Engineering Master Degree

Major in Scientific Computing Methods & Applied Mathematics

Evaluation of novel, high-level programming frameworks for exascale HPC architectures and their application in numerical plasma physics.

Victor Artigues

MACS 3, Sup Galilée, Université Paris 13

Supervisors:

Dr. Markus Rampp and Dr. Klaus Reuter

Max Planck Computing and Data Facility

Dr. Katharina Kormann

Max Planck Institute for Plasma Physics

September 21, 2018

Contents

1 Introduction

The ever-increasing capabilities of supercomputers allow scientists to compute simulations faster and run larger and more realistic cases. Optimizing such applications requires a knowledge of the computer architecture often resulting in specialized codes, sections of codes, or kernels which are designed to run extremely well on one or only few specific architectures.

To avoid deteriorating the code, not in terms of performance, but of maintainability, there is a need to improve code portability. For real world applications, when starting a project, the target architectures may not be completely known yet, not available or not fully specified. Very often codes are developed on a desktop computer or small cluster before being run on supercomputers. This would represent a case where the end architecture is known, but not accessible yet. Now speaking in terms of architecture changes during the development of a code, there are a lot more issues. Codes developed over the last 10 years have seen drastic changes in CPU, GPU and accelerator architecture. The Advanced Vector Extension, Intel's AVX, capable of performing vector operations on CPUs was announced in 2008. After AVX 256-bit, modern CPU are now using AVX-512 (from 2016). Intel's Xeon Phi processors have appeared in the last 10 years and seem to be loosing momentum recently. On the GPU side, NVIDIA has created 6 different architectures since 2008, from Tesla to the Volta architecture. Roadrunner[1], the supercomputer ranked 1 in 2008 has less than 1% of Summit's[2] Rmax performance, the number 1 supercomputer in 2018.

New architectures, such as GPUs, may be added to an existing cluster. Today, changing a code to run on a different architecture may mean rewriting most of it. After months or even years of development, one or more person has to go over the entire code and optimize it for the new architecture. Such tasks can take several months or more for a full time developer.

For all these reasons, there is a need of code abstraction from the hardware. One would benefit a lot from a reduction of the amount of work needed to port a code to a different architecture. This would increase performance reliability and reproducibility, as the architecture-specific kernels would be written and validated independently from the user's work, e.g. as a library. And the savings in terms of human efforts are naturally a driver for the development of more general, multi-architecture libraries, frameworks and APIs.

Multiple solutions which aim at hiding the optimization process from the user by the use of templates, classes and macros are already available. The solutions come as libraries which try to provide as much features needed for parallelism as possible while hiding the target architecture from the user. The goal is a "one-source" code that can run on multiple architectures. Ideally, the architecture dependent decision would be taken at compile time. Every conditional

[1] https://www.top500.org/lists/2008/06/, accessed on 10/09/2018
[2] https://www.top500.org/lists/2018/06/, accessed on 10/09/2018

statement evaluated during the compilation results in less computation during code execution. The conditions do not have to be evaluated as they are removed by the compiler, and only the right branch of the conditional statement is left. All excluded sections are completely removed from the code, making sure the different architecture-specific implementations of functions do not add any overhead at run time.

At present, Kokkos [11] and RAJA [16] are the two leading C/C++ libraries to provide such features, and shall be investigated closer in this thesis.

Kokkos is a package from the Trilinos Project [15]. The Trilinos Project regroups different independent packages implementing algorithms mostly around linear algebra. One of the numerous packages is Kokkos, a C++ performance portability programming ecoSystem. It is mainly being developed by Carter Edwards and Christian Trott at the Computer Science Research Institute of the Sandia National Laboratories[3].

RAJA is a C++ library developed at the Center for Applied Scientific Computing of the Livermore National Laboratory[4] (LLNL) mainly by Richard Hornung and David Beckingsale.

Both libraries break down parallel computation to 6 abstractions: Execution Space, Execution Pattern, Execution Policy, Memory Space, Memory Layout and Memory Trait. These abstractions are the constant features of a parallel section. By modifying the hyper-parameters for each abstraction the same code can result in a parallel loop or a reduction, it can run on CPU or GPU, and so on.

To our knowledge, there exist very few comparisons of Kokkos, RAJA and other parallel frameworks. The two articles found [18, 19] do a comparison of Kokkos, RAJA, OpenACC, OpenMP 4.0, CUDA and OpenCL using the Tealeaf application. Tealeaf is a *miniapp* solving the heat conduction equation and is used to compare the different frameworks. For a parallel implementation of the heat conduction equation, the mesh used to discretize space is being split. Each thread will solve the equation on its subdomain. When splitting the mesh, the threads need the information coming from the neighbor cells. The neighbor cells needed for each subdomain is called the halo. Because the halo of different threads are overlapping, updating values has to be done correctly. In [18, 19], a 5% to 30% performance hit is being observed for Kokkos and RAJA compared to architecture-specific implementations.

To assess the performance and usability of Kokkos and RAJA for a more complex application, we use a code provided by Dr. Katharina Kormann. This code is part of the Semi Lagrangian Library (Selalib) [22] and implements a Particle-in-cell method, GEMPIC: Geometric electromagnetic particle-in-cell methods [17] to solve the Vlasov-Maxwell system of equations. This is a FORTRAN code parallelized with MPI. We analyzed it and extracted the most time consuming function. After implementing it in C++, 4 parallel version were developed using OpenMP, Kokkos, RAJA and Cuda, respectively. The difference between a heat convection problem, used in [18, 19] and a Particle-In-Cell (PIC) model, as we used in our comparison, is that with PIC the mesh is not divided among threads but the particles are. Each thread will deal with a subset of the particles, compute the contribution of each particle to the grid. Finally, one has to sum the results from all threads. The GEMPIC implementation we are working

[3]https://cfwebprod.sandia.gov/cfdocs/CompResearch/templates/insert/org_chart.cfm, accessed on 10/09/2018

[4]https://computation.llnl.gov/casc, accessed on 10/09/2018

on has not yet been fully optimized. As its optimization is a large project for the next years, we assess the performance of Kokkos and RAJA in order to provide insight and guidance on whether or not these libraries should be picked as the parallel framework of choice for the future.

This work is structured in the following way. We start by introducing the mathematical equations solved in the FORTRAN code in chapter 2, where the equations and their discretized formulation are presented in section 2.1 and 2.2, respectively. For the complete derivation to obtain the discretized equations, please refer to [17]. Section 2.3 details on a visualization tool for the particle data we have developed. In the following, chapter 3, the two performance portability frameworks Kokkos and RAJA are presented. Kokkos and RAJA's features are explained by following a code snippet from our implementation of the GEMPIC code. The next chapter 4 contains the analysis that was conducted on the GEMPIC FORTRAN code, and introduces our C++ implementation of the most expensive part from it which forms the basis and starting point for the investigations done in the following. In chapter 5 follows the review of Kokkos and RAJA, based on our use of both to implement parallel versions of the GEMPIC code. Chapter 6 presents the results and performance of the six different parallel implementations on state of the art hardware platforms, using OpenMP, Kokkos (CPU and GPU), RAJA (CPU and GPU) and Cuda, respectively. In section 6.2 we detail on an issue encountered with RAJA that heavily impacted the parallel performance of this code, namely false sharing, and discuss our strategy to resolve it. Finally, the thesis closes with a summary.

2 Mathematical background of the GEMPIC model

2.1 Equations

The GEMPIC code implements the model described in [17].
The system of equations is made of first, the non-relativistic Vlasov equation,

$$\frac{\partial f_s}{\partial t} + \boldsymbol{v} \cdot \boldsymbol{\nabla}_{\boldsymbol{x}} f_s + \frac{q_s}{m_s} (\boldsymbol{E} + \boldsymbol{v} \times \boldsymbol{B}) \cdot \boldsymbol{\nabla}_{\boldsymbol{v}} f_s = 0. \tag{2.1}$$

where f_s is the distribution function of the particles position $\boldsymbol{x}$ and velocity $\boldsymbol{v}$. $\boldsymbol{E}$ denotes the electric field and $\boldsymbol{B}$ the magnetic field, s is the particle species, q_s its charge and m_s its mass. The Vlasov equation is nonlinearly coupled to the Maxwell equations,

$$\frac{\partial \boldsymbol{E}}{\partial t} - \boldsymbol{\nabla} \times \boldsymbol{B} = -\boldsymbol{J}, \tag{2.2}$$

$$\frac{\partial \boldsymbol{B}}{\partial t} - \boldsymbol{\nabla} \times \boldsymbol{E} = 0, \tag{2.3}$$

$$\boldsymbol{\nabla} \cdot \boldsymbol{E} = \rho, \tag{2.4}$$

$$\boldsymbol{\nabla} \cdot \boldsymbol{B} = 0. \tag{2.5}$$

Suitable initial and boundary conditions are needed to solve these coupled equations. The sources for Maxwell equations, the charge density ρ and the current density $\boldsymbol{J}$, are obtained from the distribution functions f_s by,

$$\rho = \sum_s q_s \int f_s d\boldsymbol{v}, \qquad \boldsymbol{J} = \sum_s q_s \int f_s \boldsymbol{v} d\boldsymbol{v}. \tag{2.6a),(2.6b}$$

The continuity equation for charge conservation reads

$$\frac{\partial \rho}{\partial t} + \mathrm{div} \boldsymbol{J} = 0. \tag{2.7}$$

and is obtain by taking the divergence of equation (2.2) and using equation (2.4).

The Maxwell's equations are ill-posed when equation (2.7) is not satisfied, (2.7) serves as a compatibility condition.

2.2 Discretization

We will now introduce the discretized equations solved in the GEMPIC code. The formulation presented in this section is the one implemented in the code. For the complete demonstration on how to derive them from the initial Vlasov-Maxwell equations (2.1)-(2.5), please refer to [17].

First, we are using a particle based model meaning that the distribution function f_s is approximated as follow

$$f_h(\boldsymbol{x}, \boldsymbol{v}, t) = \sum_{a=1}^{N_p} w_a \delta(\boldsymbol{x} - \boldsymbol{x}_a(t)) \delta(\boldsymbol{v} - \boldsymbol{v}_a(t)). \tag{2.8}$$

where N_p is the number of particles and a the particle's label.

For each particle a we have its mass m, charge q, weights w_a, position $\boldsymbol{x}_a$, and velocity $\boldsymbol{v}_a$.

The dynamic variables in this discretization are $\boldsymbol{u} = (\boldsymbol{X}, \boldsymbol{V}, \boldsymbol{e}, \boldsymbol{b})^T$ where $\boldsymbol{X}$ is the $3N_p$ vector of the particle's position $(\boldsymbol{x}_1, ..., \boldsymbol{x}_{N_p})^T$. Similarly, $\boldsymbol{V}$ is the $3N_p$ vector of the particle's velocity $(\boldsymbol{v}_1, ..., \boldsymbol{v}_{N_p})^T$. $\boldsymbol{e}$ and $\boldsymbol{b}$ are the coefficient vectors of the discrete electric $\boldsymbol{E}_h$ and magnetic fields $\boldsymbol{B}_h$, respectively.

The discrete fileds $\boldsymbol{E}_h$ and $\boldsymbol{B}_h$ are defined as

$$\boldsymbol{E}_h(t, \boldsymbol{x}) = \sum_{i=1}^{3N_1} \boldsymbol{e}_i(t) \boldsymbol{\Lambda}_i^1(\boldsymbol{x}), \qquad \boldsymbol{B}_h(t, \boldsymbol{x}) = \sum_{i=1}^{3N_2} \boldsymbol{b}_i(t) \boldsymbol{\Lambda}_i^2(\boldsymbol{x}), \tag{2.9a,2.9b}$$

with Λ^i the B-spline finite element basis function described in [5][6].

The dimensions of the spaces where the discretized electric field and magnetic field live are $N1$ and $N2$ respectively.

2.2.1 Semi-discretized equations

In [17], a semi-discrete Poisson bracket is derived from the analytic bracket of the Vlasov-Maxwell system (2.1)-(2.5) [20][25]. The semi-discrete Poisson bracket written in matrix form yields,

$$\begin{aligned}
\{F, G\}[\boldsymbol{X}, \boldsymbol{V}, \boldsymbol{e}, \boldsymbol{b}] = {} & \frac{\partial F}{\partial \boldsymbol{X}} \mathbb{M}_p^{-1} \frac{\partial G}{\partial \boldsymbol{V}} - \frac{\partial G}{\partial \boldsymbol{X}} \mathbb{M}_p^{-1} \frac{\partial F}{\partial \boldsymbol{V}} \\
& + \left(\frac{\partial F}{\partial \boldsymbol{V}}\right)^T \mathbb{M}_p^{-1} \mathbb{M}_q \mathbb{\Lambda}^1(\boldsymbol{X})^T \mathbb{M}_1^{-1} \left(\frac{\partial G}{\partial \boldsymbol{e}}\right) \\
& - \left(\frac{\partial F}{\partial \boldsymbol{e}}\right)^T \mathbb{M}_1^{-1} \mathbb{\Lambda}^1(\boldsymbol{X}) \mathbb{M}_q \mathbb{M}_p^{-1} \left(\frac{\partial G}{\partial \boldsymbol{V}}\right) \\
& + \left(\frac{\partial F}{\partial \boldsymbol{V}}\right)^T \mathbb{M}_p^{-1} \mathbb{M}_q \mathbb{B}(\boldsymbol{X}, \boldsymbol{b}) \mathbb{M}_p^{-1} \left(\frac{\partial G}{\partial \boldsymbol{V}}\right) \\
& + \left(\frac{\partial F}{\partial \boldsymbol{e}}\right)^T \mathbb{M}_1^{-1} \mathbb{C}^T \left(\frac{\partial G}{\partial \boldsymbol{b}}\right) - \left(\frac{\partial F}{\partial \boldsymbol{b}}\right)^T \mathbb{C} \mathbb{M}_1^{-1} \left(\frac{\partial G}{\partial \boldsymbol{e}}\right).
\end{aligned} \tag{2.10}$$

$\mathbb{M}_1$ is the $3N_1 \times 3N_1$ mass matrix defined as

$$(\mathbb{M}_1)_{ij} = \int_\Omega \Lambda_i^1(x) \cdot \Lambda_j^1(x) \, \mathrm{d}x. \tag{2.11}$$

with again, Λ^i the B-spline finite element basis function described in [5][6]. The $3N_p \times 3N_1$ matrix $\mathbb{\Lambda}^i(\boldsymbol{X})$ with generic term $\boldsymbol{\Lambda}_I^1(\boldsymbol{x}_a)$ and the $3N_p \times 3N_p$ block diagonal matrix $\mathbb{B}(\mathbf{X}, \mathbf{b})$ with blocks

$$\widehat{\mathbf{B}}_h(\mathbf{x}_a,t)=\sum_{i=1}^{N_2}\begin{bmatrix}0 & b_{i,3}(t)\lambda_i^{2,3}(\mathbf{x}_a) & -b_{i,2}(t)\lambda_i^{2,2}(\mathbf{x}_a)\\ -b_{i,3}(t)\lambda_i^{2,3}(\mathbf{x}_a) & 0 & b_{i,1}(t)\lambda_i^{2,1}(\mathbf{x}_a)\\ b_{i,2}(t)\lambda_i^{2,2}(\mathbf{x}_a) & -b_{i,1}(t)\lambda_i^{2,1}(\mathbf{x}_a) & 0\end{bmatrix} \tag{2.12}$$

are introduced to rewrite the semi-discretized equations into matrix form.
$\mathbb{C}$ is the curl matrix, a difference approximation of the curl.

(2.10) can be written,

$$\{F,G\}=DF^T\mathbb{J}(\boldsymbol{u})DG, \tag{2.13}$$

where D is the derivative with respect to the variables

$$\boldsymbol{u}=(\boldsymbol{X},\boldsymbol{V},\boldsymbol{e},\boldsymbol{b})^T, \tag{2.14}$$

and $\mathbb{J}$ is the Poisson matrix

$$\mathbb{J}(\boldsymbol{u})=\begin{bmatrix}0 & \mathbb{M}_p^{-1} & 0 & 0\\ -\mathbb{M}_p^{-1} & \mathbb{M}_p^{-1}\mathbb{M}_q\mathbb{B}\left(\boldsymbol{X},\boldsymbol{b}\right)\mathbb{M}_p^{-1} & \mathbb{M}_p^{-1}\mathbb{M}_q\mathbb{\Lambda}\left(\boldsymbol{X}\right)\mathbb{M}_1^{-1} & 0\\ 0 & \mathbb{M}_p^{-1}\mathbb{M}_q\mathbb{\Lambda}\left(\boldsymbol{X}\right)\mathbb{M}_1^{-1} & 0 & \mathbb{M}_1^{-1}\mathbb{C}^T\\ 0 & 0 & -\mathbb{C}\mathbb{M}_1^{-1} & \end{bmatrix} \tag{2.15}$$

$\mathbb{J}(\boldsymbol{u})$ is anti-symmettric and [17] shows that it satisfies the Jacobi identity, needed for analytical properties such as conservation.

