Master in High Performance Computing

Solving non-linear chemical equations for gas cooling with GPU offloading using OpenACC

Supervisor(s):

Ivan Girotto,
Umberto Maio,
Luca Tornatore

Candidate:

Avinash Anand

7th edition
2020–2021

Master in
High Performance Computing

Contents

Abstract

Simulation of cooling effects in a gaseous system involves solving a set of non-linear PDEs for chemical reactions iteratively. The complexity and the computational requirement of the simulation grow heavily as the number of particles and the number of cooling mechanisms is increased. Fortunately, it is now possible to offload such simulations on a variety of scalable computing platforms. In this work, we refactored an implementation of gas cooling in GADGET-2. We improved the implementation further by enabling the offloading of computation on multiple GPUs using MPI and OpenACC. Finally we performed a detailed comparative study of the code performance on x64_86 (Intel Xeon CPU), ppc64le (IBM Power9 CPU), Pascal (NVIDIA P100 GPU) and Volta (NVIDIA V100 GPU) architecture based computing platforms.

Acknowledgment

I would like to express my sincere gratitude to my supervisors Prof. Umberto Maio for his invaluable advice and feedback over multiple discussions; Prof. Luca Tornatore for his suggestion and support to carry out this project; and Ivan Girotto for his unwavering guidance and keeping me in the loop of positive feedback and encouragement.

I would also like to extend my warm regards to the course instructors and my colleagues. It was only because of their patient and extended help, that I was able to finish a large part of the coursework online from the other part of the world.

And last but not least, I am thankful to ICTP visa office and the office of external activities for working relentlessly over months to make my physical presence in Trieste possible.

Chapter 1

Introduction

Microphysical phenomena that involve chemical reactions and hydrodynamics play important roles in structure formation at small length and time scales. Heating and cooling of gas and variation in free-electron densities due to chemical reactions are some of these. Such phenomena are more effective when their time scales are much smaller than the dynamical time scales. For example, when cooling is dominant over other microphysical processes and cooling time is longer than the recombination time, the abundant free-electrons in primordial environment combines with neutral hydrogen atoms, forming the H^- ion. The H^- ion further combines with hydrogen atoms to form hydrogen molecules. The photon emitted in this process escapes if the gas cloud is optically thin. So the overall effect is that the energy level excitation and de-excitation caused by atomic collisions causes the conversion of thermal energy of the gas into radiative energy which escapes the medium, hence cooling the system. Upon cooling over a long time, the clouds abundant in hydrogen molecules collapse under gravitation leading to the formation of stars and other structures.

In this project, we are working on improving the implementation of the effect of gas cooling and non-equilibrium chemistry in the Gadget-2[1] simulation, as originally done by Maio et al.[2] This implementation uses different chemical reactions and their kinetic rate equations to compute the temperature of the gas in different regimes. In order to accomplish our goal, we extracted the code section that treats these physical processes from the entire code (which counts approximately 200k lines) in order to be much more agile in testing and development.

In a typical cosmological simulation, like in Gadget-2, one cannot implement gas chemistry model that follows individual atoms and molecules because of the limited computational resources. So the different chemical species are modeled as constituents of the resolved baryon particles. While the two baryon particles interact with each other by gravitational force, the chemical species corresponding to them are considered as non-interacting.

Our improvement to the implementation is focused on code refactoring and GPU offloading using OpenACC[i]. In the following sections, we will discuss briefly the chemical reactions with the corresponding cooling functions and rate equations. The next chapter deals with coding implementation and is followed by benchmarking.

[i]OpenACC website: https://www.openacc.org/

Index	Chemical reaction	Chemical process
1	$H + e \longrightarrow H^+ + 2\,e$	Hydrogen collisional ionization
2	$H^+ + e \longrightarrow H + \gamma$	Hydrogen recombination
3	$He + e \longrightarrow He^+ + 2\,e$	Helium collisional ionization
4	$He^+ + e \longrightarrow He + \gamma$	Helium recombination
5	$He^+ + e \longrightarrow He^{++} + 2\,e$	He^+ collisional ionization
6	$He^{++} + e \longrightarrow He^+ + \gamma$	He^+ recombination
7	$H + e \longrightarrow H^- + \gamma$	Photo-attachment
8	$H^- + H \longrightarrow H_2 + e$	H_2 formation path
9	$H + H^+ \longrightarrow {H_2}^+ + \gamma$	H^+ channel
10	${H_2}^+ + H \longrightarrow H_2 + H^+$	H^+ channel
11	$H_2 + H \longrightarrow 3\,H$	
12	$H_2 + H^+ \longrightarrow {H_2}^+ + H$	
14	$H_2 + e \longrightarrow 2\,H + e$	Collisional dissociation
16	$H^- + e \longrightarrow H + 2\,e$	
17	$H^- + H \longrightarrow 2\,H + e$	
18	$H^- + H^+ \longrightarrow 2\,H$	
19	$H^- + H^+ \longrightarrow {H_2}^+ + e$	
20	${H_2}^+ + e \longrightarrow 2\,H$	
21	${H_2}^+ + H^- \longrightarrow H + H_2$	
1a	$D + H_2 \longrightarrow HD + H$	
2a	$D^+ + H_2 \longrightarrow HD + H^+$	
3a	$HD + H \longrightarrow D + H_2$	
4a	$HD + H^+ \longrightarrow D^+ + H_2$	
5a	$H^+ + D \longrightarrow H + D^+$	
6a	$H + D^+ \longrightarrow H^+ + D$	
7a	$D^+ + e \longrightarrow D + \gamma$	Deuterium recombination
1b	$He + H^+ \longrightarrow HeH^+ + \gamma$	
2b	$HeH^+ + H \longrightarrow He + {H_2}^+$	
3b	$HeH^+ + \gamma \longrightarrow He + H^+$	

Table 1.1: List of chemical reactions implemented in the code

1.1 Chemical reactions and kinetic rate equations

In the early universe, where the heavy atoms were not present, the primordial gas clouds were mainly composed of H, D, He, and molecules based on them, like H_2, and HD. These species, interact in gaseous clouds via a complex series of chemical reactions, releasing energy in form of photons and cooling the gaseous medium eventually.

To evaluate the change in energy by cooling, we need to quantify the effects of the chemical reactions and the chemical abundance of the coolant species involved. For this purpose, a set of 28 self-consistent chemical reactions, accounting for the major collisional cooling processes was identified[2] and is listed in table 1.1. In the code, we tracked the abundances of all subspecies that were involved in these chemical reactions: e^-, H, H^+, He, He^+, He^{++}, H_2, ${H_2}^+$, H^-, D, D^+, HD and HeH^+.

Let us consider the rate of change of population of an species i:

$$\mathrm{d}n_i/\mathrm{dt} = C_i - D_i n_i \tag{1.1}$$

where n_i is the number density, t is the time period, C_i is the collective rate of formation and D_i is the coefficient for the collective rate of decomposition of i^{th} species. We can further write $C_i = \sum\sum k_{pq,i} n_p n_q$ where the summation is over all the pairs of p and q species that combine to produce the i^{th} species at the temperature-dependent rate $k_{pq,i}$. Similarly, $D_i = \sum k_{ij} n_j$ where summation is over all the j species that interact with i^{th} species leading to its decomposition at the temperature-dependent rate k_{ij}.

We can discretize the eq.(1.1) using backward difference formula:

$$\frac{n_i^{t+\Delta t} - n_i^t}{\Delta t} = C_i^{t+\Delta t} - D_i^{t+\Delta t} n_i^{t+\Delta t} \tag{1.2}$$

It allows us to update the number density of a species in advanced time step $(t+\Delta t)$ using the values from the previous step:

$$n_i^{t+\Delta t} = \frac{C_i^{t+\Delta t}\Delta t + n_i^t}{1 + D_i^{t+\Delta t}\Delta t} \tag{1.3}$$

Here the coefficients C and D are obtained from the rates of chemical reactions at time $t + \Delta t$.

We notice that since the concentration of H^- and ${H_2}^-$ are very low, and since they do not much affect the concentration of other abundant species, they can be considered in equilibrium throughout the simulation. So for these two species, we can use the expression for number density derived from the equilibrium formulation instead of eq. 1.3. Considering chemical equations 7, 8, 16, 17, 18 and 19 from table 1.1, we can write the equilibrium concentration of H^- as

$$n_{\mathrm{H}^-} = \frac{k_7 n_{\mathrm{H}} n_e}{(k_8 + k_{17}) n_{\mathrm{H}} + (k_{18} + k_{19}) n_{\mathrm{H}^+} + k_{16} n_e} \tag{1.4}$$

where k_i is the rate of i_{th} chemical reaction from table 1.1. Similarly, considering chemical equations 9, 12, 19, 2b, 10, 20 and 21 from table 1.1, we can write the equilibrium concentration of ${H_2}^+$ as

$$n_{H_2^+} = \frac{k_9 n_{\mathrm{H}} n_{\mathrm{H}^+} + k_{12} n_{\mathrm{H}_2} n_{\mathrm{H}^+} + k_{19} n_{\mathrm{H}^-} n_{\mathrm{H}^+} + k_{2b} n_{\mathrm{HeH}^+} n_{\mathrm{H}}}{k_{10} n_{\mathrm{H}} + k_{20} n_e + k_{21} n_{\mathrm{H}^-}} \tag{1.5}$$

Since the equation (1.3) for all the species is inter-dependent we can update the concentration of all the species either all at the same step or sequentially. In this project, we updated the species abundances sequentially in the following order: H^-, ${H_2}^+$, H_2, H, H^+, He, He^+, He^{++}, HD, D, D^+, HeH^+, e^-.

While free electron is driving most of the equation, we must ensure that the time step in equation (1.3) is smaller than the depletion time of free-electrons, t_{elec}.

$$t_{elec} = n_{\mathrm{e}}/\dot{n}_{\mathrm{e}} \tag{1.6}$$

This was done by setting the maximum allowed time step to $\delta t = \epsilon t_{elec}$ with $\epsilon < 1$. This prevents the abundance of free electron to become zero in a few iterations.

1.2 Heating and cooling of gases

Now that we are equipped with the equations to solve the rate equations of the non-equilibrium chemical reactions, we can proceed to evaluate the temperature and energy of the system and the cooling rate. The internal energy per unit volume of the gaseous system is evaluated as

$$E = \frac{P}{\gamma - 1} = \frac{k_B T}{\gamma - 1} \sum_i n_i \tag{1.7}$$

where P is the pressure, γ is adiabatic coefficient of the gas, k_B is Boltzmann constant, T is temperature, and the summation is over the number density n_i of all the species present in the gas cloud. Since the primordial gas clouds are dominated with monoatomic gases, we can safely assume $\gamma = 5/3$. On the other hand, the rate of change in energy of the system per unit time and per unit volume is given by

$$\dot{E} = \dot{E}_{Compton} + \sum_{(i,j)} \Lambda_{ij} n_i n_j \tag{1.8}$$

Here the $\dot{E}_{Compton}$ is the heating rate due to Compton effect. The summation in the second term is over all the interacting (i, j) pairs and Λ_{ij} is the respective cooling function in erg cm^3 s^{-1} which is a temperature dependent quantity. The cooling mechanisms and the corresponding cooling rates that we adopted, are listed below:

(a) Collisional excitation

Cooling by collisional excitation happens when a bounded electron in an atom jumps to a higher energy state following a collision with another species, and then decays to the lower energy state by emitting photon. We adopted the following cooling rates for this process from Anninos et al.[3]:

$$\dot{E}_{ce,1} = 7.50 \times 10^{-19} \big(1 + \sqrt{T_5}\big)^{-1} \times \exp(-118348/T) \times n_{\rm e} n_{\rm H} \text{ erg cm}^{-3}\text{ s}^{-1} \tag{1.9}$$
$$\dot{E}_{ce,2} = 9.10 \times 10^{-27} \big(1 + \sqrt{T_5}\big)^{-1} T^{-0.1687} \times \exp(-13179/T) \times n_{\rm e}^2 n_{\rm He} \text{ erg cm}^{-3}\text{ s}^{-1} \tag{1.10}$$
$$\dot{E}_{ce,3} = 5.54 \times 10^{-17} \big(1 + \sqrt{T_5}\big)^{-1} T^{-0.397} \times \exp(-473638/T) \times n_e n_{\rm He^+} \text{ erg cm}^{-3}\text{ s}^{-1} \tag{1.11}$$

where the temperature T is in Kelvin, number densities are in cm^{-3}, and $T_5 = T/10^5$K.

(b) Collisional ionization

A moving species loses its energy when it knocks out a bounded electron from an atom followed by a collision. The following cooling rates for the corresponding processes were taken from Anninos et al.[3]:

$$\dot{E}_{ci,1} = 2.18 \times 10^{-11}\, k_1\, n_{\rm e}\, n_{\rm H} \text{ erg cm}^{-3}\text{ s}^{-1} \tag{1.12}$$
$$\dot{E}_{ci,2} = 3.94 \times 10^{-11}\, k_3\, n_{\rm e}\, n_{\rm He} \text{ erg cm}^{-3}\text{ s}^{-1} \tag{1.13}$$
$$\dot{E}_{ci,3} = 8.72 \times 10^{-11}\, k_5\, n_{\rm e}\, n_{\rm He^+} \text{ erg cm}^{-3}\text{ s}^{-1} \tag{1.14}$$
$$\dot{E}_{ci,4} = 5.01 \times 10^{-27} \big(1 + \sqrt{T_5}\big)^{-1} T^{-0.1687} \times \exp(-55338/T) \times n_{\rm e}^2 n_{\rm He^+} \text{ erg cm}^{-3}\text{ s}^{-1} \tag{1.15}$$

where k_i is the rate of i^{th} chemical equation from table 1.1.

(c) Recombination

Recombination happens when two ions combine to form one species while losing some of their kinetic energy in the process. We used the following cooling rates for this process from Anninos et al.[3]:

$$\dot{E}_{re,1} = 8.70 \times 10^{-27} \times T^{0.5} T_3^{-0.2} \left(1 + T_6^{0.7}\right)^{-1} n_{\rm e}\, n_{\rm H^+} \ {\rm erg\ cm^{-3}\ s^{-1}} \tag{1.16}$$

$$\dot{E}_{re,2} = 1.55 \times 10^{-26} \times T^{0.3647} n_{\rm e}\, n_{\rm He^+} \ {\rm erg\ cm^{-3}\ s^{-1}} \tag{1.17}$$

$$\begin{aligned}\dot{E}_{re,3} =& 1.24 \times 10^{-13} \times T^{-1.5} \left[1 + 0.3 \times \exp(-94000/T)\right] \\ & \times \exp(-470000/T) \times n_{\rm e}\, n_{\rm He^+} \ {\rm erg\ cm^{-3}\ s^{-1}}\end{aligned} \tag{1.18}$$

$$\dot{E}_{re,4} = 3.48 \times 10^{-26} \times T^{0.5} T_3^{-0.2} \left(1 + T_6^{0.7}\right)^{-1} n_{\rm e}\, n_{\rm He^{++}} \ {\rm erg\ cm^{-3}\ s^{-1}} \tag{1.19}$$

where $T_n = T/10^n$K.

