Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Επιτάχυνση του προσομοιωτή BLonD με την χρήση GPU

Διπλωματικη Εργασια

του

ΤΣΑΠΑΤΣΑΡΗ Β. ΠΑΝΑΓΙΩΤΗ

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής

Αθήνα, Αύγουστος 2021

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Επιτάχυνση του προσομοιωτή BLonD με την χρήση GPU

Διπλωματικη Εργασια

του

ΤΣΑΠΑΤΣΑΡΗ Β. ΠΑΝΑΓΙΩΤΗ

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 31 Αύγουστου 2021.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Δημήτριος Σούντρης	Παναγιώτης Τσανάκας	Γεώργιος Γκούμας
Καθηγητής	Καθηγητής	Καθηγητής

Αθήνα, Αύγουστος 2021

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

ΔΗΛΩΣΗ ΜΗ ΛΟΓΟΚΛΟΠΗΣ ΚΑΙ ΑΝΑΛΗΨΗΣ ΠΡΟΣΩΠΙΚΗΣ ΕΥΘΥΝΗΣ

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ενυπογράφως ότι είμαι αποκλειστικός συγγραφέας της παρούσας Πτυχιακής Εργασίας, για την ολοκλήρωση της οποίας κάθε βοήθεια είναι πλήρως αναγνωρισμένη και αναφέρεται λεπτομερώς στην εργασία αυτή. Έχω αναφέρει πλήρως και με σαφείς αναφορές, όλες τις πηγές χρήσης δεδομένων, απόψεων, θέσεων και προτάσεων, ιδεών και λεκτικών αναφορών, είτε κατά κυριολεξία είτε βάσει επιστημονικής παράφρασης. Αναλαμβάνω την προσωπική και ατομική ευθύνη ότι σε περίπτωση αποτυχίας στην υλοποίηση των ανωτέρω δηλωθέντων στοιχείων, είμαι υπόλογος έναντι λογοκλοπής, γεγονός που σημαίνει αποτυχία στην Πτυχιακή μου Εργασία και κατά συνέπεια αποτυχία απόκτησης του Τίτλου Σπουδών, πέραν των λοιπών συνεπειών του νόμου περί πνευματικών δικαιωμάτων. Δηλώνω, συνεπώς, ότι αυτή η Πτυχιακή Εργασία προετοιμάστηκε και ολοκληρώθηκε από εμένα προσωπικά και αποκλειστικά και ότι, αναλαμβάνω πλήρως όλες τις συνέπειες του νόμου στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής άλλης πνευματικής ιδιοκτησίας.

(Υπογραφή)

.....
Τσαπατσάρης Παναγιώτης

31 Αυγούστου 2021

Περίληψη

Το BLonD είναι ένα εργαλείο που αναπτύχθηκε μία ομάδα φυσικών στο CERN, με σκοπό την υλοποίηση προσομοιώσεων. Αυτές χρησιμοποιούνται βοηθητικά για κάποια από τα πειράματα που συμβαίνουν στο CERN, καθώς επιτρέπει στους επιστήμονες να μελετούν τα αποτελέσματα των πειραμάτων. Επίσης χρησιμοποιούνται αρκετά στην διαδικασία του σχεδιασμού μελλοντικών μηχανημάτων και στην αναβάθμιση των υπάρχοντων. Οι προσομοιώσεις αυτές είναι αρκετά χρονοβόρες, κάποιες από αυτές παίρνουν ακόμα και εβδομάδες για να ολοκληρωθούν και έτσι είναι σημαντική η προσπάθεια μείωσης του χρόνου τους.

Οι κάρτες γραφικών είναι ένα εργαλείο που χρησιμοποιείται πλέον ευρέως για την επιτάχυνση προγραμμάτων γενικού σκοπού. Πιο συγκεκριμένα, οι εφαρμογές που επιταχύνονται από την χρήση τους είναι αυτές που παρουσιάζουν έντονη παραλληλία.

Στόχοι της διπλωματικής αυτής είναι (α) η υλοποίηση πολλών λειτουργιών σε κάρτες γραφικών, με σκοπό την επιτάχυνση των προσομοιώσεων με τέτοιο τρόπο ώστε να υπάρχει ευκολία στην χρήση της, και (β) η σύγκριση των αποτελεσμάτων της με την παλιά υλοποίηση σε επεξεργαστές. Οι μετρήσεις πραγματοποιήθηκαν στον ελληνικό υπερυπολογιστή ARIS.

Λέξεις Κλειδιά

Κάρτα γραφικών, beam longitudinal dynamics, GPU, GPGPU, CUDA, HPC

Abstract

BLonD is a tool, developed by a team of physicists at CERN, that is helpful for creating simulations. Worldwide research centers e.g. CERN, Fermilab etc. use it to develop simulations that help scientists to understand better the results of their experiments. Apart from that, it is a decisive factor in the procedure of designing the new machines. The goal of these simulations, is to produce highly accurate predictions, while keeping the run-time low.

Graphics Processing Unit or GPU, is hardware designed for displaying images, that today is also used to accelerate general purpose workloads. They are well known because of the success in accelerating neural network training, because the special feature of them, is that it can perform a massive amount of operations in parallel. So, if a program is parallel it can be benefited by the use of it.

Since BLonD most time-consuming methods are embarrassingly parallel, we decided that it is a perfect candidate for GPU acceleration, which is the aim of this diploma thesis. Firstly, we design some of the BLonD methods for the GPU, and optimize them. Secondly, we implement a mechanism that will let physicists use the GPU implementation without having knowledge of how a GPU works. Finally, using the Greek supercomputer ARIS, we use some benchmarks to compare our version to the CPU version. Our implementation demonstrates a 5.46 x speedup over the CPU version when run on 32 computing nodes.

Keywords

GPU, GPGPU, CUDA, beam longitudinal dynamics, HPC

στους γονείς μου

Ευχαριστίες

First of all I would like to thank my professor, Dimitrios Sountris for giving me the opportunity to do my thesis in his lab and under his supervision. Also, I would like to thank Dr. Helga Timko and CERN in general for letting me do a task for their team that helped me develop my skills and learn a lot of new and interesting things. Additionally, I would like to thank Kostis Iliakis. His guidance played a significant role not only for the completion of this thesis, but also for their writing part of it. Also our collaboration was excellent. Finally, I would like to thank my parents for their support during the past years.

Αθήνα, Αύγουστος 2021

Τσαπατσάρης Παναγιώτης

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Κεφάλαιο 1

Εκτεταμένη Περίληψη

1.1 Εισαγωγή

1.1.1 CERN

Το CERN είναι ο Ευρωπαϊκός οργανισμός για την έρευνα στην πυρηνική ενέργεια. Ιδρύθηκε το 1954 από 12 χώρες κοντά στην Γενεύη. Έιναι γνωστό για τους επιταχυντές σωματιδίων που έχει και παρέχει σε επιστήμονες διάφορων τομέων, όπως η φυσική, για να κάνουν πειράματα. Ένας τομέας της φυσικής που έχει γνωρίσει ανάπτυξη τα τελευταία χρόνια, κυρίως λόγω της συμβολής του CERN, είναι αυτός των beam longitudinal dynamics [1], που ασχολείται με την μελετη της συμπεριφοράς των σωματιδίων σε πεδία επιτάχυνσης. Πειράματα σχετικά με αυτό πραγματοποιούνται στους επιταχυντές σωματιδίων, όπως είναι ο LHC [2] οδηγώντας σε νέα αποτελέσματα.

Τα πειράματα που χρειάζεται να γίνουν είναι αρκετά και απαιτούν πολύ μεγάλη προετοιμασία. Επίσης χρειάζεται αρκετά μεγάλη προετοιμασία προκειμένου να αποφασιστούν τα χαρακτηριστικά των μηχανημάτων που θα σχεδιαστούν. Αυτό απαιτεί τόσο χρόνο όσο και πόρους, γεγονός που καθιστά πιο δύσκολο το έργο των επιστημόνων. Σε όλα αυτά έρχεται να προστεθεί και το ότι το CERN αναστέλει την λειτουργία του ανά κάποια χρονικά διαστήματα, για συντήρηση και αναβάθμιση. Σε αυτές τις περιόδους η εκτέλεση πειραμάτων είναι αδύνατη.

1.1.2 BLonD module

Για τους παραπάνω λόγους αναπτύχθηκε το BLonD module [3], μία συλλογή εργαλείων υλοποιημένα στην γλώσσα προγραμματισμού Python που βοηθούν τους επιστήμονες να υλοποιήσουν τέτοιου είδους προσομοιώσεις. Με αυτό το εργαλείο οι μελετητές, μπορούν να κατανοούν τα πειραματικά αποτελέσματα και ταυτόχρονα να αποφασίζουν τι είδους μηχανήματα θα σχεδιαστούν και με ποιες προδιαγραφές. Αυτές οι προσομοιώσεις τρέχουν για αρκετούς γύρους $\simeq$ 1 billion και διαρκούν για αρκετά μεγάλο χρονικό διάστημα. Συνεπώς είναι πολύ συμαντική η προσπάθεια για μείωση του χρόνου.

1.1.3 Κάρτες Επεξεργασίας Γραφικών

Οι κάρτες γραφικών, ή αλλίως GPU (Graphics Processing Unit) [4], είναι ένα εργαλείο που αναπτύχθηκε για την προβολή εικόνας σε συσκευές εξόδου. Σήμερα χρησιμοποιούνται

σε πολλές συσκευές όπως είναι οι προσωπικοί υπολογιστές, τα κινητά και τα drones. Το χαρακτηριστικό τους είναι πως μπορούν να κάνουν αρκετές πράξεις ταυτόχρονα. Τα τελευταία χρόνια έχουν ξεκινήσει να χρησιμοποιούνται και σε εφαρμογές γενικού σκοπού, επιταχύνοντας αυτές που παρουσιάζουν μεγάλη παραλληλία, και έτσι χρησιμοποιείται ο όρος GPGPU δηλαδή General Purpose GPU [5]. Το BLonD είναι μια εφαρμογή που παρουσιάζει αρκετά μεγάλη παραλληλία στα κομμάτια του που καταναλώνουν το μεγαλύτερο ποσοστό του χρόνου, και αυτό το καθιστά μία από τις εφαρμογές που μπορούν να επιταχυνθούν από τις κάρτες γραφικών.

Στα πλαίσια αυτής της διπλωματικής υλοποιήσαμε πολλές από τις λειτουργίες του BLonD για GPU, με τέτοιο τρόπο ώστε ο χρήστης να μπορεί να χρησιμοποιήσει την κάρτα γραφικών αν το επιθυμεί χωρίς να έχει κάποια γνώση πάνω σε αυτές. Αφού ολοκληρώσαμε όσα αναφέρθηκαν, κάναμε πειράματα προκειμένου να συγκρίνουμε την υλοποίηση μας με αυτην της CPU. Για να το κάνουμε αυτό χρησιμοποιήσαμε τον ελληνικό υπερυπολογιστή ARIS. Η δομή που θα ακολουθηθεί σε αυτό το κεφάλαιο είναι η εξής.

- Στην ενότητα 1.2 θα αναφέρουμε κάποιες πληροφορίες σχετικά με το BLonD και τις GPU.
- Έπειτα στην ενότητα 1.3 θα δείξουμε πως υλοποιήσαμε κάποιες συναρτήσεις στην ΓΠΥ και κάποια από τα αποτελέσματα.
- Στην επόμενη ενότητα 1.4 θα δείξουμε τον μηχανισμό με τον οποίο χρησιμοποιείται η GPU στα πειράματά μας.
- Στην ενότητα 1.5 θα δείξουμε τα αποτελέσματα που πήραμε από τα benchmarks και τις προσεγγιστικές μεθόδους που χρησιμοποιήσαμε.

1.2 Θεωρητικό Υπόβαθρο

Σε αυτό το κεφάλαιο παρουσιάζουμε περιληπτικά το BLonD module και το πως μοιάζουν οι προσομοιώσεις που γίνονται με αυτό. Τέλος παραθέτουμε κάποιες πληροφορίες για τις GPU.

1.2.1 BLonD Module

Οι προσομοιώσεις που αναπτύσσονται με την χρήση του BLonD αποτελούνται από 2 μέρη. Στο πρώτο μέρος περιλαμβάνονται κάποιες αρχικοποιήσεις και στο δεύτερο μέρος υπάρχει ένας βρόχος που επαναλαμβάνεται για αρκετές επαναλήψεις στον οποίο αυτά τα αντικείμενα αλληλεπιδρούν μεταξύ τους. Στα αντικείμενα που υπάρχουν στην προσομοίωση τα σημαντικότερα αναφέρονται παρακάτω.

- Η ακτίνα (Beam) που περιλαμβάνει δύο πίνακες με τις συντεταγμένες των σωματιδίων, τους dE και dt, που είναι η ενέργεια και ο χρόνος σε σχέση με κάποια σημεία αναφοράς. Σκοπός της προσομοίωσης είναι η επεξεργασία αυτών των δύο πινάκων. Αυτοί οι πίνακες έχουν μέγεθος ίσο με τον αριθμό των σωματιδίων του πειράματος.

- Το ιστόγραμμα (profile) είναι ένα αντικείμενο που χρησιμοποιείται για να υπολογίσει τον πίνακα του ιστογράμματος του πίνακα dt της ακτίνας. Ο αριθμός των διαστημάτων του ιστογράμματος είναι συγκεκριμένος για το πείραμα και το μέγεθος του είναι σε τάξη 1000 φορές μικρότερο από τον αριθμό σωματιδίων.
- Ο tracker είναι το αντικείμενο που αναλαμβάνει να τροποποιήσει τις τιμές των πινάκων της ακτίνας, μέσω των συναρτήσεων kick και drift.

Πιο περιγραφικά στο πείραμα μας γίνονται τα εξής. Δίνουμε τα χαρακτηρηστικά της ακτίνας ως είσοδο, και κάποια άλλα χαρακτηριστικά του επιταχυντή. Στην συνέχεια, επαναλαμβάνεται η εξής διαδικάσια.

- Υπολογίζεται το ιστόγραμμα του πίνακα dt της ακτίνας.
- Σύμφωνα με το ιστόγραμμα τροποποιούνται κατάλληλα κάποιοι πίνακες δυναμικων.
- Με βάση τους παραπάνω πίνακες, τροποποιούνται οι πίνακες της ακτίνας.

Η παραπάνω διαδικασία γίνεται για αρκετά μεγάλο αριθμό γύρων και είναι αρκετά χρονοβόρα, ειδικά για μεγάλο αριθμό σωματιδίων. Στην προσπάθεια να επιταχυνθεί, κάποιο κομμάτι του BLonD που καταναλώνει χρόνο γράφτηκε στην C++ και παραλληλοποιήθηκε με το OpenMP [6] [7]. Στην συνέχεια δημιουργήθηκε μία νέα έκδοση που χρησιμοποιούσε πολλούς κόμβους που επικοινωνούν μεταξύ τους με το MPI [8] [9]. Η δική μας υλοποίηση θα μπορεί να χρησιμοποιείται είτε από έναν κόμβο μόνο του, είτε από πολλούς με το MPI.

1.2.2 GPU

Οι κάρτες γραφικών αναπτύχθηκαν όπως είπαμε και στην εισαγωγή για την δημιουργία και επεξεργασία εικόνας σε συσκευές εξόδου. Σήμερα βρίσκονται σε πάρα πολλές συσκευές όπως κινητά τηλέφωνα, κονσόλες, αυτόνομα αυτοκίνητα. Οι δύο εταιρίες που είναι οι κύριοι προμηθευτές τους είναι η NVidia και η AMD. Αφού χρησιμοποιήσαμε κάρτα γραφικών της NVidia τόσο για την ανάπτυξη της υλοποίησης μας όσο και για τα benchmarks θα ασχοληθούμε κυρίως με κάρτες γραφικών αυτής.

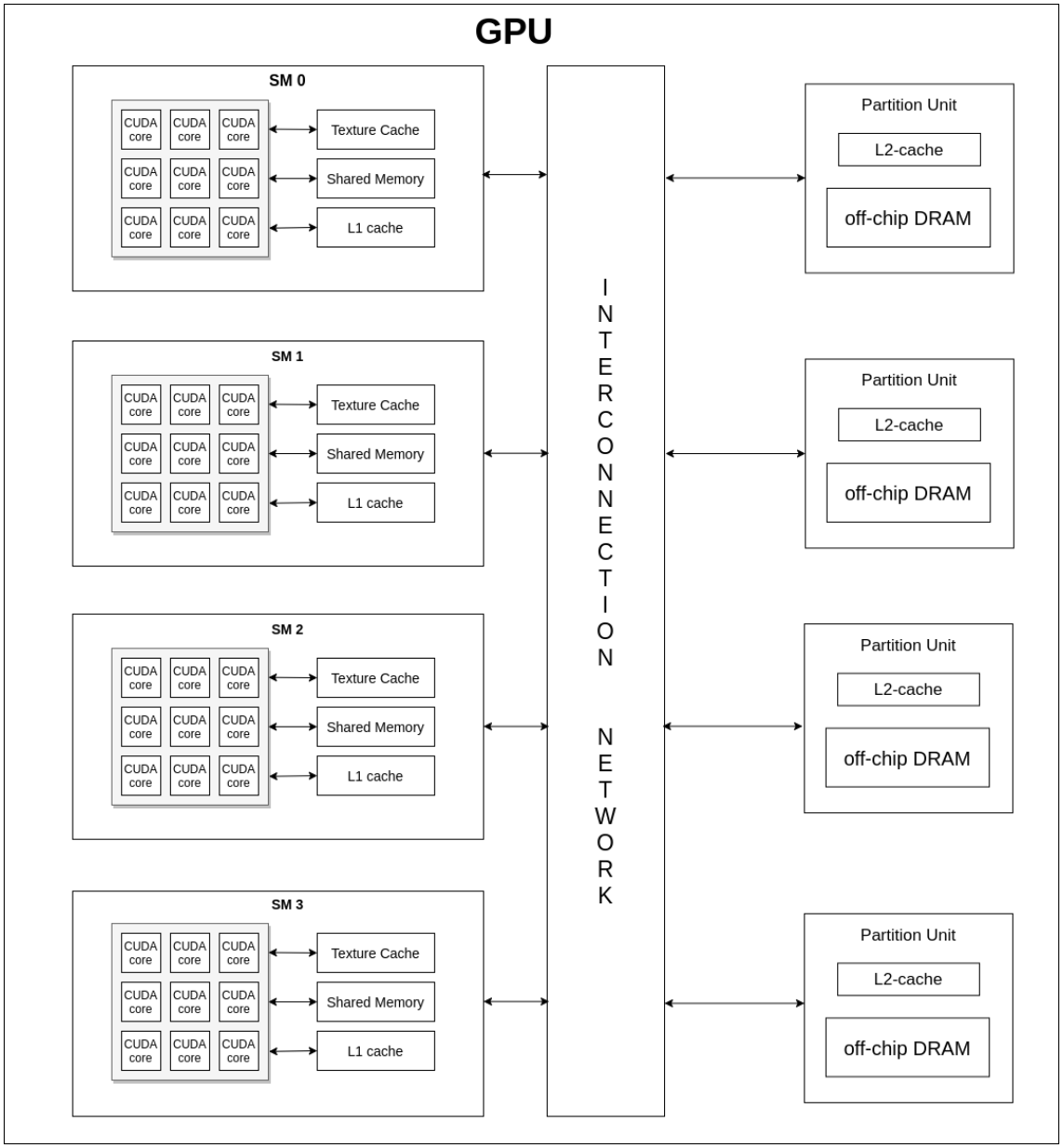
GPGPU Παρ'όλη την αρχική τους εφαρμογή, οι κάρτες γραφικών χρησιμοποιήθηκαν από επιστήμονες που θέλησαν να κάνουν μέσω αυτών πράξεις γραμμικής άλγεβρας. Από τότε με τον όρο GPGPU αναφερόμαστε στην χρήση των καρτών γραφικών για να επιταχύνουμε προγράμματα γενικού σκοπού. Ιδιαίτερη αναφορά αξίζει να γίνει στον τομέα της μηχανικής μάθησης. Φαίνεται πως οι κάρτες γραφικών είναι ιδιαίτερα σημαντικές εδώ, αφού είναι αρκετά πιο γρήγορες από τους επεξεργαστές στην εκπαίδευση των πολυεπίπεδων νευρωνικών δικτύων [10]. Αυτή τους η ικανότητα, σε συνδυασμό με την άνθιση της μηχανικής μάθησης στις μέρες μας, έχει ωθίσει τους προμηθευτές καρτών γραφικών να προσπαθούν παράγουν κάρτες γραφικών όλο και πιο ισχυρές.

Αρχιτεκτονική

Σε αυτήν την υποενότητα θα μιλήσουμε για την αρχιτεκτονική μίας κάρτας γραφικών [11]. Μία κάρτα γραφικών αποτελείται από πολλούς πηρύνες που ονομάζονται Streaming

Multiprocessors ή αλλιώς SMs. Στους SMs εκτελείται ένας kernel που είναι μια συνάρτηση γραμμένη για GPU. Κάθε SM έχει πολλά νήματα, που τρέχουν την ίδια εντολή του kernel, σαν ένα SIMD πρόγραμμα. Τα νήματα αυτά μπορούν ακόμα και να επικοινωνούν μεταξύ τους, με την χρήση κοινής μνήμης ή barriers.