2.2.2 Matrix discrete Hamiltonian

[17] gives the matrix notation of the discrete Hamiltonian as,

$$H=\frac{1}{2}\boldsymbol{V}^T\mathbb{M}_p\boldsymbol{V}+\frac{1}{2}\boldsymbol{e}^T\mathbb{M}_1\boldsymbol{e}+\frac{1}{2}\boldsymbol{b}^T\mathbb{M}_2\boldsymbol{b} \tag{2.16}$$

and the semi-discrete equations of motion are given by,

$$\dot{\boldsymbol{X}}=\{\boldsymbol{X},H\},\quad \dot{\boldsymbol{V}}=\{\boldsymbol{V},H\},\quad \dot{\boldsymbol{e}}=\{\boldsymbol{e},H\},\quad \dot{\boldsymbol{b}}=\{\boldsymbol{b},H\}. \tag{2.17a-2.17d}$$

which are equivalent to

$$\dot{\boldsymbol{u}}=\mathbb{J}\left(\boldsymbol{u}\right)DH\left(\boldsymbol{u}\right). \tag{2.18}$$

with $DH\left(\boldsymbol{u}\right)=\left(0,\mathbb{M}_p\boldsymbol{V},\mathbb{M}_1\boldsymbol{e},\mathbb{M}_2\boldsymbol{b}\right)^T$, we obtain

$$\dot{\boldsymbol{X}}=\boldsymbol{V} \tag{2.19}$$

$$\dot{\boldsymbol{V}}=\mathbb{M}_p^{-1}\mathbb{M}_q\left(\mathbb{\Lambda}^1\left(\boldsymbol{X}\right)\boldsymbol{e}+\mathbb{B}\left(\boldsymbol{X},\boldsymbol{b}\right)\boldsymbol{V}\right) \tag{2.20}$$

$$\dot{\boldsymbol{e}}=\mathbb{M}_p^{-1}\left(\mathbb{C}^T\mathbb{M}_2\boldsymbol{b}\left(t\right)-\mathbb{\Lambda}^1\left(\boldsymbol{X}\right)^T\mathbb{M}_q\boldsymbol{V}\right) \tag{2.21}$$

$$\dot{\boldsymbol{b}}=-\mathbb{C}\boldsymbol{e}\left(t\right) \tag{2.22}$$

2.2.3 Hamiltonian splitting

To discretize in time, a Hamiltonian splitting [10][14][21] is used. The discrete Hamiltonian (2.16) is split into three parts,

$$H = H_p + H_E + H_B \tag{2.23}$$

with

$$H_p = \frac{1}{2}\boldsymbol{V}^T \mathbb{M}_p \boldsymbol{V}, \qquad H_E = \frac{1}{2}\boldsymbol{e}^T \mathbb{M}_1 \boldsymbol{e}, \qquad H_B = \frac{1}{2}\boldsymbol{b}^T \mathbb{M}_2 \boldsymbol{b}, \tag{2.24a-2.24c}$$

writing $\boldsymbol{u} = (\boldsymbol{X}, \boldsymbol{V}, \boldsymbol{e}, \boldsymbol{b})^T$, we split the discrete Vlasov-Maxwell equations (2.19-2.22) into three subsystems

$$\dot{\boldsymbol{u}} = \{\boldsymbol{u}, H_p\}, \qquad \dot{\boldsymbol{u}} = \{\boldsymbol{u}, H_E\}, \qquad \dot{\boldsymbol{u}} = \{\boldsymbol{u}, H_B\}, \tag{2.25a-2.25c}$$

2.2.4 Operator HE

The discrete equations of motion for H_E are

$$\dot{\boldsymbol{X}} = 0, \tag{2.26}$$
$$\mathbb{M}_p \dot{\boldsymbol{V}} = \mathbb{M}_q \mathbb{\Lambda}^1 (\boldsymbol{X})\, \boldsymbol{e}, \tag{2.27}$$
$$\dot{\boldsymbol{e}} = 0, \tag{2.28}$$
$$\dot{\boldsymbol{b}} = -\mathbb{C}\boldsymbol{e}(t). \tag{2.29}$$

For initial conditions $(\boldsymbol{X}(0), \boldsymbol{V}(0), \boldsymbol{e}(0), \boldsymbol{b}(0))$ the exact solutions at time Δt are defined as

$$\boldsymbol{X}(\Delta t) = \boldsymbol{X}(0), \tag{2.30}$$
$$\mathbb{M}_p \boldsymbol{V}(\Delta t) = \mathbb{M}_p \boldsymbol{V}(0) + \Delta t \mathbb{M}_q \mathbb{\Lambda}^1 (\boldsymbol{X}(0))\, \boldsymbol{e}(0), \tag{2.31}$$
$$\boldsymbol{e}(\Delta t) = \boldsymbol{e}(0), \tag{2.32}$$
$$\boldsymbol{b}(\Delta t) = \boldsymbol{b}(0) - \Delta t \mathbb{C}\boldsymbol{e}(0). \tag{2.33}$$

2.2.5 Operator HB

The discrete equations of motion for H_B are

$$\dot{\boldsymbol{X}} = 0, \tag{2.34}$$
$$\dot{\boldsymbol{V}} = 0, \tag{2.35}$$
$$\mathbb{M}_1 \dot{\boldsymbol{e}} = \mathbb{C}^T \mathbb{M}_2 \boldsymbol{b}(t), \tag{2.36}$$
$$\dot{\boldsymbol{b}} = 0. \tag{2.37}$$

For initial conditions $(\boldsymbol{X}(0), \boldsymbol{V}(0), \boldsymbol{e}(0), \boldsymbol{b}(0))$ the exact solutions at time Δt are defined as

$$\boldsymbol{X}(\Delta t) = \boldsymbol{X}(0), \tag{2.38}$$
$$\boldsymbol{V}(\Delta t) = \boldsymbol{V}(0), \tag{2.39}$$
$$\mathbb{M}_1 \boldsymbol{e}(\Delta t) = \mathbb{M}_1 \boldsymbol{e}(0) + \Delta t \mathbb{C}^T \mathbb{M}_2 \boldsymbol{b}(0), \tag{2.40}$$
$$\boldsymbol{b}(\Delta t) = \boldsymbol{b}(0). \tag{2.41}$$

2.2.6 Operator HP

The discrete equations of motion for H_P are

$$\dot{\boldsymbol{X}} = \boldsymbol{V}, \tag{2.42}$$
$$\mathbb{M}_p \dot{\boldsymbol{V}} = \mathbb{M}_q \mathbb{B}(\boldsymbol{X}, \boldsymbol{b}) \boldsymbol{V}, \tag{2.43}$$
$$\mathbb{M}_1 \dot{\boldsymbol{e}} = \mathbb{\Lambda}^1(\boldsymbol{X})^T \mathbb{M}_q \boldsymbol{V}, \tag{2.44}$$
$$\dot{\boldsymbol{b}} = 0. \tag{2.45}$$

For general magnetic field coefficient $\boldsymbol{b}$, this system cannot be exactly integrated (He *et al.* 2015).

Note that each component $\dot{\boldsymbol{V}}_{\boldsymbol{\mu}}$ of the equation for $\dot{\boldsymbol{V}}$ does not depend on $\boldsymbol{V}_\mu$, where $\boldsymbol{V})\mu = (v_{1,\mu}, ..., v_{N_p,\mu})^T$, etc., with $1 \leq \mu \leq 3$. Therefore we can split this system once more into

$$H_p = H_{p1} + H_{p2} + H_{p3}, \tag{2.46}$$

with

$$H_{p_\mu} = \frac{1}{2} \boldsymbol{V}_\mu^T \boldsymbol{M}_p \boldsymbol{V}_\mu \quad \text{for } 1 \leq \mu \leq 3. \tag{2.47}$$

For concise notation we introduce the $N_p \times N_1$ matrix $\boldsymbol{\Lambda}^{\mathbf{1}}_{\boldsymbol{\mu}}(\boldsymbol{X})$ with generic term $\boldsymbol{\Lambda}^{\mathbf{1}}_{\boldsymbol{i,\mu}}(\boldsymbol{x}_a)$ and the $N_p \times N_p$ diagonal matrix $\boldsymbol{\Lambda}^{\mathbf{2}}_{\boldsymbol{\mu}}(\boldsymbol{b}, \boldsymbol{X})$ with entries $\sum_{i=1}^{N_2} b_i(t) \boldsymbol{\lambda}^2_{i,\mu}(\boldsymbol{x}_a)$, where $1 \leq \mu \leq 3$, $1 \leq a \leq N_p$, $1 \leq i \leq N_1$, $1 \leq j \leq N_2$. Then, for H_{p1} we have

$$\dot{\boldsymbol{X}}_1 = \boldsymbol{V}_1(t), \tag{2.48}$$
$$\boldsymbol{M}_p \dot{\boldsymbol{V}}_2 = -\boldsymbol{M}_q \boldsymbol{\Lambda}^2_3((\boldsymbol{b}(t), \boldsymbol{X}(t))^T \boldsymbol{V}_1(t), \tag{2.49}$$
$$\boldsymbol{M}_p \dot{\boldsymbol{V}}_3 = \boldsymbol{M}_q \boldsymbol{\Lambda}^2_2(\boldsymbol{b}(t), \boldsymbol{X}(t))^T \boldsymbol{V}_1(t), \tag{2.50}$$
$$\mathbb{M}_1 \dot{\boldsymbol{e}} = -\boldsymbol{\Lambda}^1_1(\boldsymbol{X}(t))^T \boldsymbol{M}_q \boldsymbol{V}_1(t), \tag{2.51}$$

for H_{p2} we have

$$\dot{\boldsymbol{X}}_2 = \boldsymbol{V}_2(t), \tag{2.52}$$
$$\boldsymbol{M}_p\dot{\boldsymbol{V}}_1 = \boldsymbol{M}_q\boldsymbol{\Lambda}_3^2(\boldsymbol{b}(t), \boldsymbol{X}(t))^T \boldsymbol{V}_2(t), \tag{2.53}$$
$$\boldsymbol{M}_p\dot{\boldsymbol{V}}_3 = -\boldsymbol{M}_q\boldsymbol{\Lambda}_1^2(\boldsymbol{b}(t), \boldsymbol{X}(t))^T \boldsymbol{V}_2(t), \tag{2.54}$$
$$\mathbb{M}_1\dot{\boldsymbol{e}} = -\boldsymbol{\Lambda}_2^1(\boldsymbol{X}(t))^T \boldsymbol{M}_q\boldsymbol{V}_2(t), \tag{2.55}$$

and for H_{p3} we have

$$\dot{\boldsymbol{X}}_3 = \boldsymbol{V}_3(t), \tag{2.56}$$
$$\boldsymbol{M}_p\dot{\boldsymbol{V}}_1 = -\boldsymbol{M}_q\boldsymbol{\Lambda}_2^2(\boldsymbol{b}(t), \boldsymbol{X}(t))^T \boldsymbol{V}_3(t), \tag{2.57}$$
$$\boldsymbol{M}_p\dot{\boldsymbol{V}}_2 = \boldsymbol{M}_q\boldsymbol{\Lambda}_1^2(\boldsymbol{b}(t), \boldsymbol{X}(t))^T \boldsymbol{V}_3(t), \tag{2.58}$$
$$\mathbb{M}_1\dot{\boldsymbol{e}} = -\boldsymbol{\Lambda}_3^1(\boldsymbol{X}(t))^T \boldsymbol{M}_q\boldsymbol{V}_3(t), \tag{2.59}$$

For H_{p1} and initial conditions $(\boldsymbol{X}(0), \boldsymbol{V}(0), \boldsymbol{e}(0), \boldsymbol{b}(0))$ the exact solutions at time Δt are defined as

$$\boldsymbol{X}_1(\Delta t) = \boldsymbol{X}_1(0) + \Delta t\boldsymbol{V}_1(0), \tag{2.60}$$
$$\boldsymbol{M}_p\boldsymbol{V}_2(\Delta t) = \boldsymbol{M}_p\boldsymbol{V}_2(0) - \int_0^{\Delta t} \boldsymbol{M}_q\boldsymbol{\Lambda}_3^2(\boldsymbol{b}(0), \boldsymbol{X}(t))\boldsymbol{V}_1(0)\,\mathrm{d}t, \tag{2.61}$$
$$\boldsymbol{M}_p\boldsymbol{V}_3(\Delta t) = \boldsymbol{M}_p\boldsymbol{V}_3(0) + \int_0^{\Delta t} \boldsymbol{M}_q\boldsymbol{\Lambda}_2^2(\boldsymbol{b}(0), \boldsymbol{X}(t))\boldsymbol{V}_1(0)\,\mathrm{d}t, \tag{2.62}$$
$$\mathbb{M}_1\boldsymbol{e}(\Delta t) = \mathbb{M}_1\boldsymbol{e}(0) - \int_0^{\Delta t} \boldsymbol{\Lambda}_1^1(\boldsymbol{X}(t))^T \boldsymbol{M}_q\boldsymbol{V}_1(0)\,\mathrm{d}t, \tag{2.63}$$

for H_{p2} the exact solutions are defined as

$$\boldsymbol{X}_2(\Delta t) = \boldsymbol{X}_2(0) + \Delta t\boldsymbol{V}_2(0), \tag{2.64}$$
$$\boldsymbol{M}_p\boldsymbol{V}_1(\Delta t) = \boldsymbol{M}_p\boldsymbol{V}_1(0) - \int_0^{\Delta t} \boldsymbol{M}_q\boldsymbol{\Lambda}_3^2(\boldsymbol{b}(0), \boldsymbol{X}(t))\boldsymbol{V}_2(0)\,\mathrm{d}t, \tag{2.65}$$
$$\boldsymbol{M}_p\boldsymbol{V}_3(\Delta t) = \boldsymbol{M}_p\boldsymbol{V}_3(0) + \int_0^{\Delta t} \boldsymbol{M}_q\boldsymbol{\Lambda}_1^2(\boldsymbol{b}(0), \boldsymbol{X}(t))\boldsymbol{V}_2(0)\,\mathrm{d}t, \tag{2.66}$$
$$\mathbb{M}_1\boldsymbol{e}(\Delta t) = \mathbb{M}_1\boldsymbol{e}(0) - \int_0^{\Delta t} \boldsymbol{\Lambda}_2^1(\boldsymbol{X}(t))^T \boldsymbol{M}_q\boldsymbol{V}_2(0)\,\mathrm{d}t, \tag{2.67}$$

and for H_{p3} as

$$\boldsymbol{X}_3(\Delta t) = \boldsymbol{X}_3(0) + \Delta t \boldsymbol{V}_3(0), \tag{2.68}$$

$$\boldsymbol{M}_p \boldsymbol{V}_1(\Delta t) = \boldsymbol{M}_p \boldsymbol{V}_1(0) - \int_0^{\Delta t} \boldsymbol{M}_q \boldsymbol{\Lambda}_2^2(\boldsymbol{b}(0), \boldsymbol{X}(t)) \boldsymbol{V}_3(0) \, \mathrm{d}t, \tag{2.69}$$

$$\boldsymbol{M}_p \boldsymbol{V}_2(\Delta t) = \boldsymbol{M}_p \boldsymbol{V}_2(0) + \int_0^{\Delta t} \boldsymbol{M}_q \boldsymbol{\Lambda}_1^2(\boldsymbol{b}(0), \boldsymbol{X}(t)) \boldsymbol{V}_3(0) \, \mathrm{d}t, \tag{2.70}$$

$$\mathbb{M}_1 \boldsymbol{e}(\Delta t) = \mathbb{M}_1 \boldsymbol{e}(0) - \int_0^{\Delta t} \boldsymbol{\Lambda}_3^1(\boldsymbol{X}(t))^T \boldsymbol{M}_q \boldsymbol{V}_3(0) \, \mathrm{d}t, \tag{2.71}$$

where all components not specified are constant.

The different operators, H_E, H_B, H_{p1}, H_{p2} and H_{p3}, can be combined by Lie splitting, Strang splitting [23] or higher order decomposition methods [13].

2.3 Visualisation

Plotting the particles gives an easy way of visualize the particles behavior over time. To visualize the particles, the authors of [17] use 1D histograms to represent the density. 3d3v simulations are hard to visualize because of the number of dimensions. We developed a post-processing script that produces a 3D visualization of the particles at a certain time step with a trail behind the particles corresponding to the N last time iterations. This script will be included in the post processing scripts of the project. To do the visualization without changing the source code, we apply our script to the already existing output function of the Selalib GEMPIC code. The files are set to a standard CSV format using common shell programs. Although the *sed* command is very efficient, this script could be replaced by an implementation in the GEMPIC code of a function writing the output in the CSV format. Displaying the particles themselves and not a density function requires the amount of particles to be comparably small. We subsample the particles in the script by keeping only the X first particles. Because the particles are not sorted and generated with a Sobol sequence, a low-discrepancy sequence, the first particles do spread over the entire domain.

The second script uses Paraview's Python module to produce the visualization. Here the main steps of the Python script are described. After getting the file prefix from the user, the files are opened with the CSVreader.

```
particles = CSVReader(FileName=all_names)
```

Particles are created from the CSV table. In 2D, only the X and Y fields need to match the coordinates x_1 and x_2.

```
tableToPoints = TableToPoints(Input=particles)
tableToPoints.XColumn = 'x1'
tableToPoints.YColumn = 'x2'
```

This produces points at the position of each particle.
The sense of motion is obtained by displaying multiple time steps of the particles positions.

```
temporalParticlesToPathlines =
        TemporalParticlesToPathlines(Input=tableToPoints, Selection=None)
```

Using input values, the next steps set the visual settings such as particle size, previous time steps, dots size, number of time steps displayed, and so on.