(d) Bremsstrahlung

A free electron loses its some of its kinetic energy as photons when it is accelerated under the influence of the electric field of a nearby ion. This process is called Bremsstrahlung. Its cooling rate is given by the following equation taken from Anninos et al.[3]:

$$\begin{aligned}\dot{E}_{brem} =& 1.43 \times 10^{-27} \times T^{0.5} \times \left[1.1 + 0.34 \times \exp\left(-\frac{(5.5 - \log_{10} T)^2}{3}\right)\right] \\ & \times \left(n_{\rm H^+} + n_{\rm He^+} + n_{\rm He^{++}}\right) n_{\rm e} \ {\rm erg\ cm^{-3}\ s^{-1}}\end{aligned} \tag{1.20}$$

(e) Compton cooling/heating

Compton effect is the scattering of photons by free electrons. The free electrons in a gas cloud may gain or lose their kinetic energy depending on the temperature of background radiation compared to the local gas temperature. The rate of cooling/heating caused by this process is adopted from Anninos et al.[3] and it is given below:

$$\dot{E}_{Compton} = 5.65 \times 10^{-36} \times (1+z)^4 \times \left[T - 2.73(1+z)\right] \times n_{\rm e} \ {\rm erg\ cm^{-3}\ s^{-1}} \tag{1.21}$$

(f) $\mathbf{H_2}$ and $\mathbf{H_2}^+$ cooling

The collision between H and H_2 causes the de-excitation of rotational and vibrational energy states of the H_2 molecules. The rate of cooling/heating caused by this process is given by:

$$\dot{E}_{H_2,{\rm H}} = \left[\Lambda_{\rm H_2}(n_{{\rm H},T_{gas}}) - \Lambda_{\rm H_2}(n_{\rm H}, T_{CMB})\right] \times n_{\rm H}\, n_{\rm H_2} \ {\rm erg\ cm^{-3}\ s^{-1}} \tag{1.22}$$

where T_{gas} is the temperature of gas cloud, T_{CMB} is the temperature of cosmic microwave background and the term $\Lambda_{\rm H_2}(n_{\rm H}, T)$ is given as

$$\Lambda_{\rm H_2}(n_{\rm H}, T) = \frac{\Lambda_{\rm H_2,LTE}(n_{\rm H}, T)}{\left[1 + \dfrac{\Lambda_{\rm H_2,LTE}(n_{\rm H}, T)}{\Lambda_{\rm H_2}(n_{\rm H} \to 0, T)}\right]} \tag{1.23}$$

In this equation, $\Lambda_{\rm H_2,LTE}(n_{\rm H}, T)$ is the LTE cooling function for both rotational and vibrational energy levels evaluated with the equations 6.37 and 6.38 of Hollenbach and McKee[4], and

$\Lambda_{\mathrm{H_2}}(n_{\mathrm{H}} \to 0, T)$ is the low density limit of the cooling function evaluated with equation A7 from Galli and Palla[5].

Charge exchange of $\mathrm{H_2}^+$ with H and its dissociation with collision by e^- causes cooling via photon emission. The cooling rates for these reactions taken from Galli and Palla[5] are given below:

$$\dot{E}_{\mathrm{H_2}^+,\mathrm{H}} = \exp(-650/T) \times 10^{-28}\, T^2\, n_{\mathrm{H_2}^+} n_{\mathrm{H}}\, \mathrm{erg\, cm^{-3}\, s^{-1}} \tag{1.24}$$

$$\dot{E}_{\mathrm{H_2}^+,\mathrm{e}} = 3.5 \times 10^{-27} \times \exp(-800/T)\, T^2\, n_{\mathrm{H_2}^+} n_{\mathrm{e}}\, \mathrm{erg\, cm^{-3}\, s^{-1}} \tag{1.25}$$

(g) HD cooling

Collision between HD molecules and hydrogen atoms causes the de-excitation of rotational and vibrational energy levels of the HD molecules. The resulting cooling rate is given by:

$$\dot{E}_{\mathrm{HD}} = \Lambda_{\mathrm{HD,H}}(T) \times n_{\mathrm{HD}}\, n_{\mathrm{H}}\, \mathrm{erg\, cm^{-3}\, s^{-1}} \tag{1.26}$$

where the cooling function $\Lambda_{\mathrm{HD,H}}(T)$ taken from Lipovka, Núñez-López, and Avila-Reese[6] is given by:

$$\begin{aligned}\log_{10}[\Lambda_{\mathrm{HD,H}}(T)] = &- 42.45906 + 21.90083 \times \log_{10} T - 10.1954 \times (\log_{10} T)^2 \\ &+ 2.19788 \times (\log_{10} T)^3 - 0.17286 \times (\log_{10} T)^4\end{aligned} \tag{1.27}$$

After adding all the contributions, we obtain the overall cooling rate $\dot{E}$. We use this quantity to compute the cooling time of gas t_{cool}.

$$t_{cool} = \frac{E}{\dot{E}} \tag{1.28}$$

Finally we update the system temperature as

$$T_{i+1} = T_i + \frac{T_i \times \delta t}{t_{cool}} \tag{1.29}$$

where we chose $\delta t = \epsilon t_{cool}$ with $\epsilon < 1$.

1.3 Time-stepping in the simulation

For updating the number density of different species, we need $\delta t = \epsilon t_{elec}$ and while updating the temperature of the system, we need $\delta t = \epsilon t_{cool}$. Since our simulation is limited to time period specified by cosmic scale factors a_{start} and a_{end}, we decided to chose the common time step δt both for updating species abundances and to update system temperature as

$$\delta t_i = \min\left\{\epsilon t_{cool},\ \epsilon t_{elec},\ \frac{a_i - a_{end}}{a_i H(a_i)}\right\} \tag{1.30}$$

After updating different quantities using δt, we finally updated the scale factor as

$$a_{i+1} = a_i + a_i H(a_i) \delta t_i \tag{1.31}$$

We remind the readers not familiar with basic Cosmology that the $H(a)$ is the function governing the Cosmological background of the Universe, as obtained from General Relativity. As such, in this context it should be considered merely a function that returns the evolution of "Cosmic time".

Chapter 2

Coding implementation

2.1 Original code

Let's review the original code used by Maio et al.[2]. The pseudocode referring to the structure of the original code is given in algorithm 1 and 2.

This code is embedded within the GADGET-2[1], but for our study, we ran the code independent of GADGET-2 by initializing the particle data structures by hand. We chose $\epsilon = 0.1$, MAX_ITERATIONS = 2000000000, and ran the simulation from one scale factor, a_{start} to another, a_{end}. For this work we focused in evolving the particles from $a_{start} = 0.1428$ to $a_{end} = 0.1666$ or from $a_{start} = 0.0909$ to $a_{end} = 0.01$, which are typical ages of interest in real simulations.

In the rest of this Thesis, we will set the initial values of the physical quantities to be used for the calculations accordingly to the 11 different sets listed in table A, which have been chosen as representative of some average or extreme situations (very dense and warm/very hot medium, underdense and cold/warm/hot medium).

It is worth to stress here that we spotted two bugs[i,ii] in the current version of the code that in some rare situations led to either *(i)* a wrong floating-point operation, resulting in a `nan`, or *(ii)* in a extremely long run-time (of the order of ∼30 minutes) per particle. That, embedded in a typical GADGET-2 simulation, where a large number of particles is used, for say 1 billion, may cause a serious increase of the CPU-hours consumption in the cases in which some of those situations appear for some particles. We fixed these two issues without distubing the code semantics, recovering a reasonable run-time of the order of ∼10 seconds.

So in order to improve the code, we re-adapted it which we discuss in the next section.

[i] **[First bug]** If n_e goes to zero (this depends on initial parameter set and scale factor a), it evaluates $t_{elec} =$ `inf` and then $t_{elec} =$ `nan` in the next iteration. At this point, dt_i becomes `nan` as it is evaluated as minimum of three different time steps (see line 6 of alg. 2). This leads to the abrupt termination of the program.

[ii] **[Second bug]** When $t_{elec} \ll dt_i$ in the loop of the function `species_solver()`, that delays the convergence by cycling through all the MAX_ITERATIONS number of iterations which is set to 2×10^9.

Algorithm 1: Functions used in the original code

```
Function energy_solver(i_particle, $a_i$, $dt_i$, $T_{i-1}$, $*T_i$, $*t_{cool}$):
    $t_{count} = 0$
    for $i$ = 0 to MAX_ITERATIONS do
        Evaluate: Energy of the system                                   (see eq. 1.7)
        Interpolate: $\Lambda_{ij}$ of eq. 1.8 at temperature $T_{i-1}$ using expressions from section 1.2
        Evaluate: $\dot{E}$                                                   (see eq. 1.8)
        Evaluate: $t_{cool}$                                                 (see eq. 1.28)
        $dt_{in} = \min\{\epsilon t_{cool}, dt_i/2, dt_i - t_{count}\}$
        $T_i = T_{i-1} + dt_{in} T_{i-1}/t_{cool}$
        $t_{count} += dt_{in}$
        if $dt - t_{count} < dt/\mathrm{MAX\_ITERATIONS}$ then
            break;
        end
    end
return

Function species_solver(i_particle, $a_i$, $dt_i$, $T_i$, $*\dot{n}_e$):
    $t_{count} = 0$
    Interpolate: rate coefficient $k$ at temperature $T_i$
    for $i$ = 0 to MAX_ITERATIONS do
        Evaluate: $\dot{n}_e$                                                 (see eq. 1.1)
        Evaluate: $t_{elec}$                                                 (see eq. 1.6)
        $dt_{in} = \min\{\epsilon t_{elec}, dt_i/2, dt_i - t_{count}\}$
        Update: $n_{H^-}$ and $n_{H_2^+}$                                     (see eq. 1.4 and 1.5)
        Update: $n_{H_2}$, $n_H$, $n_{H^+}$, $n_{He}$, $n_{He^+}$, $n_{He^{++}}$, $n_{HD}$, $n_D$, $n_{D^+}$, $n_{HeH^+}$, $n_{e^-}$  (see eq. 1.3)
        $t_{count} += dt_{in}$
        if $dt_i - t_{count} < dt_i \times 10^{-3}$ then
            break;
        end
    end
return
```

2.2 Code refactoring

2.2.1 Structure

We improved the first logical issue of the code as referred in previous section by implementing a condition to assign dt_i correctly in every condition. The second issue was corrected by fixing $dt_{in} = dt_i/2$ in both energy_solver() and species_solver() functions. This choice was motivated from the observation that before getting t_{elec} = `inf`, the two functions in the original code were performing just two iterations each.

At this point, launching the modified code with parameter index 0 as listed in table A, from $a_{start} = 0.1428$ to $a_{end} = 0.1666$, we found that after a few iterations, dt_{in} was exceeding t_{cool} in

Algorithm 2: Structure of the original code

```
Initialization: Lookup tables
for $i\_particle = 1$ to $num\_particles$ do
    Initialization: $T_{gas}$
    for $i = 0$ to $MAX\_ITERATIONS$ do
        Evaluate: $t_{cool}$, $t_{elec}$                                   (see eq. 1.28 and 1.6)
        $dt_i = \min\left\{\epsilon t_{cool}, \epsilon t_{elec}, \frac{a_i - a_{end}}{a_i H(a_i)}\right\}$
        energy_solver($i\_particle$, $a_i$, $dt_i$, $T_{i-1}$, &$T_i$, &$t_{cool}$)     (see alg. 1)
        species_solver($i\_particle$, $a_i$, $dt_i$, $T_i$, &$\dot{n}_e$)            (see alg. 1)
        $a_i = a_{i-1} + a_{i-1}H(a_{i-1})dt_i$
        if $a_i >= a_{end}$ then
            break;
        end
    end
    Evaluate: System Entropy, Pressure
end
```

energy_solver() function. It means the cooling was getting faster in the intermediate loop reducing the energy of the system to zero abruptly. A restructuring of this inner loop to treat these cases properly is in order, which is in the agenda of the code's authors but out of the scope of this work. Since we were actually focused on implementing and optimizing the GPU offloading, which is the core of our work, and the CPU run-time is quite large for a massive amount of particles (i.e. of the order of 10^6 or larger), we damped the inner loop of integration in energy_solver() and species_solver() functions. The integration is less accurate, but the resulting code is much faster.

The final refactored code we have at the end is given in alg. 3 and 4. With the changes we made, the program execution takes only ∼4 seconds compared with ∼10 seconds for original code, to evolve the system from $a_{start} = 0.1428$ to $a_{end} = 0.1666$ on a 3 GHz CPU with the parameter index 0 as listed in table A.

Algorithm 3: Functions used in the revised code

```
Function compute_elec_dot($i\_particle$, $T_i$, $dt_i$, $*\dot{n}_e$):
    Interpolate: rate coefficient $k$ at temperature $T_i$
    Update: $n_{H^-}$ and $n_{H_2^+}$                                        (see eq. 1.4 and 1.5)
    Update: $n_{H_2}$, $n_H$, $n_{H^+}$, $n_{He}$, $n_{He^+}$, $n_{He^{++}}$, $n_{HD}$, $n_D$, $n_{D^+}$, $n_{HeH^+}$, $n_{e^-}$   (see eq. 1.3)
    Evaluate: $\dot{n}_e$                                                     (see eq. 1.1)
return

Function compute_t_cool($i\_particle$, $a_i$, $T_i$, $*t_{cool}$):
    Interpolate: $\Lambda_{ij}$ of eq. 1.8 at temperature $T_{i-1}$ using expressions from section 1.2
    Evaluate: $\dot{E}$                                                       (see eq. 1.8)
    Evaluate: Energy of the system                                            (see eq. 1.7)
    Evaluate: $t_{cool}$                                                      (see eq. 1.28)
return
```

Algorithm 4: Structure of the revised code

```
Initialization: Lookup tables
for $i\_particle = 1$ to $num\_particles$ do
    Initialization: $T_{gas}$
    Evaluate: $t_{cool}$, $t_{elec}$                                (see eq. 1.28 and 1.6)
    for $i = 0$ to $ITER\_LIMIT$ do
        $dt_i = \min\left\{\epsilon t_{cool}, \epsilon t_{elec}, \frac{a_i - a_{end}}{a_i H(a_i)}\right\}$
        $T_i = T_{i-1} + dt_i T_{i-1}/t_{cool}$
        compute_elec_dot($i\_particle$, $T_i$, $dt_i$, $\&\dot{n}_e$)          (see alg. 3)
        compute_t_cool($i\_particle$, $a_i$, $T_i$, $\&t_{cool}$)            (see alg. 3)
        $a_i = a_{i-1} + a_{i-1} H(a_{i-1}) dt_i$
        if $a_i >= a_{end}$ then
            break;
        end
    end
    Evaluate: System Entropy, Pressure
end
```

2.2.2 Result validation

We already noticed that in the refactored version which we are using to study the relative improvement due to the GPU offloading, the integration is expected to be less accurate. However, as we discuss in this section, this accuracy loss lies within the ∼30-40% and hence the code maintains the same order-of-magnitude than the original code, which supports our choice. The results shown here for comparison were obtained by modifying the original code by fixing just the two issues mentioned in section 2.1 without disturbing the semantics. From now on, we will refer to this version as original-corrected code.