Σε επίπεδο μνήμης, κάθε αυτόνομη κάρτα γραφικών έχει την δικιά της κύρια μνήμη που μπορεί να φτάνει μεγέθη όπως τα 32GB. Η κύρια μνήμη είναι χωρισμένη σε τμήματα και κάθε SM επικοινωνεί με αυτά με ένα δίκτυο διασύνδεσης. Έτσι πετυχαίνουμε μεγαλύτερο memory bandwidth. Επίσης υπάρχουν on-chip μνήμες στους SMs που λειτουργούν σαν caches, όπως είναι οι L1-cache, η shared memory και η texture cache. Τα παραπάνω μπορούν να φανούν στην 1.1.



Σχήμα 1.1: *Αρχιτεκτονική GPU*

Μοντέλο Εκτέλεσης

Τα νήματα που υπάρχουν στους SMs χωρίζονται σε μικρότερες ομάδες ανά 32 που ονομάζονται warps. Τα νήματα που ανήκουν στο ίδιο warp εκτελούν την ίδια εντολή κάθε φορά, με διαφορετικά δεδομένα φυσικά, από SIMD hardware. Ο διαμοιρασμός των νημάτων σε warps είναι στατικός και με τέτοιο τρόπο ώστε τα πρώτα 32 νήματα να ανήκουν στο πρώτο warp, τα επόμενα 32 στο επόμενο κλπ. Κάθε νήμα έχει μοναδικό id και με βάση αυτό προσδιορίζουμε τι πρέπει να κάνει. Σε κάθε SM υπάρχουν ακόμα ένας οι περισσότεροι warp schedulers που αναλαμβάνουν να τοποθετήσουν ένα warp προς εκτέλεση όταν αυτό είναι έτοιμο αφού έχει συλλέξει τα δεδομένα που χρειάζεται από την μνήμη.

Υπάρχει ωστόσο και η περίπτωση στο πρόγραμμα μας να υπάρχει control flow με αποτέλεσμα δύο νήματα του ίδιου warp να πρέπει να εκτελέσουν διαφορετικές εντολές. Στην περίπτωση αυτή τα νήματα που εκτελούν διαφορετικές εντολές το κάνουν σε ξεχωριστά βήματα, και έτσι χάνεται ένα μέρος της παραλληλίας.

Προγραμματισμός σε CUDA C

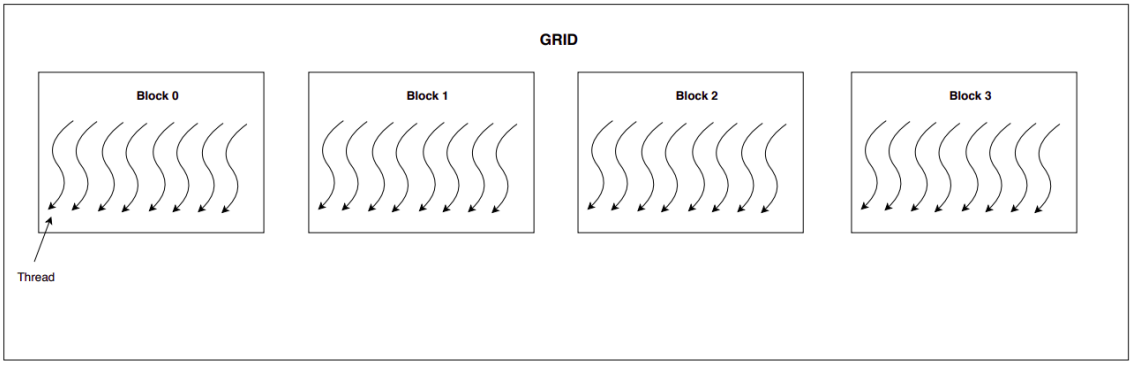
Η CUDA C είναι η γλώσσα που χρησιμοποιούμε για να περιγράψουμε στην GPU την συνάρτηση που θέλουμε να εκτελέσει. Θα προσπαθήσουμε να περιγράψουμε το πως γίνεται αυτό με ένα παράδειγμα. Θα χρησιμοποιήσουμε το παράδειγμα άθροισης δύο πινάκων A και B με το αποτέλεσμα να μπαίνει στον C.

Πριν προχωρήσουμε στην εκτέλεση της συνάρτησης θα πρέπει να έχουμε εξασφαλίσει χώρο στην κύρια μνήμη της κάρτας γραφικών για τους πίνακες μας A, B και C. Έπειτα θα πρέπει να μεταφέρουμε στην GPU τα περιεχόμενα των A και B. Αφού εκτελέσουμε την πράξη θα πρέπει να μεταφέρουμε τον πίνακα C στην κύρια μνήμη της CPU. Ας δούμε τώρα πως εκτελούμε την πρόσθεση.

Προκειμένου να εκτελέσουμε μία συνάρτηση της GPU καλούμε έναν αριθμό απο νήματα. Αυτά τα νήματα ακολουθούν μια ιεραρχία. Αρχικά υπάρχουν τα μπλοκ νημάτων. Ένα μπλοκ αποτελείται από έναν αριθμό νημάτων που σήμερα φτάνει μεχρι τα 1024. Τα νήματα που ανήκουν στο ίδιο μπλοκ εκτελούνται στο ίδιο SM και ομαδοποιούνται περαιτέρω σε warps. Όταν καλούμε έναν kernel προσδιορίζουμε πόσα μπλοκς θα δημιουργήσουμε και πόσα νήματα θα έχει το κάθε μπλοκ. Το κάθε νήμα όταν τρέχει μπορεί μέσω ειδικών μεταβλητών να γνωρίζει ποιο είναι το id μέσα στο μπλοκ αλλά και το id του μπλοκ στο οποίο ανήκει στο σύνολο των μπλοκς. Το σύνολο όλων των μπλοκς που καλούμε ονομάζεται και grid. Αυτό μπορεί να φανεί στην εικόνα 1.2. Συνεπώς για να καλέσουμε μία συνάρτηση GPU πρέπει να προσδιορίσουμε αυτές τις παραμέτρους. Η διαδικασία που περιγράφεται παραπάνω φαίνεται στο παρακάτω κομμάτι κώδικα.

```
int main(){
    //declare CPU arrays
    int A[10000];
    int B[10000];
    int C[10000];

    // Initialize CPU arrays
```

```
    for (int i = 0; i < 10000; i++){
        A[i] = rand();
        B[i] = rand();
    }

    // Declare GPU pointers and allocate
    // memory for them with cudaMalloc
    int *d_A;
    cudaMalloc(&d_A, 10000 * sizeof(int));
    int *d_B;
    cudaMalloc(&d_B, 10000 * sizeof(int));
    int *d_C;
    cudaMalloc(&d_C, 10000 * sizeof(int));

    // Copy arrays from CPU to GPU
    cudaMemcpy(d_A, A, 10000 * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, 10000 * sizeof(int), cudaMemcpyHostToDevice);

    // Invoke the kernel
    addVecs<<< (10000 + 512 - 1)/512 , 512>>>(A, B, C, 10000);

    // Copy the result back in the CPU
    cudaMemcpy(C, d_C, 10000 * sizeof(int), cudaMemcpyDeviceToHost);

    // Free GPU memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```



Σχήμα 1.2: *Πλέγμα με 4 μπλοκς και 8 νήματα ανά μπλοκ*

Τώρα απομένει να δούμε τι γίνεται μέσα στον kernel. Κάθε νήμα θα αναλάβει να προσθέσει κάποια στοιχεία του πίνακα με κάποια άλλα. Έστω ότι στο grid μας έχουμε 1024 νήματα και οι πίνακες που προσθέτουμε έχουν 2048 στοιχεία. Το πρώτο νήμα σε όλο το grid θα πρέπει αναλάβει να προσθέσει τα στοιχεία στις θέσεις 0 και 1024, το δεύτερο νήμα στις θέσεις 1 και 1025 και ούτω καθεξής. Αυτό φαίνεται στην συνάρτηση addVecs που παρατίθεται.

```
void addVecs(int *A, int *B, int *C, int size){
    //find the global index
    //of thread inside the grid
    int tid = threadIdx.x + blockIdx.x*blockDim.x;
    // Each thread will compute all the indexes
    // that are like total_threads * n + tid
    for (int i = tid; i < size; i+= blockDim.x*gridDim.x)
        C[i] = A[i] + B[i];
}
```

Στην γραμμή 4 υπολογίζεται το ολικό id του νήματος και στην γραμμή 8 φαίνεται η πράξη που κάνει για να υπολογίσει τις θέσεις που του αναλογούν.

Χρήση CUDA στην Python

Προκειμένου να καλέσουμε συναρτήσεις για κάρτες γραφικών στην Python χρησιμοποιούμε ένα module που ονομάζεται PyCUDA [12]. Αυτό μας βοηθάει αρκετά καθώς παρέχει τρόπο να έχουμε πίνακες στην GPU αλλά και για να καλούμε συναρτήσεις. Μπορεί να χρησιμοποιηθεί με δύο τρόπους. Ο πρώτος από τους δύο και πιο απλός είναι ο παρακάτω.

```
import pycuda.autoinit # initialize pycuda
from pycuda import gpuarray
import numpy as np

a = np.random.randint(1000, size=1000)
b = np.random.randint(1000, size=1000)
```

```
d_a = gpuarray.to_gpu(a)
d_b = gpuarray.to_gpu(b)

d_c = d_a + d_b
c = d_c.get()
```

Στην γραμμή 11 βλέποντας τα ορίσματα η PyCUDA δημιουργεί εκείνη την στιγμή τον kernel που χρειάζεται και τον κάνει compile. Επίσης αποφασίζει αυτή για τον αριθμό και το μέγεθος των μπλοκς. Υπάρχει ωστόσο και ο παρακάτω τρόπος, τον οποίο εμείς χρησιμοποιούμε για να μπορούμε να γράφουμε πιο περίπλοκους αλγορίθμους και να έχουμε καλύτερο έλεγχο.

```
from pycuda.autoinit
from pycuda.compiler import SourceModule
import numpy as np

mod = SourceModule("""
    __global__ void addVecs(int *a, int *b, int *c, int sz)
    {
        int tid = threadIdx.x + blockIdx.x*blockDim.x;
        for (int i = tid; i < sz; i += blockDim.x*gridDim.x)
            c[i] = a[i] + b[i];
    }
""")
addVecs = mod.get_function("addVecs")

a = np.random.randint(1000, size=1000)
b = np.random.randint(1000, size=1000)