Finally the image is saved using the SaveScreenshot function.

```
SaveScreenshot(save_file, renderView,
        ImageResolution=[resX, resY], CompressionLevel='0')
```

Fig. 2.1 shows the result for data obtained from the 3d3v code. The particles are shown as white spheres and their trajectory as the colored trail. The color coding refers to the particle's ID number.

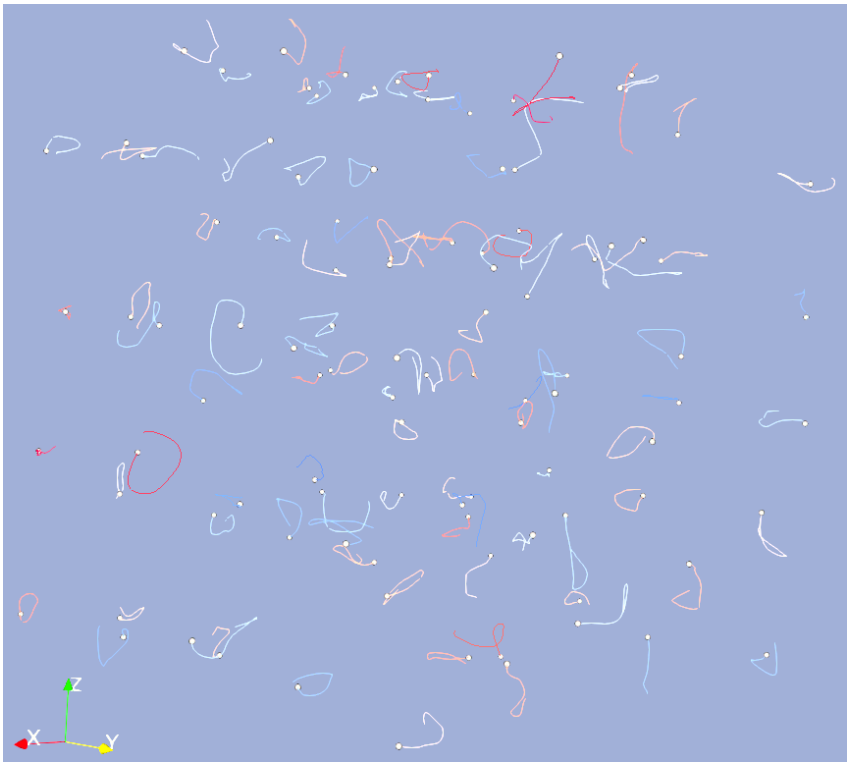

Figure 2.1: Output of our visualization tool applied to a 3D3V simulation with 100 particles. The domain is a cube with periodic boundary conditions. The trail shows the position of the particles for the last 200 time steps. Trail color coding refers to the particle's ID number.

3 Introduction to the performance portability frameworks Kokkos and RAJA

To point out the similarities and differences of Kokkos and RAJA on a real use case implementation, our codes, using Kokkos for one and RAJA for the other, are analyzed side-by-side. We will present the codes' structures and differences. The code sample shown has been cleaned from auxiliary lines such as adding, updating physical values, etc., to improve the clarity of the presentation. Similar code sections are presented just once, for example, array initialization will be written for one array only. These code snippets keep the essential lines concerning Kokkos and RAJA.

Lastly, variables have been renamed to fit the codes side by side on one page and these shorter names are used in the following sections. The original names can be found in Tab. 3.1.

Original name	Short name
hamiltonian_splitting	HS
view_j_dofs_local_host	j_dofs_h
view_j_dofs_local	j_dofs
i_species	i_sp
view_n_species_host	ns_h
view_n_species	ns
particle_mesh_coupling	pmc
Kokkos	K
RAJA	R

Table 3.1: Renaming of the code's variables for improved readability.

Listing 3.1: Kokkos implementation of the operator Hp1

```
void HS::operator_Hp1(double dt,
              int n_threads,
              int n_teams) {
  for(int i=0;
      i<this->j_dofs_h.size(); i++) {
    this->j_dofs_h(i) = 0.0;
  }
  K::deep_copy(this->j_dofs,
                    this->j_dofs_h);









  int shared_size = ...*sizeof(double);
  auto policy = K::TeamPolicy
                  <Hp1, ExecSpace>
                  (n_teams, n_threads)
                  .set_scratch_size
                  (1,K::PerThread
                        (shared_size));


  K::parallel_for(policy, *this);
}
```

Listing 3.2: RAJA implementation of the operator Hp1

```
void HS::operator_Hp1(double dt,
              int n_threads,
              int n_teams) {
  for(int i=0;
      i<this->pmc.n_dofs; i++) {
    this->j_dofs[i] = 0.0;
  }
  #if defined(RAJA_ENABLE_CUDA)
   && defined(I_USE_CUDA)
    cudaMemcpy(this->d_j_dofs,
      this->j_dofs,
      this->pmc.n_dofs*sizeof(double),
      cudaMemcpyHostToDevice);
  #else
    this->d_j_dofs = this->j_dofs;
  #endif

  #if defined(RAJA_ENABLE_CUDA)
   && defined(I_USE_CUDA)
    R::forall<R::cuda_exec<128>>
    (R::RangeSegment(0,
      this->particles.group[0].n_particles),
      [=,*this] RAJA_DEVICE (int i_part){.});
  #else
    double d_x_old[n_threads*3];
    RajaVector2DType raja_x_old(d_x_old,
                                n_threads, 3);
  R::forall<R::omp_parallel_for_exec>
  (R::RangeSegment(0,
  this->particles.group[0].n_particles),
    [&,this] (int i_part){.});
  #endif
}
```

Listings 3.1 and 3.2 contain the highest level function regarding the parallel section. The function operator_Hp1 from the class HS is called directly from the main loop when testing the code. The only function required to be run before operator_Hp1 is the constructor of HS to initialize most of the variables needed.

3.1 View

The first action taken by the operator_Hp1 function is to set values to zero on line 5. For Kokkos it is written

```
5 this->j_dofs_h(i) = 0.0;
```

and for RAJA

```
5 this->j_dofs[i] = 0.0;
```

This introduces the View, the data type used by both Kokkos and RAJA to abstract the location of the data.
Most of scientific codes spend a lot of time processing arrays of data. Because these computations often take a significant part of the computing time, programmers invest a lot in making those operations as fast as possible. Such optimizations are tightly linked to the underlying hardware, environment, language and programming model. As said earlier, optimal layouts can be different depending on the hardware. Because low-level details can have such a high impact on performance, porting a code optimized for a specific architecture is complicated. Porting a code to a different architecture often means rewriting a significant amount of the code.

Abstracting the data type allows for the optimization of array management and access internally to relieve the programmer from this architecture-specific task. The optimal layout and pad allocation for ideal alignment depending on the architecture can also be chosen.

Kokkos and RAJA have, in general, a different approach regarding the architecture-specific decisions. On the one hand, Kokkos tries to have as many default settings as possible. Meaning that when not specified, the settings for a View will be picked according to the fastest architecture accessible. RAJA, on the other hand, has less default values and wants the user to be aware of his choices.

A View contains only meta-data such as dimensions, sizes and layout plus a pointer to the data. This pointer can be on host memory or on GPU device memory.

3.1.1 Memory allocation

When declaring a View, Kokkos allocates the memory on the architecture defined by the programmer or the default setting. The first parameter of a View, a string, is a name for Kokkos to use when debugging. Accessing directly the View from the host code can result in an error as the data may not be accessible from the CPU. Initializing a 1D View of size n in Kokkos, with a secondary View for access on the Host, looks as follows:

```
1 K::View<double*> j_dofs("j_dofs", n);
2 K::View<double*>::HostMirror j_dofs_h = K::create_mirror_view(j_dofs);
3 K::deep_copy(j_dofs, j_dofs_h);
```

In contrast, RAJA does not allocate memory. When initializing a View in RAJA, one more parameter is needed: the pointer to the data, previously allocated by the user code.

A small MemoryManager code is given in RAJA's example folder[1] which uses the *#ifdef* sections to allocate memory on CPU or GPU. Although it is in the example folder, it is not part of the RAJA library. Initializing a 1D View of size n in RAJA works as follows:

```
1 double *j_dofs = new double[n]();
2 double *d_j_dofs = memoryManager::allocate<double>(n);
3 #if defined(RAJA_ENABLE_CUDA) && defined(I_USE_CUDA)
4   cudaMemcpy(d_j_dofs, j_dofs, n*sizeof(double), cudaMemcpyHostToDevice);
5 #else
6   d_j_dofs = j_dofs;
7 #endif
8 R::View<double, R::Layout<1>> r_j_dofs = R::View<double, R::Layout<1>>
9                                                   (d_j_dofs, n);
```

Both codes have a host View/array, j_dofs_h (Kokkos) and j_dofs (RAJA), which can be used from the host side.

Memory Spaces

The Memory space defines where the data is stored. Depending on the hardware, it gives access to high bandwidth memory, on-die scratch memory and non-volatile bulk storage. Kokkos also provides access to UVM (Unified Virtual Memory, or Unified Memory), which allows access to the data from both the host and the device by duplicating the data and synchronizing the memory to keep the host copy of the device data updated and vice versa.

A View type with its memory space can be defined as follows.

```
1 Kokkos::View<long>
2 Kokkos::View<long, Kokkos::HostSpace>
3 Kokkos::View<long, Kokkos::OpenMP>
4 Kokkos::View<long, Kokkos::CudaSpace>
5 Kokkos::View<long, Kokkos::CudaUVMSpace>
```

The first type, *Kokkos::View<long>*, will use the default memory spaces *HostSpace*, *OpenMP* or *CudaSpace*, according to the highest priority architecture available. This will therefore match the default execution space.

RAJA does not define Memory spaces with names as they do not allocate memory.

3.1.2 Kokkos' mirror View

The memory pointer of a Kokkos View can, but should not be accessed directly. Only Kokkos functions should be used to modify a View. So to set the values of a View e.g. from a text file, plain memory allocated on the CPU is needed. This is accomplished by the mirror View.

The mirror View is a View with memory allocated on the CPU only. If the original View is already on the host memory, the mirror will just copy the pointer to the View's data. If the View has memory on the GPU, the mirror view will allocate the same amount of memory on the host.

[1] https://github.com/LLNL/RAJA/blob/develop/examples/memoryManager.hpp, accessed on 05/16/2018, commit 3eb858d87acce64dcd03bede5b84dee87b10ced7

Once the mirror View as been initialized with the right values, we do a deep copy to the original View. The deep_copy function will check, in the case of a View and its mirror, if the View has its data on the host or the device. If the memory is on the host, because the pointers to the data are the same, nothing needs to be done. The View's data has already been initialized as it shares the same memory as the mirror View.

If the View has its data on the device, a host-to-device copy is made.

The three lines, View allocation, HostMirror initialisation and call to *deep_copy* allow efficient View initialization regardless of the architecture used.

In Listing 3.1, lines 4-7, we are setting the mirror View to zero. In lines 8-9, we copy it to the original View.

3.1.3 RAJA's explicit memory allocation

RAJA does not have mirror Views. But the View's data is accessible to the user as it is declared by the user. So the memory management is very similar to what we would do with CUDA. A host and a device array are initialized. Once the memory is set on the host array, it is copied to the device using CUDA's functions.

Overall, we are doing explicitly what Kokkos does implicitly with mirror Views and the deep_copy function.

In Listing 3.2, lines 4-7, we are setting the host array to zero. In lines 8-16, we copy it to the device's array or we set both array pointers to the same value depending on the architecture.

3.2 Layout

In the case of the RAJA View as shown in section 3.1 one could ask what the View is useful for if we already manipulate explicitly the plain old data array. Its main advantage is that the View allows to abstract the indexing. This is less relevant for 1D data but the advantages of the Layout abstraction become obvious starting from 2 dimensions.

Memory layouts specify how to map logical indices to address offsets. With different architectures, appropriate Memory layouts are needed to optimize memory access patterns for a given algorithm. Most notably, for a 2D array, if the CPU would require the rows to be stored contiguously, the GPU may perform better with the columns stored contiguously. These differences come from the design of the parallel architecture. To illustrate, we take an example of a 4-by-n matrix. We used different colors where the computation of different rows shall be independent. For both the CPU and the GPU, to obtain the best performance, the threads have to work on independent tasks.

In Fig. 3.1, we illustrate the memory layout in the case of a CPU with 4 threads. Each colored data represents a group of data with dependencies. Because each thread is independent, we give them one row to work on. Each thread will process a row, element by element, so that at any given time only one element per row is being processed. At the same time, cache lines are not shared between different threads, which is important, see section 6.2.

On the GPU, we have warps of threads that are tightly linked. They are in groups of 32 threads, we represented them as groups of 4 in Fig. 3.2 for clarity. All threads of a warp execute the same operation on different data. Even if less than 32 threads are needed warps use 32 threads. Masks are used to stop specific threads of a wrap from computing an operation.

Because threads of a warp are synchronized and work on contiguous data, here we need adjacent data in memory to be independent. Storing the matrix column by column gives the desired layout as shown by the alternating colors.

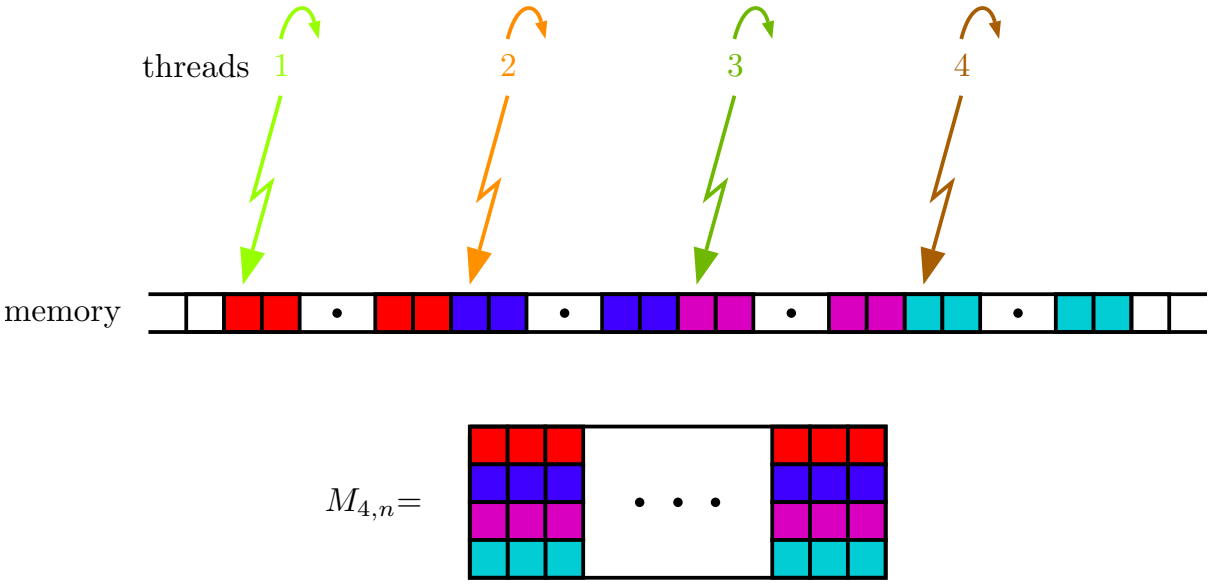

Figure 3.1: Illustration of data assignation for multi-threaded processing on the CPU. Independent tasks need to be physically distant from each other in the memory so that each thread gets contiguous data to work with. Same color for dependent data.

This difference in architecture is taken into account by Kokkos and RAJA, but RAJA requires explicit choice of the Layout to match the architecture while Kokkos has a different default layout for the CPU and the GPU.

The available Memory Layouts are LayoutRight, LayoutLeft and LayoutStride for Kokkos. On RAJA, also a possibility with Kokkos, the Layout is defined only by the strides, with LayoutRight (as defined for Kokkos) as default. LayoutLeft accesses the elements of an array starting with the leftmost index. If we store a n by m matrix, with standard notations n being the number of rows and m the number of columns, with a LayoutLeft the leftmost index, n, is stored first meaning that values of the columns will be stored consecutively. This is illustrated in Fig. 3.3. Following the same logic, the LayoutRight stores the rightmost index first. Our n by m matrix will have consecutive rows values in the memory. See Fig. 3.4. The names LayoutRight and LayoutLeft refer to the common names, row-major and column-major, respectively. The stride for an array will be the distance in memory between two adjacent elements of the array. For our n by m matrix with LayoutLeft, the strides will be $(1, n)$. Elements only one row apart will be consecutive in memory (stride of 1). Elements one column appart will be n entries apart. For the our LayoutRight example, the stride is $(m, 1)$.