The Fig. 2.1 shows the comparison between the time evolution of the temperature from the two codes using the parameter index 0 from table A. The plots from the two codes overlap and the result from the revised code lies between ∼17% in the beginning to ∼0.1% in the end, compared to the original-corrected code. As the internal specific energy is directly proportional to the system temperature, its time evolution follows the similar trend.

Fig. 2.2 compares the final total energy of the system for 11 different initial parameter sets listed in table A at corresponding a_{end}. The total energy in the new code lies within 0.9-32% of that of original-corrected code.

Fig. 2.3 compares another key quantity, the ionization fraction of He^+ at a_{end} for 11 parameter sets listed in table A. Here we observe that the result from the revised code lies between 0.3-41% of that of original-corrected code with an outlier at 66%. Though we can compare the ionization fraction of H^+ and $H_2{}^+$ as well, but for all the 11 parameter sets we used, those quantities are too small ($\sim 10^{-100}$) for reliable comparison.

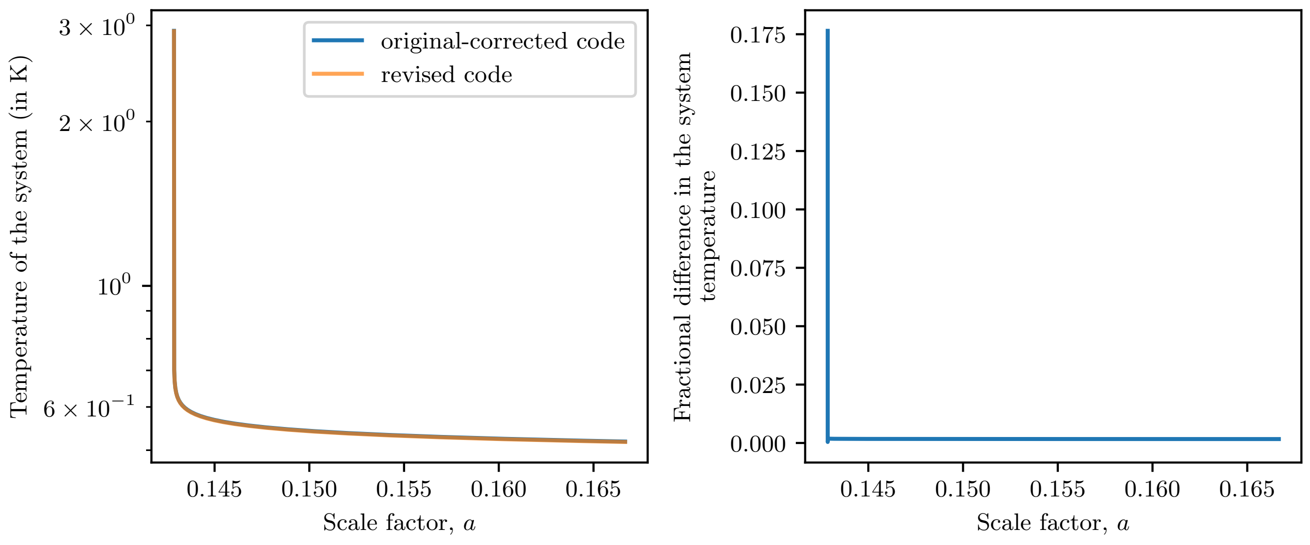


Figure 2.1: Comparing the time evolution of the system temperature between the original-corrected and the revised code

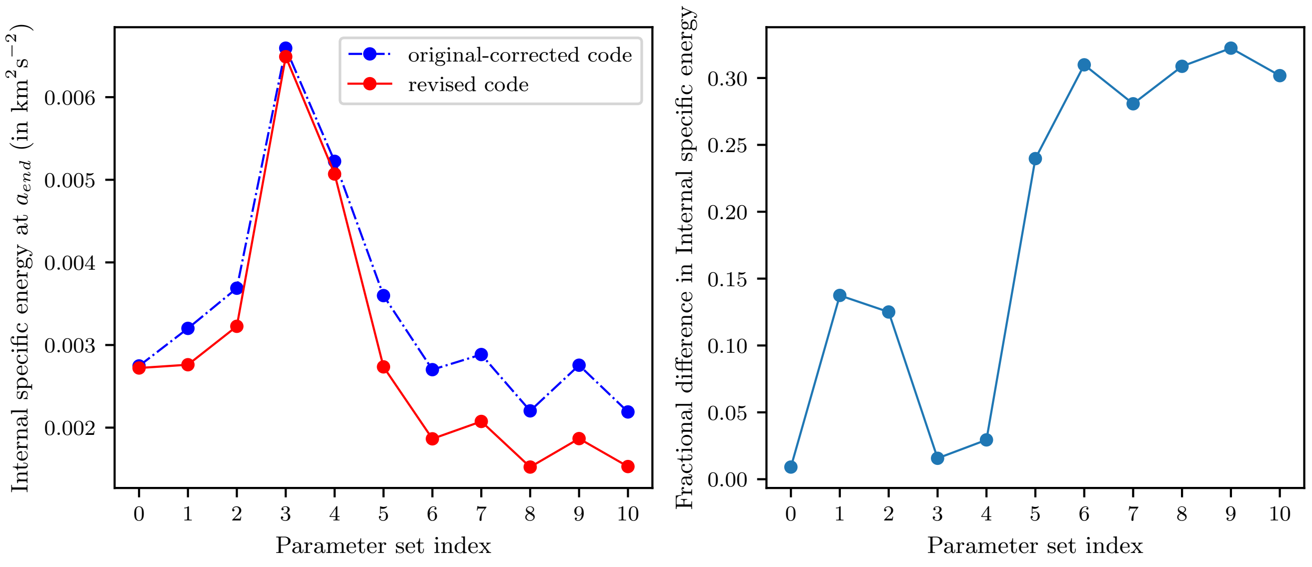


Figure 2.2: Comparing the final total energy for 11 different parameter sets

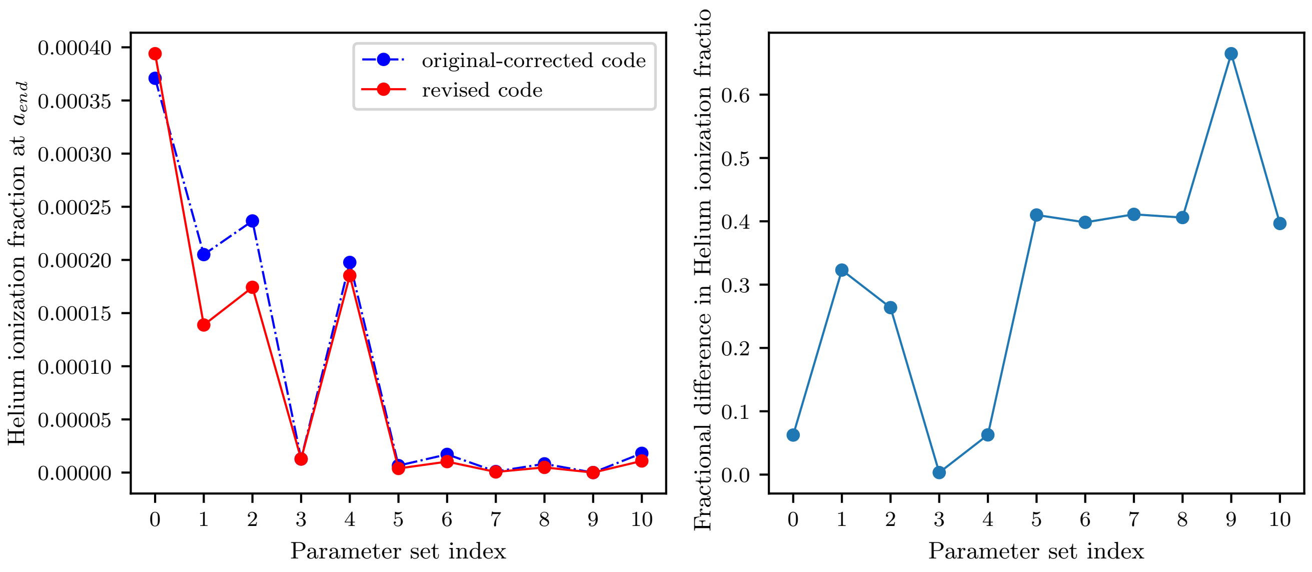


Figure 2.3: Comparing the time evolution of He^+ ionization fraction for 11 different parameter sets

2.2.3 Performance comparison

After verifying the correctness of physics implementation of the revised code, let's compare the performance counters of the two codes. The performance counters were collected using `perf`[iii] performance analysis tool on two HPC clusters `Ulysses v2` [iv] and `Marconi100` [v]. `Ulysses v2` is the HPC cluster hosted at SISSA and `Marconi100` is the HPC cluster hosted by CINECA. The table 2.1 lists the one node specification of the two clusters. We collected the hardware counters `cycles, instructions, cache-references, cache-misses, branches-references, branch-misses`.

For the comparison, we used 44 particles with the parameters sets listed in table A and averaged the performance counters over 3 runs. The counters are listed in table 2.2 and 2.3. We observe that even though the number of instruction executed per cycle is larger for the original-corrected code, the total number of executed instructions, branch references, and branch misses are smaller for the revised version, which is responsible for its performance improvement. The revised code exhibits a 2.53x speedup on `Ulysses v2` and 3.03x speedup on `Marconi100` . The relative performance of the same codes on the two clusters are consistent with the ratio of respective CPU frequency.

2.3 GPU offloading of the revised code

As noted in the previous section, the code refactoring introduces a massive performance improvement over the original code. Given that the computation for each particle is independent of the others, it makes the problem embarrassingly parallel. We can use this to our advantage to further improve the code performance by offloading the computation of our program on GPUs. A GPU has hundreds of cores that can handle thousands of threads simultaneously which is in contrast to a CPU that offers a

[iii] `perf` performance analysis tool: https://perf.wiki.kernel.org/index.php/Main_Page
[iv] `Ulysses v2` cluster: https://www.itcs.sissa.it/services/computing/hpc
[v] `Marconi100` cluster: https://www.hpc.cineca.it/hardware/marconi100

Specification	**Host**	
	`Ulysses v2`	`Marconi100`
CPU model	Intel Xeon E5-2683v4	IBM POWER9 AC922
Architecture	x86_64	ppc64le
Sockets	2	2
Cores/socket	16	16
Threads/core	2	4
Total physical cores	32	32
Total logical cores	64	128
CPU Max frequency	3.0 GHz	3.8 GHz
L1d cache	32K	32K
L1i cache	32K	32K
L2 cache	256K	512K
L3 cache	40960K	10240K
RAM	64 GB DDR4	256 GB DDR4
GPU (on specific nodes)	2× Tesla P100-PCIE 16GB	4× Tesla V100-SXM2 16GB

Table 2.1: Hardware specification of the one node of `Ulysses v2` and `Marconi100` clusters[7],[8]

	Original-corrected code		**Revised code**	
Event	**Count**	**Summary**	**Count**	**Summary**
Cycles	1.76E+12	2.552 GHz	6.96E+11	2.555 GHz
Instructions	2.07E+12	1.18 IPC	6.32E+11	0.910 IPC
Cache references	5.09E+07	0.074 M/sec	1.18E+07	0.043 M/sec
Cache misses	3.68E+07	72.281% of all cache refs	8.62E+06	73.107% of all cache refs
Branch references	1.51E+11	219.641 M/sec	6.14E+10	255.15 M/sec
Branch misses	2.78E+08	0.18 % of all branches	1.13E+07	0.02 % of all branches
Total elapsed time	689.39 sec		272.79 sec	

Table 2.2: `perf` counter comparison on `Ulysses v2` cluster

	Original-corrected code		**Revised code**	
Event	**Count**	**Summary**	**Count**	**Summary**
Cycles	1.83E+12	3.788 GHz	5.93E+11	3.722 GHz
Instructions	2.29E+12	1.25 IPC	6.68E+11	1.13 IPC
Cache references	4.89E+11	1011.661 M/sec	1.13E+11	711.846 M/sec
Cache misses	1.25E+09	0.255% of all cache refs	6.26E+08	0.552% of all cache refs
Branch references	1.49E+11	309.229 M/sec	6.07E+10	380.605 M/sec
Branch misses	4.33E+08	0.29 % of all branches	7.81E+07	0.13 % of all branches
Total elapsed time	483.34 sec		159.40 sec	

Table 2.3: `perf` counter comparison on `Marconi100` cluster

few cores capable of handling just a few threads. Also, if we consider a computing cluster where each node is equipped with multiple GPUs, the GPUs alone provide a more scalable computing platform than all the CPUs. So we can offload our computations to GPUs by mapping each particles to a GPU thread and expect a massive performance gain.

Though the a GPU uses SIMT model to parallelize the computations, it is not much useful in our case. It must be noted that the execution path of each particle in our code is independent of the others and is strictly depends on the initial parameter. So as the execution proceeds on a GPU, the threads will eventually lose synchronization, limiting the full utilization of SIMT. Even in that case, just the massive amount of threads and arithmetic units on the GPU will boost the code performance significantly.

For offloading a code on GPU, one can use either API based frameworks like CUDA and OpenCL or directive based interfaces like OpenACC and OpenMP. Our problem is limited to parallelizing a main loop which is embarrassingly parallel. Also in every iteration, it uses same data set over and over again. This requires to transfer data between CPU and GPU only at the beginning and at the end of the program execution on GPU. So in our case, directive based approach is more appropriate as they are more suitable for loop-level parallelization. On the top of that, the two directive based methods are portable, allowing us to use the same code for both the multicore execution and GPU offloading.