d_a = gpuarray.to_gpu(a)
d_b = gpuarray.to_gpu(b)
d_c = gpuarray.empty_like(a)
addVecs(d_a, d_b , d_c, np.int32(d_a.size),
        grid=(20,1), block=(512,1,1))
c = d_c.get()
```

Εδώ ορίζουμε εμείς τον kernel στις γραμμές 5 με 12 και ορίζουμε εμείς το μέγεθος και τον αριθμό των μπλοκς στην γραμμή 22.

1.3 Υλοποίηση συναρτήσεων Cuda και Βελτιστοποιήσεις

Σε αυτό το κεφάλαιο θα περιγράψουμε πως υλοποιήσαμε κάποιους από τους βασικούς kernels που χρειαστήκαμε και κάποιες τι βελτιστοποιήσεις κάναμε. Ακόμα θα δείξουμε αποτελέσματα σχετικά με την σύγκριση επεξεργαστή και κάρτας γραφικών, καθώς και δύο διαφορετικών καρτών γραφικών. Τέλος, θα μιλήσουμε για μια βελτιστοποίηση που κάνουμε στο πλαίσιο μιας ολόκληρης προσομοίωσης και πως αυτή μας κερδίζει χρόνο.

1.3.1 Cuda Kernels

Σε αυτήν την υποενότητα θα περιγράψουμε την λογική με την οποία υλοποιήσαμε κάποιους kernels και θα δείξουμε αποτελέσματα μέσω εικόνων. Οι kernels αυτοί είναι οι εξής.

- histogram
- kick
- drift
- linear_interp_kick
- FFTs

Τους πρώτους τέσσερις τους υλοποιήσαμε εμείς, ενώ τους FFTs τους χρησιμοποιούμε από την βιβλιοθήκη CuFFT μέσω του Python module με όνομα scikit-cuda. Εκτός από αυτούς, υλοποιήσαμε και αρκετές άλλες συναρτήσεις για GPU αλλά επειδή καταναλώνουν σχεδόν ασήμαντο ποσοστό του χρόνου δεν αξίζει να αναφερθούμε ιδιαίτερα σε αυτές.

Για ότι υλοποιήσαμε παρακάτω, χρησιμοποιούμε την εξής τεχνική για να υπολογίσουμε τον αριθμό των μπλοκς και το μέγεθος τους. Γνωρίζουμε ότι κάθε SM μπορεί να πάρει αριθμό νημάτων μικρότερο ή ίσο από 2048, και το μέγιστο μέγεθος μπλοκ είναι 1024. Έτσι θέτουμε το μέγεθος του μπλοκ στο 1024, και τον αριθμό των μπλοκς στο διπλασίο του αριθμού των SMs, έτσι ώστε να μπορέσουμε να τα καλύψουμε όλα και να πετύχουμε 100% χρησιμοποίηση. Στην πράξη, η τεχνική αυτή έδειξε πολύ καλά αποτελέσματα, και για αυτό την υιοθετήσαμε.

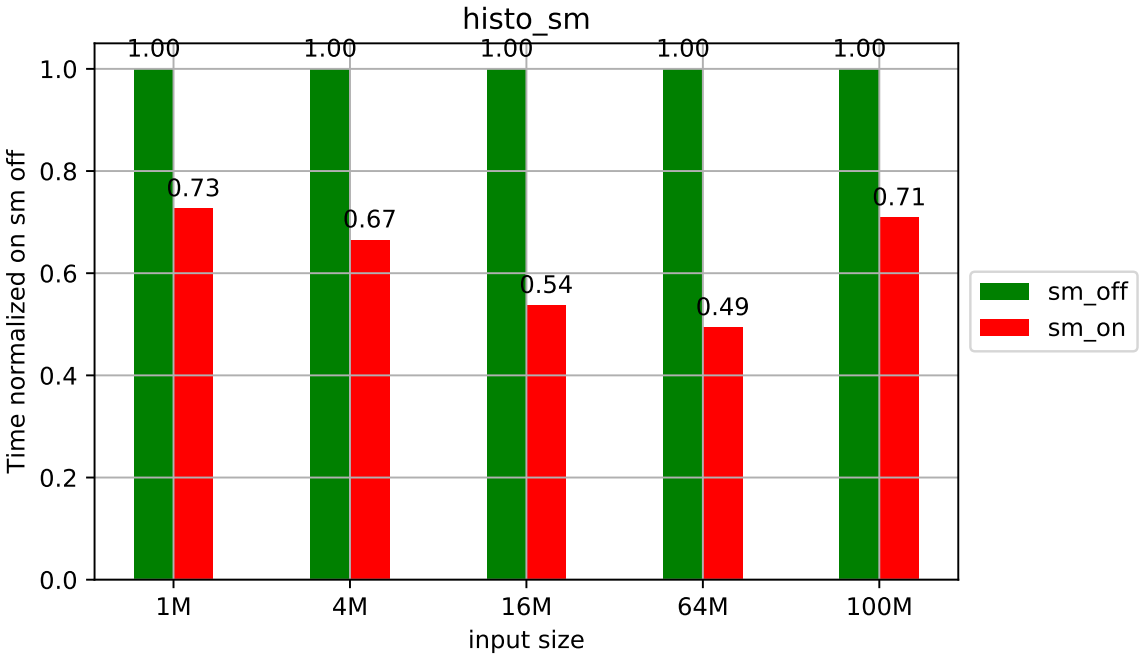
Προκειμένου να εξηγήσουμε την λογική που ακολουθούμε, είναι σημαντικό να εξηγήσουμε, πως σχεδόν όλες οι παραπάνω συναρτήσεις είναι κάποιο for-loop, άλλες απλό και άλλες εμφολευμένο. Τα for-loops κάνουν κάποιες πράξεις πάνω σε σωματίδια, τροποποιώντας πίνακες όπως ο dE. Για να τις υλοποιήσουμε στην GPU αναθέτουμε σε κάθε νήμα κάποια σωματίδια, με τέτοιον τρόπο ώστε διαδοχικά νήματα να αναλαμβάνουν διαδοχικά σωματίδια. Τώρα για κάθε kernel θα δούμε τα αποτελέσματα, και σε περίπτωση που σε κάποιον χρησιμοποιήθηκε κάποιο optimization θα το αναφέρουμε.

Τέλος, πριν περάσουμε στα αποτελέσματα, χρησιμοποιήσαμε και συγκρίναμε μεταξύ τους 3 διαφορετικές συσκευές. Αρχικά έναν επεξεργαστή Ιντελ Ξεον 26603, μία κάρτα γραφικών Νιδια κ40 και μία αρκετά πιο σύγχρονη κάρτα γραφικών Νιδια 100. Τα χαρακτηριστικά τους υπάρχουν στην ενότητα 6.

Histogram

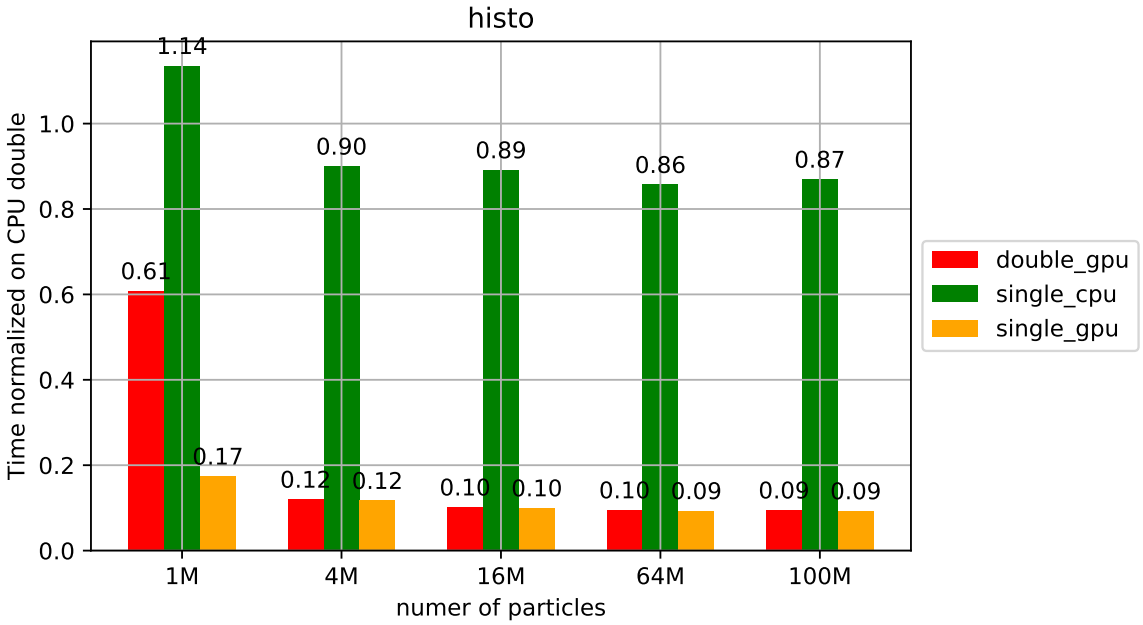
Η συνάρτηση του ιστογράμματος όπως προδίδει και το όνομα αναλαμβάνει να υπολογίσει το ιστόγραμμα του πίνακα dt. Για να το κάνει αυτό κάθε νήμα υπολογίζει το διάστημα στο οποίο ανήκει το σωματίδιο του, και αυξάνει την τιμή του κατά ένα. Στην απλή υλοποίηση γίνεται ακριβώς αυτό, με την διαφορά ότι η αύξηση πρέπει να γίνει με ατομική εντολή, αφού υπάρχει η περίπτωση δύο νήματα να θέλουν να αυξήσουν την τιμή του ίδιου διαστήματος.

Είναι εύκολο να καταλάβουμε πως το να υπάρχουν πολλά νήματα που θέλουν να αυξήσουν τιμές ατομικά, σε έναν όχι και τόσο μεγάλο πίνακα όπως αυτός του ιστογράμματος, μειώνει κατά πολύ την παραλληλία λόγω των ατομικών εντολών. Έτσι αναπτύξαμε μία δεύτερη έκδοση του histogram, στην οποία κάθε μπλοκ νημάτων υπολογίζει τοπικά το ιστόγραμμα των σωματιδίων που του αναλογούν και έπειτα το προσθέτει στο ολικό. Το τοπικό ιστόγραμμα υπάρχει σε μία ειδικού τύπου on-chip μνήμη που ονομάζεται shared-memory. Με αυτόν τον τρόπο πετυχαίνουμε το να μειώσουμε τον ανταγωνισμό των νημάτων, αφού πλέον θα έχουμε μόνο 1024 νήματα και όχι 30.720 (τόσα θα είχαμε σε μία GPU με 15 SMs). Επίσης αυτού του τύπου η μνήμη είναι πολύ πιο γρήγορη από την κύρια μνήμη, τόσο ώστε να συγκρίνεται σε ταχύτητα με αυτήν ενός καταχωρητή. Επειδή ωστόσο είναι περιορισμένη, και σε ορισμένες περιπτώσεις δεν μπορεί να χωρέσει ένα αντίγραφο του ιστογράμματος σε αυτήν, τροποιήσαμε την έκδοση μας ώστε όταν συμβαίνει αυτό, να διατηρεί τοπικά τις κεντρικές θέσεις του ιστογράμματος που συνήθως είναι πιο πυκνές, και για τα διαστήματα εκτός αυτών να κάνει πράξεις απευθείας στο τελικό ιστόγραμμα. Στις εικόνες 1.3, 1.4 και 1.5 μπορούμε να δούμε την σύγκριση μεταξύ των δύο εκδόσεων, αλλά και την σύγκριση μεταξύ των διαφορετικών συσκευών.



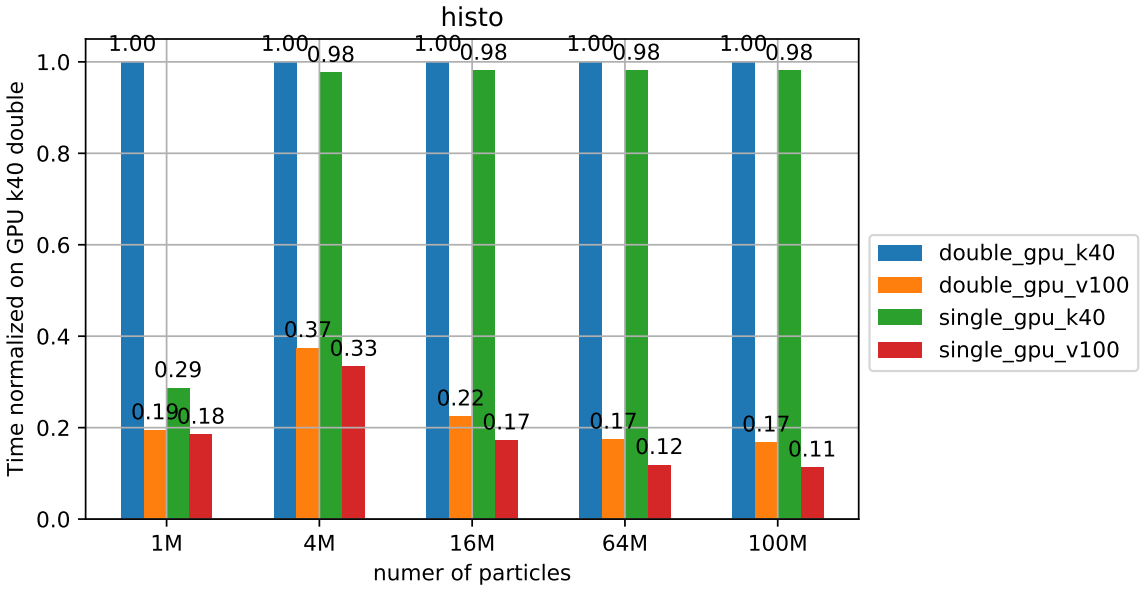
Σχήμα 1.3: *Σύγκριση εκδόσεων του ιστογράμματος*

Στην εικόνα 1.3 βλέπουμε πως η βελτιστοποιημένη έκδοσή μας φτάνει να είναι ακόμα και δυό φορές πιο γρήγορη από την απλή. Ωστόσο για μεγαλύτερα μεγέθη, που το ποσοστό του ιστογράμματος στην shared-memory γίνεται αρκετά μικρό, η διαφορά μικραίνει αισθητά.



Σχήμα 1.4: *Σύγκριση μεταξύ επεργαστή και κάρτας γραφικών στο ιστόγραμμα*

Στην εικόνα 1.4 παρατηρούμε πως η GPU μας είναι 10 φορές πιο γρήγορη από την CPU. Το άλλο που παρατηρούμε είναι πως το να χρησιμοποιούμε 32 bits για να περιγράψουμε τον dt πίνακα δεν μας επιφέρει κέρδος στον χρόνο. Αυτό ισχύει τόσο για την GPU αλλά όπως φαίνεται και για την CPU.

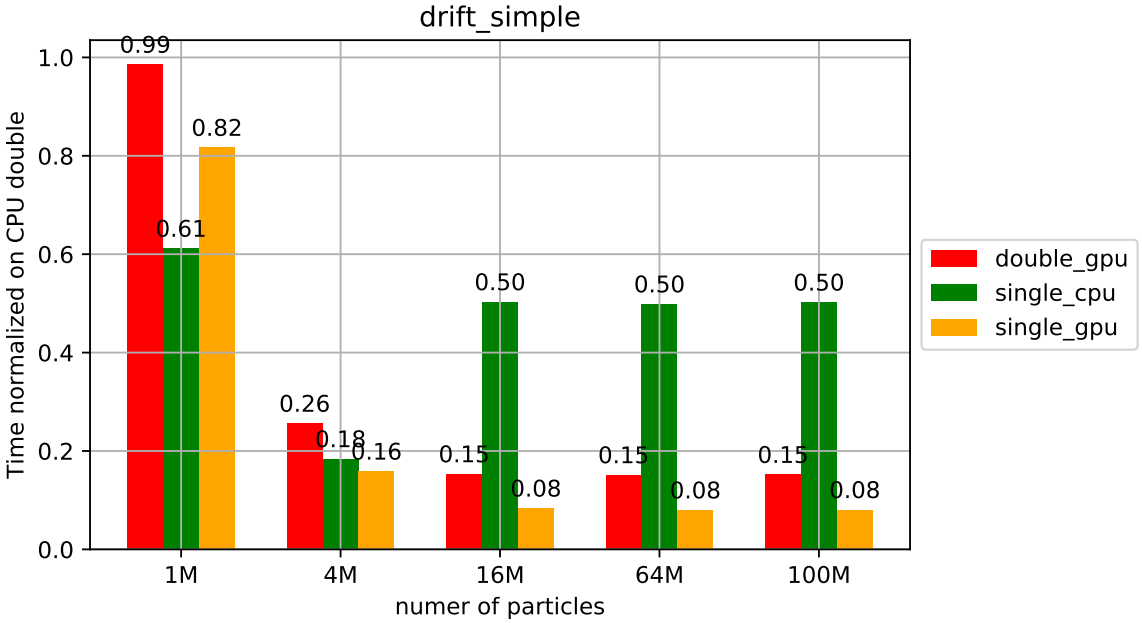


Σχήμα 1.5: *Σύγκριση μεταξύ διαφορετικών καρτών γραφικών για το ιστόγραμμα*

Στην εικόνα που φαίνεται παραπάνω μπορούμε να δούμε την σύγκριση μεταξύ της K40 και της v100 και να παρατηρήσουμε πως κατά μέσο όρο η v100 είναι 5 φορές πιο γρήγορη. Ακόμα και σε αυτήν την κάρτα η μετατροπή από τα 64 στα 32 bits δεν επιφέρει πολύ μεγάλη μείωση του χρόνου, ωστόσο μεγαλύτερη από ότι πριν.

Drift

Η συνάρτηση drift έχει 5 παραλλαγές, που όλες είναι αρκετά παρόμοιες μεταξύ τους. Γίνονται οι ίδιες προσβάσεις στην μνήμη αλλά αλλάζουν οι πράξεις που γίνονται στην κάθε έκδοση. Επειδή η συμπεριφορά είναι σε γενικές γραμμή ίδια, θα δείξουμε γραφικές παραστάσεις μόνο από την πρώτη έκδοση αυτή με τον solver να παίρνει την τιμή simple. Παρακάτω φαίνονται τα αποτελέσματα.



Σχήμα 1.6: *Σύγκριση μεταξύ επεργαστή και κάρτας γραφικών στο drift*

Στην εικόνα 1.6 παρατηρούμε πως η GPU μας είναι 6 φορές πιο γρήγορη από την CPU. Το άλλο που παρατηρούμε είναι πως το να χρησιμοποιούμε 32 bits μας επιφέρει κέρδος στον χρόνο αφού αυτός μειώνεται στο μισό, πράγμα που βλέπουμε τόσο στην GPU όσο και στην CPU. Βλέποντας την σύγκριση μεταξύ της K40 και της v100 στην εικόνα 1.7, παρατηρούμε πως και στο drift η v100 είναι περίπου 5 φορές πιο γρήγορη, τόσο στα 64 bits όσο και στα 32.

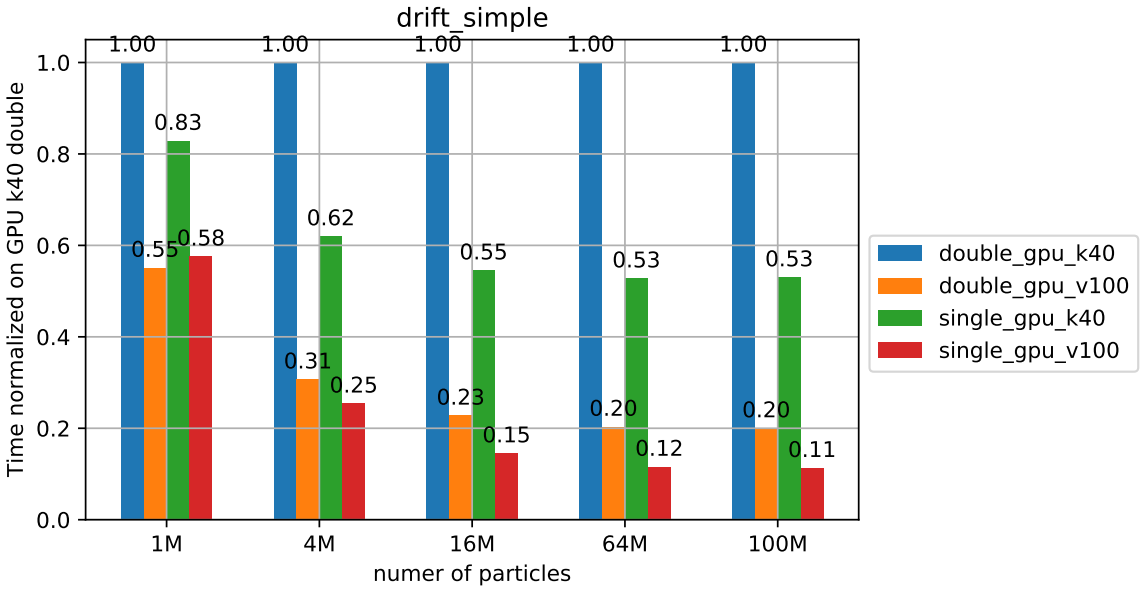
Kick

Η συνάρτηση kick είναι σχετικά απλή, έχει ένα for-loop στο οποίο γίνονται αλλαγές στον πίνακα dE. Στις εικόνες 1.8 και 1.9 φαίνονται τα αποτελέσματα.

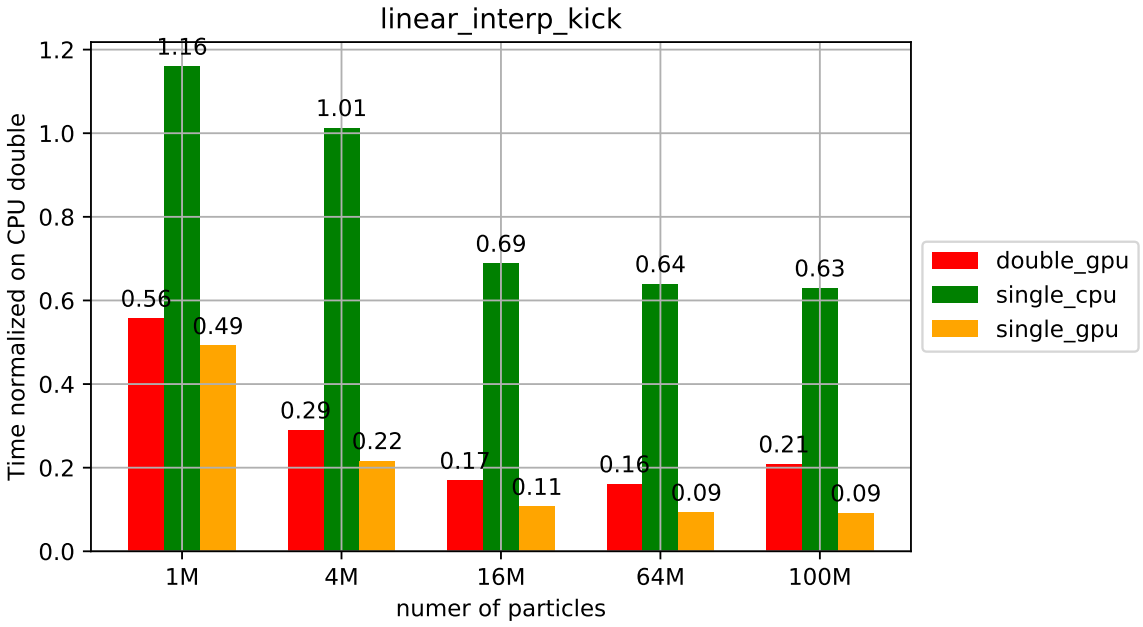
Στην εικόνα 1.8 παρατηρούμε πως η GPU μας είναι έως και 20 φορές πιο γρήγορη από την CPU. Όπως και στο drift παρατηρούμε μείωση του χρόνου στο μισό όταν χρησιμοποιούμε 32 bits. Στην εικόνα 1.9, παρατηρούμε πως στο kick η v100 είναι περίπου 3 φορές πιο γρήγορη, στα 64 bits αλλά στα 32 συγκλίνει στις 5 φορές, όσο ανεβαίνουμε μέγεθος.

Linear Interpolation Kick

Η συνάρτηση linear_interp_kick είναι παρόμοια με την kick, ωστόσο περιλαμβάνει κάποιες πράξεις στην αρχή που γίνονται για τον υπολογισμό δύο άλλων πινάκων που χρειάζονται να μπουν σε άλλο kernel.

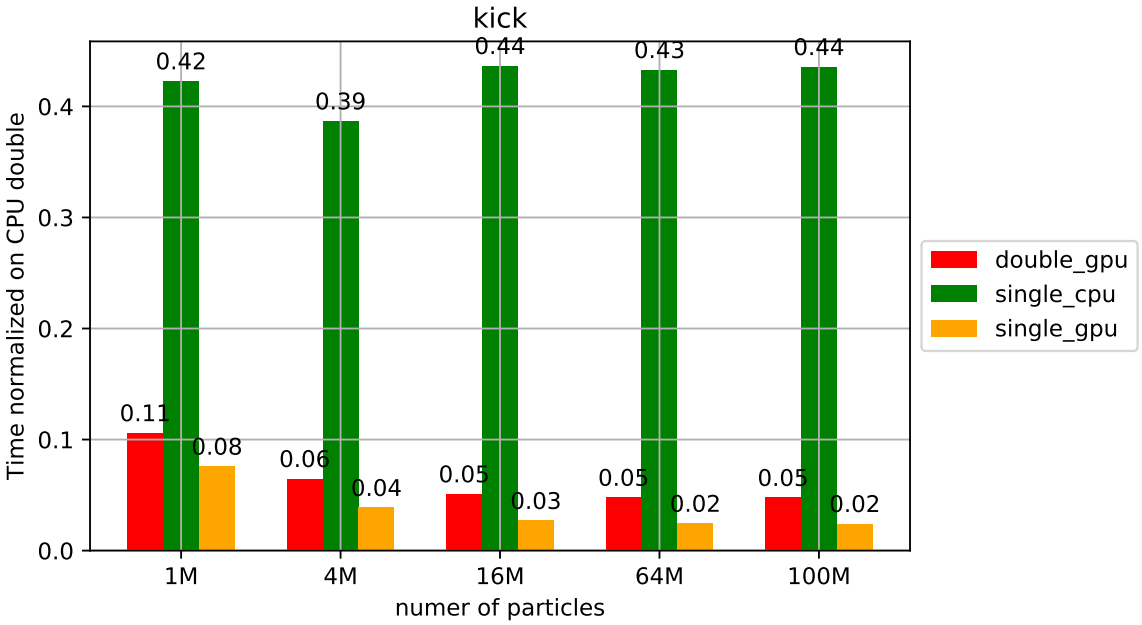


Σχήμα 1.7: *Σύγκριση μεταξύ διαφορετικών καρτών γραφικών για το drift*

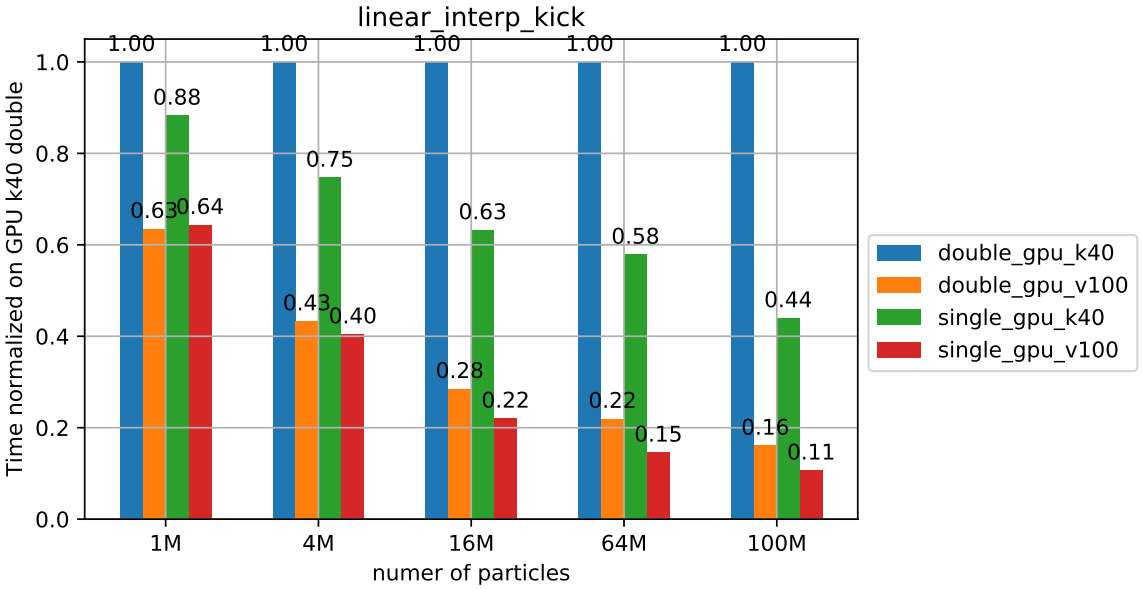


Σχήμα 1.10: *Σύγκριση μεταξύ επεργαστή και κάρτας γραφικών στο linear_interp_kick*

Στην εικόνα 1.10 παρατηρούμε πως η GPU μας είναι έως και 20 φορές πιο γρήγορη από την CPU. Ακόμα και σε αυτή την συνάρτηση παρατηρούμε μείωση στην μέση όταν πηγαίνουμε στα 32 bits.



Σχήμα 1.8: *Σύγκριση μεταξύ επεργαστή και κάρτας γραφικών στο kick*

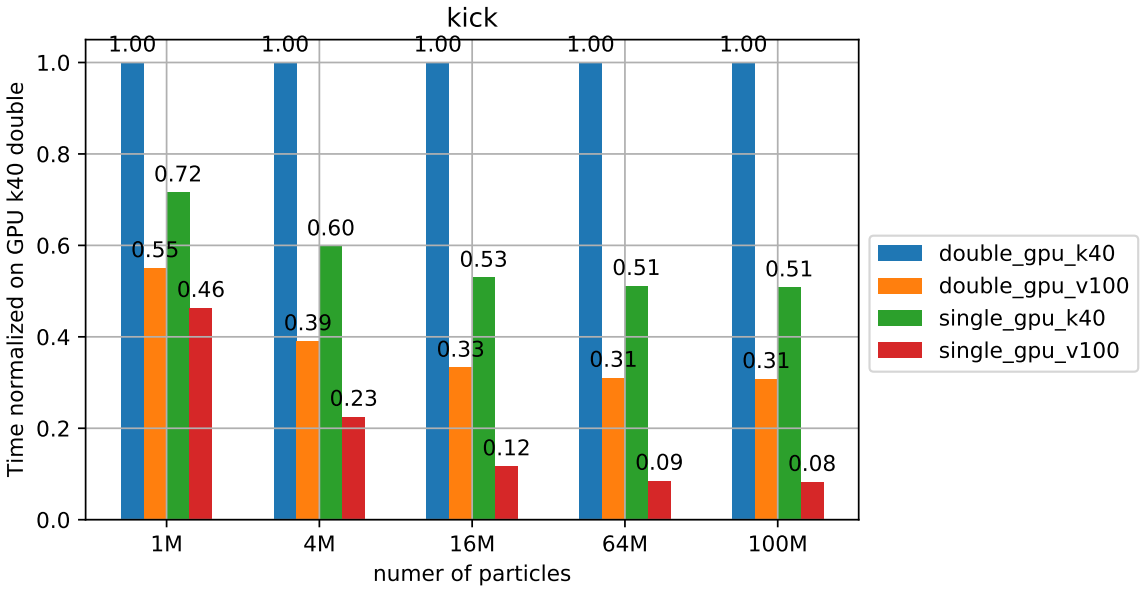


Σχήμα 1.11: *Σύγκριση μεταξύ διαφορετικών καρτών γραφικών για το linear_interp_kick*

Στην εικόνα 1.11, παρατηρούμε πως στο linear_interp_kick η v100 γίνεται πιο γρήγορη σε σχέση με την K40 όσο αυξάνουμε το μέγεθος του πειράματος και συγκλίνει στο 6. Παρατηρούμε ακόμα πως η βελτίωση με την μείωση της ακρίβειας είναι σημαντικότερη στην K40 από ότι στην v100, καθώς η πρώτη κερδίζει περισσότερο από τον μισό χρόνο ενώ η δεύτερη περίπου το ένα τρίτο του χρόνου, όσο μεγαλώνει το μέγεθος.

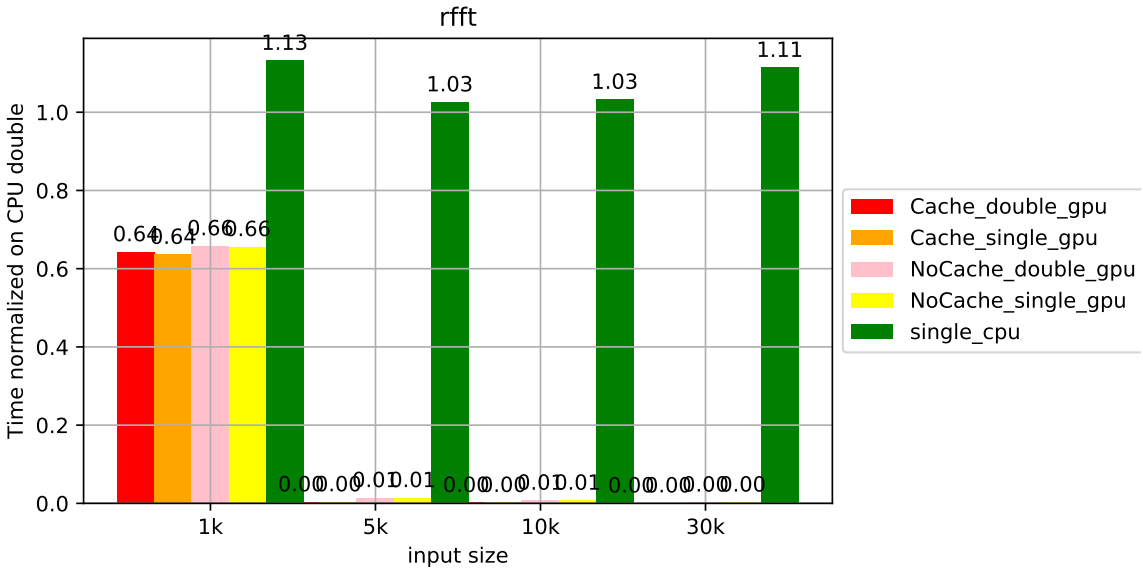
FFTs

Στους μετασχηματισμούς Fourier περιλαμβάνονται 2 συναρτήσεις, ο rrft και ο irfft, τις οποίες όπως αναφέραμε και πριν, παίρνουμε έτοιμες από την βιβλιοθήκη CuFFT. Τις χρησιμοποιούμε κατάλληλα ώστε η συμπεριφορά των δικών μας συναρτήσεων να είναι ίδια με

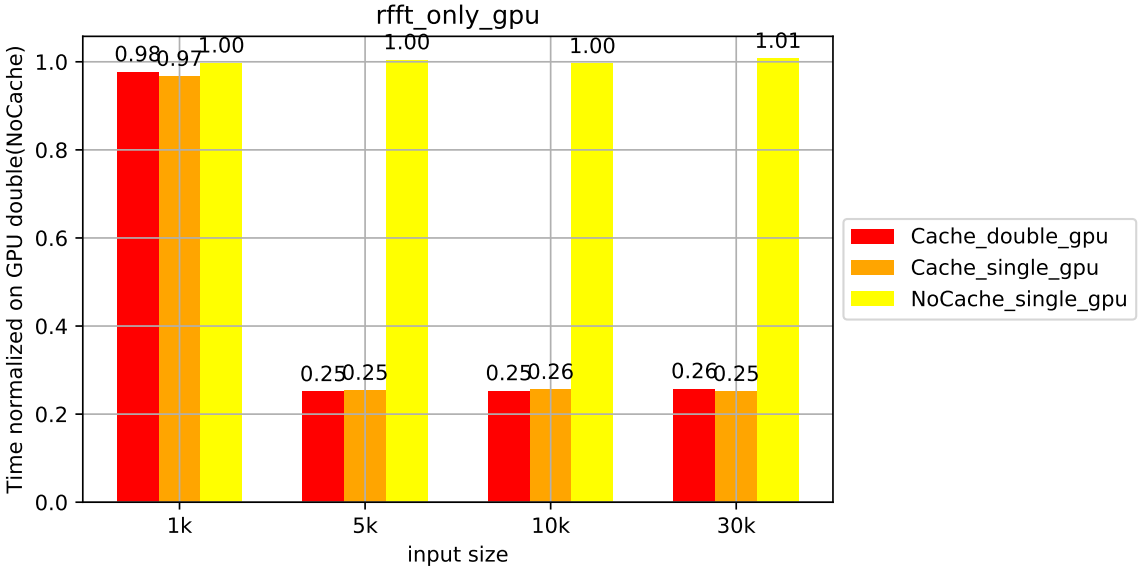


Σχήμα 1.9: *Σύγκριση μεταξύ διαφορετικών καρτών γραφικών για το kick*

αυτήν των αντίστοιχων συναρτήσεων του πακέτου numpy. Σε αυτές τις συναρτήσεις χρησιμοποιείται ένα optimization που θα δούμε αναλυτικά παρακάτω η GPU_cache. Στα αποτελέσματα που θα δούμε παρακάτω, φαίνεται το πόσο βελτιώνει αυτό την επίδοση των συναρτήσεων αυτών.

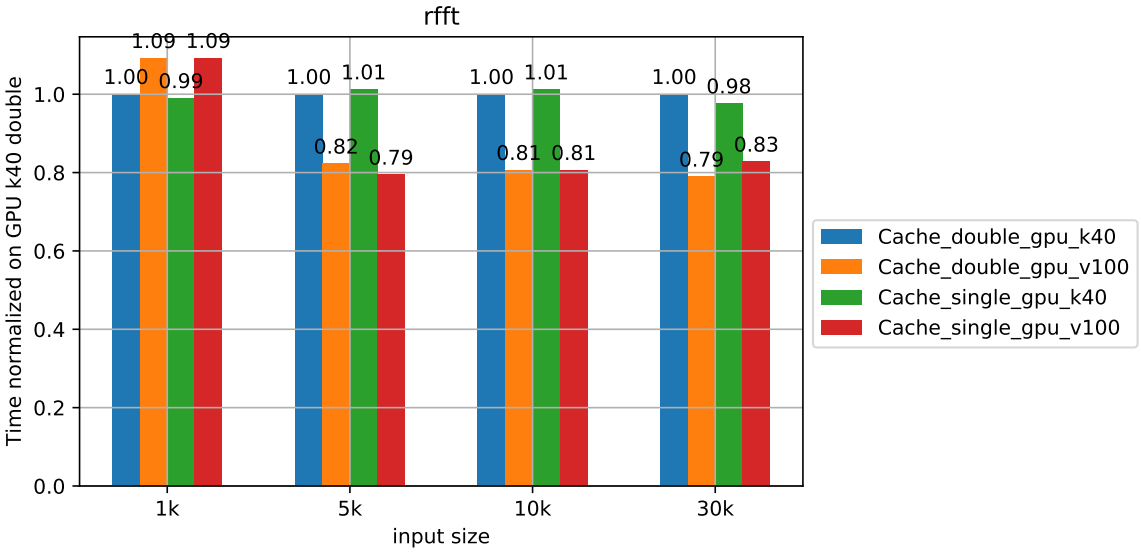


Σχήμα 1.12: *Σύγκριση μεταξύ επεργαστή και κάρτας γραφικών στο rfft*



Σχήμα 1.13: *Σύγκριση μεταξύ διαφορετικών εκδόσεων του rfft σε κάρτα γραφικών*

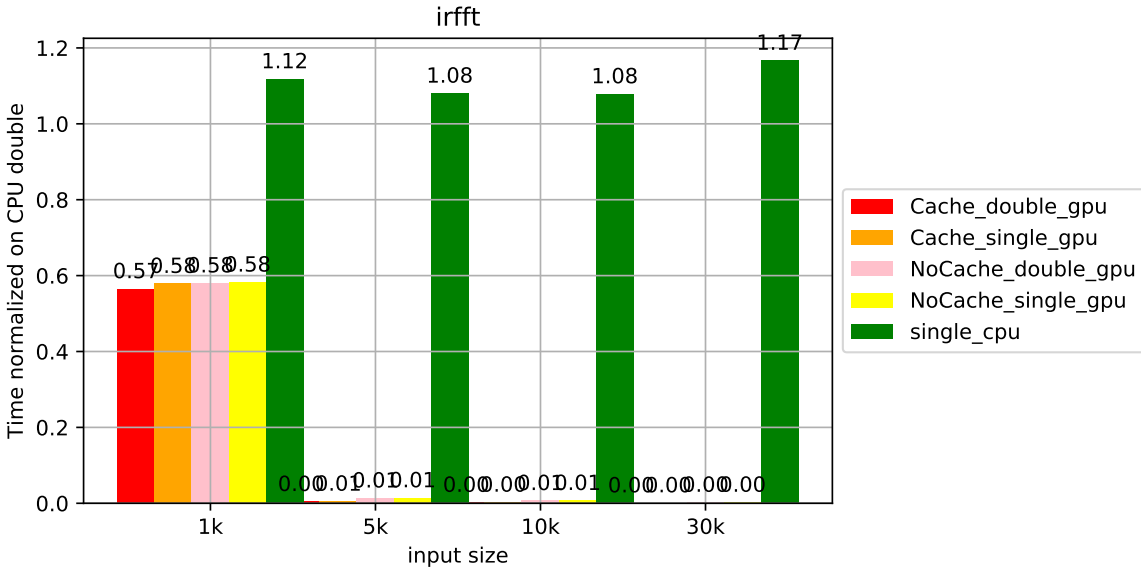
Στην εικόνα 1.12 παρατηρούμε πως η GPU μας δεν μπορεί να συγκριθεί με την CPU αφού φαίνεται να είναι πάνω από 100 φορές ταχύτερη. Για αυτό έχουμε ξεχωριστή γραφική παράσταση στην εικόνα 1.13 που φαίνεται η διαφορά μεταξύ των εκδόσεων. Παρατηρούμε ότι η διαφορά στα bits δεν παίζει κανέναν απολύτως ρόλο, ενώ η χρήση της GPU_cache ρίχνει τον χρόνο στο ένα τέταρτο του αρχικού.



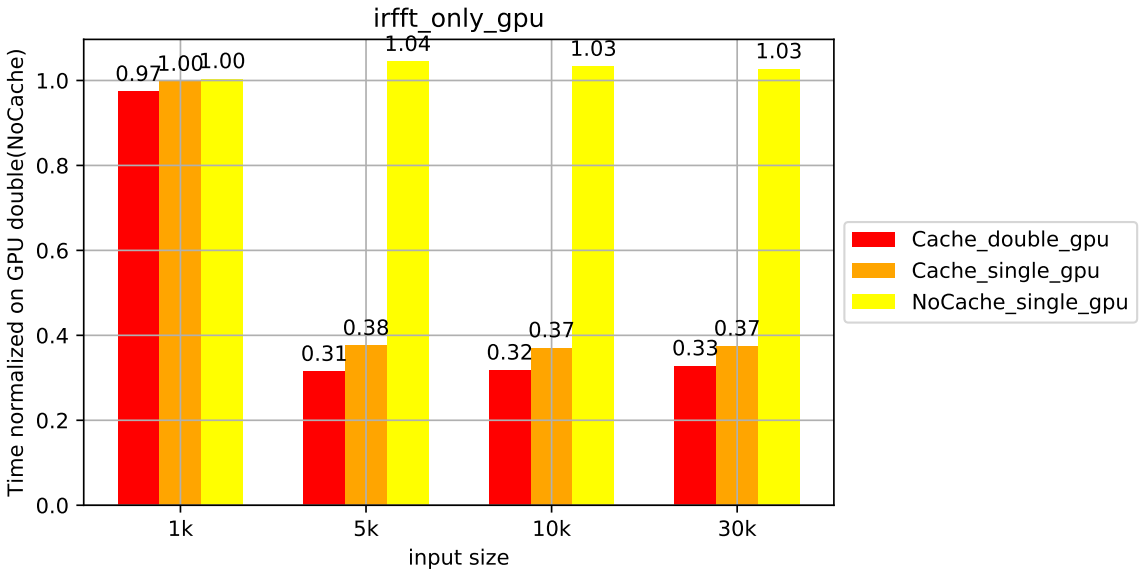
Σχήμα 1.14: *Σύγκριση μεταξύ διαφορετικών καρτών γραφικών για το rfft*

Στην εικόνα 1.14 η σύγκριση των δύο καρτών γραφικών μας δείχνει πως η v100 δεν βελτιώνει κατά πολύ την επίδοση, καθώς ο χρόνος μειώνεται κατά μόλις 20%.

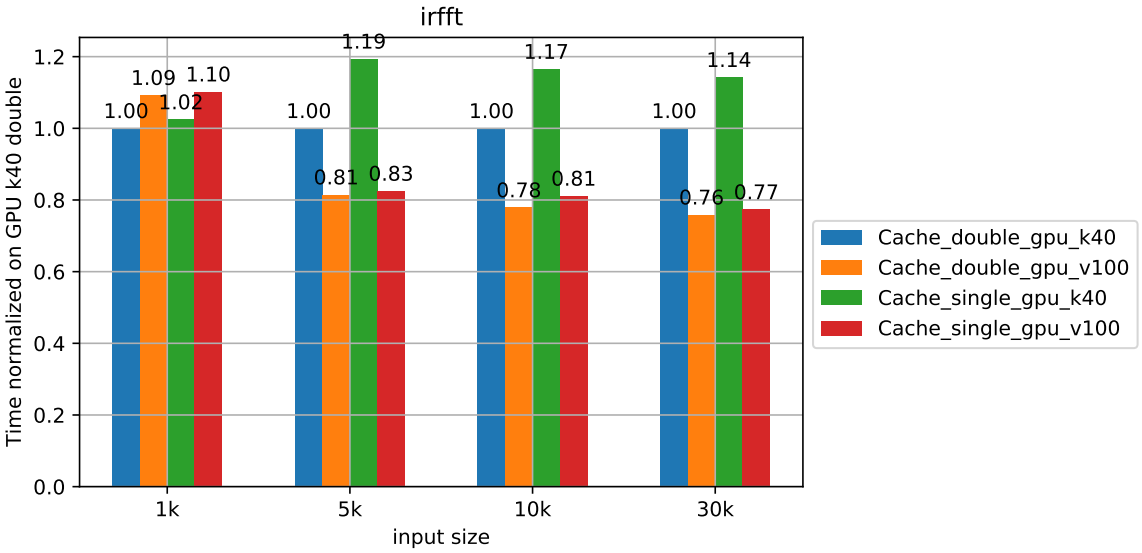
Τα ίδια ακριβώς αποτελέσματα παρατηρούνται και στην συνάρτηση irfft όπως βλέπουμε εικόνες 1.15, 1.16 και 1.17.



Σχήμα 1.15: *Σύγκριση μεταξύ επεργαστή και κάρτας γραφικών στο irfft*



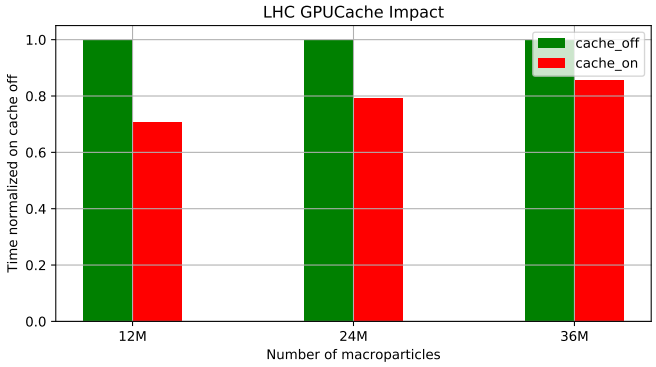
Σχήμα 1.16: *Σύγκριση μεταξύ διαφορετικών εκδόσεων του irfft σε κάρτα γραφικών*



Σχήμα 1.17: *Σύγκριση μεταξύ διαφορετικών καρτών γραφικών για το irfft*

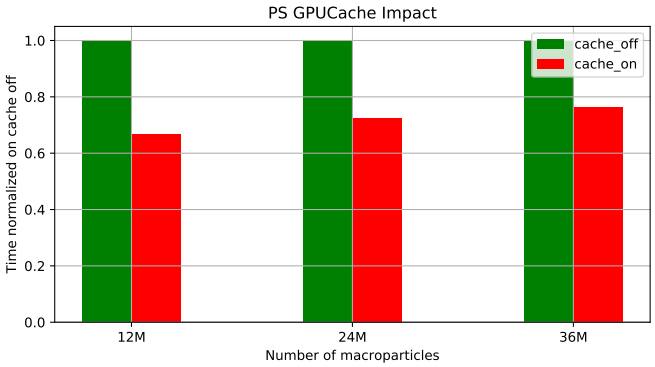
1.3.2 GPU_cache

Η GPU_cache είναι μία βελτιστοποίηση που σκεφτήκαμε προκειμένου να γλυτώσουμε χρόνο από την δημιουργία και την διαγραφή πινάκων στην μνήμη της κάρτας γραφικών. Αυτό που παρατηρείται στο BLonD είναι πως σε κάποια σημεία και κυρίως στους FFTs δημιουργούνται κάποιοι προσωρινοί πίνακες που δεν χρειάζονται στην συνέχεια της εκτέλεσης και έτσι αφού δεν υπάρχει αναφορά σε αυτούς διαγράφονται. Προκειμένου να μην γίνεται αυτό σε κάθε γύρο χωρίς να χρειάζεται, αποθηκεύουμε αυτούς τους πίνακες και τους επαναχρησιμοποιούμε. Στις 1.18, 1.19 και 1.20 φαίνεται το αποτέλεσμα που έχει η χρήση αυτού σε 3 από τα benchmarks που έχουμε.

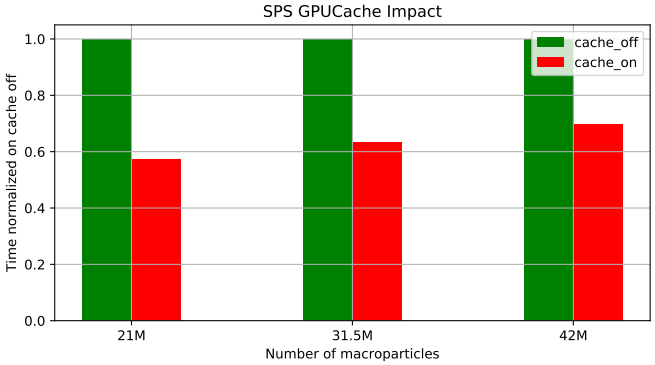


Σχήμα 1.18: *Επίδραση της GPU_cache στο LHC*

Στις παραπάνω είκονες βλέπουμε ότι η GPU_cache, ιδιαίτερα στα μικρά πειράματα μπορεί να μειώσει τον χρόνο του πειράματος ακόμα και κατά 40%. Ωστόσο, όσο μεγαλώνει το μέγεθος και οι συναρτήσεις drift και kick κυριαρχούν στο χρόνο εκτέλεσης, μειώνεται η συνεισφορά του χαρακτηριστικού αυτού.



Σχήμα 1.19: *Επίδραση της GPU_cache στο PS*



Σχήμα 1.20: *Επίδραση της GPU_cache στο SPS*

1.4 Χρήση της GPU στα πειράματα

Η GPU στα πειράματα θα χρησιμοποιηθεί προφανώς για να επιταχύνει τις συναρτήσεις που καταναλώνουν το μεγαλύτερο ποσοστό του χρόνου. Ωστόσο, προκειμένου να αποφύγουμε τις μεταφορές από την CPU προς την GPU μεταφέραμε και αρκετά άλλα κομμάτια του κώδικα στην GPU. Αυτό έχει ως αποτέλεσμα σε κάποιες προσομοιώσεις να μην υπάρχει καμία μεταφορά δεδομένων μεταξύ των δύο. Προκειμένου να κάνουμε χρήση της GPU σε κάποια προσομοίωση χρειάζεται να τρέξουμε κάποιες εντολές πριν την αρχή του κυρίως βρόχου. Η πρώτη εντολή που πρέπει να εκτελεστεί είναι η παρακάτω.