This brings the last Memory Layout, LayoutStride. This gives the user the possibility of defining the stride as $(s_1, s_2, ...)$. LayoutStride makes the creation of a subview very simple. To extract the 2 by 2 matrix $M(\{2,3\},\{2,3\})$ create from the second and third rows and columns of our matrix M in Fig. 3.3, we have to go through the elements, $M(2,2)$, $M(3,2)$, $M(2,3)$ and $M(3,3)$ as shown in Fig. 3.5. The consecutive elements are $M(2,2)$, $M(3,2)$ and $M(2,3)$,

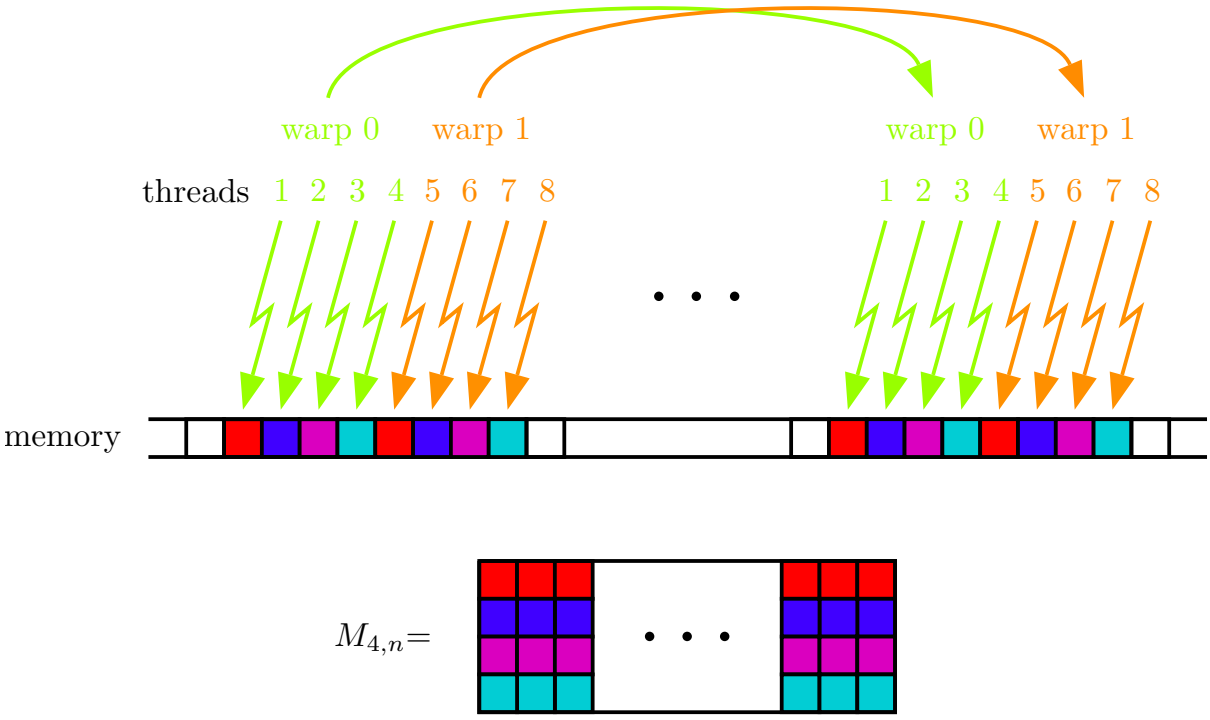

Figure 3.2: Illustration of data assignation for multi-threaded processing on the GPU. Because threads of a warp are tied and work best when memory adresses respect the threads ordering, we want consecutive data in memory to be independent. On current architecture warps regroup 32 threads. We illustrated 4 threads per warp for better readability. Same color for dependent data.

$M(3,3)$. So the stride of the first index is $s_1 = 1$. We can see in Fig. 3.5 with the dashed line that $M(3,2)$ and $M(2,3)$ have one more element between them. So $M(3,3)$ is the third element after $M(3,2)$ in memory. The stride of the second index is $s_2 = 3$. Without allocating memory, our subview is created by defining its size as 2 by 2, the address of its first element as the address of $M(2,2)$ and the stride as $(s_1 = 1, s_2 = 3)$.

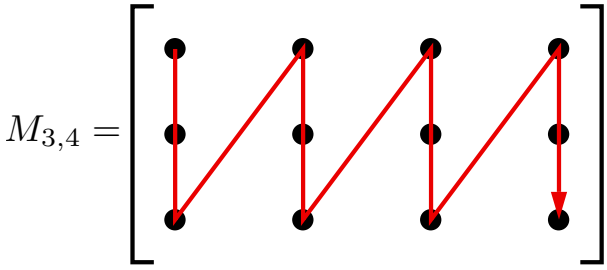

Figure 3.3: Representation of consectutive elements for a 3 by 4 matrix stored with LayoutLeft (stride $(1, 3)$). This is the column-major format.

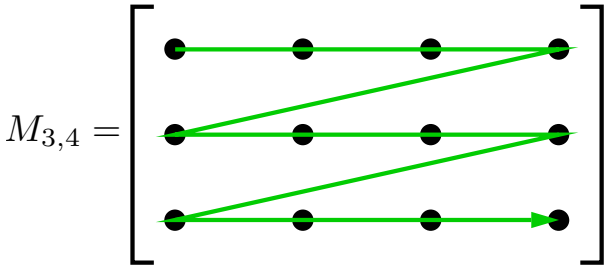

Figure 3.4: Representation of consectutive elements for a 3 by 4 matrix stored with LayoutRight (stride $(4, 1)$). This is the row-major format.

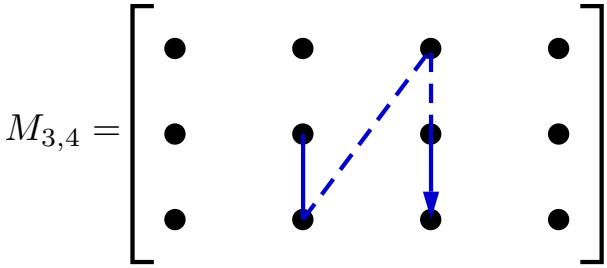

Figure 3.5: Representation of consectutive elements for a 2 by 2 submatrix stored with LayoutStride.

3.3 Traits

Kokkos adds a Traits abstraction to its View. Memory traits define how the memory is being accessed in an algorithm. Different settings are available such as atomic access, random access and streaming loads and stores. This attributes will allow the insertion of optimal load and store operations in the implementation of the programming model. A compiler aware of the specific access traits might therefore produce a more optimized code.

It this example we show how an array is set to have atomic access.

```
K::View<double*, K::MemoryTraits<K::Atomic>>  j_dofs;
```

If an atomic access is required for a variable, e.g. to avoid race conditions when other solutions are not available, this memory traits are a convenient way of making sure the variable will never be accessed simultaneously by different threads anywhere in the code.
The RandomAccess traits gives access to the texture cache on the GPU. The texture cache is a read-only memory (initialized from the CPU beforehand) that can be more efficient than global memory depending on the access pattern. On the CPU, this traits will have no effect because such special memory does not exist. This reflects the idea of having default values and a single source code. One can always specify a RandomAccess trait without worrying about the consequences on other hardware.

```
Kokkos::View<double*, Kokkos::RandomAccess>  j_dofs;
```

Kokkos may introduce optimizations in the future for random access on the CPU. This could improve performance without the user noticing any changes to his existing code.

3.4 Execution space, pattern and policy

3.4.1 Execution pattern

For a better understanding, we will now explain lines 27 of Listing 3.1 and 20,28 of Listing 3.2.

```
27 K::parallel_for(policy, *this);
```

```
20 R::forall<R::cuda_exec<128>>
28 R::forall<R::omp_parallel_for_exec>
```

Both Kokkos and RAJA use Execution patterns to call the parallel function. Here they are *K::parallel_for* for Kokkos and *R::forall* for RAJA.

The execution pattern defines the basic parallel algorithm such as simple loops with parallel_for (Kokkos) and forall (RAJA), reductions with parallel_reduce (Kokkos) and omp_reduce (RAJA), cuda_reduce (RAJA). The reductions are set differently as RAJA relies on special reduction variables and Kokkos has an extra input parameter on parallel_reduce for the reduction.

Scans are also available for both Kokkos and RAJA but were not used in our project.

Kokkos' pattern takes 2 parameters. First, the policy, which will be explained in the next section, 3.4.2. Second, the content of the loop is defined by **this* meaning that the class is used as a functor. The *operator()* is used as body-function of the parallel_for. Using the *operator()* as body of the parallel function is only possible with Kokkos. Another option, and the only option in case of RAJA, is to use lambda functions.

3.4.2 Execution policy

At line 19-24 of Listings 3.1 and line 20-23, 28-31 of Listing 3.2, the Execution policy of the parallel section is defined.

```
18 auto policy = K::TeamPolicy
19                    <Hp1, ExecSpace>
20                    (n_teams, n_threads)
21                    .set_scratch_size
22                    (1,K::PerThread
23                          (shared_size));
```

Kokkos first defines the Execution Policy: RangePolicy or TeamPolicy. The range policy simply defines a range of indices over which the function will be called. The user has no control over the order of execution or concurrency. It is not allowed to synchronize the threads.

The Team policy, used at line 18, adds features over the Range policy and is therefore more complex. It adds hierarchical parallelism, and scratch pad memory.

The first template parameter, Hp1, of the TeamPolicy is optional and will be explained in section 3.5. The second parameter, ExecSpace, defines the architecture running the code. It can be Cuda, OpenMP, PThreads or just CPU serial. When multiple architectures are available and enabled, the default architecture will be choosen with this ordering, Cuda > OpenMP > PThreads > serial. If using the default execution space, the memory space will be set correctly as Kokkos' default values will match. Otherwise, the user has to make sure the memory space is compatible with the execution space. Host memory space (CPU), not being accessible from the Device (GPU), will not be compatible with Device execution space.

Here are the 6 execution spaces for Kokkos.

```
Kokkos::DefaultExecutionSpace
Kokkos::Serial
Kokkos::Threads
Kokkos::OpenMP
Kokkos::Cuda
```

With the Team policy, one defines a number of Teams without size restriction and a number of threads per Team limited by the hardware. The two values are passed as parameters of TeamPolicy at line 20.

The Team policy introduces scratch pad memory and hierarchical parallelism, not available with the simpler RangePolicy. As for CUDA's shared memory, Kokkos' scratch pad memory size needs to be known prior to the parallel section.

It is either shared between threads of a Team or thread private. The total size of the Scratch memory is defined before the parallel section.

It is a very important feature for a PIC application as each thread needs private variables to compute updated particle positions, velocity, and so on. The impact of not having private variables (not available with RAJA) will be discussed in the case of the RAJA code.

At lines 21-23 of Listings 3.1, we request scratch memory:

```
21                    .set_scratch_size
22                    (1,K::PerThread
23                          (shared_size));
```

The first parameter, 1, is the memory level. It can be 0 or 1. Memory level 0 is faster but smaller. On GPU this allows to access the shared memory. The second parameter,

K::PerThread(shared_size), defines if the memory is accessible to the thread only, *PerThread()*, or shared to a team of threads, *PerTeam()*. The *shared_size* parameter sets the number of bits to allocate, per thread or per team. Once it is specified, we can initialize a scratch View inside the parallel loop. We demonstrate it here, with a code line extracted from the Hp1 function.

```
K::View<long[3], ExecSpace::scratch_memory_space,
                 K::MemoryTraits<K::Unmanaged>>
                 view_xi(team_member.thread_scratch(1));
```

The variable view_xi is thread private and only exists inside the parallel section. The parameter 1 in the *thread_scratch* function specifies which memory level to access. It has to match with the first parameter of the *set_scratch_size* function.

Lastly, the Team policy introduces hierarchical parallelism by assigning an execution pattern to threads within a Team inside an execution pattern on the Teams. The most straight forward example is a nested parallel_for. The first parallel for will be distributed among Teams and the second parallel_for will be distributed to the Team's threads. We can apply the same with nested parallel_reduce or parallel_scan. More sophisticated behavior can be easily obtained by nesting different execution patterns. The inner or outer loop can be a parallel_reduce while the other is a parallel_for.

Although we presented RAJA's "pattern" in section 3.4.1, RAJA does not really have a "pattern". But for the comparison to Kokkos, we presented the forall policy with Kokkos' pattern. Kokkos calls the parameter of the pattern the "policy". RAJA names both concepts under policy.

```
20 R::forall<R::cuda_exec<128>>
21     (R::RangeSegment(0,
22       this->particles.group[0].n_particles),
23       [=,*this] RAJA_DEVICE (int i_part){.});

28 R::forall<R::omp_parallel_for_exec>
29   (R::RangeSegment(0,
30   this->particles.group[0].n_particles),
31     [&,this] (int i_part){.});
```

The template parameter of the forall policy describes where the execution will occur. Line 20, cuda_exec, will execute on the GPU and line 28 omp_parallel_for_exec will execute on the CPU using OpenMP. The first parameter of the policy is segment iteration policy, line 21-22 and 29-30. In our case, a RangeSegment starting at 0 with n_particles elements. This is similar to Kokkos' RangePolicy.

A feature that only RAJA provides is the definition of complex List segments, i.e. the indices the loop will be ran on with can be defined from 0 to $n-1$ (0, 1, ...,n-1). Or with constant stride to get every second (0, 2, 4,...), third (0, 3, 6,...),... n-th number. But the list can also contain arbitrary numbers (1, 2, 5, 7, 9). The list can also be defined with a concatenation of the previously mentioned possibilities. A use case would be to have a mesh split in a number of subsets in which the computation at each vertex is independent from the other vertex of the same subset. This indices of each subset would be given to different lists and the parallel loop will be called on each list.

3.5 Parallel section

As introduced in section 3.4.1, there are different parallel operations such as parallel_for, parallel_reduce, parallel_scan for Kokkos. For RAJA there are, among others, forall and scan. These operations are called collectively *parallel dispatch* for Kokkos. With these 3 operations, "for loops" with independent iterations, reductions and prefix scans can be implemented. Currently the reduction loop supported on both CPU and GPU only covers scalar reductions. Kokkos provides additional experimental solutions for vector reductions. It will also be discussed in more detail in section 5.1 as it is a key element of the GEMPIC algorithm and many more algorithms.

3.5.1 Vector reduction on Kokkos

On RAJA we had to implement the reduction ourselves. However, on Kokkos we could use a *scatter_view* from the experimental class. This type required two different definition for CPU and GPU. But we expect Kokkos to add default settings to this type later as it is today still in the Experimental section.

```
#if defined(KOKKOS_ENABLE_CUDA)
typedef K::Experimental::ScatterView
        <double *, K::LayoutLeft, ExecSpace, 0, 0, 1> ViewScatterType;
typedef K::Experimental::ScatterAccess
        <double *, 0, ExecSpace, K::LayoutLeft, 0, 1, 1> ViewScatterAccessType;
#else
typedef K::Experimental::ScatterView
        <double *, Layout, ExecSpace, 0, 1, 0> ViewScatterType;
typedef K::Experimental::ScatterAccess
        <double*, 0, ExecSpace, Layout, 1, 0, 0> ViewScatterAccessType;
#endif
```

We defined two types: *ViewScatterType*, for the main variable, *ViewScatterAccessType*, to access the *ViewScatterType* from each threads inside a parallel section. The *ViewScatterAccessType* variable only exist inside the parallel section. First the main variable is initialized with a View as parameter.

```
ViewScatterType scatter;
this->scatter = Kokkos::Experimental::create_scatter_view
      < Kokkos::Experimental::ScatterSum>(this->j_dofs);
```

Then, from the parallel section, the *ViewScatterAccessType* variable is initialized

```
ViewScatterAccessType j_dofs_scatter_access = j_dofs_scatter->access();
```

We use the *ViewScatterAccessType* as a standard View inside the parallel section. Once the parallel section is closed, the data needs to be gathered using the *contribute* function.

```
K::Experimental::contribute(hamiltonian.j_dofs, hamiltonian.scatter_view);
```

Finally, *j_dofs* contains the reduction result. This type will behave differently on CPU and GPU. The difference will be explained in section 5.1.

3.5.2 Functors and lambda functions

The parallel calls rely mostly on lambda functions, but a functor can also be used for Kokkos. In any case, the inputs of the functions have to match the expected input parameters of the

execution pattern and policy. This prohibits to pass any data as arguments. Therefore lambda functions are much easier to work with on a small scale. Functors require all the data as members.

Functors are easier to test and make for a more readable code for important functions. But it brings one problem, what if multiple parallel functions need to be implemented? To answer this, Kokkos uses an Execution Tag. The tag is an empty structure that is passed as the first argument of the *operator()*. The tag is also passed as a template parameter of the Execution Policy, which will then feed it to the *operator()* making it a compile-time decision. This gives the parallel dispatch the ability to call one specific declaration of the *operator()*. This is used at line 19 of Listing 3.1.

```
18 auto policy = K::TeamPolicy
19                  <Hp1, ExecSpace>
                    ...;
```

The first template parameter, Hp1, is the Tag that will define which *operator()* to use.

4 GEMPIC code analysis

4.1 Integration loop

The GEMPIC code (selalib3d3v) integrates the equations (2.30-2.33, 2.38-2.41, 2.60-2.71) using a loop over time steps. At each iteration, the electric and magnetic fields are computed and the particles are pushed according to the fields. The method uses the so called Strang splitting [23], giving the loop shown in Fig.4.1.