To test our implementation, we had access to NVIDIA P100 GPU on `Ulysses v2` and NVIDIA V100 GPU on `Marconi100` . While GPU offloading using OpenACC is supported on both accelerators, OpenMP based offloading is only supported for NVIDIA V100 GPUs and higher[vi]. So in order to utilize both kind of GPU resources and to do comparative study, we decided to use OpenACC. Using OpenACC has another advantage that it leaves on the compiler to choose automatically the level of parallelization based on underlying hardware unlike OpenMP where the user has to make specific declaration[9]. Our OpenACC implementation is based on the OpenACC API version 2.7.[vii]

Before executing the program on GPU, one has to make sure that the relevant data is available on GPU memory. This was ensured by first allocating memory on GPU using explicit data directive `acc enter data create()` and then initializing lookup table on GPU using the directive `acc parallel loop`. Since the arrays of lookup tables were very small in size and initialization happened in less than a second, there was no need for fine-tuning. The clause `acc update device()` was used to copy the variables that were initialized on CPU host. The directive `acc declare create()` was needed as well for all the `extern` variables.

Coming to the offloading of main loop over all the particles, in order to utilize maximum GPU resources, it is necessary to use optimized kernel launch configuration. In OpenACC, it is modified using the clauses `num_gang()`, for the number of CUDA blocks, and `num_workers()` and `vector_length()` for the size of blocks (number of threads per block) which is the product of these two. To offload the main loop on GPU, we used the following directive and clauses: `acc parallel loop gang vector_length() independent`. The `parallel` directive with `vector_length()` clause launches multiple gangs (blocks) with given vector length (blocksize). Since we are providing the blocksize explicitly, the number of blocks is determined implicitly and it is given by the size of loop divided by the blocksize. The `loop gang` construct make the loop iterations to be shared across the gangs of the parallel region and `independent` specifies that all the loop iterations

[vi]https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html#openmp-use

[vii]The OpenACC API v2.7: https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf

Listing 2.1: Kernel execution time on a P100 GPU obtained using nvprof tool

```
==29867== NVPROF is profiling process 29867, command: ./chemistry_nvc
==29867== Profiling application: ./chemistry_nvc
==29867== Profiling result:
   Time(%)      Time  Calls       Avg       Min       Max  Name
   100.00%  631.073s      1  631.073s  631.073s  631.073s  main_336_gpu
     0.00%  1.0874ms     10  108.74us  1.1520us  981.53us  [CUDA memcpy HtoD]
     0.00%  768.99us      1  768.99us  768.99us  768.99us  [CUDA memcpy DtoH]
     0.00%  44.447us      1  44.447us  44.447us  44.447us  init_chem_216_gpu
     0.00%  11.104us      1  11.104us  11.104us  11.104us  init_chem_77_gpu
     0.00%  8.8950us      1  8.8950us  8.8950us  8.8950us  init_chem_429_gpu
     0.00%  2.3040us      2  1.1520us  1.1520us  1.1520us  [CUDA memset]
     0.00%  1.8240us      1  1.8240us  1.8240us  1.8240us  init_chem_389_gpu
```

are independent and can be executed in parallel.

In order to utilize multiple GPUs on a node, we used MPI-OpenACC based hybrid code. As a simple case, the particles were distributed over all the MPI processes and then each MPI processes were allowed to offload its computations to available GPUs within the same node in round-robin fashion. For this purpose, the OpenACC APIs `acc_get_device_type()`, `acc_get_num_devices()` and `acc_set_device_num()` were used. Since OpenACC is directive based offloading approach, we can also use our hybrid code as MPI only code with CPU only execution.

For all the GPU offloading using OpenACC, we used Nvidia `nvc`[viii] compiler of Nvidia HPC SDK. To compile the MPI-OpenACC hybrid code, on `Ulysses v2` we used Open MPI[ix] that comes with Nvidia HPC SDK v20.9.0. For the same on `Marconi100` , we used IBM Spectrum MPI v10.4.0[x] with Nvidia `pgcc` compiler v21.5.0 available as wrapper `mpipgicc`. For all the compilations, we used compiler based optimization flag `-O2`.

Listing 2.1 shows the typical execution time for different kernels offloaded on GPU, obtained with Nvidia `nvprof` tool.[xi] It shows that the time taken by data transfer between host and device, and initialization of data structures and lookup tables are negligible in comparison with the execution of main loop of algorithm 4. So we only need improve the performance of main loop which we discuss in the next sub-section.

2.3.1 Performance optimization of GPU offloading

On an Nvidia GPU, the threads are launch in a bunch of 32 threads called warps. A warp executes one common instruction at a time. The warps are created, managed, scheduled and executed by an independent unit called streaming multiprocessor (SM). An SM has limited number of arithmetic pipelines, registers, shared memory and warp schedulers. There are also the limitations on number of warp and blocks that an SM can host. These specifications of an SM varies with the particular devices and is listed in table 2.4 for P100 and V100 GPUs.

[viii] Nvidia HPC compiler user's guide: https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html

[ix] Open MPI documentation: https://www.open-mpi.org/doc/

[x] IBM Spectrum MPI documentation: https://www.ibm.com/docs/en/smpi

[xi] Nvidia nvprof profiling tool: https://docs.nvidia.com/cuda/profiler-users-guide/index.html

Specification	Tesla P100	Tesla V100
Architecture	Pascal	Volta
CUDA capability version	6.0	7.0
Number of SMs	56	80
FP32 cores/SM	64	64
Total FP32 cores	3584	5120
Peak FP32 TFLOPS	9.3	15.7
FP64 cores/SM	32	32
Total FP64 cores	1792	2560
Peak FP64 TFLOPS	4.7	7.8
GPU max clock rate	1329 MHz	1530 MHz
Memory interface	4096-bit HBM2	4096-bit HBM2
Memory size	16 GB	16 GB
Memory bandwidth	732 GB/s	900 GB/s
Peak power comsumption	250 W	300 W
Warp schedulers/SM	2	4
Threads/warp	32	32
Max warp/SM	64	64
Max thread blocks/SM	32	32
Max 32-bit registers/SM	65536	65536
Max registers/block	65536	65536
Max registers/thread	255	255
Max threads/block	1024	1024

Table 2.4: Hardware specification for P100 and V100 GPUs[10],[11]

As a first step toward optimization, we can try manipulating the occupancy of the SMs. The pool of warps that are allocated on a SM are called active warps. The occupancy is the ratio of the number of active warps to the maximum number of possible active warps. The occupancy can be changed by changing the kernel launch configuration that is, the grid size and blocksize. We can determine the optimal launch configuration simply by looking at the time it takes to complete the program execution for different configurations of blocksize. But since both P100 and V100 GPUs allow maximum 1024 threads and 65536 registers per block, as we try to increase the blocksize, the number of registers available per thread will get reduced. So while we modified the blocksize using the clause `vector_length()`, we also specified the maximum number of registers per thread using the compiler flag `-gpu=maxregcount:<num>`. While there is also limitation on shared memory available per block and shared memory per multiprocessor, it is not a constraint since our program is not data intensive.

Table 2.5 and 2.6 list the elapsed time for the execution of main loop for different launch configurations on `Ulysses v2` and `Marconi100` respectively. We used total $2^{20} = 1,048,576$ number of particles with the parameter sets listed in table A. The elapsed time is reported in seconds unless specified otherwise.

From the results, we infer that the best launch configuration is 512 threads per block, 2048 number of blocks with maximum 128 registers per thread. We also observe that for all launch configurations, a V100 GPU delivers a performance more than twice that of a P100 GPU. This difference in performance is reflected by the difference in their respective architecture. A V100 GPU

		Number of threads per block (number of blocks = 2^{20}/#threads)					
		32	**64**	**128**	**256**	**512**	**1024**
Max register count per thread	**32**	>12 hrs	>12 hrs	>12 hrs	>12 hrs	>12 hrs	>12 hrs
	64	>12 hrs	>12 hrs	>12 hrs	>12 hrs	>12 hrs	>12 hrs
	128	13155.59	13213.84	13182.33	13091.13	13014.41	
	256	13331.69	13257.79	13180.75	13171.60		
	512	13331.79	13260.08	13183.37			
	1024	15347.91	15142.11				
	2048	15367.45					

Table 2.5: Elapsed time for executing the main loop for different launch configurations on `Ulysses v2` with 2^{20} particles. The timings are listed in seconds except for the red-banded rows where the runtime exceeded the maximum time-limit of the job. The green cell in the table shows the minimum elapsed time among others.

		Number of threads per block (number of blocks = 2^{20}/#threads)					
		32	**64**	**128**	**256**	**512**	**1024**
Max register count per thread	**32**	>24 hrs	>24 hrs	>24 hrs	>24 hrs	>24 hrs	>24 hrs
	64	29181.94	29269.70	29313.00	29555.42	30906.89	31687.69
	128	6187.04	5992.84	5949.61	5680.77	5046.40	
	256	6367.12	6343.20	6306.21	6020.38		
	512	6363.49	6344.08	6309.00			
	1024	6365.32	6346.93				
	2048	6362.37					

Table 2.6: Elapsed time for executing the main loop for different launch configurations on `Marconi100` with 2^{20} particles. The timings are listed in seconds except for the red-banded rows where the runtime exceeded the maximum time-limit of the job. The green cell in the table shows the minimum elapsed time among others.

has 80 multiprocessors, total 5120 CUDA cores, compared to 56 and 3584 respectively that a P100 GPU has. Also the GPU and memory clock frequency of V100 is higher than the other. In the next section and onward, we will use 512 threads per block and maximum 128 registers per thread as launch configuration unless otherwise mentioned.

2.3.2 Profiling the GPU offloading

Though we obtained the optimal launch configuration for our kernel, we don't know yet if it brings out the best of a GPU. So in this section, we will review the code performance on GPU using the NVIDIA Nsight Compute tool[xii], find out the bottlenecks, and then explore the possibility of further performance improvement. Since the pascal GPU architecture is not supported by Nsight compute, we will only profile our code on V100. For all the profiling, we will use 81920 particles which is

[xii] NVIDIA Nsight Compute: https://docs.nvidia.com/nsight-compute/index.html

twice of the size of wave that is possible with blocksize 512 and maximum 128 registers per thread (discussed in section 3.1).

First we take a look at the occupancy of the code on GPU given in the Fig. 2.4. The theoretical active warp per SM is determined by the launch configuration and is given by

$$\text{Theoretical active warps/SM} = \frac{\text{registers/SM}}{\text{max registers/thread} \times \text{threads/warp}} \tag{2.1}$$

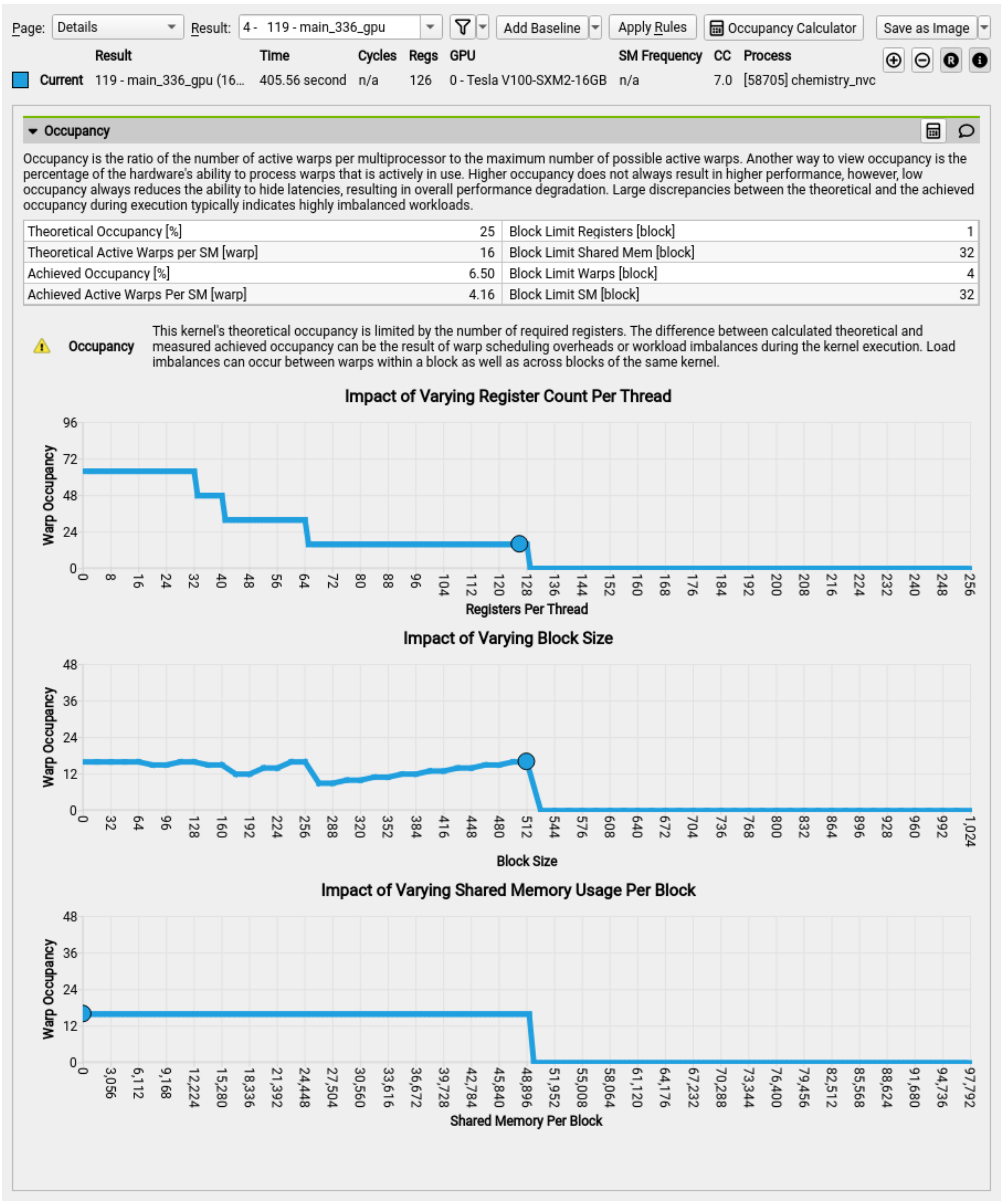
With the values listed in table 2.4 and $\text{max registers/thread} = 128$, we get $\text{theoretical active warp/SM} = 16$. The occupancy percentage is computed against the maximum number of warp that is possible on an SM (64 on V100). As the chart shows, we are getting only 4.16 active warps per SM against 16. The chart also shows the change in occupancy as the launch configuration is changed. But as we observed in previous section, even with high theoretical occupancy, any other launch configuration results in poor performance. So as suggested in the chart, we will take a look at scheduler statistics.