```
import blond.utils.bmath as bm
bm.use_gpu(gpu_id)
```

Η συνάρτηση use_gpu στο αρχείο blond.utils.bmath.py αρχικοποιεί την PyCUDA για την GPU με το συγκεκριμένο id στο μηχάνημα μας. Εκτός αυτού αντικαθιστά κάποιες συναρτήσεις που τρέχουν σε CPU, με τις αντίστοιχες τους σε GPU, μέσω ενός dictionary. Αυτές οι συναρτήσεις είναι Python συναρτήσεις που καλούν kernels της PyCUDA, και κάποια παραδείγματα αυτών είναι τα kick, drift, linear_interp_kick και τα FFTs.

Αφού γίνει αυτό, χρειάζεται να αλλάξουμε και κάποιες μεθόδους στα αντικείμενα της προσομοίωσης. Για τον σκοπό αυτό, για κάθε κλάση που εμπλέκεται στο κύριο βρόχο, έχουμε δημιουργήσει μία αντίστοιχη με διαφορετικές μεθόδους όπου απαιτείται. Έτσι μετά τις παραπάνω εντολές, χρειάζεται να τρέξουμε την μέθοδο use_gpu και για τα αντικείμενα που εμπλέκονται στον κύριο βρόχο. Συνεπώς με τις εντολές αυτές μόνο, μπορεί κάποιος

να χρησιμοποιήσει την GPU για τα πειράματα του. Ένα τυπικό παράδειγμα παρουσιάζεται παρακάτω.