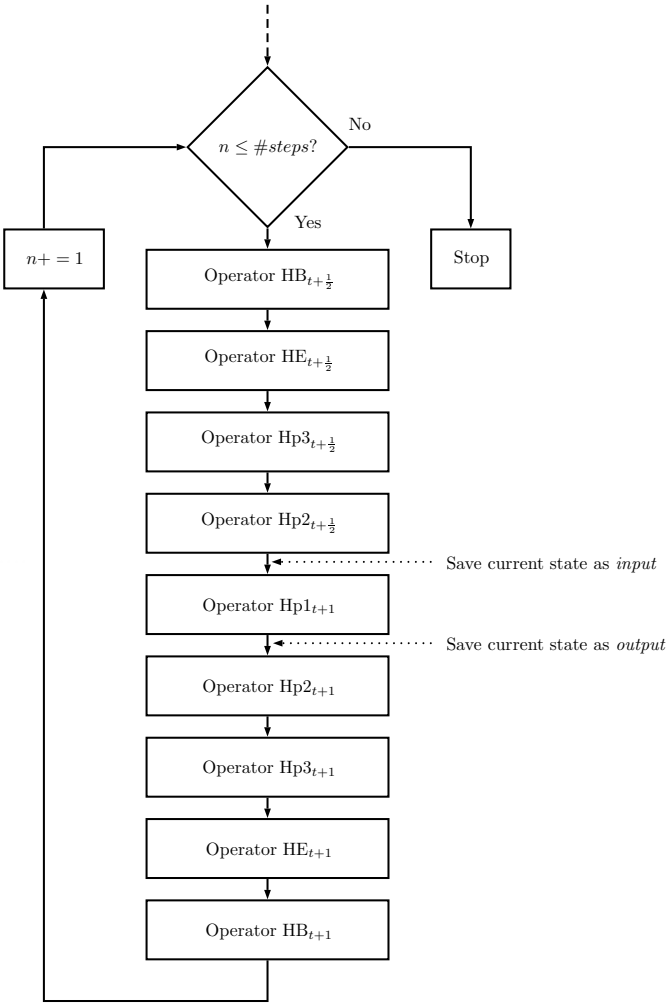

Figure 4.1: Schematic of the time loop resulting from the Strang splitting of the equations (2.30-2.33, 2.38-2.41, 2.60-2.71), as implemented in the GEMPIC code. The moments we extract the variables to produce our benchmarks are shown with arrows.

4.2 Performance profile

As stated by Amdahl's law [1] the speed-up of a parallel program is bound by its percentage of serial code. We are therefore interested in finding the most time consuming section of the GEMPIC code to obtain the most significant improvements by parallelizing. The analysis on the selalib3d3v code was performed using the gprof profiler for GNU FORTRAN. From the analysis we obtained the five most time consuming functions of the code, shown in Tab. 4.1.

They match the hints that Dr. Katharina Kormann had given to us, the most time consuming section of the code is pushing the particles and computing the contribution of each particle to the mesh. Together they take 80-85% of the compute time.

We observe in the analysis that the OperatorHp2 and OperatorHp3 routines are taking each around twice the time of the OperatorHp1. This is simply due to the Strang splitting that computes one operator only once per time step. While all other operators are computed twice over half time steps.

The performance when varying the time steps, number of particles or mesh size was evaluated for our five functions.

In Fig. 4.2, we see the total time of the computation increasing significantly with the size of the mesh but not our five functions' execution time. For the increased mesh sizes, other functions dominate the CPU time. As expected we can ignore the mesh size for the operator Hp1 function.

In Fig. 4.3, we see that all the functions scale linearly with the number of particles as we would expect for a loop going through each particle once.

In Fig. 4.4, we see all the functions scaling linearly with the number of time steps. This is particularly important in a PIC simulation. Because of the computation of the contribution of each particle to the mesh, there is a highly randomized access to the mesh data if the particles are not sorted. PIC simulations starting with sorted particles will have an increasing time per iteration as the particles lose their ordering. After verification, we can confirm that the particles are never ordered, hence constant time per iteration is seen over the entire simulation.

The addCurrent functions are not ranked first as computationally intensive individually, but together they are. The separation of the 3 functions comes from the Hamiltonian splitting. These functions perform exactly the same task along a different axis. In C++, template parameters can be used so that a single function is written for the 3 different axis. The axis will be picked according to the template parameters at compile time, hence adding no overhead to

Name	Percentage
evaluateFieldSingleSpline3dFeec	30.43%
addCurrentUpdateVPrimitiveComponent3Spline3d	15.66%
addCurrentUpdateVPrimitiveComponent2Spline3d	15.56%
addChargeSingleSpline3d	9.67%
addCurrentUpdateVPrimitiveComponent1Spline3d	9.65%

Table 4.1: Top 5 functions sorted by the percentage of the total execution time the program spent in each, obtained using the gprof FORTRAN profiler. This 3D3V simulation runs on 1 core with 10000 time steps, 1 million particles and a mesh containing 1024 cells.

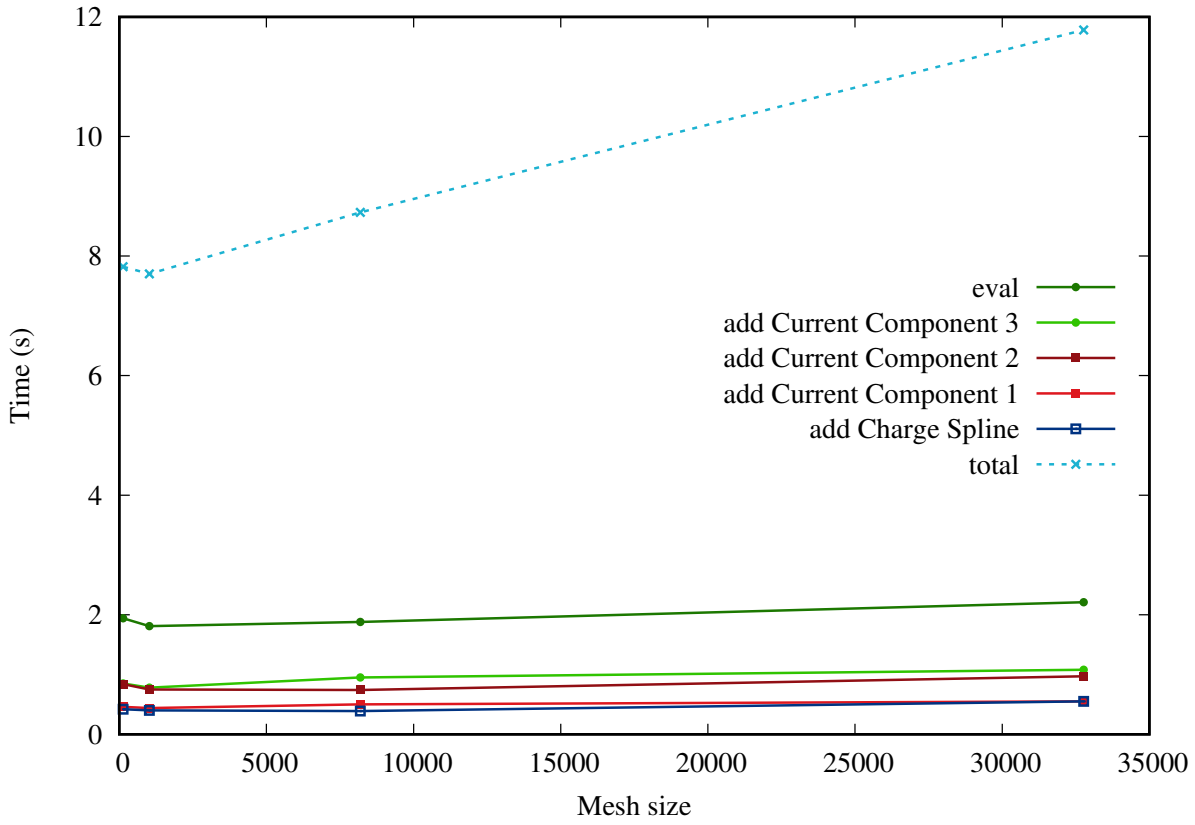

Figure 4.2: Execution time of the five main functions depending on the mesh size. This 3D3V simulation runs on 1 core with 100 time steps and 10000 particles.

the function.

They are all functions applied to one particle and belong to a "For all species, For all particles" loop.

The addCurrent functions are called by the OperatorHp1, OperatorHp2 and OperatorHp3 functions, respectively. Again, the functions perform the same task for the 3 spatial axes. One of the main tasks performed by these functions is to push the particles, update their position, using their velocity. The OperatorHp1, 2, 3 are the functions running the "For all species, For all particles" loops. Together with the two other function OperatorHE and OperatorHB (electric and magnetic field) they form one time step iteration. To compute the electric and magnetic fields, the contribution of all particles to the mesh are needed. In the FORTRAN code, the particles are split over multiple processes that communicate using MPI. The contribution is gathered at the end of each OperatorHp1, 2, 3 function.

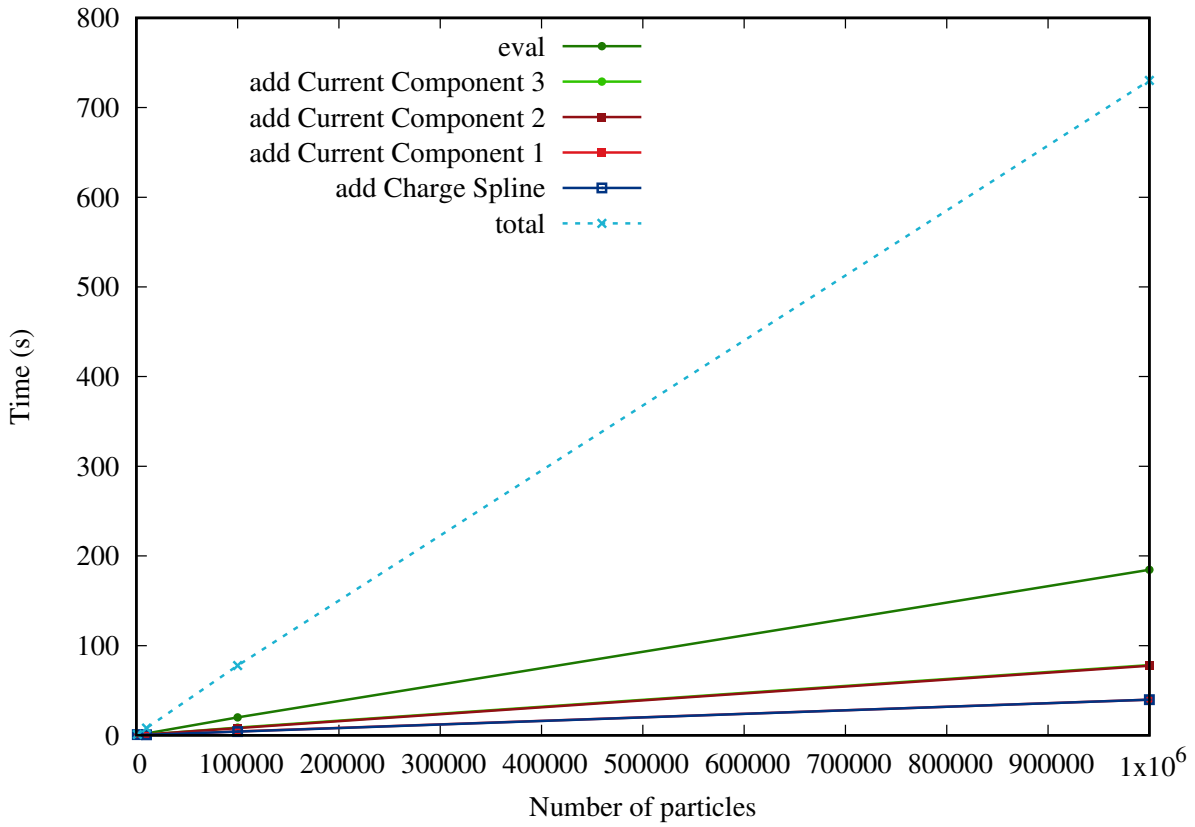

Figure 4.3: Execution time of the five main functions depending on the number of particles. This 3D3V simulation runs on 1 core with 100 time steps and a mesh containing 1024 cells. The *add Current Component 3* and *2* curves as well as *add Current Component 1* and *add Charge Spline* curves are overlapping in the plot.

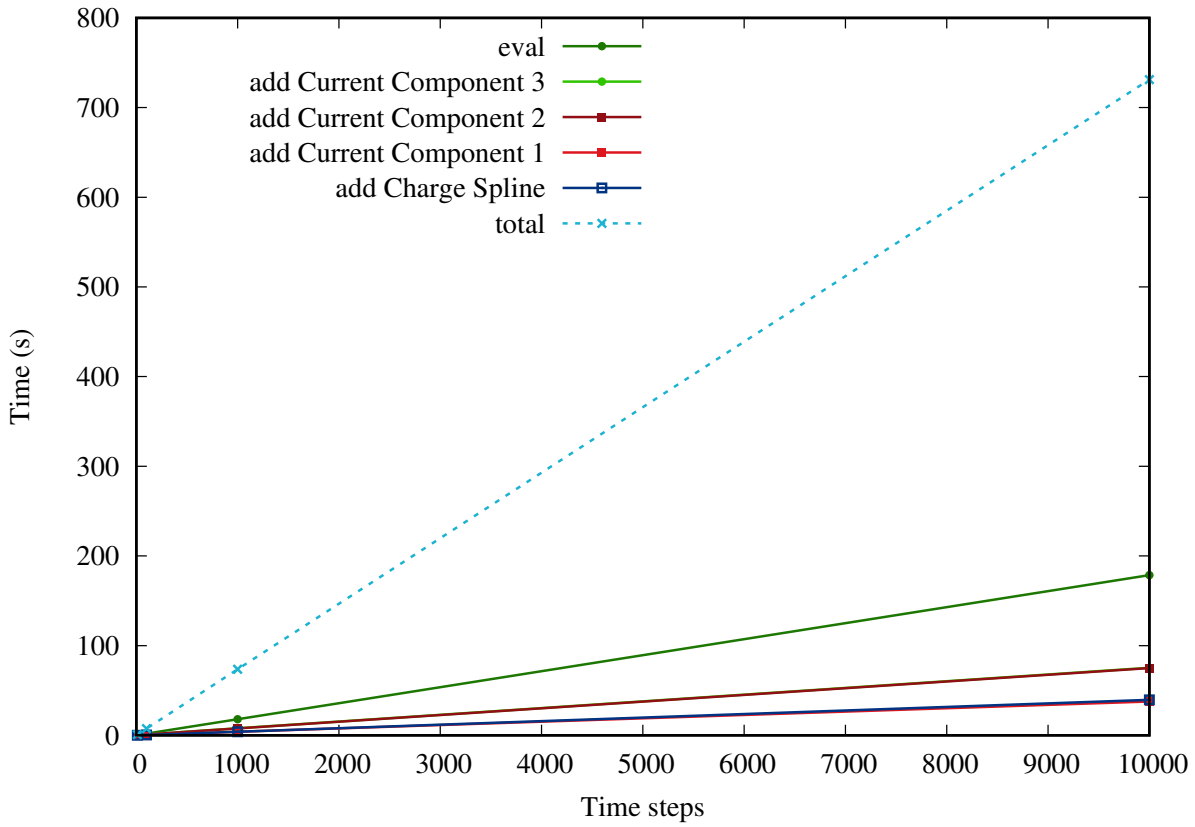

Figure 4.4: Execution time of the five main functions depending on the number of timesteps. This 3D3V simulation runs on 1 core with 10000 particles and a mesh containing 1024 cells. The *add Current Component 3* and *2* curves as well as *add Current Component 1* and *add Charge Spline* curves are overlapping in the plot.

4.3 C++ implementation

We consider the OperatorHp1 function up to the MPI call to be our test code for this project. As seen in Fig. 4.1, all physical values are copied before and after the OperatorHp1 function. Because not the entire code has been implemented in C++, text files are used as input and output. Knowing the input and output of the FORTRAN code, we check the correctness of the C++ implementation by comparing the results with the output file. Three time steps were run. To obtain meaningful execution time and to debug easily, we vary the number of particles between 100 and $1e7$ and the mesh size from 128 cells to 1024. In a MPI-distributed simulation on a supercomputer, having $1e7$ particles per node is considered a large simulation, as it would results in billions of particles overall, counting each process. Thus, our largest test case, that we perform on a single node, is representative of a very heavy workload. In PIC simulations, memory access can be optimized by ordering the particles such that particles close in space will be close in memory. This is not implemented in the GEMPIC code. A Sobol sequence, a low-discrepancy sequence, is used to initialize the particles so they are not sorted initially. This allows us to use the first 3 frames as representative sample of a random iteration of the simulation. During an entire run, there will be very little difference in the memory access pattern for the particles as they will never be sorted.

The sorting of the particles is an important step of PIC simulations. In [4], the author obtains around 50% improved performance from his implementation of the sorting algorithm in a PIC simulation. Another way of managing the cache-trashing coming with unsorted particles is to change the data structure used. For example, storing the relative position to the cells of the particles serves as an automatic sorting system. The particles are directly accessed cells by cells by the result of the data structure used. The implementation of the GEMPIC model being mostly a proof of concept, the question of sorting the particles has not been addressed yet[1].

[1] Dr. Kormann, personal communication.

5 Implementation and usability review

In order to compare Kokkos to RAJA but also to see how they perform compared to architecture-specific frameworks, we developed multiple versions of the same function Hp1 from the original FORTRAN routine. These are the 5 different implementations:

C++ We first developed a standalone code by re-implementing the Hp1 function from the GEMPIC FORTRAN code in C++. This code includes all the necessary input/output functions to run the same computation as the FORTRAN code. This test bench is also used as the baseline for the other codes below.

OpenMP Extending from the C++ code, we implemented a parallelized version for the CPU using OpenMP. The results obtained from this code in terms of performance will serve as reference for Kokkos' and RAJA's CPU implementations.

Kokkos Getting this version of the code to work well took approximately two months from installing the library, testing it, learning the different features, implementing our section of the GEMPIC code and correcting all the small issues related to performance, such as variables passed by value/reference.