The warp scheduler statistics for main loop kernel is given in fig. 2.5. A V100 GPU has 4 warp schedulers per SM, each responsible for handling maximum 16 warps. The eligible warps are the active warps that are ready to issue the next instruction. At every cycle, the scheduler picks up one warp from the eligible one and issues instruction to it. When there are no eligible warps, the issue slot is skipped. This happens when the active warps are still working on the previous instructions to complete. The lost issue slots indicates lack of eligible warps to hide the latency.

To see the reason behind the warp stall, we take a look at the warp state statistics, given in Fig. 2.6. The primary contributor to warp stall is the wait for fixed latency execution to complete. On an average, we lose 3.3 cycles on this between every two instruction issue. The second highest stall reason is the wait for math pipeline to be available. This is followed by the cycle misses where the warps are waiting to receive an instruction. The next important stall reason is the LG (local and global memory) throttle. This happens when a warp waits for an LG operation to finish for the L1 instruction queue.

Finally the selected warps are the one for which the scheduler issues the instruction. A small difference between selected and not selected warp state indicates there are enough number of warps to cover the latencies which is contradictory to the other stall reasons. The contradiction can be explained by the inherent thread divergence. This happens when one thread completes execution earlier than the other threads in the same warp OR when different threads in a warp branch off to different execution path. In our case, it happens because the different particles take different number of iterations and hence different execution path to converge. When warp divergence happens, only the threads with taken branch are executed forward and the others are disabled. This makes the stall wait more prominent even if the compute resources are utilized optimally.

Now let's take a look at the pipeline and SM utilization of the main loop kernel shown in Fig. 2.7. FP64 pipeline which is responsible for double-precision floating point computations has the highest 70.3% pipeline utilization. The same keeps the SMs busy for 70.27% of elapsed time. It correlates with math pipe throttle stall that we saw in warp state statistics. ALU, FMA and XU are other arithmetic pipelines while LSU is the load storage unit pipeline that is responsible for issuing load and store instructions to L1 cache. The CBU pipeline is the convergence barrier unit that is for branch level convergence, barrier and branch instructions[12]. Overall load of arithmetic pipeline over



	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
🟦 Current	119 - main_336_gpu (16...	405.56 second	n/a	126	0 - Tesla V100-SXM2-16GB	n/a	7.0	[58705] chemistry_nvc

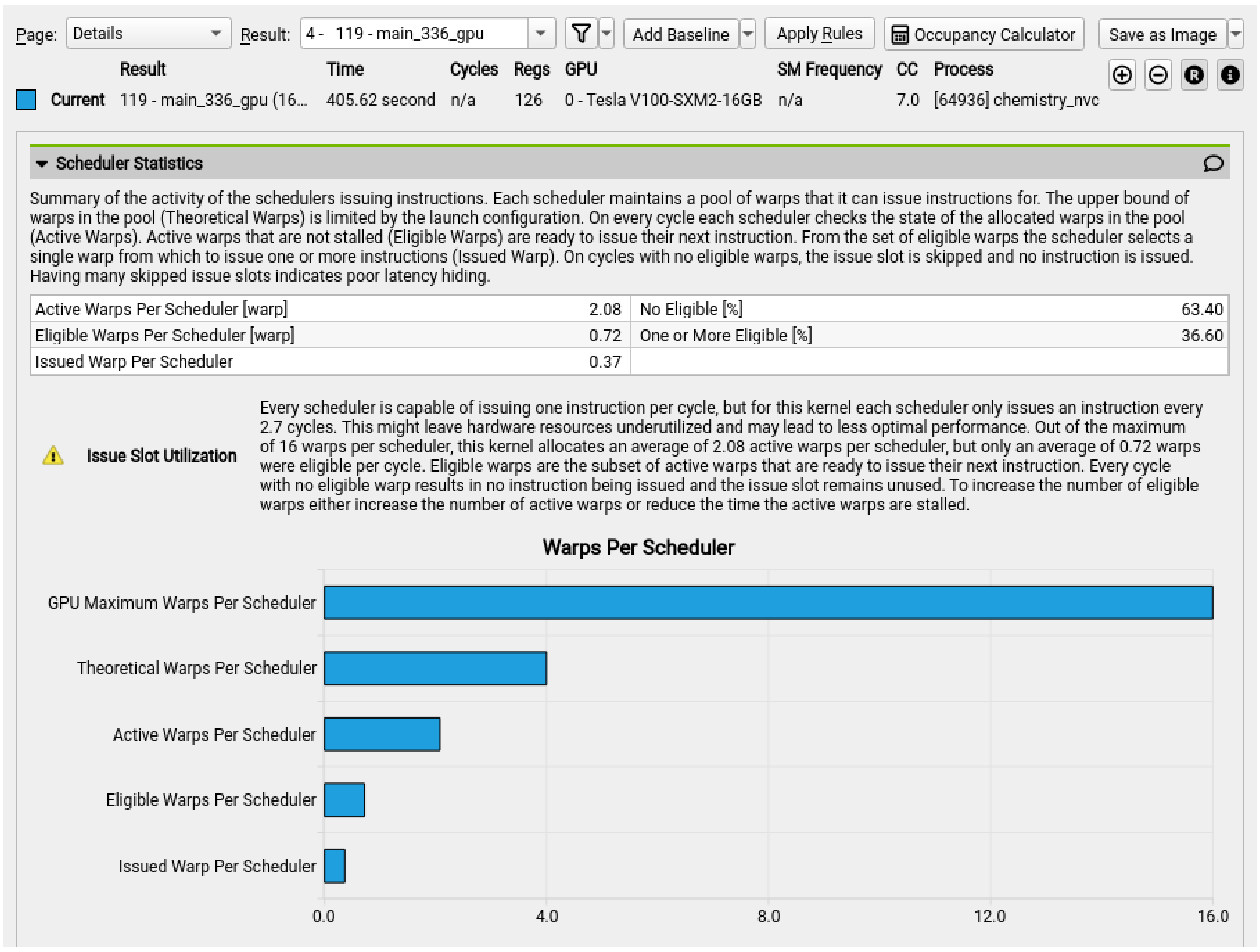
Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	25	Block Limit Registers [block]	1
Theoretical Active Warps per SM [warp]	16	Block Limit Shared Mem [block]	32
Achieved Occupancy [%]	6.50	Block Limit Warps [block]	4
Achieved Active Warps Per SM [warp]	4.16	Block Limit SM [block]	32

⚠ **Occupancy** This kernel's theoretical occupancy is limited by the number of required registers. The difference between calculated theoretical and measured achieved occupancy can be the result of warp scheduling overheads or workload imbalances during the kernel execution. Load imbalances can occur between warps within a block as well as across blocks of the same kernel.

Figure 2.4: Occupancy of the main loop kernel on V100 with 81920 particles



	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
🟦 Current	119 - main_336_gpu (16...	405.62 second	n/a	126	0 - Tesla V100-SXM2-16GB	n/a	7.0	[64936] chemistry_nvc

Scheduler Statistics

Summary of the activity of the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (Theoretical Warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (Active Warps). Active warps that are not stalled (Eligible Warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions (Issued Warp). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.

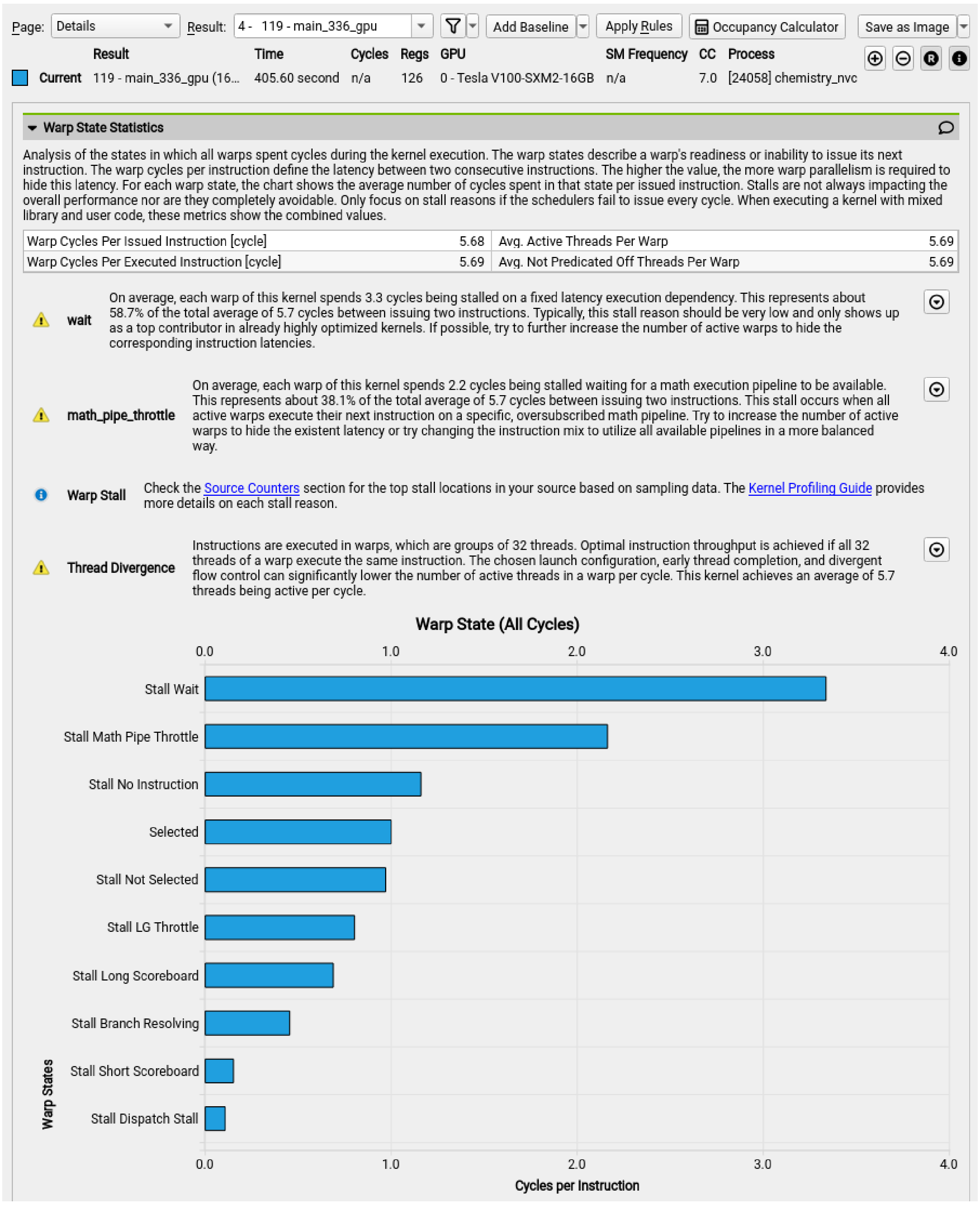
Active Warps Per Scheduler [warp]	2.08	No Eligible [%]	63.40
Eligible Warps Per Scheduler [warp]	0.72	One or More Eligible [%]	36.60
Issued Warp Per Scheduler	0.37		

Issue Slot Utilization

Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 2.7 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 16 warps per scheduler, this kernel allocates an average of 2.08 active warps per scheduler, but only an average of 0.72 warps were eligible per cycle. Eligible warps are the subset of active warps that are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps either increase the number of active warps or reduce the time the active warps are stalled.

Warps Per Scheduler
GPU Maximum Warps Per Scheduler
Theoretical Warps Per Scheduler
Active Warps Per Scheduler
Eligible Warps Per Scheduler
Issued Warp Per Scheduler
0.0
4.0
8.0
12.0
16.0

Figure 2.5: Warp scheduler statistics for the main loop kernel on V100 with 81920 particles



	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
🟦 Current	119 - main_336_gpu (16...	405.60 second	n/a	126	0 - Tesla V100-SXM2-16GB	n/a	7.0	[24058] chemistry_nvc

Warp State Statistics

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

Warp Cycles Per Issued Instruction [cycle]	5.68	Avg. Active Threads Per Warp	5.69
Warp Cycles Per Executed Instruction [cycle]	5.69	Avg. Not Predicated Off Threads Per Warp	5.69

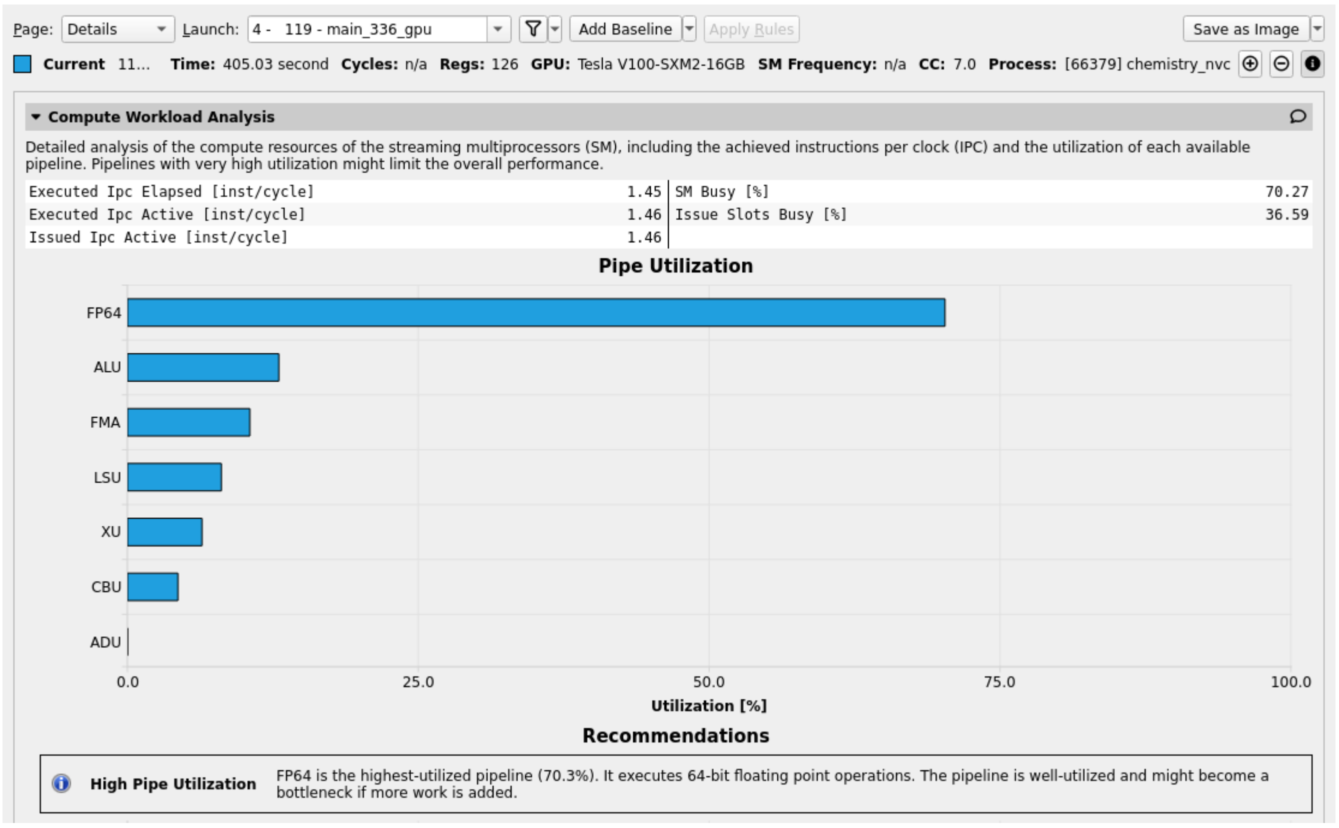
wait On average, each warp of this kernel spends 3.3 cycles being stalled on a fixed latency execution dependency. This represents about 58.7% of the total average of 5.7 cycles between issuing two instructions. Typically, this stall reason should be very low and only shows up as a top contributor in already highly optimized kernels. If possible, try to further increase the number of active warps to hide the corresponding instruction latencies.

math_pipe_throttle On average, each warp of this kernel spends 2.2 cycles being stalled waiting for a math execution pipeline to be available. This represents about 38.1% of the total average of 5.7 cycles between issuing two instructions. This stall occurs when all active warps execute their next instruction on a specific, oversubscribed math pipeline. Try to increase the number of active warps to hide the existent latency or try changing the instruction mix to utilize all available pipelines in a more balanced way.