```
import blond.utils.bmath as bm
bm.use_gpu()
beam.use_gpu()
tracker.use_gpu()
rf_station.use_gpu()
profile.use_gpu()
```

Τελικώς, είναι πολύ σημαντικό να αναφερθούμε στο τι συμβαίνει με τους πίνακες της προσομοίωσης. Όπως είδαμε κάποια αντικείμενα έχουν δικούς τους πίνακες. Παραδείγματα αυτών είναι η ακτίνα με τα dE και dt, το profile με τα n_macroparticles δηλαδή το ιστόγραμμα και το bin_centers. Αυτοί οι πίνακες πρέπει να υπάρχουν στην GPU για να μπορεί να τρέξειτο πείραμα. Ωστόσο, σε κάποιες περιπτώσεις οι φυσικοί θέλουν να διαβάζουν δεδομένα από αυτούς, και σε άλλες μάλιστα θέλουν να τους αλλάζουν. Για να μπορεί να γίνει αυτό, έχουμε δημιουργήσει μία κλάση με όνομα CGA που κάνει ακριβώς αυτό το πράγμα. Παρέχει ένα αντικείμενο με δύο πίνακες, έναν στην CPU και έναν στην GPU που είναι συνδεδεμένοι μεταξύ τους. Όταν ο ένας από τους δύο πίνακες, αλλάξει, τότε ο άλλος θεωρείται λάθος, και έτσι όταν ζητηθεί, θα χρειάζεται ενημέρωση από τον πρώτο. Ένα παράδειγμα για να γίνει πιο σαφές είναι ο κώδικας που φαίνεται παρακάτω.