RAJA The parallel implementation using RAJA was the next-to-last version implemented. It is important to point out that RAJA was approached with the knowledge acquired while implementing the Kokkos code. It was therefore easier at first as most of the abstractions are the same as with Kokkos. However, some features we used in Kokkos, such as per thread scratch memory, are lacking in RAJA.

CUDA Lastly, to have a relevant reference for Kokkos' and RAJA's GPU implementations, a CUDA version was implemented.

All codes have also had an algorithmic optimization compared to the FORTRAN implementation. As it will be explained, instead of computing the mesh index of the particles every time they are called, we precompute them and store them in an array. The performance benefit will be shown in chapter 6.

5.1 Kokkos

Kokkos provides a default execution space that will look for the available architectures in the ordering mentioned previously. This is the first step for a very simple implementation of a portable code. This default execution space relieves the user from thinking about to which architecture he will deploy his code. Complex algorithms can of course have different parallel

sections to use different execution spaces.

The first issue encountered was getting a member function to run in Kokkos' parallel dispatch. To do so, the *operator()* is used, as it is the only member function usable for Kokkos. It can be used to implement the desired function or simply as a wrapper to call another member function. The parallel dispatch takes the class as a functor, and therefore calls the *operator()* function by default.

This removes the drawbacks of using lambda functions, allows for easier testing, and leads to more readable code. But it introduces one problem, what if multiple parallel functions need to be implemented? To answer this, Kokkos uses an Execution Tag. The tag is an empty structure that is passed as first argument of the *operator()*. The tag is also passed as a template parameter of the Execution Policy, which will then feed it to the *operator()* making it a compile-time decision. This gives the parallel dispatch the ability to call one specific declaration of the *operator()*.

Another necessary element for our application is a vector reduction. This feature was a little bit harder to find. The Kokkos documentation only, but extensively covers the case of scalar reduction in the documentation[1], examples and test files. This issue had already been mentioned to the Kokkos team a year prior to the writing of this thesis. Now Kokkos provides a ScatterView class under the Experimental namespace. The ScatterView class has a different behavior on CPU and GPU.

On the CPU, the ScatterView class will duplicate its View and provide internally one View to each thread. Each thread works on its View. For the user, it is only visible as a single ScatterView. After the parallel section, the *contribute* function has to be called to compute the reduction of all the internal Views.

On the GPU, the ScatterView does not duplicate its View. Duplicating the View to give one copy to each thread removes race conditions, the undefined behavior occurring when two or more threads are trying to modify the same variable at the same time. Race conditions can be avoided by adding a rule to the access of the variable. A rule, the so-called atomic operation, forces the processor to verify that there is no thread accessing the variable before modifying it. The obvious drawback is the slowdown that can occur from restraining threads to access memory when dealing with atomic operations. But nowadays, the atomic add operation is extremely efficient on the GPU [12]. This makes the atomic operation in the case of a reduction, instead of a variable duplication, more and more used.

The type ScatterView is templated with a value to specify if we want the duplication or the atomic access but there are no default settings depending on the architecture. A typedef of this type was used and separates the template settings for CPU and GPU with a #ifdef. Using Kokkos, this is the only time two versions of a code section (2 lines) had to be implemented.

Lastly, we need scratch memory for each thread. With OpenMP, this is achieved simply by starting the parallel section, before the "for loop" and declaring all the needed variables there such that allocation occurs per thread. In CUDA using shared memory can produce faster codes. Kokkos manages both cases with Views on their "scratch_memory_space" and an "Unmanaged" Memory Traits. This gives access to PerTeam and PerThread memory. The total size of the scratch memory inside a parallel loop needs to be known before the parallel dispatch

[1] https://github.com/kokkos/kokkos/wiki/Custom-Reductions:-Build-In-Reducers, accessed on 09/11/2018

is launched, similarly to CUDA. Different memory levels with different size cap, corresponding to L1 and L2 cache sizes, can be accessed on CPU and shared memory on the GPU.

To finish this user review of Kokkos, we can give two examples of interactions with the developers. We opened two threads in Kokkos' issue[2,3] section, one concerning overheads on parallel section when the functor contains a large *std::vector* and the second concerning issues when switching from a Tesla K40m GPU [9] to a Tesla P100 GPU [7]. Multiple developers from Kokkos provided guidance on solving the problems within less than 24 hours. For the overhead, the reason is the copy of the functor by Kokkos when launching a parallel section using the *operator()*. Using *std::shared_ptr* solved it. The issue that appeared on the P100 should have already appeared on the K40m as it was due to a wrong memory management. We speculate that the P100 with Compute Capability 6.0 has better error detection than the K40m with Compute Capability 3.5.

5.2 RAJA

Once correct benchmarks for Kokkos on the CPU and the GPU had been obtained, we turned towards the RAJA framework.

The first difference observed is that Kokkos tries to have default values depending on the architecture. For simple tasks, omitting the template parameters will result in code working on the CPU and the GPU depending on the compilation. RAJA does not implement such default values and lets the user specify explicitly all the settings. This makes the user aware of what is happening but also requires to duplicate every line of code to include a CPU and a GPU version.

RAJA does not use functors for the parallel dispatch but only lambda functions. This was not a big issue in our situation, it just requires to shape the code differently.

As for the Kokkos implementation, scratch memory is needed. RAJA has an implementation of shared memory for the GPU. In November 2017, a new issue thread was posted discussing writing an example of the Shared Memory for the GPU and the CPU using macros. Six month later, it can be found as part of the examples provided by RAJA. They use shared memory in solving the 2D wave equation. In this case, shared memory can be used to store the matrix containing the values at each cell of the mesh. If we would use this implementation for our per thread scratch memory, we would need an array of $n_{threads} \times ScratchMemorySizePerThread$. Each thread would be assigned one row (or column) to work with.

This is the most common solution. But passing the different arrays to our function instead of using the lambda capture was not trivial. After not getting an answer on the issue thread[4] of RAJA asking if this was the right way to implement scratch memory, we ended up using regular Views. On the K40m GPU, the code ran faster than Kokkos' version but much slower on the P100 GPU.

Lastly, we modified RAJA's source code slightly to make it work with our code design. We now explain the code design to show where the problem comes from. Class A, a C++ class named A, contains a member V of type View. In RAJA, the View type does not have a default constructor. So when the constructor of A is called, the constructor of V should be in the

[2]https://github.com/kokkos/kokkos/issues/1615, accessed on 10/09/2018
[3]https://github.com/kokkos/kokkos/issues/1686, accessed on 10/09/2018
[4]https://github.com/LLNL/RAJA/issues/492, accessed on 10/09/2018

initializer list. Not knowing the size of the View before the constructor of A has read the input file, having V in the initializer list is impossible.

To remove this constraint, a simple empty constructor was added to the RAJA View source file.

```
RAJA_INLINE RAJA_HOST_DEVICE constexpr View() {}
```

The second modification was to remove the const type qualifier on the Layout variable from the View structure.

```
layout_type /*const*/ layout;
```

Removing it restored the *operator=* of the structure View.
The question concerning our modifications to the source code is still unanswered[5].

In fairness to RAJA, we note that no benchmark will test every aspect of parallel programming. In a side-by-side comparison, a library can be disadvantaged by the nature of the benchmark. In [18, 19], by working on a heat conduction problem, the authors tackle 2 of the "Dwarfs of Parallelism" introduced in [2], which discuses a list of 13 high-level classes of problems that try to capture a large variety of parallelism applications. This list includes dense and sparse linear algebra, spectral methods, N-body methods, structured and unstructured grids. One of the goals of this list is to establish a variety of benchmarks to evaluate parallel performance. Because there is no application tackling every problem, each benchmark focuses on a few of the Dwarfs. The heat convection problem deals with the two classes structured grid and sparse linear algebra. Our application, a PIC model, although it come close to the N-body methods, it does not quite fit in the list of 13 Dwarfs. This illustrates that no single benchmark can correctly evaluate all aspects of parallel computing. Therefore, for completeness, we will briefly introduce a feature of RAJA that is not required in our application. This is presented as an important feature of the framework and indeed, Kokkos does not have it. Our remarks and conclusion regarding Kokkos and RAJA only covers the features we used so RAJA may be evaluated and judged differently on another benchmark.

RAJA is promoting its ability to do parallelization over a list of indices. Traditionally, the indices are defined in the "for loop" for OpenMP or by the thread index in CUDA. The user can also use a formula with the thread index to describe more complex behaviors. To parallelize over even numbers, we would multiply by 2 the loop index and the loop index would be scaled accordingly. With RAJA, a list of indices can be created and used as indices for the parallel dispatch. Updating every even index of a vector, then updating the odd indices is simply done by listing the indices beforehand and giving the list to the Execution Policy. This is commonly referred to as indirect indexing. This interesting feature, not provided by Kokkos, is not used by our application, hence we do not have any comparison on this.

[5] https://github.com/LLNL/RAJA/issues/508, accessed on 05/09/2018

6 Performance benchmarks

In this chapter, we will first present the results obtained on the computer used to develop the different codes. The node provides two Intel Xeon E5-2680 v2 @ 2.80GHz CPUs and two Tesla K40m GPUs [9] (only one is used in our codes). This cluster also has P100 GPUs [7] with IBM POWER8 processors. In section 6.2 the analysis and corrections presented were done on a second cluster with an Intel Haswell Xeon E5-2698 v3 @ 2.30GHz CPU, because this cluster had different debugging tools. For our last benchmark, we obtained the access to a third cluster and its Intel Xeon Gold 6130 @ 2.10GHz CPU and two Tesla V100 GPUs [8] (only one is used in our codes). This final benchmark provides the most relevant information regarding the expected performance on present-day hardware.

Our codes utilize one CPU and one GPU of a single cluster node. We will use the following names to denote the hardware used from each cluster:

Ivy-Kepler Single node with an Intel Xeon E5-2680 v2 @ 2.80GHz CPU and a NVIDIA Tesla K40m GPU.

IBM-Pascal Single node with an IBM POWER8 processor and a NVIDIA Tesla P100 GPU.

Haswell-N Single node with an Intel Haswell Xeon E5-2698 v3 @ 2.30GHz CPU without GPU.

Skylake-Volta Single node with an Intel Xeon Gold 6130 @ 2.10GHz CPU and a NVIDIA Tesla V100 GPU.

6.1 Algorithmic optimization

We performed performance analysis and optimization on all codes within a limited time, going from a few days to a couple of weeks, e.g. to understand and try different solutions for RAJA, see section 6.2. So we acknowledge that performance could be improved although it should already be representative of the performance one can obtain with the same amount of work on the different libraries.

The test are done on an Ivy-Kepler. We also ran the GPU codes on IBM-Pascal.

Because of the algorithm we are working on, the parallelization seemed simple at first. The loop over all particles is parallelized. The main computation of the algorithm is the particle-to-mesh transfer for which a kernel function centered on each particle is computed at each grid node. This requires the localization of the particles within the mesh. The high number of modulo operations performed is the highest load of the program. Finding optimal data structures for the particles is the subject of ongoing research (see e.g. [3]). One could store the position on the mesh and the relative position of the particle within a cell. In our case, the code originally computed the grid index every time it was needed.

The FORTRAN code uses MPI to parallelize the workload. With 100000 particles on 100 CPUs, the simulation has a total of 10 million particles. We focus on one MPI process and use 3 time steps to test the performance and verify the solution. To get significant run times, we scale the number of particles to 10 millions on one CPU with the original 16 by 8 by 8 mesh. With this setting, the FORTRAN code runs in 7.5 seconds per iteration.

In Fig.6.1 we compare the log_2 of the execution time as a function of number of CPU cores for the different implementations. First we note the C++ implementations are about 2 times slower than the FORTRAN one. With OpenMP, the speed-up stays very close to the ideal line with a little degradation for 8 to 10 cores. Kokkos CPU is 15% slower than pure C++ with one thread but scales perfectly, bringing its performance closer to OpenMP's performances when running on more than 4 cores.

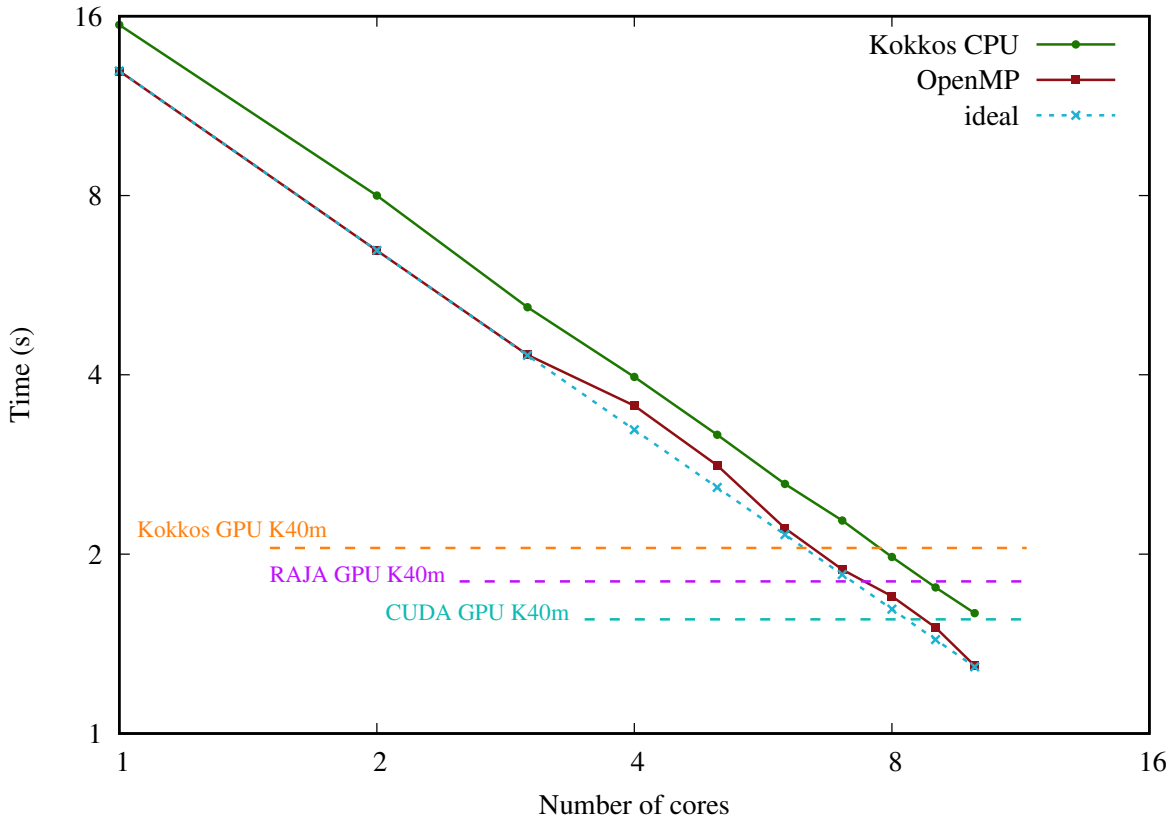

Figure 6.1: Log-log plot of the compute time as a function of number of cores. We compare the performance of Kokkos on CPU and GPU to OpenMP, Cuda and RAJA on GPU. The codes, with the original non-optimized modulo computation, were run on an Intel Xeon E5-2680 v2 @ 2.80GHz and a NVIDIA Tesla K40m. Results for the GPU are independent of the number of CPU cores and are placed arbitrarily on the abscissa.

As said previously, the most expensive part of the code are the modulo operations computed to get the grid position of each particle. Without changing the data structure, we optimized the

code by rearranging the modulo computation and storing it in arrays. We replaced computation by memory storage and reuse. The arrays are of the size of the 16 by 8 by 8 mesh. Instead of computing the positions on the fly, we are now precomputing them for each particle. This reduced the processing time by a factor of 2 for the C++ implementations but did very little to the FORTRAN implementation.

In Fig.6.2, the "Opti" suffix means that we are looking at the code with the improved modulo computations. The OpenMP code is still as fast as the serial code with 1 thread but the scaling is slightly worse. Kokkos CPU on 1 thread is 15% slower compared to the serial code, but again, scales perfectly bringing it very close to the performance of OpenMP around 6 to 10 cores. For the GPU implementations, we are only comparing the optimized codes but used two different graphics cards. The oldest, K40m gave results comparable to the 8 to 10 cores CPU codes, with CUDA being faster than RAJA and Kokkos being the slowest.

Using the newer P100 GPU drastically changed the results. RAJA GPU is slightly faster than the 10 cores CPU codes but is now the slowest of the GPU implementations. Compared to the K40m, RAJA is 3.5 times faster. The second fastest implementation is CUDA with a speed-up of 3.75x compared to the K40m. Kokkos has an impressive 7x speed-up compared to the K40m performance placing its performance close to a fourth of the 10 cores CPU speed.

We did not manage to get as good results for RAJA CPU as for the rest of the codes. The single thread performance are the same as with Kokkos but we have identified a false sharing problem causing the code to not scale as one would expect, see section 6.2.

In [18, 19], the authors compare Kokkos, RAJA, and different standard parallel implementation languages such as OpenMP, CUDA, OpenACC and OpenCL. The comparison is done on a simpler code. They used a simple heat conduction problem for the basis of their tests. Similarly to us, they arrive with Kokkos and RAJA between 5% and 30% of OpenMP's performance.