Warp Stall Check the Source Counters section for the top stall locations in your source based on sampling data. The Kernel Profiling Guide provides more details on each stall reason.

Thread Divergence Instructions are executed in warps, which are groups of 32 threads. Optimal instruction throughput is achieved if all 32 threads of a warp execute the same instruction. The chosen launch configuration, early thread completion, and divergent flow control can significantly lower the number of active threads in a warp per cycle. This kernel achieves an average of 5.7 threads being active per cycle.

Figure 2.6: Warp state statistics for the main loop kernel on V100 with 81920 particles



```
Page: Details   Launch: 4 - 119 - main_336_gpu   Add Baseline   Apply Rules   Save as Image
Current 11...  Time: 405.03 second  Cycles: n/a  Regs: 126  GPU: Tesla V100-SXM2-16GB  SM Frequency: n/a  CC: 7.0  Process: [66379] chemistry_nvc

Compute Workload Analysis
Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available
pipeline. Pipelines with very high utilization might limit the overall performance.
Executed Ipc Elapsed [inst/cycle]          1.45 | SM Busy [%]                 70.27
Executed Ipc Active [inst/cycle]           1.46 | Issue Slots Busy [%]        36.59
Issued Ipc Active [inst/cycle]             1.46 |

Recommendations
High Pipe Utilization   FP64 is the highest-utilized pipeline (70.3%). It executes 64-bit floating point operations. The pipeline is well-utilized and might become a
                        bottleneck if more work is added.
```

Figure 2.7: Compute workload analysis for the main loop kernel on V100 with 81920 particles

the other pipelines strongly indicates that our code is strictly compute bound.

Similarities between the different types of instruction per cycle (Ipc) counters in compute workload analysis explains the only little difference between selected and not selected warp in Fig. 2.6. The scheduler was able to issue instructions to one warp per cycle. That is why we see more than one active issued Ipc in compute workload analysis.

Chapter 3

Benchmark and performance comparison

In this chapter, we measure and compare the performance of revised code on different CPU and GPU architectures. To obtain all the results, we used the parameter set listed in table A where i^{th} particle was assigned the parameter set index $i \mod 11$. For all the GPU offloading, we used 512 threads per block and maximum 128 registers per thread. Unless otherwise mentioned, we will keep the number of MPI process same as the number of GPUs to which we offload the computation.

3.1 Performance trend with increasing number of particles

We see the increase in elapsed time for the execution of main loop in Fig. 3.1 as the number of particles is increased. The single core performance on CPU decreases linearly and it decreases with multiple of 32 particles when all 32 cores of a node are used. Though the GPUs are slower for smaller number of particles, they overshadow CPUs fairly quickly. It crosses single core performance at 12 and 13 number of particles, and it crosses 32 cores performance at 417 and 449 number of particles on `Ulysses v2` and `Marconi100` respectively.

Despite the performance takeover, however, we can see that elapsed time on GPU remains nearly constant as the number of particles is increased. This indicates the under-utilization of GPU resources for the range of particles we used. We examine further the execution of code on GPUs to determine the number of particles for which, the GPU resources are fully utilized. Fig. 3.2 shows the performance trend of the code on a single GPU as the number of particles is increased.

We observe that elapsed time on P100 GPU remains almost constant till it reaches 28672 particles ($\sim$302s) and then till 57344 particles ($\sim$598.5s). The same happens for V100 GPU for 40960 particles ($\sim$175.5s) and 81920 particles ($\sim$351s). It means that for the parameter sets we chose, a P100 GPU can process maximum 28672 particles at a time and a V100 GPU can process maximum 40960 particles at a time. This result perfectly correlates with the system architecture. With 512 threads per block and maximum 128 registers per thread, we are using total 65536 registers per block. It is the maximum number of registers available to a block and it is same as the number of registers we have on each SM (see table 2.4). It means we are offloading one block per SM with blocksize being 512. As P100 has 56 and V100 has 80 SMs, we can thus offload total $56 \times 512 = 28672$ threads on P100 and $80 \times 512 = 40960$ threads on V100 GPU at a time which is the same as the periodicity we

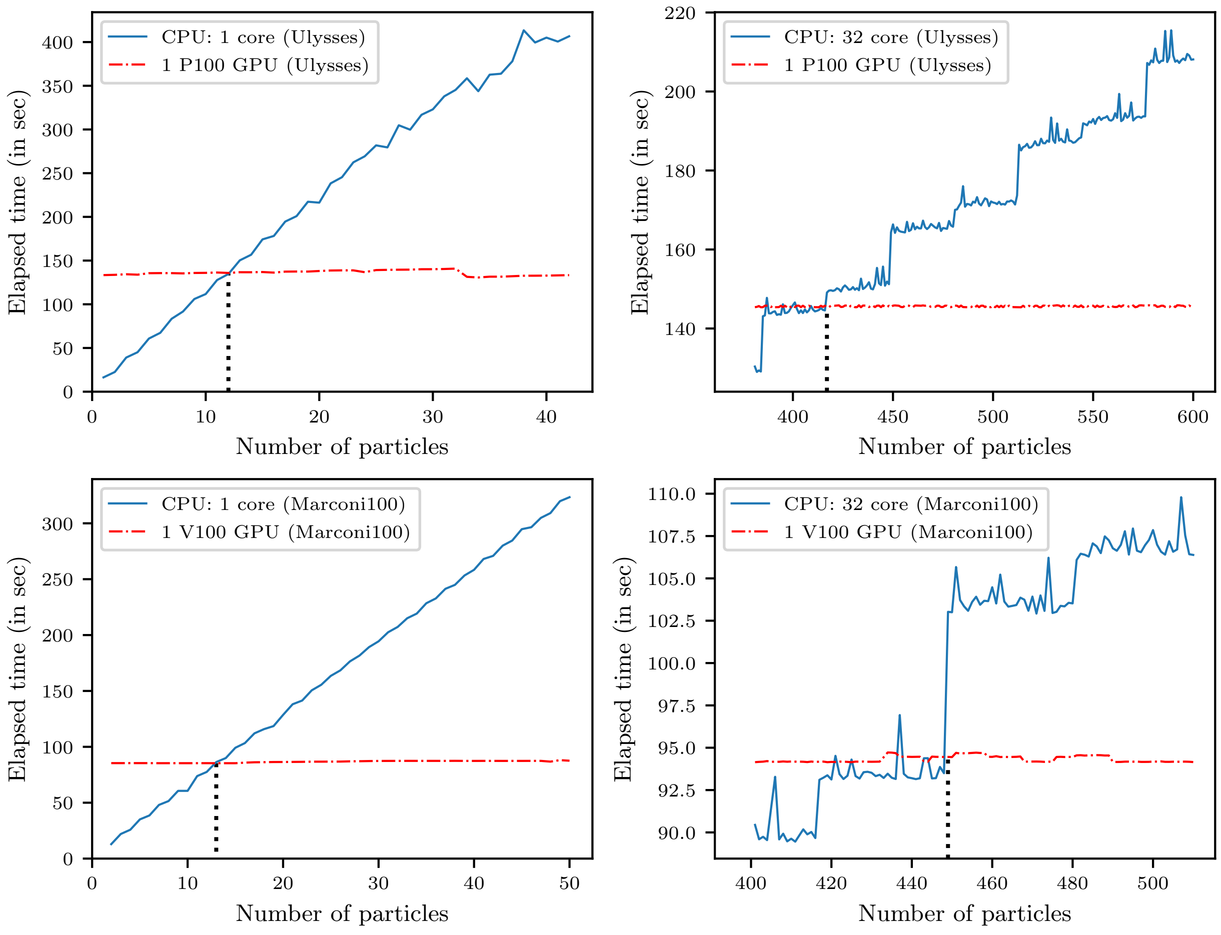


Figure 3.1: Execution time for the main loop as the number of particles is increased. The plots in the first row refer to the `Ulysses v2` and the ones in the second row refer to the `Marconi100` cluster.

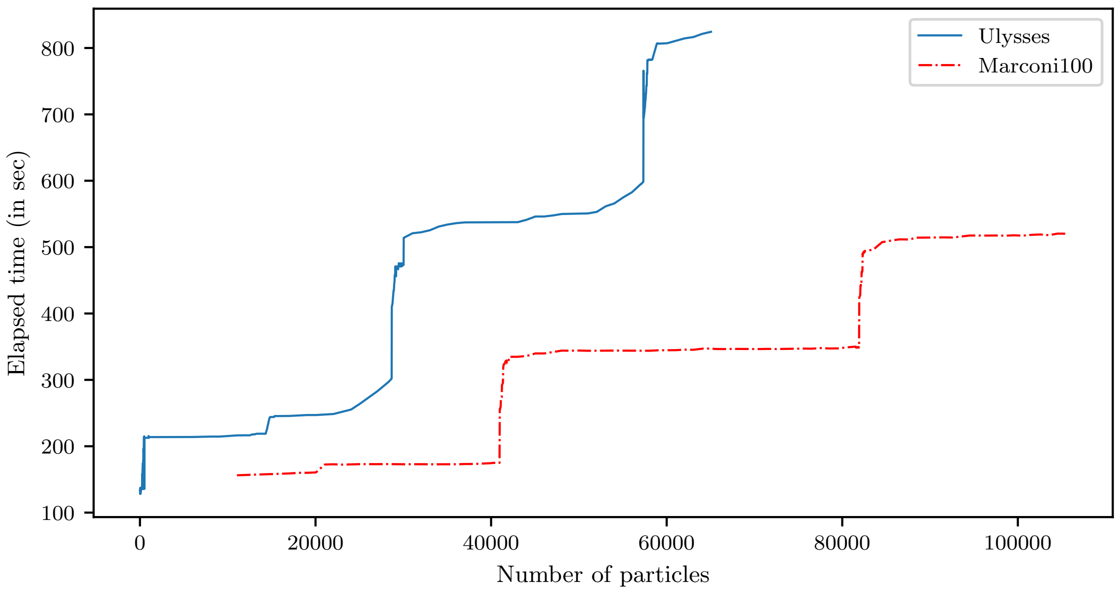


Figure 3.2: Execution time for the main loop on a single GPU as the number of particles is increased

observe in fig. 3.2. The number of threads that can run concurrently on a GPU is called wave. So we can say that with the launch configuration we use, the size of a wave on P100 is 28672 and it is 40960 on V100.

3.2 Performance comparison on single node

As we now have the revised code available with GPU offloading, let's compare its performance on different targets available on a single compute node. Fig. 3.3 shows the performance comparison between MPI only execution using all 32 cores on a node, OpenACC multicore execution and OpenACC execution offloaded to a single GPU with $2^{16} = 65536$ particles. Unlike MPI, OpenACC multicore uses the host multicore CPU as a shared-memory computing platform - like OpenMP. By default OpenACC multicore uses all the available physical cores during execution. While NVIDIA HPC SDK offers an environment variable `ACC_NUM_CORES` to change the number of cores to use at runtime, but as of OpenACC version 2.7, there are no APIs to query the number of cores in use.

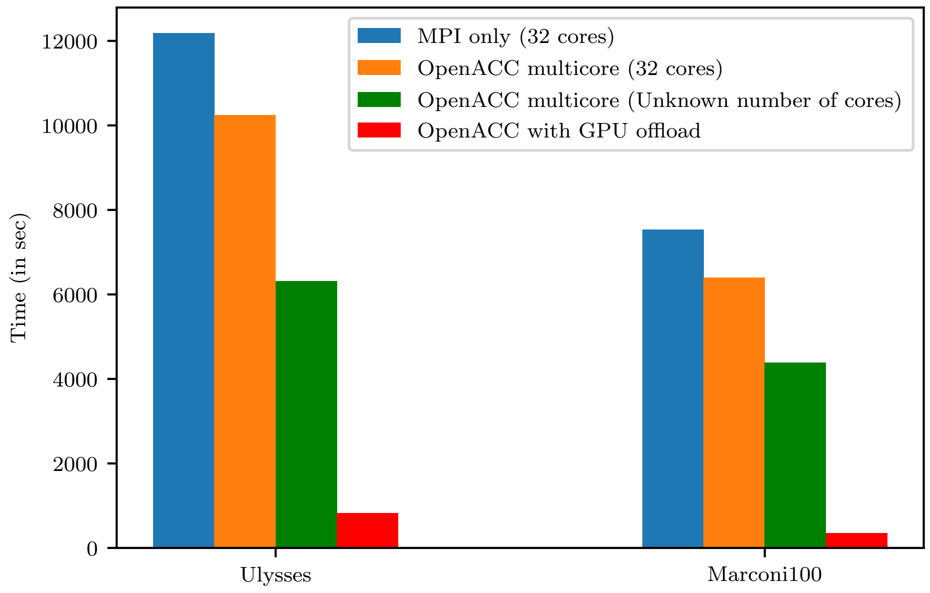


Figure 3.3: Performance comparison of the revised code with $2^{16} = 65536$ particles on different targets available on a single compute node of `Ulysses v2` and `Marconi100`

Along with others, we report the OpenACC multicore performance with default number of cores in use - that is unknown and 32 cores, that is available number of physical cores on a node - assigned with `ACC_NUM_CORES`. We observe that OpenACC multicore offers better performance in both cases - as much as 2x - than MPI. This can be attributed to better cache efficiency as the result of shared-memory execution. On the other hand, a single GPU on `Ulysses v2` performs 14.76x better than MPI only and 7.65x better than the best of OpenACC multicore execution. On `Marconi100` , a single GPU performs 21.74x better than MPI only and 12.65x better than the best of OpenACC multicore execution.