```
for i in range(iterations):
    profile.track()
    tracker.track()
    profile.bin_centers += 0.25
```

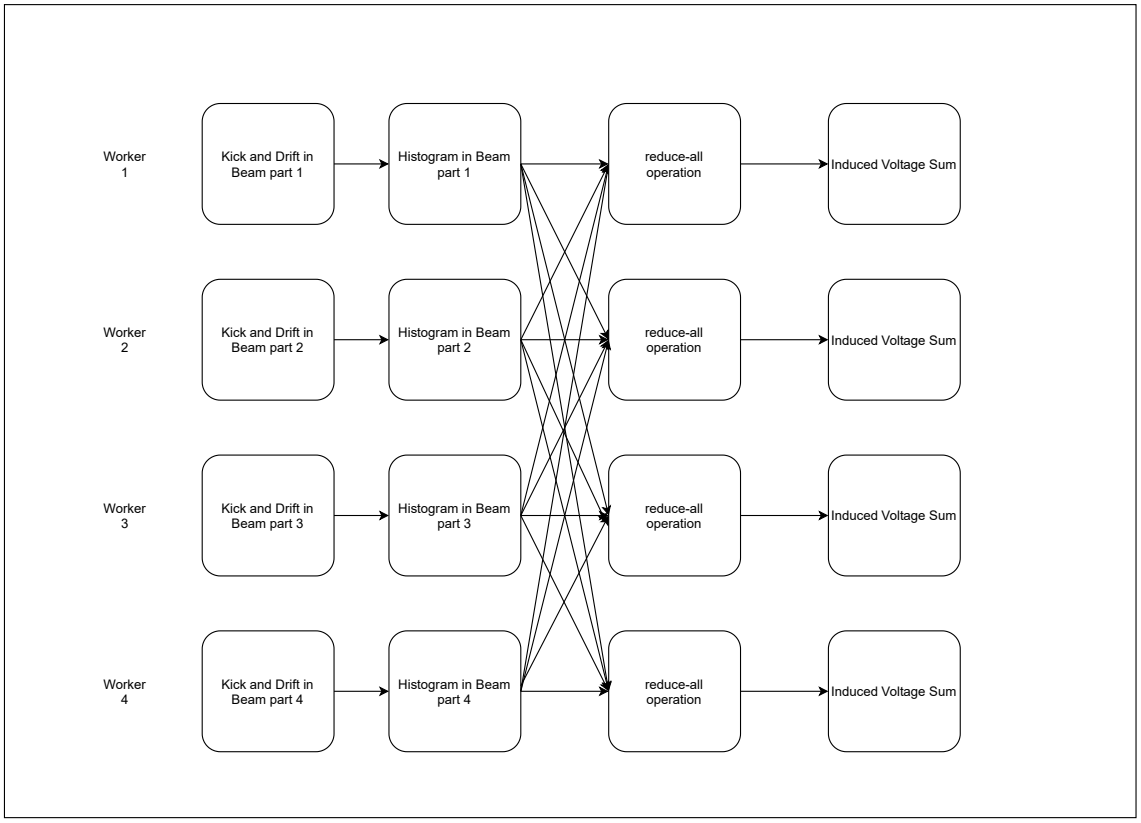
Στην γραμμή 4 ο πίνακας bin_centers που υπάρχει και στην GPU θα αλλάξει. Για να γίνει αυτό, αρχικά ο πίνακας της CPU θα ενημερωθεί, καθώς αυτός ο πίνακας αλλάζει μέσα στις GPU μεθόδους. Τέλος, αφού αλλάζει αυτός της CPU θα θεωρηθεί λάθος αυτός της GPU και την επόμενη φορά που θα χρειαστεί θα θέλει ενημέρωση.

Το συμπέρασμα που μπορεί να προκύψει είναι πως αυτή η κλάση είναι πολύ χρήσιμη για την προσομοίωση γιατι αρχικά κρύβει από τους φυσικούς που αναπτύσσουν μια προσομοίωση το πρόβλημα του συγχρονισμού μεταξύ των δύο συσκευών, και έπειτα εξασφαλίζει πως μέθοδοι που τρέχουν στην CPU και δεν θα έχουν υλοποιηθεί θα δουλεύουν σωστά.

1.5 Αξιολόγηση Πειραμάτων

Σε αυτήν την ενότητα θα παραθέσουμε αποτελέσματα από την εκτέλεση τριών πειραμάτων. Τα ονόματα αυτών είναι LHC, PS και SPS και τα αφήνουμε να πραγματοποιήσουν δέκα χιλιάδες γύρους. Το κάθε ένα έχει κάποιες ιδιαιτερότητες αλλά όλα εκτελούν τις βασικές πράξεις δηλαδή το histogram, linear_interpolation_kick και drift καθώς και τους FFTs.

Για τα πειράματα χρησιμοποιήσαμε τον ελληνικό υπερυπολογιστή ARIS, οι κόμβοι του οποίου διαθέτουν δύο GPUs NVidia K40 και 2 CPUs Intel Xeon E5-2660v3, καθώς και 64 GB RAM.



Σχήμα 1.21: *Παράδειγα ενός γύρου με το MPI*

1.5.1 MPI

Πριν δείξουμε τα αποτελέσματα είναι χρήσιμο να εξηγήσουμε κάποιες έννοιες, ώστε να είναι περισσότερο κατανοητά τα παρακάτω. Στην υλοποίηση με το MPI μοιράζουμε τους πίνακες της ακτίνας σε κόμβους και workers. Αυτοί εκτελούν τις πράξεις τους κανονικά, σαν να ήταν μόνο αυτή η ακτίνα, και έτσι ο πίνακας του ιστογράμματος που έχει ο καθένας είναι ελλειπής. Έτσι λοιπόν, μετά τον υπολογισμό του ιστογράμματος, επικοινωνούν και αθροίζουν τα ιστογράμματα τους, ώστε να να έχουν το τελικό ιστόγραμμα. Αυτό μπορεί να φανεί στην εικόνα 1.21.

Με βάση το παραπάνω κατηγοριοποιούμε τον χρόνο στις εξής τρεις ομάδες.

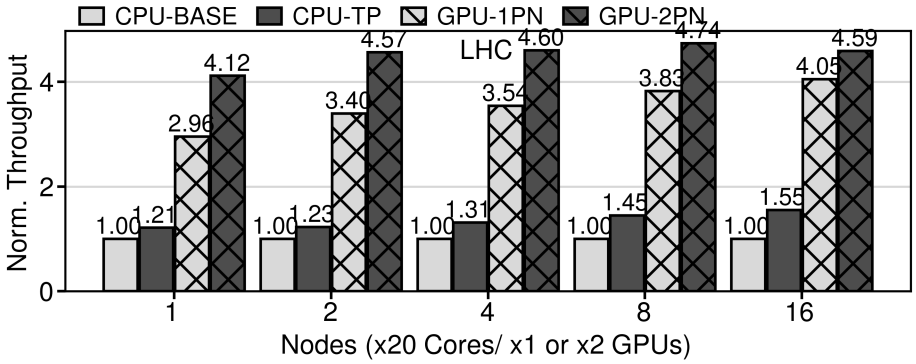
- comp ο χρόνος που εξαρτάται από τον αριθμό σωματιδίων (kick, drift, ιστόγραμμα κλπ.
- comm ο χρόνος που καταναλώνεται λόγω της επικοινωνίας
- serial ο χρόνος που είναι ανεξάρτητος από τον αριθμό σωματιδιών FFTs κλπ

Έτσι, με την αύξηση των workers μειώνουμε τον αριθμό των σωματιδίων που αναλαμβάνει ο καθένας και έτσι μειώνουμε μόνο τον comp χρόνο. Ωστόσο λόγω των περισσότερων μηνυμάτων αυξάνουμε την επικοινωνία.

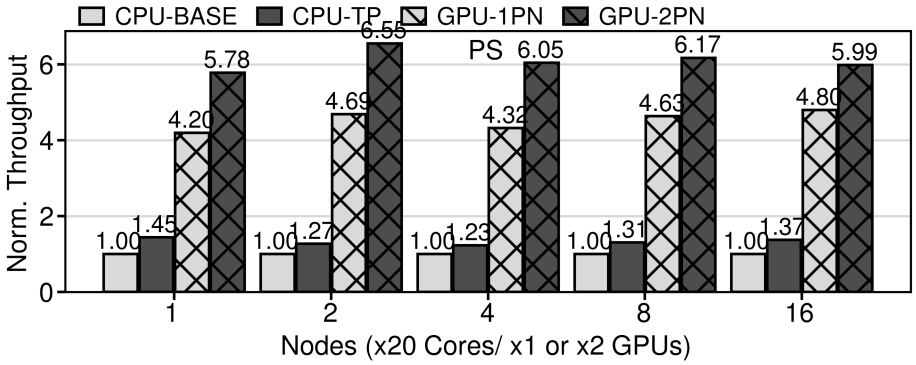
1.5.2 Σύγκριση CPU με GPU

Αρχικά θα δείξουμε πως τα πάει η GPU σε σχέση με την CPU. Για την σύγκριση έχουμε χρησιμοποιήσει 1, 2, 4, και 16 κόμβους. Για την CPU κάθε κόμβος έχει 2 workers ενώ για

την GPU έχουμε εξετάσει τόσο την περίπτωση με έναν worker ανά κόμβο όσο και με 2. Τέλος χρειάζεται να αναφέρουμε πως στην περίπτωση των CPUs έχουμε εφαρμόσει και την τεχνική του task-parallelism, δηλαδή οι serial πράξεις που γίνονται μετά το histogram, μοιράζονται στους 2 CPU workers έτσι ώστε να μειωθεί ο χρόνος τους. Τα αποτελέσματα φαίνονται στις εικόνες 1.22, 1.23 και 1.24.

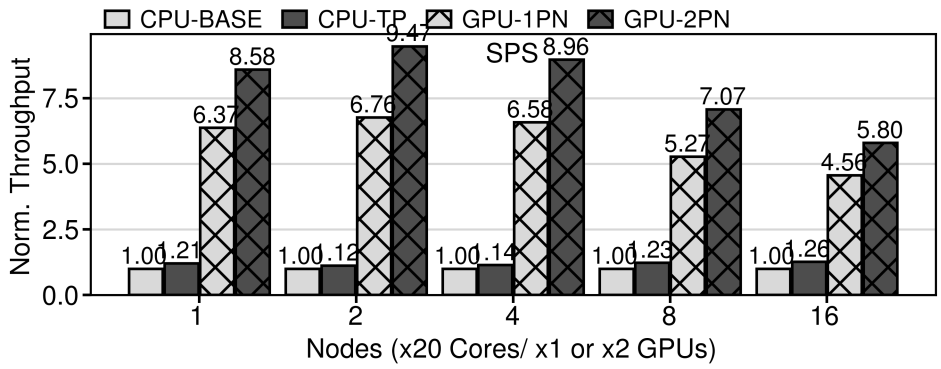


Σχήμα 1.22: *Σύγκριση μεταξύ CPU και GPU στο LHC*



Σχήμα 1.23: *Σύγκριση μεταξύ CPU και GPU στο PS*

Παρατηρούμε πως η κάρτα γραφικών με έναν worker ανά κόμβο, είναι πάνω από 3 φορές πιο γρήγορη από την CPU στο LHC και στο PS. Οι 2 workers ανά κόμβο στις GPU, μειώνουν το comp χρόνο στο μισό, αλλά αυξάνουν το κόστος της επικοινωνίας. Για αυτό τον λόγο η διαφορά μεταξύ GPU-1PN και GPU-2PN μειώνεται όσο αυξάνουμε τους κόμβους.



Σχήμα 1.24: *Σύγκριση μεταξύ CPU και GPU στο SPS*

Στο SPS η απόδοση της GPU σε σχέση με την CPU χειροτερεύει αρκετά όσο αυξάνουμε τον αριθμό των κόμβων, λόγω της αυξημένης επικοινωνίας που προκύπτει.

1.5.3 Προσεγγιστικές Μέθοδοι

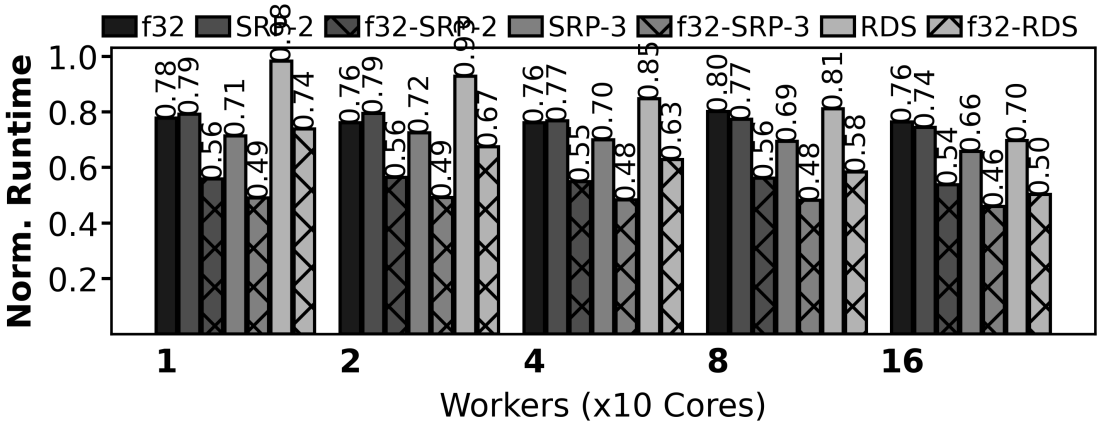
Προκειμένου να μειώσουμε τον χρόνο τον πειραμάτων, καθώς και το κόστος της επικοινωνίας, εφαρμόσαμε κάποιες μεθόδους προσέγγισης. Αυτές μειώνουν την ακρίβεια των αποτελεσμάτων, αλλά όπως βλέπουμε παρακάτω όχι αρκετά ώστε να υπάρχει πρόβλημα. Αυτές είναι οι εξής.

- f32 : μείωση των bits από τα 64 στα 32 για τους πραγματικούς αριθμούς.

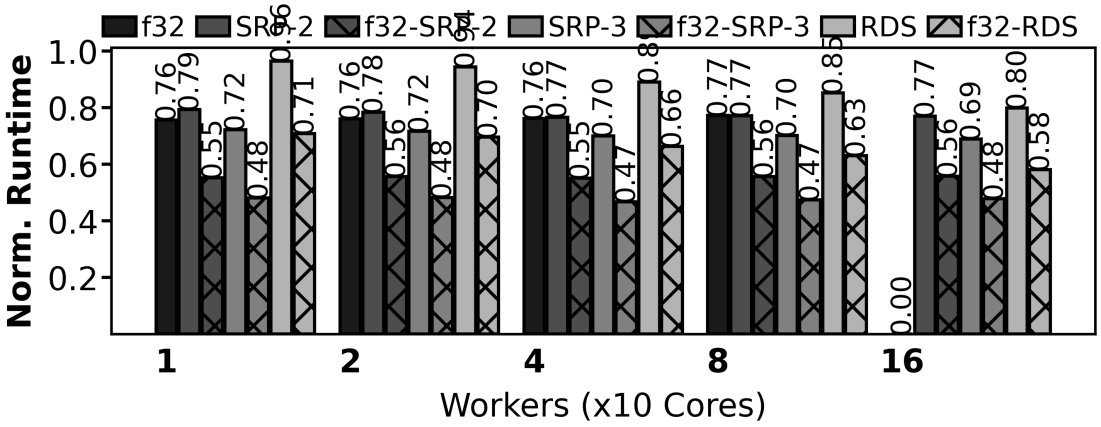
- srp-n : ο υπολογισμός του ιστογράμματος κάθε n γύρους με την παραδοχή ότι δεν αλλάζει απότομα, όταν το n παίρνει μικρές τιμές π.χ. 2 ή 3.

- rds : ο υπολογισμός του ολικού ιστογράμματος μέσω κλιμάκωσης, με την παραδοχή πως η κατανομή που έχει ο κάθε worker είναι ίδια με την κατανομή όλων των σωματιδίων και έτσι τα τοπικά ιστογράμματα πολλαπλασιασμένα με τον κατάλληλο αριθμό, μπορούν να αντικασταστήσουν το ολικό ιστόγραμμα.

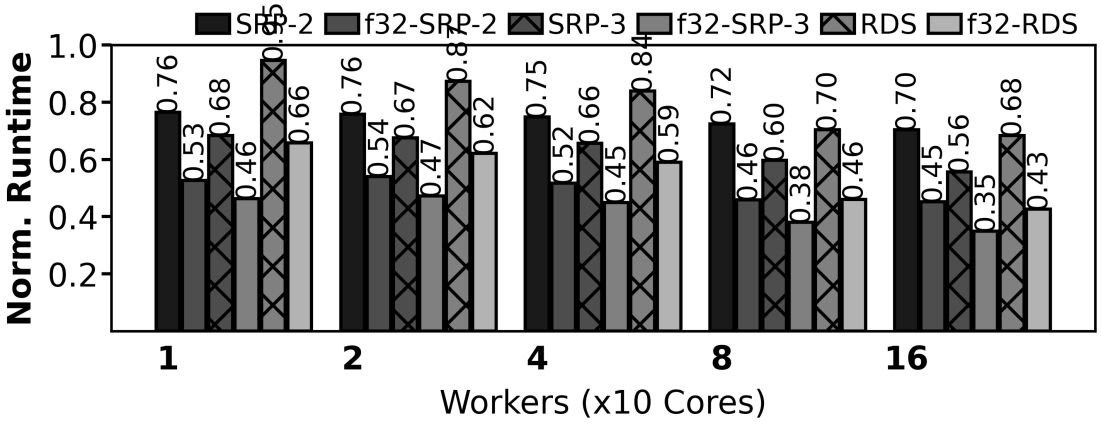
- οι παραλλαγές των παραπάνω δύο με την χρήση 32 bits για τους πραγματικούς. αριθμούς.



Σχήμα 1.25: *Προσεγγιστικές μέθοδοι στο LHC*



Σχήμα 1.26: *Προσεγγιστικές μέθοδοι στο PS*

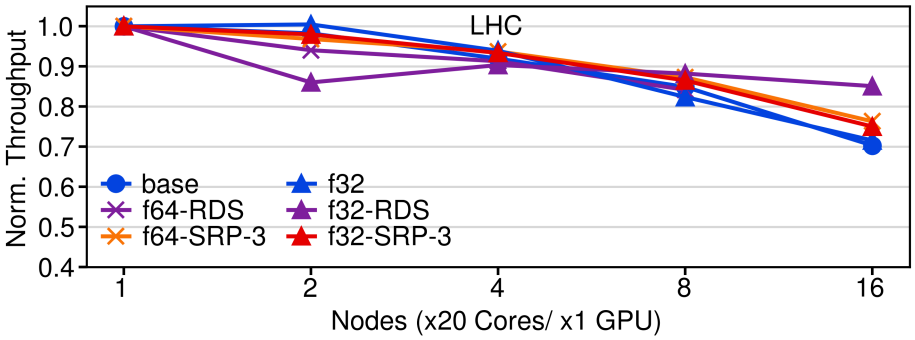


Σχήμα 1.27: *Προσεγγιστικές μέθοδοι στο SPS*

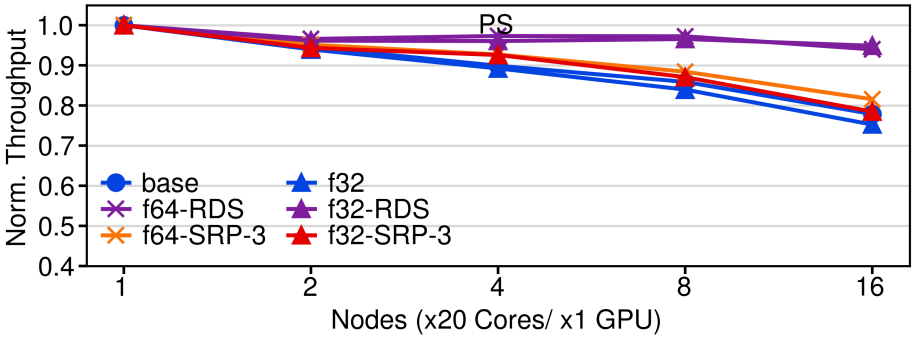
Στις εικόνες 1.25, 1.26 και 1.27 μπορούμε να παρατηρήσουμε τα αποτελέσματα από τα πειράματα. Παρατηρούμε πως το f32 ρίχνει το χρόνο σε όλα κοντά στο 75%. Το RDS

όσο ανεβαίνουμε σε αριθμό κόμβων αυξάνει την συμβολή του και αυξάνεται το ποσοστό του χρόνου που εξοικονομεί. Το ίδιο ισχύει και με το SRP. Στους συνδυασμούς του f32 με τα SRP και RDS, επειδή είναι σχεδόν ανεξάρτητα, αφού το καθένα από αυτά μειώνει τον χρόνο από διαφρετικά σημεία, μπορούμε να δούμε πως υπάρχει κάτι που θα μπορούσαμε να το χαρακτηρήσουμε επαλληλία. Δηλαδή τα ποσοστά που λείπουν στο καθένα ξεχωριστά, αθροίζονται στον συνδυασμό τους. Για παράδειγμα στο f32 του LHC λείπει το 22% και στο SRP-2 λείπει το 21%, ενώ στο f32-SRP-2 λείπει το 44%, που είναι κοντά στο άθροισμα αυτών των δύο.

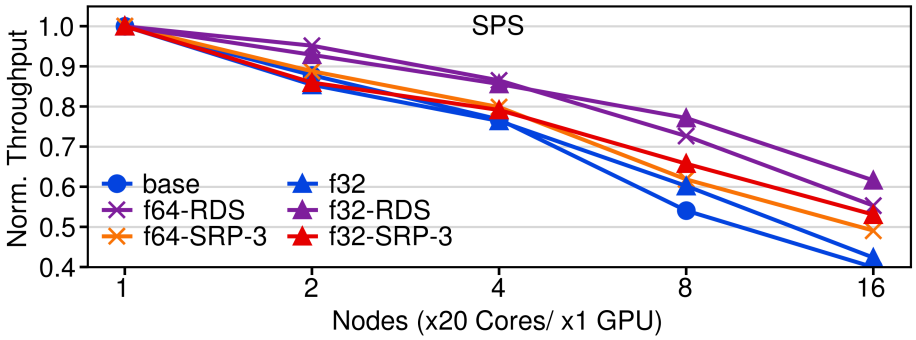
Αφού είδαμε πως επιδρούν οι παραπάνω μέθοδοι, ας δούμε τώρα πως κλιμακώνουν στις εικόνες 1.28, 1.29 και 1.30.



Σχήμα 1.28: *Weak Scaling στο LHC*



Σχήμα 1.29: *Weak Scaling στο PS*

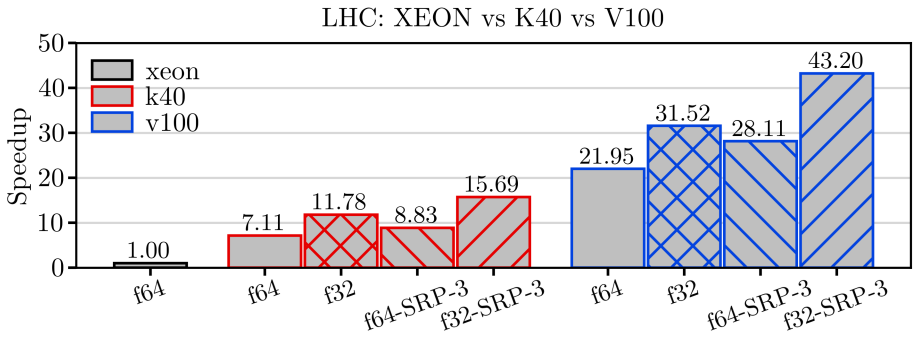


Σχήμα 1.30: *Weak Scaling στο SPS*

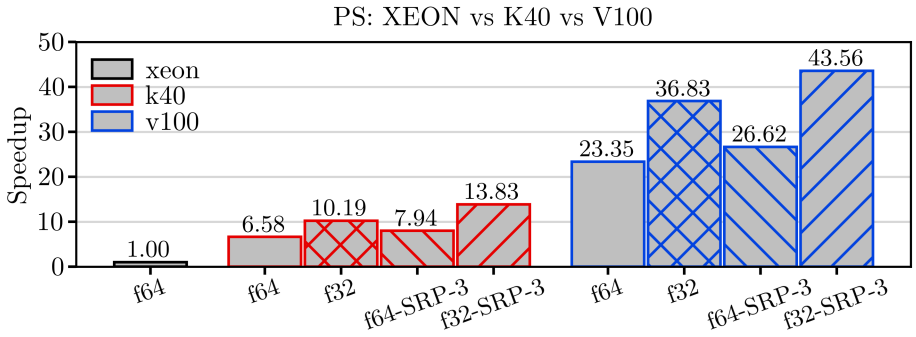
Παρατηρούμε ότι με την συμβολή των προσεγγιστικών μεθόδων που εφαρμόστηκαν, υπάρχει καλύτερη κλιμάκωση σε σχέση με πριν. Στο PS μάλιστα το throughput είναι κοντά στο 95% του αρχικού σε σχέση με την βασική έκδοση που είναι στο 75%. Όπως φάνηκε και στην εικόνα 1.27 το SPS κλιμακώνει χειρότερα από τα υπόλοιπα, με το f32-RDS να φτάνει μόλις στο 60% του αρχικού από το 40$ που φτάνει το βασικό.

1.5.4 Σύγκριση της v100 με την k40

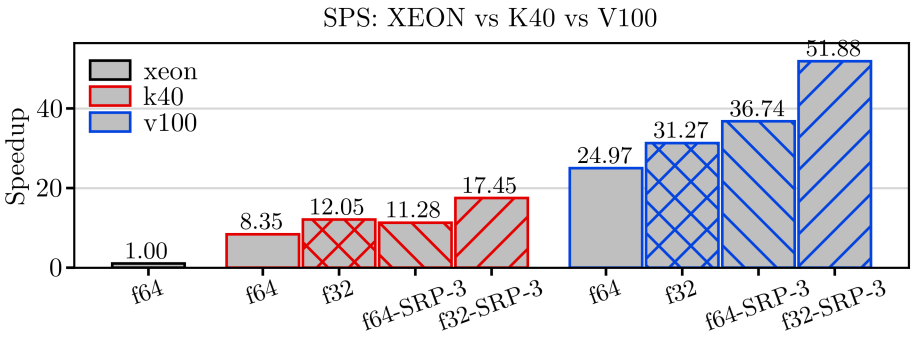
Η k40 Τέλος δείχνουμε τα αποτελέσματα που προκύπτουν από την σύγκριση των δύο διαφορετικών καρτών γραφικών που διαθέτουμε, της k40 με την v100. Η σύγκριση γίνεται μόνο σε επίπεδο ενός κόμβου με έναν worker.



Σχήμα 1.31: *Σύγκριση k40 με v100 στο LHC*



Σχήμα 1.32: *Σύγκριση k40 με v100 στο PS*



Σχήμα 1.33: *Σύγκριση k40 με v100 στο SPS*

Από τις εικόνες 1.31, 1.32 και 1.33, παρατηρούμε πως η v100 είναι 3 φορές πιο γρήγορη σε κάθε μέθοδο από την k40, και στην βασική έκδοση είναι πάνω από 20 φορές πιο γρήγορη από την CPU. Προκύπτει λοιπόν εύκολα το συμπέρασμα, πως η χρήση καλύτερο hardware οδηγεί σε ξεκάθαρη αύξηση της επίδοσης των προγραμμάτων μας.

1.6 Συμπεράσματα και Μελλοντικές επεκτάσεις

1.6.1 Συμπεράσματα

Στην εργασία αυτή, παρουσιάσαμε πως υλοποιήθηκε η επιτάχυνση του BLonD module με την χρήση της κάρτας γραφικών. Υλοποιήσαμε τις συναρτήσεις που φαίνονται να καταναλώνουν τον περισσότερο χρόνο των πειραμάτων, στην GPU. Έπειτα για κάθε συνάρτηση από αυτές με δικά μας δεδομένα κάναμε σύγκριση μεταξύ της CPU που είχαμε στην κατοχή μας και των δύο GPUs. Βλέποντας αυτά τα αποτελέσματα, καταφέραμε και κάναμε βελτιστοποιήσεις στον κώδικα της GPU έτσι ώστε να πάρουμε όσο το δυνατόν καλύτερα αποτελέσματα.

Στην συνέχεια δείξαμε πως πετύχαμε να ενσωματώσουμε αυτές τις συναρτήσεις στα πειράματα μας, με την χρήση των αντίστοιχων GPU κλάσεων και των αντικειμένων CGA. Με αυτόν τον τρόπο καταφέραμε να εξαλείψουμε εντελώς από κάποια πειράματα την μεταφορά δεδομένων μεταξύ επεξαστή και κάρτας γραφικών. Αφού ολοκληρώσαμε την υλοποίηση για τον ένα κόμβο, συμπλήρώσαμε τις συναρτήσεις που χρειάζονται προκειμένου να μπορεί να χρησιμοποιηθεί η GPU ως MPI worker, είτε με άλλες GPU αλλά ακόμα και με CPU.

Αφού πραγματοποιήθηκαν όλα τα παραπάνω, προχωρήσαμε στην σύγκριση της υλοποίησης μας με την παλιότερης CPU υλοποίησης. Με την χρήση του ARIS είδαμε πως η GPU είναι 5 φορές γρηγορότερη στις περισσότερες περιπτώσεις, και προκειμένου να λύσουμε το πρόβλημα της κλιμακωσημότητας, εφαρμόσαμε τις προσεγγιστικές τεχνικές. Έτσι βελτιώσαμε την απόδοση της υλοποίησης μας περισσότερο, χωρίς να χάσουμε σημαντική ακρίβεια, λόγω της φύσης των πειραμάτων μας.

Τέλος, προκειμένου να βεβαιωθούμε πως η υλοποίηση μας θα μπορεί να δουλέψει με πιο καινούριο hardware εξίσου καλά χωρίς αλλαγές, χρησιμοποιήσαμε την νεότερη και δυνατότερη NVidia Tesla v100 και την συγκρίναμε με την k40 παρατηρώντας πως είναι τουλάχιστον 3 φορές πιο γρήγορη.

1.6.2 Μελλοντικές Επεκτάσεις

Πέρα από τα παραπάνω που υλοποιήθηκαν, υπάρχουν κάποια εμπόδια που αν λύναμε θα μας δινόταν η δυνατότητα να κάνουμε αρκετά μεγαλύτερα και γρηγορότερα πειράματα. Παρακάτω παραθέτουμε κάποιες ιδέες που θα βοηθούσαν σε αυτήν την κατεύθυνση.

- Στην παρούσα έκδοση, η αρχικοποίηση της ακτίνας των σωματιδίων γίνεται σε έναν worker και έτσι όλη η ακτίνα οφείλει να χωρά στην μνήμη ενός worker παρ'όλο που αυτός θα κρατήσει μονάχα ένα τμήμα της. Έτσι, περιοριζόμαστε στο μέγεθος της ακτίνας από την κύρια μνήμη ενός worker και όχι από την συνολική μνήμη όλων των workers. Αν μπορούσαμε να υλοποιήσουμε την αρχικοποίηση με το MPI, θα μας δυνόταν η δυνατότητα να δοκιμάσουμε πειράματα με πολύ μεγαλύτερο αριθμό σωματιδίων από ότι τώρα.

- Στα πειράματα που κάναμε δοκιμάσαμε να χρησιμοποιήσουμε κάποιους κόμβους με CPU και κάποιους άλλους με GPU. Για να καταφέρουμε να τις συγχρονήσουμε χρησιμοποιήσαμε ένα σύστημα εξισορρόπησης του φορτίου, έτσι ώστε να φτάνουν οι κόμβοι την ίδια στιγμή στο στάδιο του reduce και έτσι να μην χάνουμε αρκετό χρόνο για συγχρονισμό. Ωστόσο αυτή η προσπάθεια απέτυχε, αφού σχεδόν πάντα ο χρόνος που κερδίζαμε από την χρήση CPU ήταν μικρότερος από αυτόν που χάναμε λόγω του συγχρονισμού, επειδή η CPU είναι αρκετά πιο αργή από την GPU. Συνεπώς θα είχε νόημα να δοκιμάσουμε την εφαρμογή μας σε ένα cluster με CPUs αντίστοιχα ισχυρές με τις GPUs ώστε να είναι ετερογενής η προσομοίωση.

Chapter 2

Introduction

2.1 CERN

CERN, the European Organization for Nuclear Research, was founded in 1954 by 12 countries in Western Europe. It is a research organization that operates the largest particle physics laboratory in the world. It is based in a northwest suburb of Geneva and the current number of its members is twenty three. CERN's main focus is to provide physicists the particle accelerators they need, in order to perform experiments. Another important contribution of CERN is the world wide web, since it started as a project there. Currently, CERN operates a network of six accelerators and a decelerator, that are involved in increasing the energy of particles needed for the experiments. Acceleration is achieved in multiple steps through a sequence of accelerators. Each one modifies the characteristics of the beam and passes it to the next one. [13] [14] [15]

2.2 Longitudinal Beam Dynamics

Longitudinal Beam Dynamics [1] is a field of Physics that study the motion of the particles in an accelerator and the issue of synchronization between the particles and the accelerating field. Experiments related to that, are happening in CERN's accelerators, like the LHC. Their need for time and resources, is the reason why its difficult to perform them very often. Additionally, CERN's accelerators are under maintenance for some periods of times, so the execution of them is infeasible.

In order to overcome these obstacles, a team of physicists in CERN, developed a tool called BLonD [16] [17], to develop simulations of these experiments. These simulations are helpful not only because they produce results, but because they also contribute in the real accelerators upgrading procedure. They involve a lot of particles, from 500 thousands particles, up to one 1 billion, and they run for a lot of turns. Also, some simulations need to be repeated a lot of times with different parameters. As a result they take a lot of time to produce results, and it is really important to reduce their runtime.

The GPU [4] is a device used to rapidly generate the output desired for a display device. Its special feature is, that it can perform a lot of calculations at the same time. Because of that exact feature, it has became extremely common to use it to speedup general purpose applications. Today, some of the world's most powerful supercomputers are using GPU

acceleration. Applications that can be benefited by them are those with a lot of heavy operations done in parallel.

Since the computational parts of BLonD are embarrassingly parallel, we chose to apply GPU acceleration to make the simulations produce results faster. and For this thesis, our goal was to apply GPU acceleration in the BLonD module. Since most parts of a BLonD simulation are embarrassingly parallel, BLonD is a perfect candidate for GPU acceleration. We had two main objectives. The first one was to implement and optimize the GPU functions, to produce better results. The second one was to make this GPU version of BLonD user friendly, so that physicists can you use it without having knowledge of how that a GPU works.

2.3 Thesis Structure

The rest chapters of thesis are organized as follows:

- In Chapter 2 we present the basics of GPU architecture and the CUDA programming model. We also make an introduction to BLonD simulations.
- In Chapter 3 we examine the core BLonD functions, their GPU implementation and some optimizations we applied to them. We also showcase and evaluate an optimization called gpu_cache.
- We demonstrate how with a few commands a user can enable the GPU in Chapter 4.
- In Chapter 5 we evaluate the performance of our GPU implementation and we compare it with the CPU. We also examine the approximation techniques and the performance we get by them.
- We conclude this thesis and refer to some things that could be future work in Chapter 6.

Chapter 3

Background Knowledge

3.1 BLonD Simulations

BLonD simulations consist of two main parts. The first part is the initialization of the simulation's objects. The most important objects that take part in a simulation are the following:

- *Beam* is the main object of the simulation. It contains the energy and time coordinates of our particles. Its size is considered the input size of the simulation and varies from 500 thousand particles for small simulation, to 1 billion for bigger ones.
- *Profile* is an object that is used to observe particles time coordinates. It is used to get the histogram of them and operate on it. The number of slices of the histogram is less than the number of particles, usually about 1000 times smaller.
- *Tracker* is an object used to modify the dE and dt coordinates of beam with functions kick and drift.
- *Impedance* is an object used to apply FFTs to the histogram, and create some arrays that affect the functions of the tracker.

After the initialization we have the main loop that is run for some number of turns. In this loop some objects use their track method. These objects usually are a tracker, a profile, and a TotalInducedVoltage object that instead of track it invokes a method that computes the FFTs of the histogram. A more abstract way of this main loop is the following.

- Modify dE, dt with kick or linear_interpolation_kick and drift methods
- Get the histogram of dt
- From histogram make some adjustments for the kick and drift of the next turn

In the process of accelerating these simulations, the functions that dominated the runtime were written in C++. This way, they were optimized and run in parallel with OpenMP [6]. In this new version, instead of calling python functions, that were serial, new C++ functions were used that were optimized and fine tuned. This new version is called blond++ [7] and it delivered up to 23X single-node speedup.

In order to be able to scale-out BLonD experiments, a new version was developed, that is called Hybrid-BLonD [9] . In this version, the workload is distributed among workers, in order to scale both horizontally and vertically. MPI [8] is used to distribute the workload, and each worker uses OpenMP to perform its computations, so the schema is MPI-over-OpenMP. This new implementation demonstrates an average 25.7X speedup when using 32 computing nodes, compared to the single-node version.

3.2 GPU (Graphic Processing Unit)

GPUs or Graphic Processing Units were introduced as devices that could efficiently manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device. Originally they were created to support advanced graphics and video-gaming. Although their initial purpose was graphics, today they exist almost everywhere, game consoles, personal computers, even on mobile phones. There are two major GPU manufacturers, Nvidia and AMD. Since we use Nvidia GPUs both for the development and for our benchmarks, and the GPU code of this thesis is CUDA C, on the next paragraphs, where we describe what a GPU is, we are focusing mostly in Nvidia GPUs.

3.2.1 GPGPU

Even though GPUs were designed to support computer graphics, Nvidia introduced programmability to its GPUs and researchers started to use GPUs to compute linear algebra operations. Since then, the term General Purpose GPU [5] is used to describe the use of GPUs to support general purpose workloads. GPUs are mostly known today for accelerating the training process of neural networks [10]. The need for more demanding machine algorithms and workloads, has driven GPU manufactures to advance the architecture and the programming model of their GPUs.

3.2.2 GPU Architecture

In this subsection we will describe the architecture [11] of a modern GPU. A GPU is composed of many cores, which are called Streaming Multiprocessors or SMs (Compute Units for AMD). Each SM can execute a Single-Instruction-Multiple-Thread (SIMT) program. In modern GPUs the maximum number of threads that can run on the SM is 1024. The threads that run on the same SM can interact with each other using a scratchpad memory and barrier operations. Also they share a first-level cache, to reduce the traffic sent to the lower levels of the memory system. When data is not found in this cache, the latency that is produced is hidden due to the large number of threads, since waiting for data overlaps with the execution of other warps. The GPU has its own main memory, that the SMs have access to. To sustain high computational throughput the GPU must have high memory bandwidth, and for that purpose there are multiple memory channels, each to a different memory partition. These are connected with the SMs via an on-chip interconnection network. This can be seen in figure 3.1

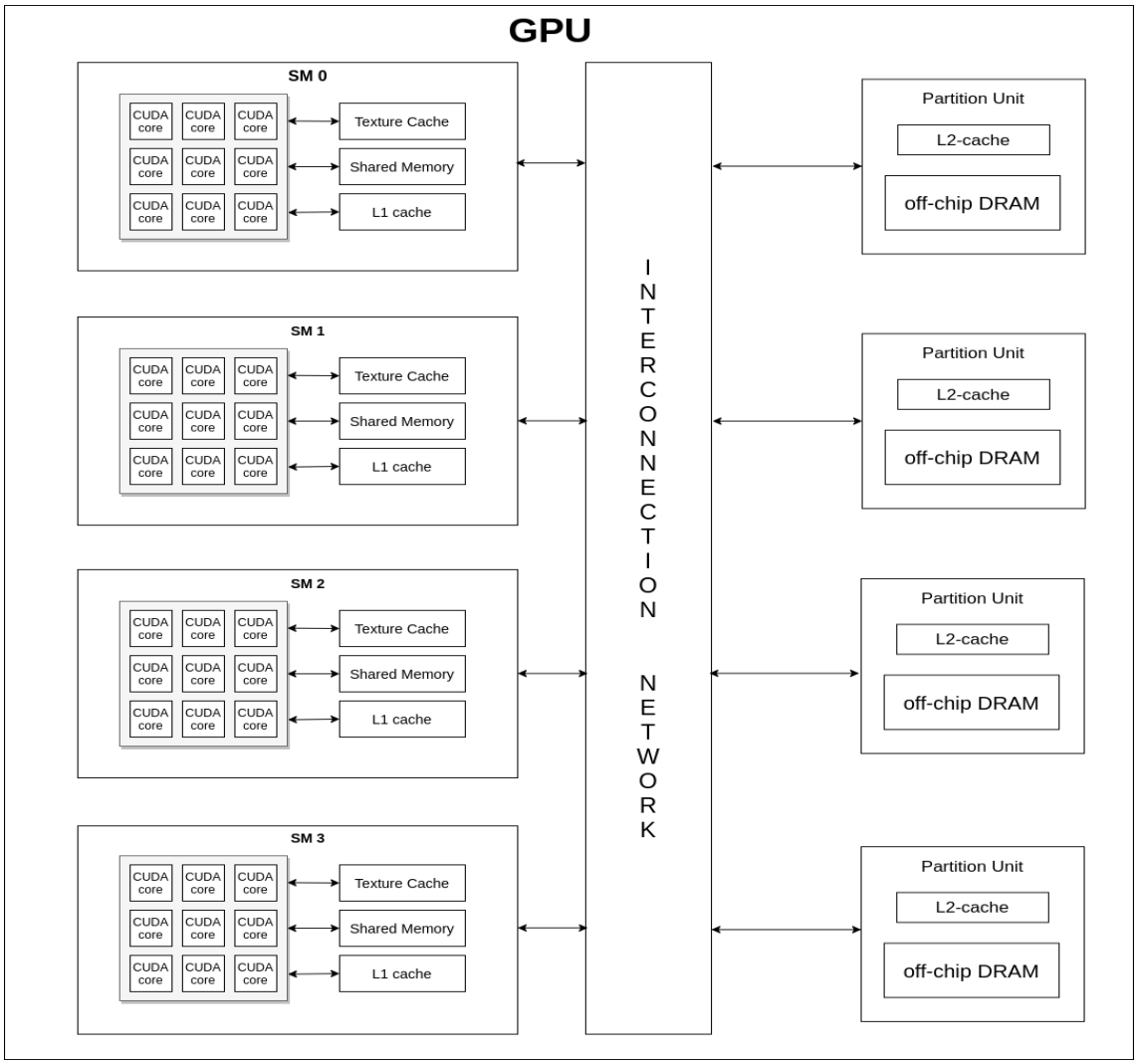


Figure 3.1: *GPU Architecture*

3.2.3 Execution Model

Threads that run on the same SM are executed in lockstep in groups of 32, or 64 for AMD GPUs, that are called warps (wavefronts by AMD), by Single-Instruction-Multiple-Data (SIMD) hardware. Threads are assigned in warps by the hardware when they spawn and their assignment is static. For example if we spawn 2048 threads, warp 1 contains threads 0-31, warp 2 contains 32-63 etc. Each of these threads has its own unique id that can is used to distribute the workload.

Since up to 2048 threads can run in a single SM, each SM can host more than one warp. A warp can be classified, based on its availability to be executed, as eligible or non eligible. A warp is eligible when it already has all the data it needs and it is ready for execution, else it is non eligible. Each SM has at least one warp scheduler that picks an eligible warp and executes its threads. If a warp stalls, due to an uncached memory read for example, it becomes inactive and changes places with an eligible warp that is available.

Its also possible for two threads of the same warp to follow different paths due to

control flow. The control flow code can be represented as a graph of Basic Blocks, straight-line code sequences with no branches in except to the entry and no branches out, where each thread can take a specific path. Threads that take different paths are being serialized. For example, in an if-else statement supposing that the first 16 threads of the warp execute the if statement and the other 16 execute the else statement, these 2 executions will happen as if they were in different warps. One approach to implement this strategy is to use a stack with three fields. A re-convergence program counter, the next Basic Block where two paths join that can be computed at compile time, the next program counter, and the active mask, a mask that describes which threads are in this statement. From what described in this paragraph it is clear that having complex control flow in the GPU source code is something that should be avoided, because it serializes the code. There is also a lot of research on how branch divergence can be handled to improve GPU performance [18] [19].

3.2.4 Memory System

In subsection 3.2.2 it was mentioned that a GPU has its own main memory. This is a DRAM memory with a size up to 32GB for modern GPUs. This memory is off-chip and is connected with SMs via an interconnection network. Just like CPUs, GPUs have part of their memory reside with their SMs.

Shared Memory is an on-chip memory. Each SM has its own and only its threads can access it. It is implemented as a static random access memory and is described as being implemented with one bank per lane with each bank having one read port and one write port. Each thread can access every bank. The access to shared memory is almost as fast as accessing the register file. Although it needs to be handled carefully since bank conflicts are happening when two or more threads try to access different elements that exist in the same bank. If this happens, hardware splits these requests and serializes them, decreasing the potential bandwidth. Successive 32-bit words are assigned to successive banks so a good practice is for successive threads to access successive elements.

L1-data cache is a different on-chip memory that is unique for each SM. This type of memory is used to help the threads of the SM access global memory. It maintains a small subset of the global memory address space and in some architectures this subset is strictly not modified, since this way cache coherence protocols are not needed, decreasing complexity. If all threads in a warp try to access elements that would fall in the same L1 data cache block and miss, then a single request will be sent. Accesses like these are said to be "coalesced". If threads within a warp try to access elements from different L1 cache blocks then multiple requests are generated. Accesses like these are said to be "uncoalesced". A good practice when we access the global memory is to access successive elements in order. This way we get as many coalesced accesses as possible.

Aside from these two memories each SM contains one last cache, named Texture

Cache that is used for computer graphics but it is not really useful in GPGPU.

Memory Partition Units The SMs of the GPU are connected with the main memory, through an interconnection network. The main memory is separated in memory partitions, where each partition has a portion of the L2 cache. Each memory partition contains along with the L2 cache one or more access schedulers that are responsible for reordering read and write operations to reduce overheads of accessing DRAM.

3.2.5 Development in GPU with CUDA C

In the previous paragraphs we mentioned how the GPU hardware is structured and now we are going to see how someone can use the GPU to speedup his workload. We will accomplish that by using an example.

Lets suppose that someone wants to add two arrays of integers, A and B, and store the result in a third array C. Arrays A and B are stored in the CPU, so first of all we have to allocate space for them in the GPU main memory, and then transfer these two arrays there. After we have to allocate the space needed to save the result. To compute the sum of these arrays we need to call a GPU function.

GPU functions, also called kernels, are lines of code that are executed by all threads. In order to call a kernel we need to pass the arguments of the kernel, like the arrays A,B,C and their size for the previous example. Then we also need to define some special parameters called block size and grid size. Threads in the GPU follow a hierarchy. The smallest unit is the thread that belongs in a block. Thread has its own id inside the block that can be accessed from CUDA. Then all blocks belong to the grid, where again each block has its own id. So the block size parameter is the number of threads inside each block, and grid size is the number of blocks inside the grid. These parameters are written inside <<< .. >>> after the kernel name, like for example *addVecs* <<< *grid_size, block_size* >>> (*A, B, C, size*). In the previous example if we set grid_size to 16 and block_size to 1024, we are going to call 16 blocks, where each block contains 1024 threads, 16384 threads in total. Determining the value of these threads is crucial if we want to get the best performance. It is also important to mention that threads of the same block run in the same SM and have ways of communicating, like shared memory and barriers. You can see that in figure 3.2.

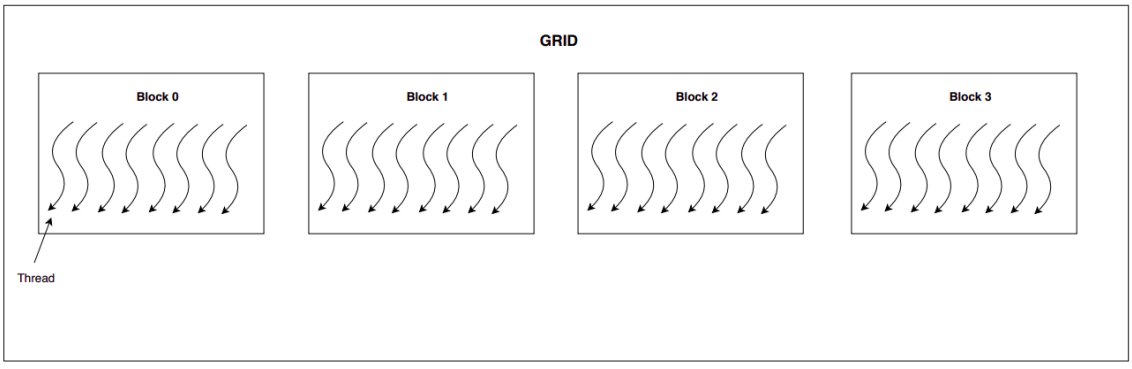


Figure 3.2: *We can see a grid with 4 blocks with each block having 8 threads*

Below we can see an example of the kernel *addVecs*(*int* ∗ *A*, *int* ∗ *B*, *int* ∗ *C*, *intsize*) and the main that calls it.