As in [18, 19], we compared RAJA, Kokkos and OpenMP implementations of the same code and arrive at the same conclusion that those new performance portable framework are relatively easy to use if all the features needed by a user are available. But it is up to the user to decide if the slight reduction in performance with the performance portability framework compared to architecture-specific frameworks is acceptable. The question is if one can tolerate a 15% performance reduction for an easy CPU/GPU implementation.

Although 15% overhead is very high in the case of the GEMPIC code and plasma physics simulations, it could be interesting to use Kokkos or RAJA as a first development step and later rewrite the critical sections with specific architectures in mind. In that case, it would be interesting to compare a complex code with multiple parallel sections in which Kokkos loops are mixed with standard C++ data and OpenMP loops with Kokkos' Views. This would bring an important answer to the question, which part of Kokkos causes the overhead, the data types or the parallel calls. Now that those libraries are available for an easy implementation of parallel codes, can they be used as a first draft and replaced with traditional OpenMP/CUDA code at the critical sections of the codes where a 15% performance hit is not acceptable? A future work would be to provide an answer to this question.

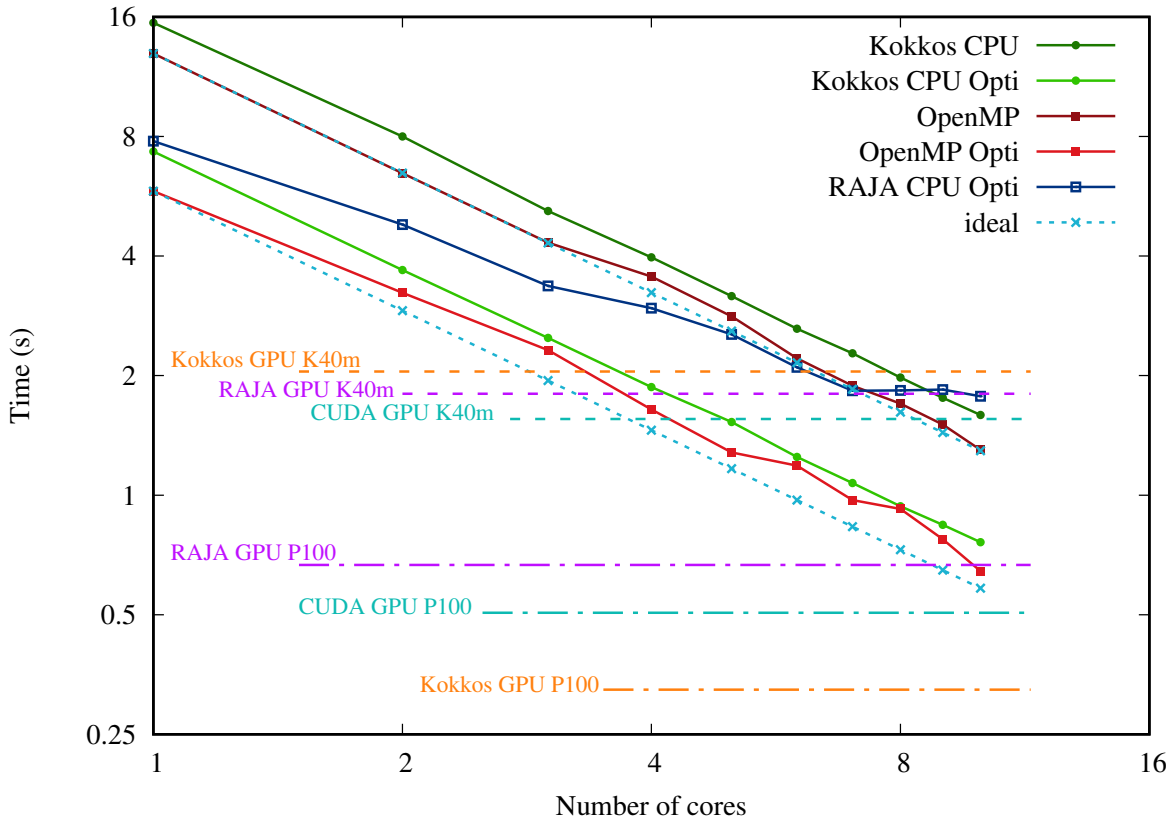

Figure 6.2: Log-log plot of the compute time as a function of number of cores. We compare the performance of Kokkos and RAJA on CPU and GPU to OpenMP and Cuda. Both non-optimized and optimized modulo computation are presented, the latter denoted with *Opti*. The codes were run on an Intel Xeon E5-2680 v2 @ 2.80GHz, a NVIDIA Tesla K40m and a NVIDIA Tesla P100. Results for the GPU are independent of the number of CPU cores and are placed arbitrarily on the abscissa.

6.2 False sharing optimization

As seen in Fig. 6.2, RAJA on CPU has the worst scaling, which is due to bad memory access. In Fig. 6.3 the performance of the 3 versions of the code are plotted. The first version was implemented similarly to the OpenMP and Kokkos code, except for the per thread scratch memory which was implemented using $\#thread * n$ Views. The second version implements the array with cache line alignment. The last version allocates the memory needed for each thread as one contiguous memory block.

Using the LIKWID performance tool [24], we were able to identify the problem. The tests and the results of the 3 RAJA codes were obtained on Haswell-N. Running a FALSE_SHARING analysis revealed that the OpenMP code has 226MB of Local LLC hit with false sharing when using 10 cores, the Kokkos code has 7MB. However, for the RAJA implementation, a much larger value of 18GB was determined, indicating a severe performance penalty due to false sharing.

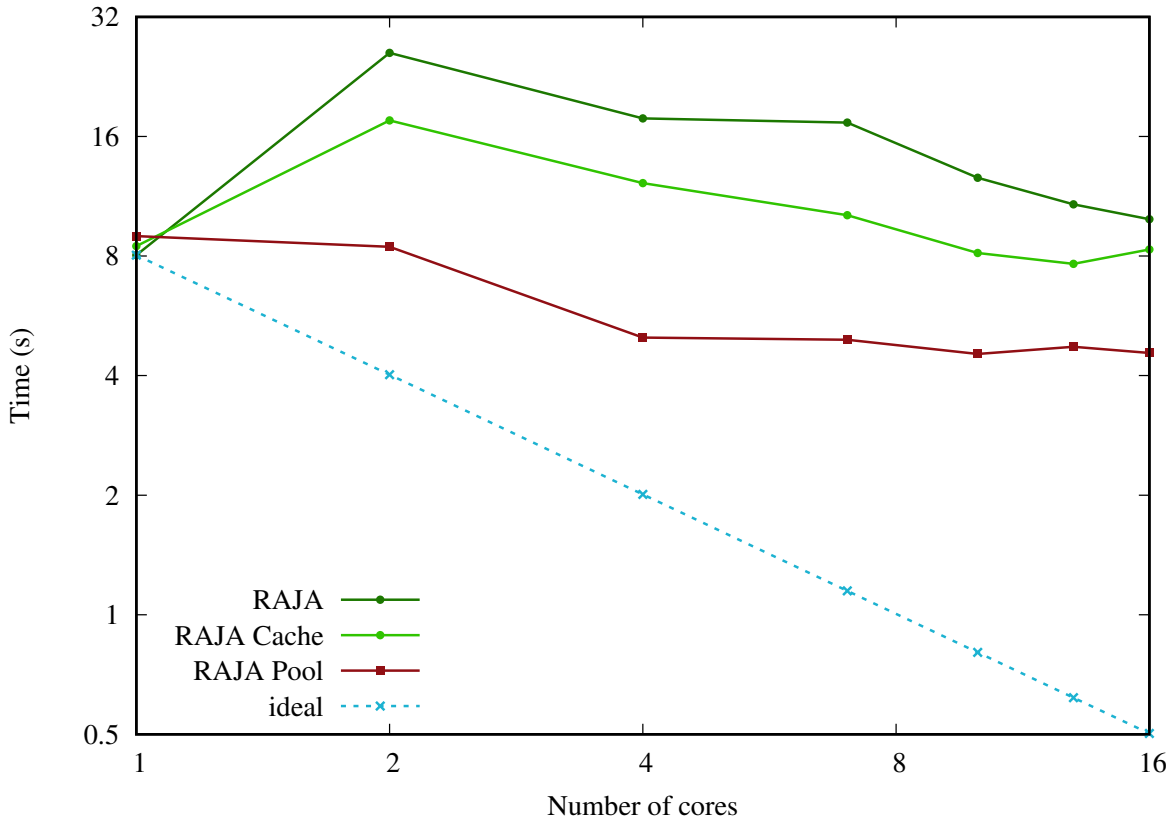

Figure 6.3: Log-log plot compute time as a function of number of cores. Three different versions of the RAJA code are compared. The first, named *RAJA*, uses arrays for each variables needing to be private for each thread. The second, named *RAJA Cache*, adds cache alignment to the previous *RAJA* code. Finally, *RAJA Pool* uses a single array as pool where each thread gets a full line to allocate variables contiguously. Although far from the ideal scaling, only *RAJA Pool* presents a bit of scaling.

The Local LLC hit with false sharing refers to the amount of memory the processor had to fetch from the Last Level Cache, the L3 cache on the Haswell architecture, because the cache line from the L1 or L2 cache it was working with was also used by another core. Cache lines are 64 bits long on x86_64 platforms.

In Fig. 6.4, we can see that Core 1 has in his L1 cache $(x_1, ..., x_8)$ just like Core 2. Core 1 is only modifying x_1 and Core 2, x_8. But when giving the result back to the L3 cache, both cache lines $(x_1, ..., x_8)$ cannot be accepted as one may have modified the values first. Let's say Core 1 modified the values first. Then the computation done by Core 2 is unvalidated and Core 2 has to copy the new $(x_1, ..., x_8)$ from the L3 cache again.

Ideally, threads work on data that are far away physically, in comparison to the size of a cache line, typically 8 doubles. So cache line interference is not common.

Our algorithm retrieves the position, velocity and charge of a particle from a large array. It then works with local variables to compute the new state of the particle before updating the

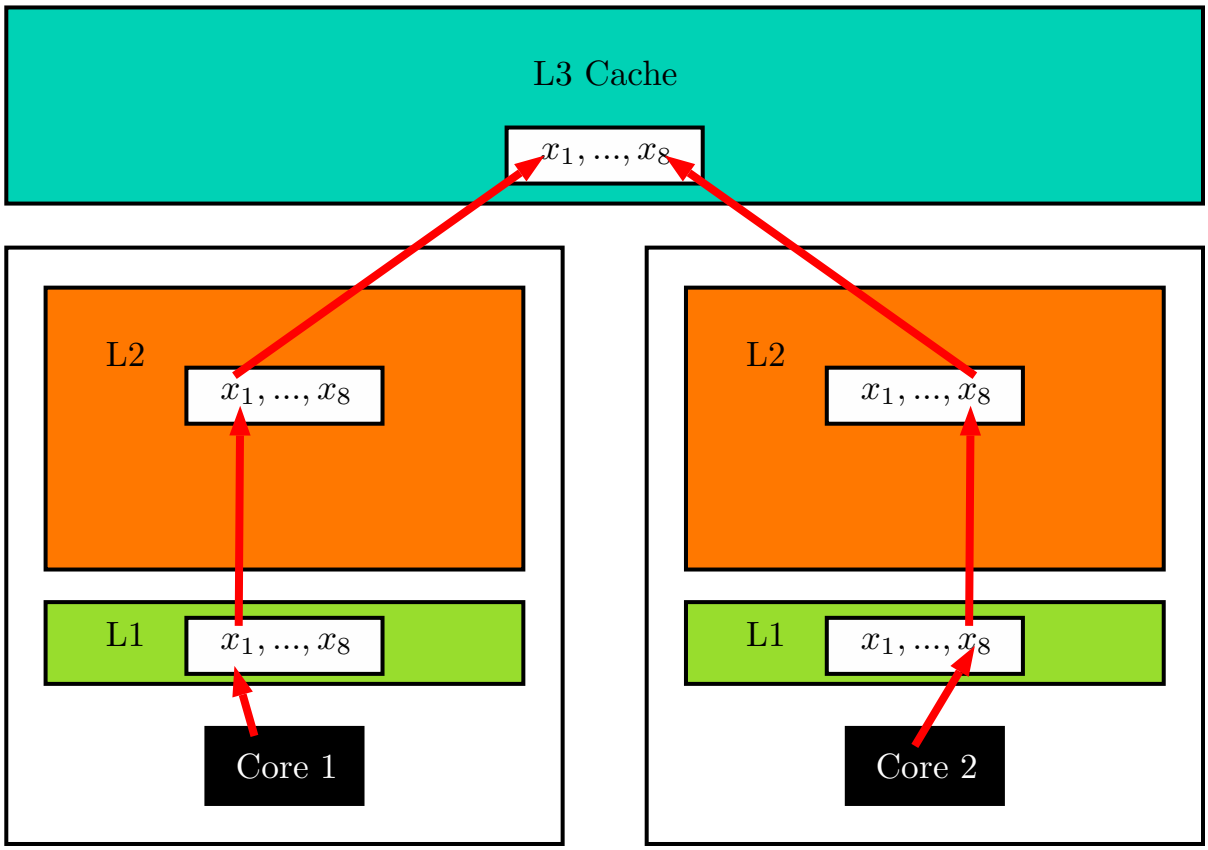

Figure 6.4: False sharing occurs when two threads having in their cache the same variables are trying to write to them. There is a conflict when trying to apply the changes done on the L1 cache back to the L3 cache, even if the variables modified by both cores differe.

large array. With OpenMP, the local variables are defined as thread private and with Kokkos, we use the per thread scratch memory. This is fine as the memory allocated to each thread is allocated as one block.

The first version of the RAJA code used a simple #threads-by-3 View. The first index is the thread number, the second index is the scratch data index, 3 in this example. Same for the velocity, and all other local arrays. In the memory we have, next to each other, packages of 3 variables accessed by different threads. When the first thread reads its vector 3, it copies 8 values in its cache. The 3 he is interested in, plus the vector 3 of its neighbor and 2 more values. This will result in false sharing issues, significantly slowing down the computation and the parallel scaling.

The first solution tried is known as cache alignment or padding. In order to avoid threads copying neighbors values in its cache, ghost values are added between the interesting values. Instead of having a #threads-by-3 View, we allocate a #threads-by-8 View where the first 3 values are used as before and the next 5 are never touched. They serve as padding to make sure the core only copies the relevant elements into its cache, and not its neighbor's vector as shown

in Fig. 6.5.

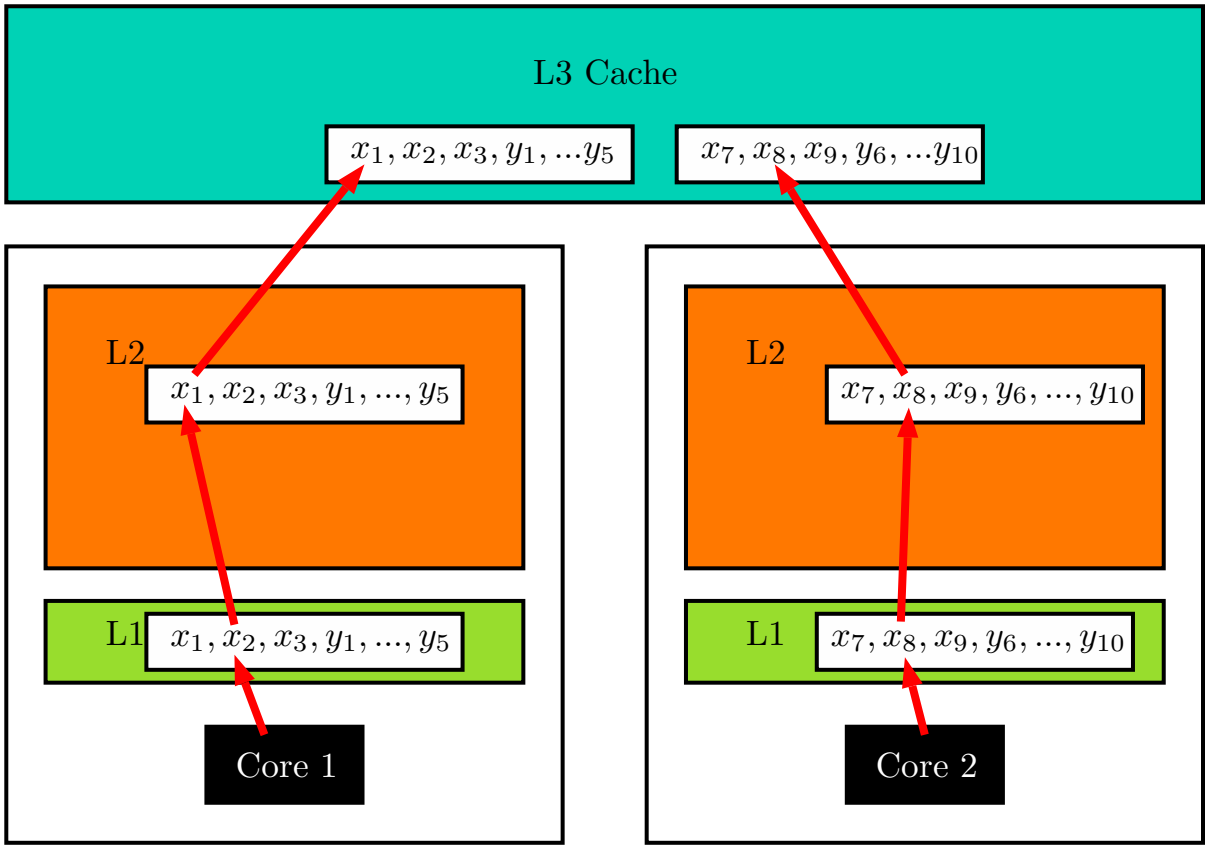

Figure 6.5: This illustrates how false sharing can be solved by adding a padding, y_i variables, to the variables of interest, x_i. The padding will place the variables accessed by the second core physically further away from the variables accessed by the first core, thus making the line in each L1 cache not share any variable.