In Fig. 3.4, we compare the performance between GPU offloading on a single compute node of `Ulysses v2` and `Marconi100` . We observe that 2 GPUs of `Marconi100` perform ∼2x better than

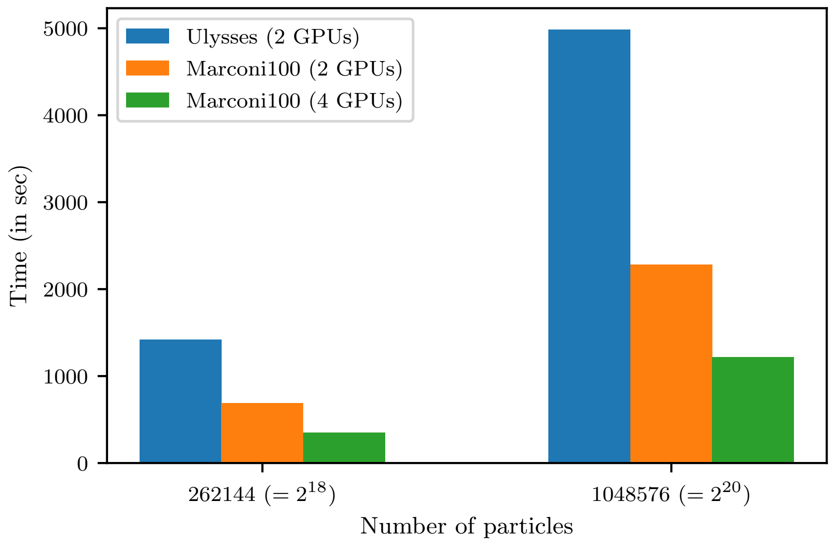


Figure 3.4: Comparing the single node GPU offloading performance between `Ulysses v2` and `Marconi100` clusters

the 2 GPUs of `Ulysses v2` , that is consistent with the result we obtained in section 2.3. The 4 GPUs on `Marconi100` perform even better due to scaling. So we conclude that given an exclusive node on `Ulysses v2` or `Marconi100` , GPU offloading is the best available option. And among the two system, GPU offloading on a single node on `Marconi100` will be 4x faster than the GPU offloading on a single node on `Ulysses v2` .

3.3 Performance scaling on GPUs

Since our problem is embarrassingly parallel, we expect perfect strong and weak scaling as the number of GPUs are increased. We observe the same in Fig. 3.5 and 3.6 respectively. For strong scaling, we used $3 \times 2^{20} = 3,145,728$ particles.

3.4 Optimal number of MPI processes per GPU

So far we have used one MPI process to offload computations to one GPU. However, Gadget-2, where we will embed our revised code, is already based on MPI. In Gadget-2, the particles are initialized and distributed over all the available MPI processes. So it becomes important to identify the optimal distribution of the MPI processes over the available GPUs. We implemented a feature in the code by which a user can specify in the command line argument, the number of MPIs (let's call them the communicating MPIs) that will communicate to GPUs. We used `MPI_Comm_split_type()` function to split the global MPI communicator `MPI_COMM_WORLD` into sub-communicators for each nodes. Then we split the nodal sub-communicator further so that the rank 0 of each sub-sub-communicator acts as an communicating MPI.

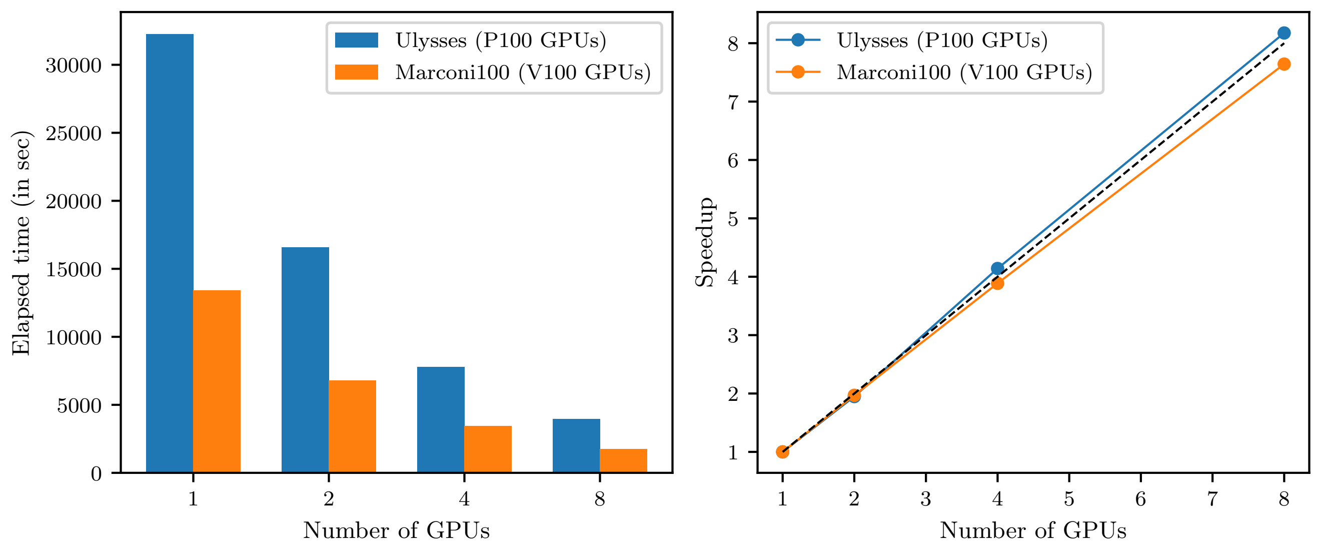


Figure 3.5: Strong scaling on GPUs

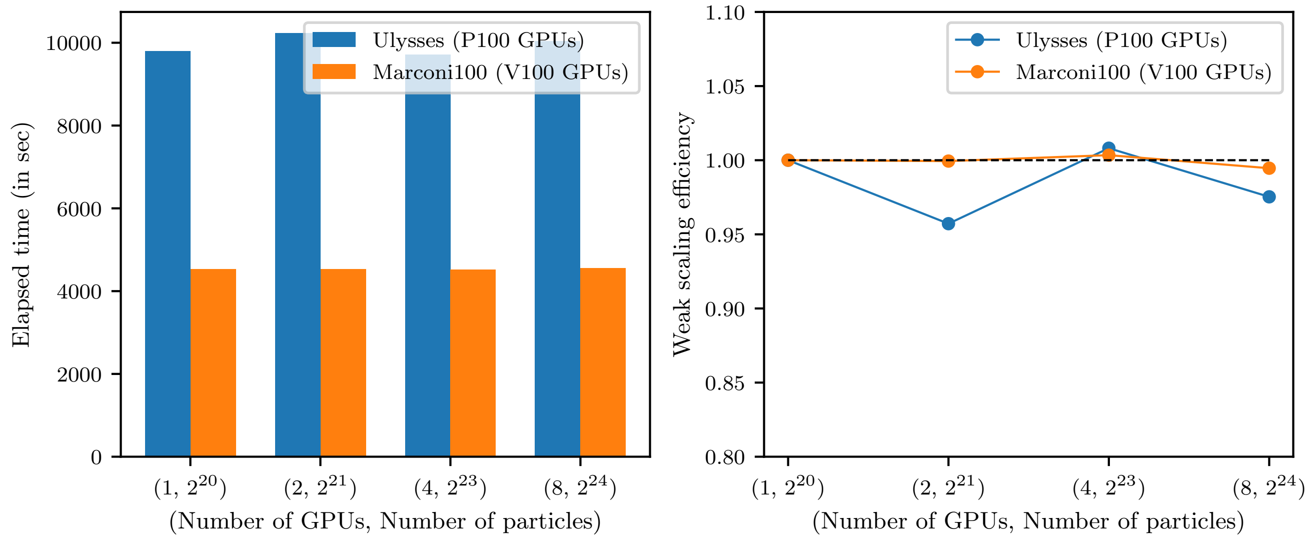


Figure 3.6: Weak scaling on GPUs

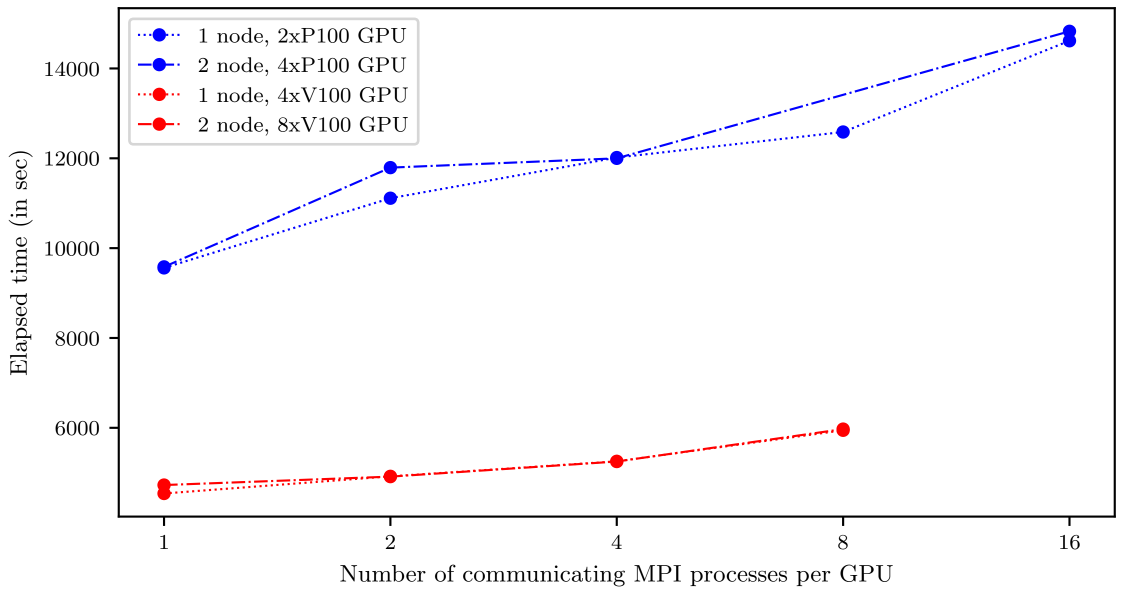


Figure 3.7: Performance comparison for MPI processes distributed evenly over allocated GPUs

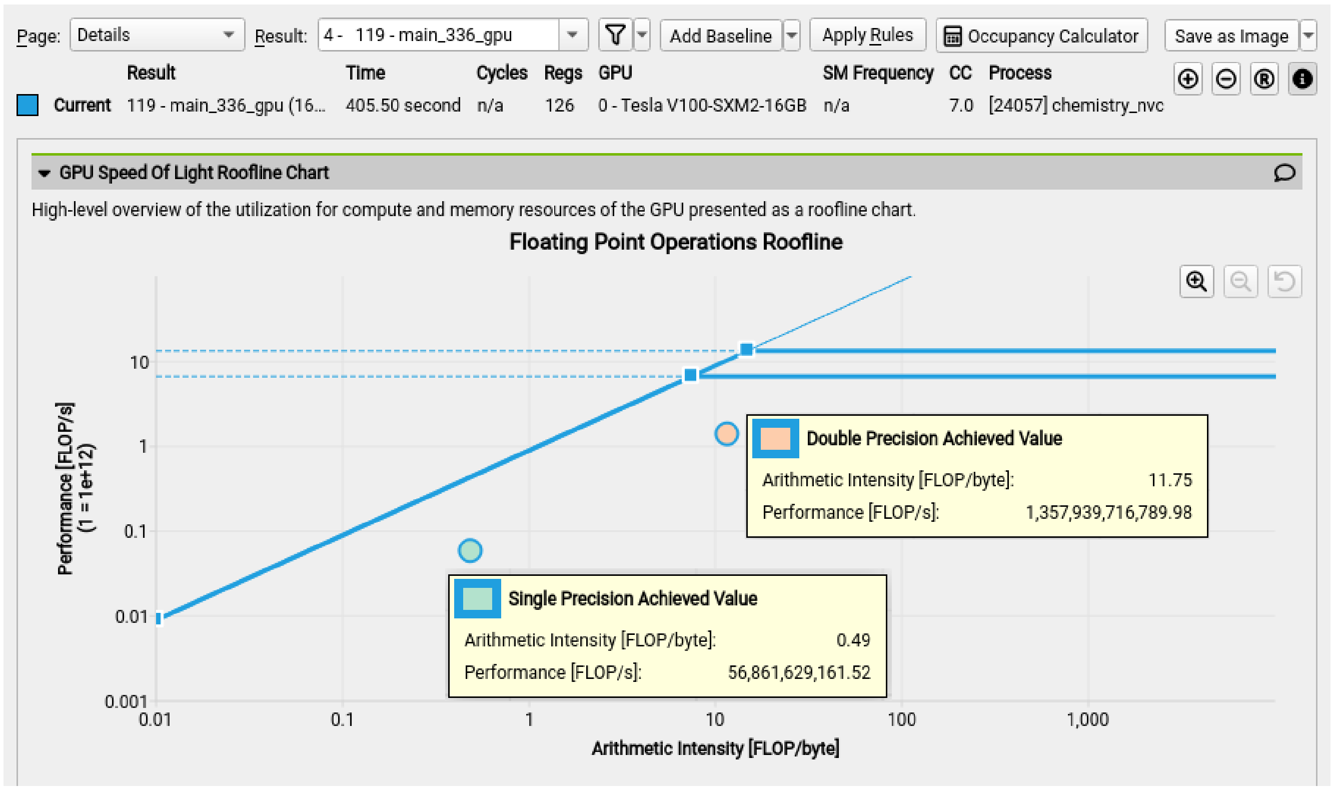
We used `MPI_Gather()` function to collect the particle data structures within the sub-sub-communicator to the communicating MPIs. These communicating MPIs then offload the computation on GPUs. After the computation is over, the communicating MPIs receive the updated data structures from GPUs and distribute them back to the respective MPIs using `MPI_Scatter()`. In this implementation, we used `acc_get_num_devices()` to obtain the total number of available GPUs. This API function returns the total number of GPUs available on the node irrespective of the actual allocation by the job. So we are forced to use all the GPUs available on a node for using this implementation.