```
void addVecs(int *A, int *B, int *C, int size){
    //find the global index
    //of thread inside the grid
    int tid = threadIdx.x + blockIdx.x*blockDim.x;
    // Each thread will compute all the indexes
    // that are like total_threads * n + tid
    for (int i = tid; i < size; i+= blockDim.x*gridDim.x)
        C[i] = A[i] + B[i];
}
```

In the above kernel first of all each thread computes its global index. To do that it adds its threadIdx.x (id inside the block) to the blockIdx.x*blockDim.x (block id inside the grid, times the number of threads inside each block). Then it operates to indexes assigned to it until it gets out of range of the arrays. The for-loop is this way constructed so even if threads are lesser than the size of the array, the whole sum is performed. In case that the threads are more than the size, each thread completes one or no iteration of the for-loop. The above kernel also satisfies the criterion "successive threads access successive memory locations", making our accesses coalesced. Next, we see the main function. It is important to keep in mind that sometimes CPU is also called Host and GPU is called Device.

```
int main(){
    //declare CPU arrays
    int A[10000];
    int B[10000];
    int C[10000];

    // Initialize CPU arrays

    for (int i = 0; i < 10000; i++){
        A[i] = rand();
        B[i] = rand();
    }

    // Declare GPU pointers and allocate
    // memory for them with cudaMalloc
    int *d_A;
    cudaMalloc(&d_A, 10000 * sizeof(int));
    int *d_B;
    cudaMalloc(&d_B, 10000 * sizeof(int));
    int *d_C;
    cudaMalloc(&d_C, 10000 * sizeof(int));
```

```
    // Copy arrays from CPU to GPU
    cudaMemcpy(d_A, A, 10000 * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, 10000 * sizeof(int), cudaMemcpyHostToDevice);

    // Invoke the kernel
    addVecs<<< (10000 + 512 - 1)/512 , 512>>>(A, B, C, 10000);

    // Copy the result back in the CPU
    cudaMemcpy(C, d_C, 10000 * sizeof(int), cudaMemcpyDeviceToHost);

    // Free GPU memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```

In this main function five operations that have to do with the GPU are performed. First we have memory allocation with cudaMalloc and then we copy the contents of A and B to d_A and d_B with cudaMemcpy. We invoke the kernel to compute the sum of these array and we call cudaMemcpy to retrieve the contents of d_C to C. Finally we free the allocated space in the GPU with cudaFree.

For the block size we chose the value 512, and for the grid size we selected (10000 + 512 - 1)/512 which is equal to 20. So the total number of threads is 20*512=10240. After we decided the block size we picked the least number that multiplied with 512 is greater or equal with the size of our arrays. These numbers affect the **occupancy** of our kernel. Occupancy is defined as the ratio of active warps on an SM to the maximum number of active warps supported by the SM. There is an upper limit for active warps that are executed, based on the compile options, the configuration and the device capabilities. Below we can see the limiters of the occupancy.

- **Active Warps Per SM** For each GPU there is maximum number of warps that can be active at the same time. Lets suppose that this number is 64. If we set the block size to 256(8 warps per block) and we have 8 blocks for each SM, each SM has 64 active warps so the occupancy is 100%. If the block size is 256(8 warps per block) and we assign each SM 6 blocks, we have 48 active out of 64 so the occupancy is 75

- **Blocks per SM** Again there is a limit for active blocks that can run in a GPU. Lets assume that warps are limited to 64 per SM and that an SM has a limit of 16 active blocks. If we try blocks with 32 threads each, we have only 16 active blocks since we are limited by that number, while each block contains one warp, so 16 warps in

total. That means 16/64 warps so 25% occupancy. If we try a greater number for block size like 256 we can have up to 4096 threads(16 blocks) but we are limited from active warps since 4096 threads mean 128 active warps. We end up with 2048 threads and 64 active warps, which means (8 active blocks) maximizing the occupancy.

- **Registers per SM** Each SM has a set of registers which is shared by all active threads, that is limited. We can see for a kernel the number of registers that needs to execute. For example lets assume that this number is 33 and the SM limit is 65536. Lets assume now that the block size is 512. Registers needed for a block are 33*512 = 16896. So we can have up to 65536/16896 = 3 active blocks. That means having 512*3/32 = 48 active warps. So the occupancy in this example is 75%.

- **Shared Memory per SM** There is also a limit for the amount of shared memory that can be used for an SM. A typical number for shared memory is 48 kB. If each block for example with 512 threads needs 22 kB of shared memory, only two of these blocks can run in the SM, since a third one can not satisfy its need for shared memory. Then we have 512*2/32 = 32 active warps and the occupancy is 32/64 = 50%.

Trying to optimize our kernel, we need to find grid and block size parameters that maximize the occupancy of our GPU. This is the general rule, which applies also to our kernels.

3.2.6 Development in GPU with PyCUDA

Since the BLonD code is written in python, PyCUDA [12] is a module that is used in this thesis. PyCUDA offers a way of invoking GPU kernels from python code while adding no overhead. Below there is the previous example of how to add two arrays A and B with PyCUDA.

```
import pycuda.autoinit # initialize pycuda
from pycuda import gpuarray
import numpy as np

a = np.random.randint(1000, size=1000)
b = np.random.randint(1000, size=1000)

d_a = gpuarray.to_gpu(a)
d_b = gpuarray.to_gpu(b)

d_c = d_a + d_b
c = d_c.get()
```

This code is pretty simple. We create two numpy arrays with size 1000. Then we use the gpuarray.to_gpu method which allocates space and transfers a to GPU. The d_a is a gpuarray object, an array that exists in the GPU memory. The same procedure is done for the b array. Then we simple perform the operation just like we would do with numpy. PyCUDA recognizes the operation, it creates a kernel to do that operation and compiles it. Because of that compilation, we have an additional overhead. Also PyCUDA defines by itself the block and grid size.

There is also a different way to call a kernel provided by PyCUDA. You can see it in the following example.

```
from pycuda.autoinit
from pycuda.compiler import SourceModule
import numpy as np

mod = SourceModule("""
    __global__ void addVecs(int *a, int *b, int *c, int sz)
    {
        int tid = threadIdx.x + blockIdx.x*blockDim.x;
        for (int i = tid; i < sz; i += blockDim.x*gridDim.x)
            c[i] = a[i] + b[i];
    }
""")
addVecs = mod.get_function("addVecs")

a = np.random.randint(1000, size=1000)
b = np.random.randint(1000, size=1000)

d_a = gpuarray.to_gpu(a)
d_b = gpuarray.to_gpu(b)
d_c = gpuarray.empty_like(a)
addVecs(d_a, d_b , d_c, np.int32(d_a.size),
        grid=(20,1), block=(512,1,1))
c = d_c.get()
```

Using the SourceModule we need to compile the code only the first time it is used, and we are also able to determine the grid and block size parameters. Finally to completely avoid compiling at runtime, we can compile our CUDA code and use the cubin file from PyCUDA. This last approach is used in this thesis, since jit (just in time) compiling was not available for our benchmarks in ARIS.

Chapter 4

GPU implementation of BLonD

In this section we will explain, how we ported some of the BLonD most crucial functions in the GPU, and what practices we used to optimize them. Also, we will demonstrate and discuss the results of these functions, run in three different devices, 2 GPUs and 1 CPU. Finally, we also illustrate an optimization that can be applied to experiments, the GPU_cache, and its results.

4.1 Implemented Kernels

The greatest percentage of time for simulations is being consumed by specific functions like histogram, kick, drift and linear_interpolation_kick. So first of all we tried to implement these kernels in the GPU. In this section we will also show some optimizations that are applied in these kernels. We will compare these kernels with their CPU equivalent functions using an Nvidia Tesla k40 GPU and an Intel Xeon E5-2660v3 CPU. The GPU has 15 Streaming Multiprocessors and 12 GBs of RAM. The CPU has 10 cores and its frequency is 2.60 GHz. We also compare a double precision version with a single precision one. Finally after their comparison we will also compare the Nvidia Tesla k40 with the Nvidia Volta V100, which is a newer and very strong GPU with 80 SM's and 32 GB of RAM. Their specifications can be seen in chapter 6.

4.1.1 Histogram Kernel

The first implemented kernel was the histogram kernel. The histogram kernel takes as input the dt coordinates of a beam, a lower and an upper limit for dt, and the number of slices. Also we need the array with the slices, that will be the result of our function. So in order to calculate the histogram with multithreading we need to find for each particle to locate its corresponding bin, and increase the value of that bin by one. It is important to notice that since a lot of threads may try to increase the value of the same bin, they must do it atomically. So a first approach is this *simple* kernel in CUDA C that can be seen below.

```
__global__ void simple_histogram(double * input,
    int * output, const double cut_left,
    const double cut_right, const int n_slices,
```

```
    const int n_macroparticles)
{
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    int target_bin;
    double const inv_bin_width = n_slices/(cut_right-cut_left);
    for (int i=tid; i<n_macroparticles; i=i+blockDim.x*gridDim.x){
        target_bin = floor((input[i] - cut_left) * inv_bin_width);
        if(target_bin<0 || target_bin>=n_slices)
            continue;
        atomicAdd(&(output[target_bin]),1);
    }
}
```

A thread first computes its global id and then for each dt coordinate that it is responsible for, calculates the target bin, and increases the value of that bin by one with the atomicAdd instruction. This is the most simplified version.

An optimization we can do for that version is to use the shared_memory. To do that, we give each block a local histogram, that exists in the shared_memory, and after finishing adding its particles, the block is responsible for adding this local histogram to the global one. This *shared memory* implementation is presented below.

```
__global__ void sm_histogram(double * input,
        int * output, const double cut_left,
        const double cut_right, const unsigned int n_slices,
        const int n_macroparticles)
{
    extern __shared__ int block_hist[];
    for (int i=threadIdx.x; i<n_slices; i+=blockDim.x)
        block_hist[i]=0;
    __syncthreads();
    int const tid = threadIdx.x + blockDim.x*blockIdx.x;
    int target_bin;
    double const inv_bin_width = n_slices/(cut_right-cut_left);
    for (int i=tid; i<n_macroparticles; i+=blockDim.x*gridDim.x){
        target_bin = floor((input[i] - cut_left) * inv_bin_width);
        if (target_bin<0 || target_bin>=n_slices)
            continue;
        atomicAdd(&(block_hist[target_bin]),1);
    }
    __syncthreads();
    for (int i=threadIdx.x; i<n_slices; i+=blockDim.x)
            atomicAdd(&output[i],block_hist[i]);
}
```

The shared memory array is declared in line 6. The size of the shared memory is

passed as a third parameter after block and grid size while calling the kernel. We need two more actions. Before using the shared memory we need to initialize it to 0, at block level. Then we use barrier at line 9 to make sure that the whole shared memory is being reset. Then the threads just like before increase their particle's bin locally. Once again we need to make sure that every thread is finished with its particles and we do that with __syncthreads() of line 19. Finally each thread add the bins that it is responsible for to the global histogram.

Obviously there is a limitation to that approach. A problem will appear if a local histogram cannot fit into the shared memory. To avoid that we have developed a third, hybrid version of the histogram kernel that includes one more parameter, capacity of the shared memory.

```
__global__ void hybrid_histogram(double * input,
            int * output, const double cut_left,
            const double cut_right, const unsigned int n_slices,
            const int n_macroparticles, const int capacity)
{
    extern __shared__ int block_hist[];
    //reset shared memory
    for (int i=threadIdx.x; i<capacity; i+=blockDim.x)
        block_hist[i]=0;
    __syncthreads();
    int const tid = threadIdx.x + blockDim.x*blockIdx.x;
    int target_bin;
    double const inv_bin_width = n_slices/(cut_right-cut_left);

    const int low_tbin = (n_slices / 2) - (capacity/2);
    const int high_tbin = low_tbin + capacity;


    for (int i=tid; i<n_macroparticles; i+=blockDim.x*gridDim.x){
        target_bin = floor((input[i] - cut_left) * inv_bin_width);
        if (target_bin<0 || target_bin>=n_slices)
            continue;
        if (target_bin >= low_tbin && target_bin<high_tbin)
            atomicAdd(&(block_hist[target_bin-low_tbin]),1);
        else
            atomicAdd(&(output[target_bin]),1);

    }
    __syncthreads();
    for (int i=threadIdx.x; i<capacity; i+=blockDim.x)
            atomicAdd(&output[low_tbin+i],block_hist[i]);
```

}

In this final version, we choose to keep the middle part of the histogram in the shared memory and the rest to the global memory. So, if a particle belongs to a bin that is inside the shared memory, we increase it locally in line 24, otherwise we increase it globally in line 26. Next we compare these versions in figure 4.1.