The performance of this code is named RAJA Cache in Fig. 6.4. Although it is better with the padding, we still do not have scaling. This code has 6GB of Local LLC hit with false sharing when using 10 cores. It is an order of magnitude bigger than for our OpenMP code.

A better solution is to mimic OpenMP and Kokkos: Allocate each thread's local variables as blocks. To do so, we regrouped all the memory needed per thread. Allocating the memory contiguously per thread will remove any false sharing except for the 8 first and last values. This implementation does show scaling over the 1 thread execution time, but it is still not perfect.

6.3 Performance results on state of the art hardware

In this section we provide a comparison of the different codes on the Ivy-Kepler, IBM-Pascal and the Skylake-Volta computer, the latter with state of the art hardware, see Fig. 6.6.

First we will look at RAJA. We can see that the single core performance are the same on both computers but the newer hardware actually performs worth with multiple cores. We can only

interpret these results by saying that although we solved a part of the memory management issue (see section 6.2), there is still some problems with the RAJA based implementation. Next, comparing OpenMP on the two computers, we observe on average a 14% improvement on Skylake-Volta. OpenMP presents some overhead going from 1 to 2 cores on both computers but then scale almost perfectly all the way up to 10 cores and 16 cores for Ivy-Kepler and Skylake-Volta, respectively. The results for Kokkos are similar, there is a good scaling and an average of 10% improvement going from Ivy-Kepler to Skylake-Volta. The difference between OpenMP and Kokkos also lowers as Kokkos is on average 16% slower than OpenMP on Ivy-Kepler and only 10% slower on Skylake-Volta.

On the GPU side, it is important to note that the two GPUs are using different connectors. The P100 is connected using the NVLink communication protocol by NVIDIA, with a transfer rate of 20 GT/s, while the V100 is using the standard PCIe connector, with a transfer rate of 8 GT/s. The copying time is 2 time smaller with the NVLink. RAJA and CUDA produce very little difference on the computation time between the P100 and the V100. The two codes are not using shared memory which could partly explain the lack of improvement from one GPU to the other. The transfer of the data to the host is 2 time faster on the P100, we therefore obtain faster results on the P100, even when adding the computation time . Kokkos' GPU result is the only that improves from the P100 to the V100, about 40%.

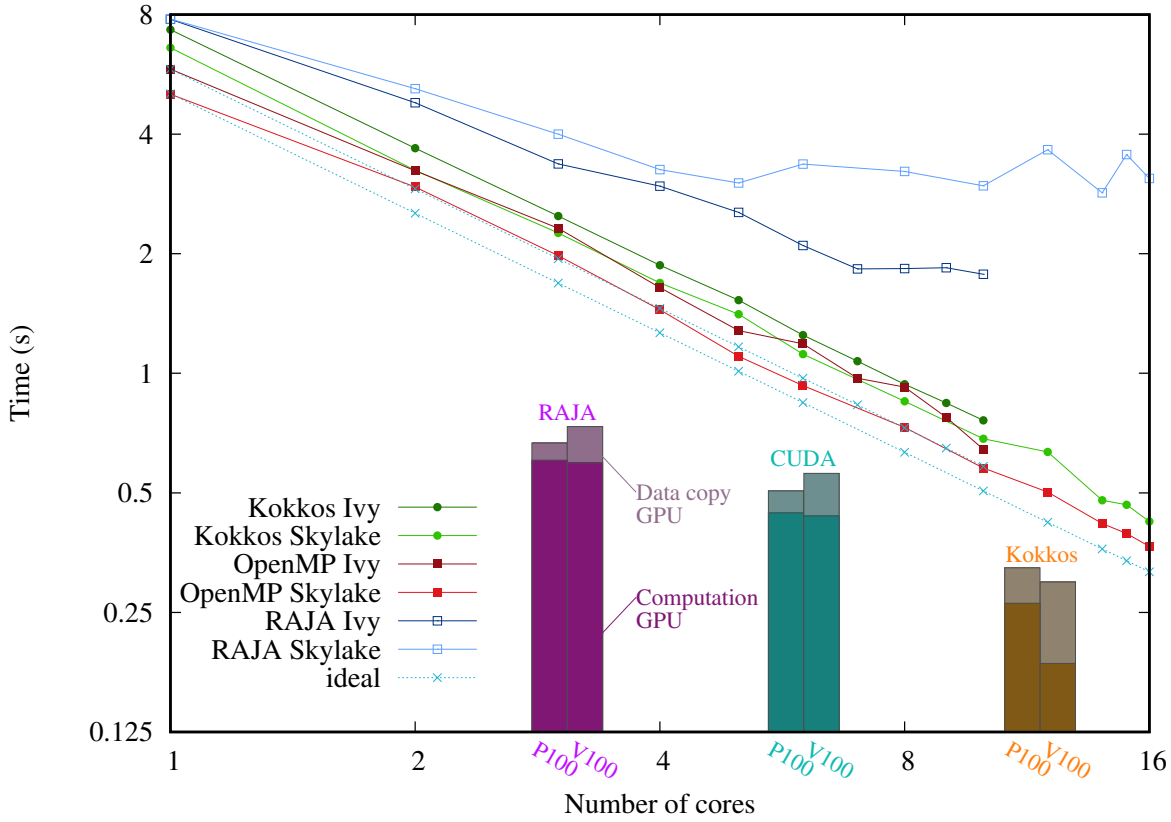

Figure 6.6: Log-log plot of the compute time as a function of number of cores. We compare the performance of Kokkos and RAJA, on CPU and GPU to OpenMP and Cuda. The codes, with the optimized modulo computation, were run on an Intel Xeon Gold 6130 @ 2.10GHz (Skylake) and an Intel Xeon E5-2680 v2 @ 2.80GHz (Ivy), a NVIDIA Tesla P100 and a NVIDIA Tesla V100. Results for the GPU are independent of the number of CPU cores and are placed arbitrarily on the abscissa. The copying time on the same GPU for the different codes (e.g. RAJA V100 and Kokkos V100) is similar but appears different due to the logarithmic scale.

7 Summary and conclusions

This thesis presents a performance and usability assessment of two performance portability libraries, Kokkos and RAJA. This work is motivated by the need of code abstraction between different architectures. The heterogeneity of architectures in supercomputers makes it hard to have a single code which is optimized for all platforms. Multiple solutions are being developed today whose goal is to provide the user the ability to write single source code with architecture-specific optimization at compile time. Here, we have investigated two prominent libraries, Kokkos [11] and RAJA [16]. The two libraries are being considered for the development of an optimized implementation of the GEMPIC model [17], and our work aims at providing valuable information to decide which framework should be adopted. The current GEMPIC code relies solely on MPI to distribute the workload on a supercomputer. We focused on the per-node computation, comparing Kokkos and RAJA to standard OpenMP and CUDA implementations.

As described in Chapter 3, RAJA and Kokkos offer a very similar concept and level of abstraction although they differ quite a bit when looking at specific implementations.

Kokkos offers data types usable in both CPU and GPU environments depending on C++ template values, i.e. compile time decisions. Kokkos tries to add default values for the largest part of the parameters. If compiled with the default values, the code will be compiled for the "fastest" architecture available. "Fastest" is defined by the following ordering, Cuda > OpenMP > PThreads > serial. With the default parameters, the execution space, CPU or GPU, of a parallel section will match the memory space, host memory or device memory, respectively. This makes implementations much simpler for the user. Kokkos manages to hide the CPU/GPU macro cases very well. The Kokkos project seems better documented and, from our experience when interacting with the developers, more active than RAJA.

RAJA, similarly to Kokkos, also provides data types usable in both CPU and GPU environments depending on C++ template values. But RAJA does not implement default values for the template parameters depending on the architecture. This forces the user to specify the template values inside #ifdef preprocessor branches, and therefore to write all CPU parameters and GPU parameters explicitly, thus limiting the idea of a single source code for multiple architectures. For this reason, RAJA forces us to have two sections in our code for CPU and GPU compilation.

This comparison of the data management is a good representation of the overall experience we have had using the two libraries. The approach of Kokkos to decide which settings to use and to allow the user to not worry about it makes programming easier. Regarding the features necessary to our application, vector reduction, scratch memory, etc..., Kokkos is also better suited, as RAJA did not have some of these features.

Our focus is on the usability of Kokkos and RAJA as performance portability frameworks for the future development of an optimized version of the GEMPIC [17] FORTRAN code.

Five versions of the most expensive function Hp1 from the FORTRAN routine were developed to compare Kokkos to RAJA and also see how they perform compared to architecture-specific frameworks. We re-implemented the FORTRAN routine in C++, with a test bench to reproduce the same computation. In addition, two versions were coded, one using Kokkos and another using RAJA. Finally, an OpenMP and a CUDA version of the code were written to complete our benchmarks with current architecture-specific frameworks.

The original code, as said in section 4.3, is a proof of concept and has not been optimized yet. Although we applied a straight forward algorithmic optimization by precomputing and storing values, we did not focus on the algorithm it self but rather the performance obtainable on the same code with different parallel frameworks.

The performance comparison of each code, presented and analyzed in chapter 6, was done on different hardware generations. We compared the computation time reduction from the algorithmic optimization and observed a 2x speed-up, showing how important the algorithmic optimization is regardless of the framework used as the same speed-up was observed on all the 5 codes.

Comparing different CPUs shows that OpenMP is faster than Kokkos and RAJA on all generations. With Kokkos we observe about 10% overhead compared to OpenMP on the older Ivy Bridge and 15% on the newer Skylake system. OpenMP and in particular Kokkos show a virtually ideal parallel scaling up to 16 cores of a single CPU.

In section 6.2 we explain in detail the issue faced with RAJA's CPU parallel scaling. We encountered false sharing issues with the RAJA code that heavily impacted its performance. False sharing is a difficult issue to deal with as it is not explicitly visible from the code, as for example, a race condition is. The false sharing can only be identified once we know what we are looking for. After successfully identifying the cause of RAJA's bad scaling we worked on two solutions. With the second solution described in section 6.2 we managed to improve the results. From having zero improvement with multiple cores comparing to the serial code, we managed to obtain some scaling. This is a first step as the scaling is not yet ideal, and we can expect improvements if we continue to optimize the code.

The GPU comparison was done using a NVIDIA Tesla K40m, a NVIDIA Tesla P100 connected with NVLink and a NVIDIA Tesla V100 connected with PCIe. With a very limited amount of time spent on the CUDA implementation, we created a baseline for a GPU code, yet our CUDA implementation is faster than Kokkos only on the K40m. We suspect the data management to be the reason of the slower CUDA code compared to Kokkos. The ordering obtained in term of GPU performances overall is RAJA as the slowest, then CUDA and finally Kokkos as the fastest on GPU.

Concerning the comparison of the CPU and GPU results, the Ivy Bridge CPU and Kepler GPU computer, and the Skylake CPU and Volta GPU computer, give a good comparison with hardware of the same generation. The best GPU result on the K40m is between the performance of 4 to 5 cores of the Ivy Bridge CPU. We see a bigger improvement on the GPUs when going to the newer generations than on the CPUs. On the Volta GPU, Kokkos outperforms both Kokkos' CPU implementation and the OpenMP implementation, making Kokkos on a V100 the fastest code we have produced.

With both the usability and performance assessment we have conducted, we now have a global picture of how the 4 frameworks compare. Because RAJA's performances are not ideal, we can say that, looking at the usability, Kokkos would be our choice today. After spending a similar, limited, amount of time on each of the 5 codes the fact is that RAJA is the only code not scaling properly. Furthermore, with the restricted amount of time, we were capable of delivering better performances with Kokkos than CUDA. Although this doesn't compare Kokkos to the perfect CUDA implementation, Kokkos on GPU requires very little changes, if any, compared to Kokkos on CPU. For us, on the usability, Kokkos wins over CUDA.

This bring us to the conclusion that from our test with the GEMPIC model, we recommend Kokkos over RAJA. Now, regarding the overhead of Kokkos CPU over OpenMP, it would depend on the application and whether the few percent lost is worth the free GPU implementation. A future work would be to answer the last paragraph of section 6.1 regarding the source of Kokkos' overhead. A usage of Kokkos would be to develop codes easily for both CPU and GPU and only implement critical sections with architecture-specific frameworks, e.g. having the critical loop implemented with OpenMP, still using Kokkos' View.

Bibliography

[1] Gene M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM. 1967, pp. 483–485.

[2] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Katherine A. Yelick. *The landscape of parallel computing research: A view from berkeley*. Tech. rep. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[3] Yann Barsamian, Sever A. Hirstoaga, and Éric Violard. "Efficient data structures for a hybrid parallel and vectorized particle-in-cell code". In: *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE. 2017, pp. 1168–1177.

[4] Kevin J. Bowers. "Accelerating a particle-in-cell simulation using a hybrid counting sort". In: *Journal of Computational Physics* 173.2 (2001), pp. 393–411.

[5] Annalisa Buffa, Giancarlo Sangalli, and Rafael Vázquez. "Isogeometric analysis in electromagnetics: B-splines approximation". In: *Computer Methods in Applied Mechanics and Engineering* 199.17-20 (2010), pp. 1143–1152.

[6] Annalisa Buffa, Judith Rivas, Giancarlo Sangalli, and Rafael Vázquez. "Isogeometric discrete differential forms in three dimensions". In: *SIAM Journal on Numerical Analysis* 49.2 (2011), pp. 818–844.

[7] NVIDIA Corporation. *NVIDIA Tesla P100*. 2016. URL: `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`.

[8] NVIDIA Corporation. *NVIDIA Tesla V100*. 2017. URL: `http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`.

[9] NVIDIA Corporation. *TESLA K40 GPU Active Accelerator*. 2013. URL: `https://www.nvidia.com/content/PDF/kepler/Tesla-K40-Active-Board-Spec-BD-06949-001_v03.pdf`.

[10] Nicolas Crouseilles, Lukas Einkemmer, and Erwan Faou. "Hamiltonian splitting for the Vlasov–Maxwell equations". In: *Journal of Computational Physics* 283 (2015), pp. 224–240.

[11] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. "Kokkos: Enabling many-core performance portability through polymorphic memory access patterns". In: *Journal of Parallel and Distributed Computing* 74.12 (2014), pp. 3202–3216.

[12] Juan Gómez-Luna. "Atomic Operations across GPU generations." University Lecture. 2015. URL: `http://ece408.hwu-server2.crhc.illinois.edu/Shared%20Documents/Slides/Presentation_ECE408_JGL.pdf`.

[13] Ernst Hairer, Christian Lubich, and Gerhard Wanner. *Geometric numerical integration: structure-preserving algorithms for ordinary differential equations*. Vol. 31. Springer Science & Business Media, 2006.

[14] Yang He, Hong Qin, Yajuan Sun, Jianyuan Xiao, Ruili Zhang, and Jian Liu. "Hamiltonian time integrators for Vlasov-Maxwell equations". In: *Physics of Plasmas* 22.12 (2015), p. 124503.

[15] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. "An overview of the Trilinos project". In: *ACM Transactions on Mathematical Software (TOMS)* 31.3 (2005), pp. 397–423.

[16] Richard Hornung and Jeffrey Keasler. *The RAJA portability layer: overview and status*. Tech. rep. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2014.

[17] Michael Kraus, Katharina Kormann, Philip J. Morrison, and Eric Sonnendrücker. "GEMPIC: Geometric electromagnetic particle-in-cell methods". In: *Journal of Plasma Physics* 83.4 (2017).

[18] Matthew Martineau, Simon McIntoshSmith, and Wayne Gaudin. "Assessing the performance portability of modern parallel programming models using TeaLeaf". In: *Concurrency and Computation: Practice and Experience* 29.15 (2017), e4117.

[19] Matthew Martineau, Simon McIntoshSmith, Mike Boulton, and Wayne Gaudin. "An evaluation of emerging many-core parallel programming models". In: *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM. 2016, pp. 1–10.

[20] Philip J Morrison. "The Maxwell-Vlasov equations as a continuous Hamiltonian system". In: *Physics Letters A* 80.5-6 (1980), pp. 383–386.

[21] Hong Qin, Yang He, Ruili Zhang, Jian Liu, Jianyuan Xiao, and Yulei Wang. "Comment on Hamiltonian splitting for the Vlasov–Maxwell equations". In: *Journal of Computational Physics* 297 (2015), pp. 721–723.

[22] *Selalib*. July 2018. URL: `http://selalib.gforge.inria.fr/`.

[23] Gilbert Strang. "On the construction and comparison of difference schemes". In: *SIAM Journal on Numerical Analysis* 5.3 (1968), pp. 506–517.

[24] Jan Treibig, Georg Hager, and Gerhard Wellein. "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments". In: *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*. IEEE. 2010, pp. 207–216.

[25] Alan Weinstein and Philip J Morrison. "Comments on: The Maxwell-Vlasov equations as a continuous Hamiltonian system". In: *Physics Letters A* 86.4 (1981), pp. 235–236.