Fig. 3.7 shows the performance results from this implementation. 32 cores per node was used for this exercise and the number of particles was decided in a way that each GPU has to perform computation for 2^{20} particles. The x-axis in the plot refers to the number of communicating MPIs. For example, if the number of communicating MPIs is one, that means one MPI process each was communicating to the available GPUs on a node. And each communicating MPI was gathering/scattering the particle data structure from/to 16 (=32/2) MPI processes, including itself.

From the results, we conclude that one MPI process per GPU is the most efficient offloading strategy. Despite having more MPI communication overhead, this choice works better because it reduces the number of host-to-device data transfer and vice versa, and limits the number of CUDA streams to one per GPU.

3.5 Performance efficiency

In the previous sections, we have shown that the revised code performance is better than the original-corrected code and GPU offloading is even better. In this section, we will explore the performance metrics of the revised code on different computing platforms.


Page: Details
Result: 4 - 119 - main_336_gpu
Add Baseline
Apply Rules
Occupancy Calculator
Save as Image
GPU Speed Of Light Roofline Chart
Floating Point Operations Roofline
Performance [FLOP/s] (1 = 1e+12)
Arithmetic Intensity [FLOP/byte]
10
1
0.1
0.01
0.001
0.01
0.1
1
10
100
1,000

	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
🟦 Current	119 - main_336_gpu (16...	405.50 second	n/a	126	0 - Tesla V100-SXM2-16GB	n/a	7.0	[24057] chemistry_nvc

High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Double Precision Achieved Value	
Arithmetic Intensity [FLOP/byte]:	11.75
Performance [FLOP/s]:	1,357,939,716,789.98

Single Precision Achieved Value	
Arithmetic Intensity [FLOP/byte]:	0.49
Performance [FLOP/s]:	56,861,629,161.52

Figure 3.8: Throughput of GPU offloading on a V100 GPU with 81920 particles

We measured the performance of MPI only code in GFLOPS on both `Ulysses v2` and `Marconi100` with the help of PAPI library[i]. We used the PAPI preset event `PAPI_DP_OPS` to collect the number of double-precision floating-point operations needed to execute the main loop of alg. 4. With this, we obtained the throughput of 1 node of `Ulysses v2` to be 25.213 GFLOPS and that of `Marconi100` to be 37.890 GFLOPS.

Fig. 3.8 shows the roofline chart of the code performance for the main loop of alg. 4 on the V100 GPU. The chart was obtained using NVIDIA Nsight Compute tool. It shows that our code has achieved 1.36 TFLOPS in double-precision compared to peak 7.8 TFLOPS listed in table 2.4. The achieved FP64 value lies in the right-hand side of the FP64 ridge point. It indicates that our code is indeed a compute bound code. The FP32 performance is not much relevant in our case. We could not produce the same result for a P100 GPU as it is not supported by Nsight Compute.

To measure the power efficiency of GPUs we used the NVIDIA's `nvprof` profiling tool. With the option of system profiling, it produces average, min and max SM clock frequency, memory clock frequency, temperature and power consumption for each GPUs. For this purpose, we used 286720 particles, that is equivalent to 10 waves on one P100 GPU and 7 waves on a V100 GPU. We obtained the average power consumption on a P100 to be 73.7W over 2797.13 seconds and 63.0W over 1236.04 seconds on a V100 GPU. So overall, for 286720 particles, a P100 GPU used ∼206,091,993 J energy while a V100 GPU used only ∼77,890,811 J. This indicates that the `Marconi100` cluster is more power efficient for GPU offloading. It is worth noticing that the average power consumption of our code is much lower than the max power consumption as listed in table 2.4. It is the result of non-utilization of complete GPU resources as discussed in the section 2.3.2.

[i]Performance Application Programming Interface (PAPI) library: https://icl.utk.edu/papi/

Chapter 4

Conclusion and future work

In this project, we have successfully managed to refactor and port the original implementation of gas cooling by Maio et al.[2] to different computing platforms using MPI and OpenACC. We found the best launch configuration for our compute kernel to utilize most of GPU resources. With the best launch configuration we observed that a single P100 GPU outperforms 32 CPU cores of `Ulysses v2` by a factor of 7.65, and a single V100 GPU outperforms 32 CPU cores of `Marconi100` by a factor of 12.65. On the top of that, GPUs on one node of `Marconi100` , equipped with 4×V100 GPU performs 4x better than the GPUs on one node of `Ulysses v2` that has 2×P100 GPU.

We have also implemented a feature to allow only a certain number of MPI processes out of all, to offload the computations on GPUs available on a node. Restricting the number of MPIs that connect to a GPU limits the number of CUDA streams and host-device data transfer. As a result, we observed that 1 MPI process per GPU offers the best performance.

There are multiple possibilities to improve the GPU offloading further. As discussed in section 2.3.2, our implementation is severely limited by the saturation of FP64 pipeline and uncoalesced memory access. We know that FP32 pipeline provides twice as much throughput than the FP64 pipeline. So using half-precision floating point computations instead of double-precision as we did in our case, will likely improve the code performance by a factor of 2. However since our code is parameter sensitive, loss of precision will also affect the accuracy of the results significantly.

On the other hand, the uncoalesced memory access issue on GPU is the result of separate, non-sequential memory chunk transaction requested by different threads. This type of transaction happens either when the lookup table is accessed separately by each thread or when the individual elements of particle data structure are updated. While the first type of transaction cannot be avoided or modified, the second type of transaction can be improved with a transformation of particle data structure. When a set of consecutive threads access an element of particle data structure, they each access it from different `struct` making the data access uncoalesced. Instead if we group the identical elements from each particle data structures in an array, every access to them will become coalesced. This will require data transformation twice - at the beginning and the end of offloading - that will add a computational overhead but at the same time, it will save us multiple compute cycles.

Another good exercise for the future development would be to use shared memory units of GPU using `cache` directive of OpenACC on the arrays of lookup tables. It will reduce the load on L1-L2 data transfer link and will likely improve the performance more.

References

[1] Volker Springel. “The cosmological simulation code gadget-2”. In: *Monthly Notices of the Royal Astronomical Society* 364.4 (Dec. 2005), pp. 1105–1134. ISSN: 0035-8711. DOI: 10.1111/j.1365-2966.2005.09655.x. eprint: https://academic.oup.com/mnras/article-pdf/364/4/1105/18657201/364-4-1105.pdf. URL: https://doi.org/10.1111/j.1365-2966.2005.09655.x.

[2] U. Maio et al. “Metal and molecule cooling in simulations of structure formation”. In: *Monthly Notices of the Royal Astronomical Society* 379.3 (July 2007), pp. 963–973. ISSN: 0035-8711. DOI: 10.1111/j.1365-2966.2007.12016.x. eprint: https://academic.oup.com/mnras/article-pdf/379/3/963/3503445/mnras0379-0963.pdf. URL: https://doi.org/10.1111/j.1365-2966.2007.12016.x.

[3] Peter Anninos et al. “Cosmological hydrodynamics with multi-species chemistry and nonequilibrium ionization and cooling”. In: *New Astronomy* 2.3 (1997), pp. 209–224. ISSN: 1384-1076. DOI: https://doi.org/10.1016/S1384-1076(97)00009-2. URL: https://www.sciencedirect.com/science/article/pii/S1384107697000092.

[4] D. Hollenbach and C. F. McKee. “Molecule formation and infrared emission in fast interstellar shocks. I. Physical processes.” In: *The Astrophysical Journal Supplement Series* 41 (Nov. 1979), pp. 555–592. DOI: 10.1086/190631.

[5] Daniele Galli and Francesco Palla. “The chemistry of the early Universe”. In: *Astronomy and Astrophysics* 335 (July 1998), pp. 403–420. arXiv: astro-ph/9803315 [astro-ph].

[6] Anton Lipovka, Ramona Núñez-López, and Vladimir Avila-Reese. “The cooling function of HD molecule revisited”. In: *Monthly Notices of the Royal Astronomical Society* 361.3 (Aug. 2005), pp. 850–854. ISSN: 0035-8711. DOI: 10.1111/j.1365-2966.2005.09226.x. eprint: https://academic.oup.com/mnras/article-pdf/361/3/850/2941777/361-3-850.pdf. URL: https://doi.org/10.1111/j.1365-2966.2005.09226.x.

[7] *Intel® Xeon® Processor E5-2683 v4 Product Specifications*. URL: https://www.intel.com/content/www/us/en/products/sku/91766/intel-xeon-processor-e52683-v4-40m-cache-2-10-ghz.html (visited on 02/14/2022).

[8] Scott Vetter, Ritesh Nohria, and Gustavo Santos. *IBM Power System AC922 Technical Overview and Introduction*. 2019. URL: http://www.redbooks.ibm.com/abstracts/redp5494.html?Open (visited on 02/14/2022).

[9] Michael Wolfe. *Burying The OpenMP Versus OpenACC Hatchet*. Jan. 2019. URL: https://www.nextplatform.com/2019/01/16/burying-the-openmp-versus-openacc-hatchet/ (visited on 01/24/2022).

[10] *NVIDIA Tesla P100*. URL: http://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (visited on 02/14/2022).

[11] *NVIDIA TESLA V100 GPU ARCHITECTURE*. 2017. URL: http://www.nvidia.com/object/volta-architecture-whitepaper.html (visited on 02/14/2022).

[12] *NVIDIA Nsight Compute Kernel Profiling Guide*. docs.nvidia.com. URL: https://docs.nvidia.com/nsight-compute/ProfilingGuide/ (visited on 02/14/2022).

Appendix A

List of initial parameter sets

Table A.1: List of 11 initial parameter sets used in code testing and benchmarking (x is the fractional number density, i is the particle index and i^{th} particle is assigned the parameter index given by i mod 11)

Physical quantity	Variable associated with the quantity	Parameter index										
		0	1	2	3	4	5	6	7	8	9	10
Density[1]	SphP[i].Density	6.77E-04	6.77E+00	6.77E-02	6.77E-06	6.77E-04	6.77E-02	6.77E+00	6.77E+00	6.77E+02	6.77E+02	6.77E+02
Temperature[2]	T_{gas}	2.56E+06	6.85E+08	3.18E+07	6.85E+04	1.48E+06	3.18E+05	6.85E+06	6.85E+04	1.48E+06	1.48E+04	1.48E+08
γ	SphP[i].Gamma	1.666E+00	1.666E+00	1.666E+00	1.666E+00	1.666E+00	1.666E+00	1.666E+00	1.666E+00	1.666E+00	1.666E+00	1.666E+00
Entropy	SphP[i].Entropy	1.00E-08	1.00E-08	1.00E-08	1.00E-08	1.00E-08	1.00E-10	1.00E-10	1.00E-12	1.00E-12	1.00E-14	1.00E-10
x_{e}	SphP[i].elec	3.92592E-03	3.92592E-04	3.92592E-04	3.92592E-04	3.92592E-04	3.92592E-04	3.92592E-04	3.92592E-04	3.92592E-04	3.92592E-04	3.92592E-04
x_{H}	SphP[i].HI	8.99607E-02	7.59607E-01	7.59607E-01	7.59607E-01	7.59607E-01	7.59607E-01	7.59607E-01	7.59607E-01	7.59607E-01	7.59607E-01	7.59607E-01
$x_{\mathrm{H}+}$	SphP[i].HII	3.92592E-03	3.92592E-04	3.92592E-04	3.92592E-04	3.92592E-04	3.92592E-04	3.92592E-04	3.92592E-04	3.92592E-04	3.92592E-04	3.92592E-04
$x_{\mathrm{H}-}$	SphP[i].HM	7.60E-21	7.60E-21	7.60E-21	7.60E-21	7.60E-21	7.60E-21	7.60E-21	7.60E-21	7.60E-21	7.60E-21	7.60E-21
x_{H_2}	SphP[i].H2I	1.52E-06	1.52E-06	1.52E-06	1.52E-06	1.52E-06	1.52E-06	1.52E-06	1.52E-06	1.52E-06	1.52E-06	1.52E-06
x_{H_2+}	SphP[i].H2II	7.60E-21	7.60E-21	7.60E-21	7.60E-21	7.60E-21	7.60E-21	7.60E-21	7.60E-21	7.60E-21	7.60E-21	7.60E-21
x_{HD}	SphP[i].HD	7.00E-10	7.00E-10	7.00E-10	7.00E-10	7.00E-10	7.00E-10	7.00E-10	7.00E-10	7.00E-10	7.00E-10	7.00E-10
x_{D}	SphP[i].DI	3.50E-05	3.50E-05	3.50E-05	3.50E-05	3.50E-05	3.50E-05	3.50E-05	3.50E-05	3.50E-05	3.50E-05	3.50E-05
$x_{\mathrm{D}+}$	SphP[i].DII	4.00E-09	4.00E-09	4.00E-09	4.00E-09	4.00E-09	4.00E-09	4.00E-09	4.00E-09	4.00E-09	4.00E-09	4.00E-09
x_{He}	SphP[i].HeI	6.00E-02	6.00E-02	6.00E-02	6.00E-02	6.00E-02	6.00E-02	6.00E-02	6.00E-02	6.00E-02	6.00E-02	6.00E-02
$x_{\mathrm{He}+}$	SphP[i].HeII	-6.00E-12	-6.00E-12	-6.00E-12	-6.00E-12	-6.00E-12	-6.00E-12	-6.00E-12	-6.00E-12	-6.00E-12	-6.00E-12	-6.00E-12
$x_{\mathrm{He}++}$	SphP[i].HeIII	6.00E-12	6.00E-12	6.00E-12	6.00E-12	6.00E-12	6.00E-12	6.00E-12	6.00E-12	6.00E-12	6.00E-12	6.00E-12
$x_{\mathrm{HeH}+}$	SphP[i].HeHII	1.00E-14	1.00E-14	1.00E-14	1.00E-14	1.00E-14	1.00E-14	1.00E-14	1.00E-14	1.00E-14	1.00E-14	1.00E-14

[1]in g cm^{-3} [2]in Kelvin, calculated using other parameters

$$a_{start}[i] = \begin{cases} 0.0909, \text{if } i \text{ is odd} \\ 0.1428, \text{otherwise} \end{cases}, \; a_{end}[i] = \begin{cases} 0.01, \text{if } i \text{ is odd} \\ 0.1666, \text{otherwise} \end{cases}$$