.com/ulitoo/HPC583` we will find the repository with different directories that contain the code for this Capstone. In summary, under the `Basics` directory we will find:

- `LibTest`: Testing C++ Libraries for CUDA, Threads, MPI, FFTW3, BLAS, LAPACK, ScaLAPACK etc.
- `BLAS`: Linear Algebra implementations of Matrix multiplication and Solvers and comparison with BLAS and LAPACK in Error and Execution time.
- `Errors`: Plot of Results with `matplotlib` of different Matrix sizes for implementations of Linear Algebra Solvers.
- `CUDA`: Test implementations of linear algebra in Nvidia GPUs.
- `FFT`: Implementations of Fourier Transforms and tests with FFTW3.
- `PBLAS`: Implementation of Linear Algebra with ScaLAPACK.
- `Python`: Other test implementations using Python.

[1] J. Barrientos, UW Masters Degree Capstone PHYS 600, https://github.com/ulitoo/HPC583 (2024).

[2] R. Courant, Variational methods for the solution of problems of equilibrium and vibrations, Bulletin of the American Mathematical Society **49**, 1 (1943).

[3] K. Goto and R. Van De Geijn, High-performance implementation of the level-3 blas, ACM Trans. Math. Softw. **35**, 10.1145/1377603.1377607 (2008).

[4] M. Kroeker, OpenBLAS Libraries, Github URL https://github.com/OpenMathLib/OpenBLAS/releases (2024).

[5] J. Schur, Über potenzreihen, die im innern des einheitskreises beschränkt sind., Journal für die reine und angewandte Mathematik **1917**, 205 (1917).

[6] E. Carrier, Computer Science Course 357, University of Illinois (2020).

[7] J. B. J. Fourier, Theorie analytique de la chaleur, F. Didot, Paris (1822).

[8] J. Cooley and J. Tukey, An algorithm for the machine calculation of complex fourier series, Mathematics of Computation **19**, 297 (1965).

[9] N. Ahmed, T. Natarajan, and K. Rao, Discrete cosine transform, IEEE Transactions on Computers **C-23**, 90 (1974).

[10] M. Frigo and S. G. Johnson, The design and implementation of FFTW3, Proceedings of the IEEE **93**, 216 (2005), special issue on "Program Generation, Optimization, and Platform Adaptation".

[11] NVIDIA, Cuda c++ programming guide, NVIDIA Corporation https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (2024).

[12] OpenMP Architecture Review Board, OpenMP application program interface version 5.2 (2021).

[13] D. W. Walker, Standards for message-passing in a distributed memory environment, CRPC work-shop (1992).

[14] J. Dongarra and P. Luszczek, Scalapack, in *Encyclopedia of Parallel Computing*, edited by D. Padua (Springer US, Boston, MA, 2011) pp. 1773–1775.

[15] J. Choi, J. Dongarra, and D. Walker, Pb-blas: a set of parallel block basic linear algebra subprograms, in *Proceedings of IEEE Scalable High Performance Computing Conference* (1994) pp. 534–541.

[16] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, Slate: design of a modern distributed and accelerated linear algebra library, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19 (Association for Computing Machinery, New York, NY, USA, 2019).

[17] F. C. Santos, V. Soares, and A. C. Tort, An English translation o Bertrand's theorem, arXiv e-prints , arXiv:0704.2396 (2007), arXiv:0704.2396 [physics.class-ph].

[18] R. Courant, K. Friedrichs, and H. Lewy, Über die partiellen differenzengleichungen der mathematischen physik, Mathematische Annalen **100**, 32 (1928).

[19] A. Albouy, Lambert's theorem: Geometry or dynamics?, Celestial Mechanics and Dynamical Astronomy **131**, 10.1007/s10569-019-9916-2 (2019).

[20] C. Runge, Ueber die numerische auflösung von differentialgleichungen, Mathematische Annalen **46**, 167 (1895).

[21] E. Hairer and G. Wanner, Stability of multistep methods, in *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems* (Springer Berlin Heidelberg, Berlin, Heidelberg, 1996) pp. 240–249.

[22] G. Söderlind and L. Wang, Adaptive time-stepping and computational stability, Journal of Computational and Applied Mathematics **185**, 225 (2006), special Issue: International Workshop on the Technological Aspects of Mathematics.

[23] S. Kumar, Fundamental limits to moore's law (2015), arXiv:1511.05956 [cond-mat.mes-hall].

[24] R. P. Feynman, Simulating physics with computers, International Journal of Theoretical Physics **21**, 467 (1982).

[25] B. Schumacher, Quantum coding, Phys. Rev. A **51**, 2738 (1995).

[26] A. W. Harrow, A. Hassidim, and S. Lloyd, Quantum algorithm for linear systems of equations, Physical Review Letters **103**, 10.1103/physrevlett.103.150502 (2009).

[27] Z. Zhao, A. Pozas-Kerstjens, P. Rebentrost, and P. Wittek, Bayesian deep learning on a quantum computer, Quantum Machine Intelligence **1**, 41–51 (2019).

[28] A. Zaman, H. J. Morrell, and H. Y. Wong, A step-by-step hhl algorithm walkthrough to enhance understanding of critical quantum computing concepts, IEEE Access **11**, 77117–77131 (2023).

[29] M. Zhang, L. Dong, Y. Zeng, and N. Cao, Improved circuit implementation of the hhl algorithm and its simulations on qiskit, Scientific Reports **12**, 13287 (2022).

[30] Raghav-Bell, Solving linear systems of equations using HHL and its qiskit implementation, Github URL https://github.com/Qiskit/textbook/blob/main/notebooks/ch-applications/hhl_tutorial.ipynb (2024).

[31] S. Aaronson, Read the fine print, Nature Physics **11**, 291 (2015).

[32] W. Kohn and L. J. Sham, Self-consistent equations including exchange and correlation effects, Phys. Rev. **140**, A1133 (1965).

[33] TOP500, 63rd edition of the top 500 supercomputers, june 2024, TOP 500 (2024).

[34] T. Dauxois, Fermi, Pasta, Ulam, and a mysterious lady, Physics Today **61**, 55 (2008).