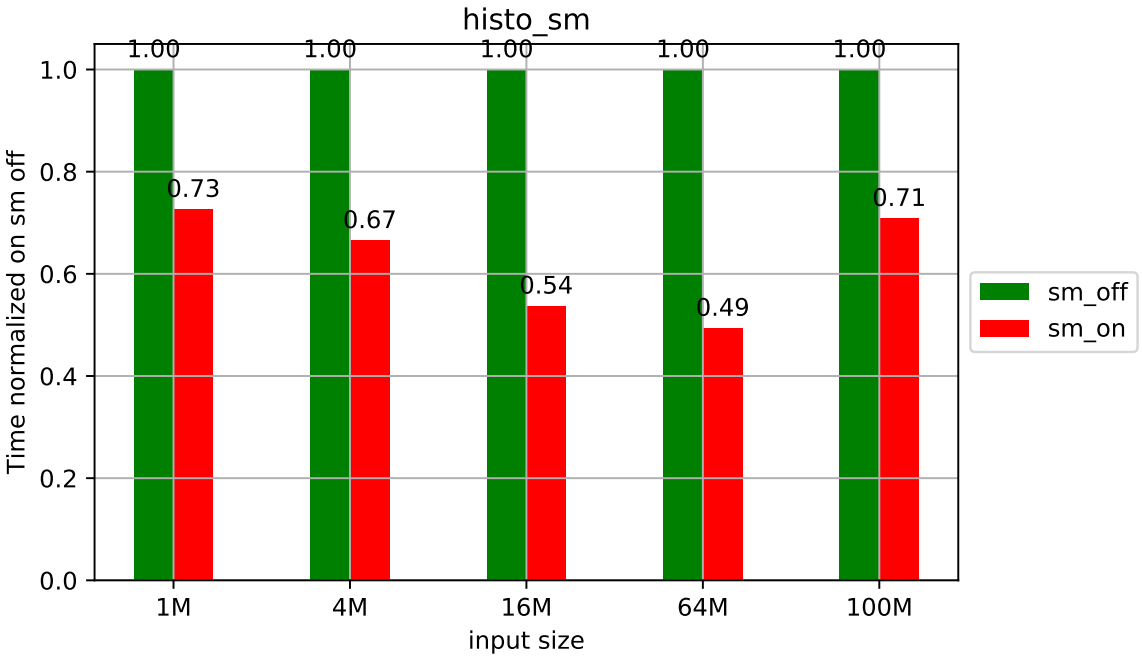


Figure 4.1: *Comparison of simple version with our shared memory version of histogram*

Our optimized version is better than the first simple kernel. The effect of it drops as we increase the number of bin and a great number of them is now located in the global memory. Its important to highlight that from this optimization we have two benefits. First of all, accessing an element that resides in shared memory is as fast as accessing a register. Also by having multiple local histograms we reduce by a lot the traffic that would exist if we were having every thread trying to atomically increase elements in one global histogram.

Now we see the comparison between the CPU and the GPU in figure 4.2

First of all we can see that the GPU is 10 times faster than the CPU. The only exception is the case with the 1 million particles, where the GPU is being underutilized. We can also see that there is almost no difference between the single and the double precision for both the CPU and the GPU. This happens first, because only the dt changes from double to float while the histogram arrays remain int32 and second, the bottleneck in this kernel are the atomic add operations, that do not gain any benefit from it.

Finally we compare the k40 with the V100, and we see the 5x speedup the more powerful GPU offers in figure 4.2.

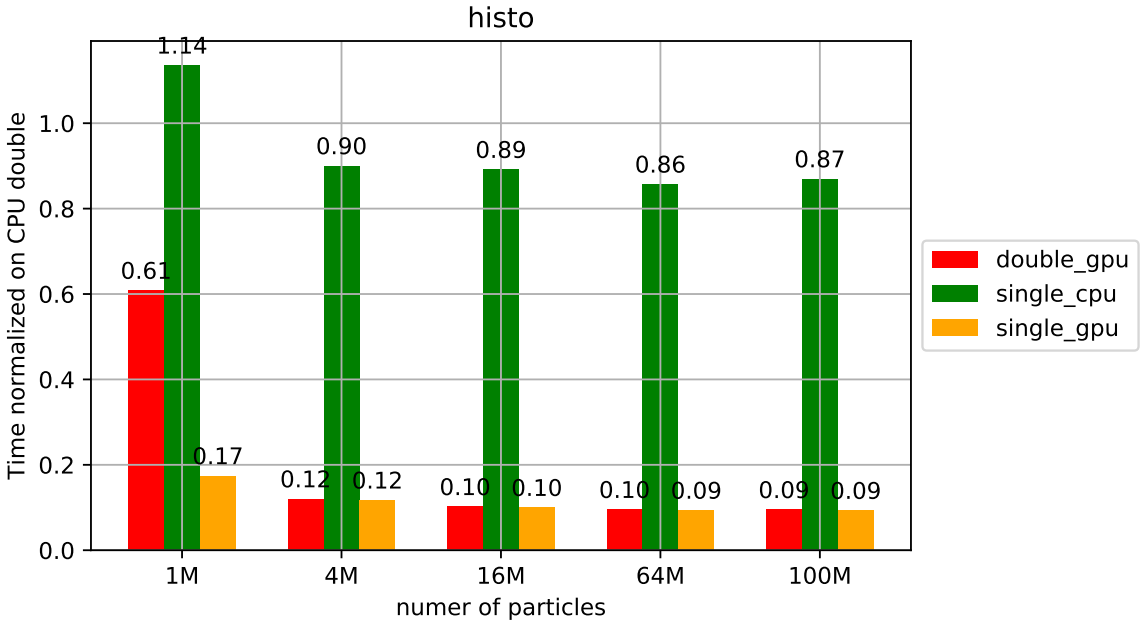


Figure 4.2: *Comparison of CPU and GPU for the histogram function*

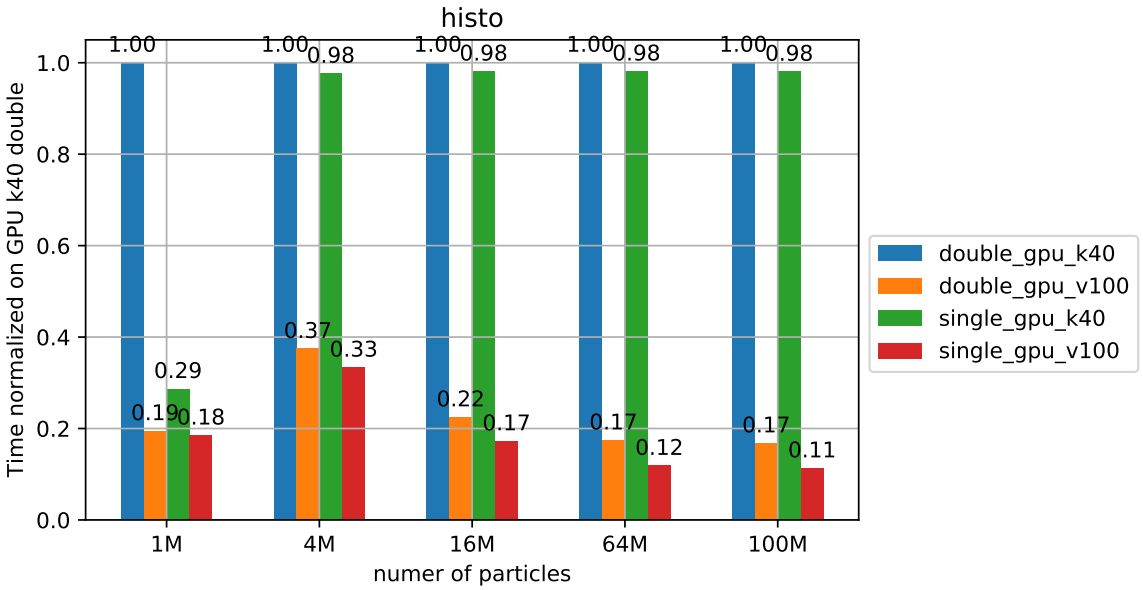


Figure 4.3: *Comparison between different GPUs for the histogram function*

4.1.2 Drift Kernel

Drift is a function that is applied to a beam object by a tracker object. It is used to update the particle arrival time to the RF station. Below you can see the equations they implement, one for each solver followed by the CPU source code.

$$\Delta t^{n+1} = \Delta t^n + \frac{L}{C} T_0^{n+1} \left[\left(1 + \sum_{i=0}^{2} \alpha_i (\delta^{n+1})^{i+1} \right) \frac{1 + (\Delta E / E_s)^{n+1}}{1 + \delta^{n+1}} - 1 \right] \quad \text{(exact)}$$

$$\Delta t^{n+1} = \Delta t^n + \frac{L}{C} T_0^{n+1} \left(\frac{1}{1 - \eta(\delta^{n+1})\delta^{n+1}} - 1 \right) \quad \text{(legacy)}$$

$$\Delta t^{n+1} = \Delta t^n + \frac{L}{C} T_0^{n+1} \eta_0 \delta^{n+1} \quad \text{(simple)}$$

$$\delta = \sqrt{1 + \beta_s^{-2} \left[\left(\frac{\Delta E}{E_s} \right)^2 + 2 \frac{\Delta E}{E_s} \right]} - 1 \quad \text{(exact)}$$

$$\delta = \frac{\Delta E}{\beta_s^2 E_s} \quad \text{(simple, legacy)}$$

```
void drift(double * __restrict__ beam_dt,
                    const double * __restrict__ beam_dE,
                    const char * __restrict__ solver,
                    const double T0, const double length_ratio,
                    const double alpha_order, const double eta_zero,
                    const double eta_one, const double eta_two,
                    const double alpha_zero, const double alpha_one,
                    const double alpha_two,
                    const double beta, const double energy,
                    const int n_macroparticles) {

    int i;
    double T = T0 * length_ratio;

    if ( strcmp (solver, "simple") == 0 )
    {
        double coeff = eta_zero / (beta * beta * energy);
        #pragma omp parallel for
        for (int i = 0; i < n_macroparticles; i++)
            beam_dt[i] += T * coeff * beam_dE[i];
    }

    else if ( strcmp (solver, "legacy") == 0 )
    {
        const double coeff = 1. / (beta * beta * energy);
        const double eta0 = eta_zero * coeff;
        const double eta1 = eta_one * coeff * coeff;
        const double eta2 = eta_two * coeff * coeff * coeff;

        if (alpha_order == 0)
            for ( i = 0; i < n_macroparticles; i++ )
                beam_dt[i] += T * (1. / (1. - eta0 * beam_dE[i]) - 1.);
```

```
        else if (alpha_order == 1)
            for ( i = 0; i < n_macroparticles; i++ )
                beam_dt[i] += T * (1. / (1. - eta0 * beam_dE[i]
                                        - eta1 * beam_dE[i] * beam_dE[i])
                                        - 1.);
        else
            for ( i = 0; i < n_macroparticles; i++ )
                beam_dt[i] += T * (1. / (1. - eta0 * beam_dE[i]
                            - eta1 * beam_dE[i] * beam_dE[i]
                            - eta2 * beam_dE[i] * beam_dE[i] * beam_dE[i])
                            - 1.);
    }

    else
    {

        const double invbetasq = 1 / (beta * beta);
        const double invenesq = 1 / (energy * energy);
        // double beam_delta;

        #pragma omp parallel for
        for ( i = 0; i < n_macroparticles; i++ )

        {

            double beam_delta = sqrt(1. + invbetasq *
                                (beam_dE[i] * beam_dE[i] * invenesq +
                                2. * beam_dE[i] / energy)) - 1.;

            beam_dt[i] +=
                        T *
                        ((1. + alpha_zero * beam_delta +
                        alpha_one * (beam_delta * beam_delta) +
                        alpha_two * (beam_delta*beam_delta*beam_delta)) *
                        (1. + beam_dE[i] / energy) / (1. + beam_delta) -
                        1.);
        }
    }
}
```

This GPU kernel is a simple one, and can't be optimized further.

```
extern "C"
__global__ void drift(double * __restrict__ beam_dt,
                      double  * __restrict__ beam_dE,
                      const int solver,
                      const double T0, const double length_ratio,
                      const double alpha_order, const double eta_zero,
                      const double eta_one, const double eta_two,
                      const double alpha_zero, const double alpha_one,
                      const double alpha_two,
                      const double beta, const double energy,
                      const int n_macroparticles)
{
    double T = T0 * length_ratio;
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    if ( solver == 0 )
    {
        double coeff = eta_zero / (beta * beta * energy);
        for (int i=tid; i<n_macroparticles; i=i+blockDim.x*gridDim.x)
            beam_dt[i] += T * coeff * beam_dE[i];
    }

    else if ( solver == 1 )
    {
        const double coeff = 1. / (beta * beta * energy);
        const double eta0 = eta_zero * coeff;
        const double eta1 = eta_one * coeff * coeff;
        const double eta2 = eta_two * coeff * coeff * coeff;

        if (alpha_order == 0)
            for (int i=tid; i<n_macroparticles; i=i+blockDim.x*gridDim.x)
                beam_dt[i] += T * (1. / (1. - eta0 * beam_dE[i]) - 1.);
        else if (alpha_order == 1)
            for (int i=tid; i<n_macroparticles; i=i+blockDim.x*gridDim.x)
                beam_dt[i] +=   T * (1. / (1. - eta0 * beam_dE[i]
                                - eta1 * beam_dE[i] * beam_dE[i])
                                - 1.);
        else
            for (int i=tid; i<n_macroparticles; i=i+blockDim.x*gridDim.x)
                beam_dt[i] +=   T * (1. /
                                (1. - eta0 * beam_dE[i]
                                - eta1 * beam_dE[i] * beam_dE[i]
```

```
                                - eta2 * beam_dE[i] * beam_dE[i] *
                                beam_dE[i])
                                - 1.);
    }

    else
    {

        const double invbetasq = 1 / (beta * beta);
        const double invenesq = 1 / (energy * energy);
        double beam_delta;


        for (int i=tid; i<n_macroparticles; i=i+blockDim.x*gridDim.x)
        {
            beam_delta = sqrt(1. + invbetasq *
                              (beam_dE[i] * beam_dE[i] * invenesq +
                              2.*beam_dE[i] / energy)) - 1.;

            beam_dt[i] += T *  ((1. + alpha_zero * beam_delta +
                              alpha_one * (beam_delta * beam_delta) +
                              alpha_two * (beam_delta * beam_delta *
                              beam_delta)) *
                              (1. + beam_dE[i] / energy) /
                              (1. + beam_delta) - 1.);
        }
    }
}
```

Below we can compare the performance of our devices in these kernels. Once again we test the sizes 1, 4, 16, 64 and 100 millions.

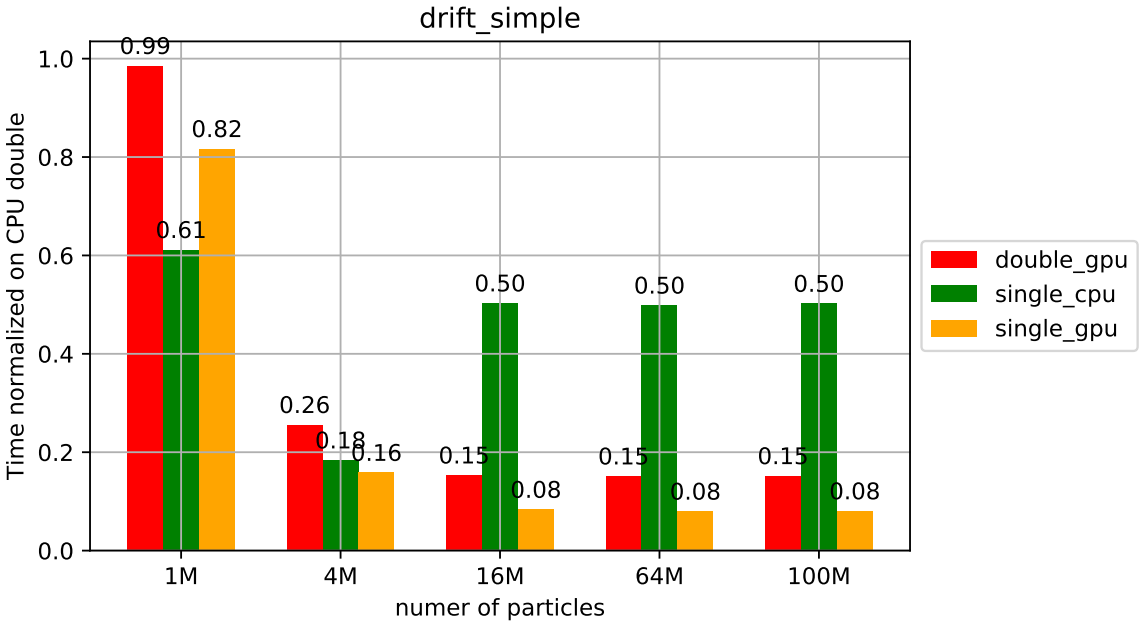


Figure 4.4: *Comparison of CPU and GPU for the drift function with the simple solver*

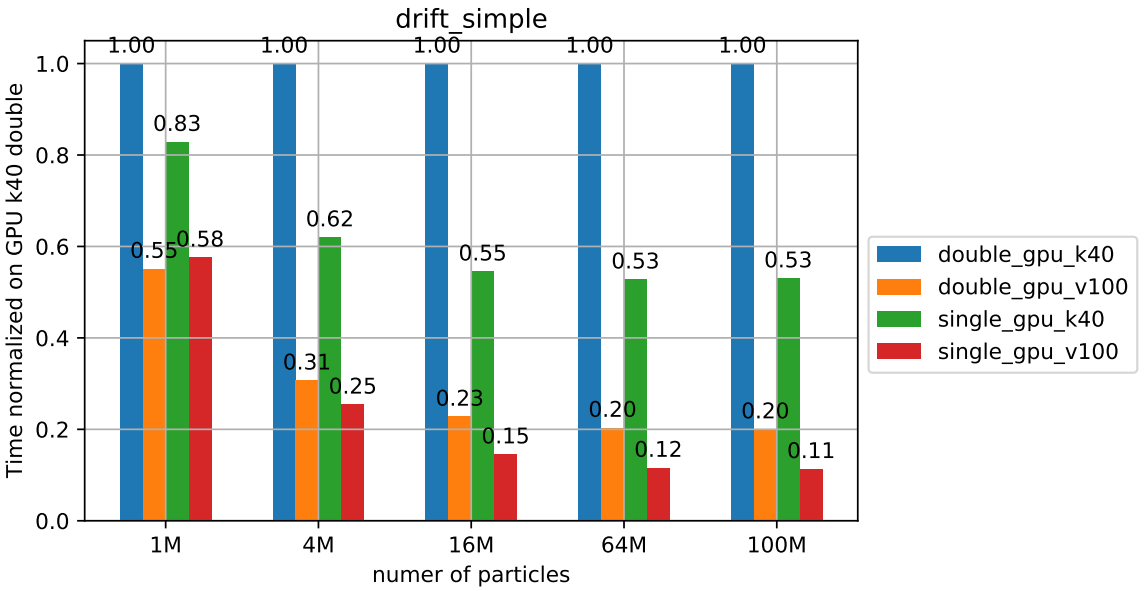


Figure 4.5: *Comparison between different GPUs for the drift function with the simple solver*

In figure 4.4 and 4.5, we can see that the simple version, because of the small number of computations per memory accesses, the speedup is limited. For small number of particles, the GPUs are underutilized, and so the speedup is also limited. As we increase the number of particles, the speedup is getting better. The same applies for the comparison between k40 and v100. As we increase the number of particles, v100 gets a lot better than k40. This happens for almost every benchmark, because v100, needs more particles to utilize the 80 SMs that it possesses.

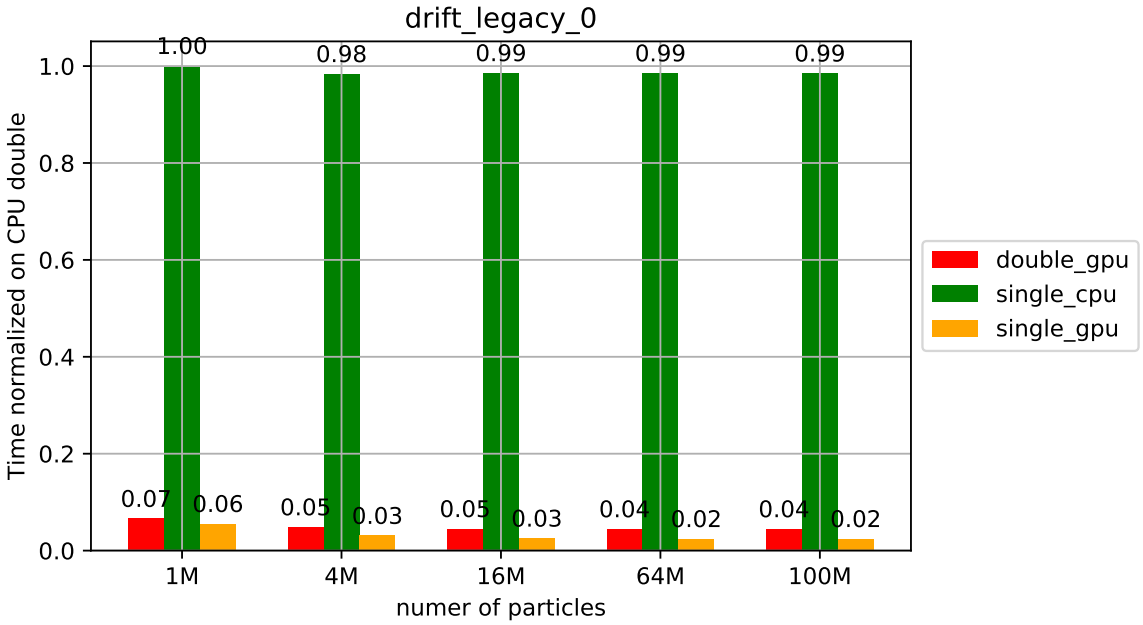


Figure 4.6: *Comparison of CPU and GPU for the drift function with the legacy solver with alpha order 0*

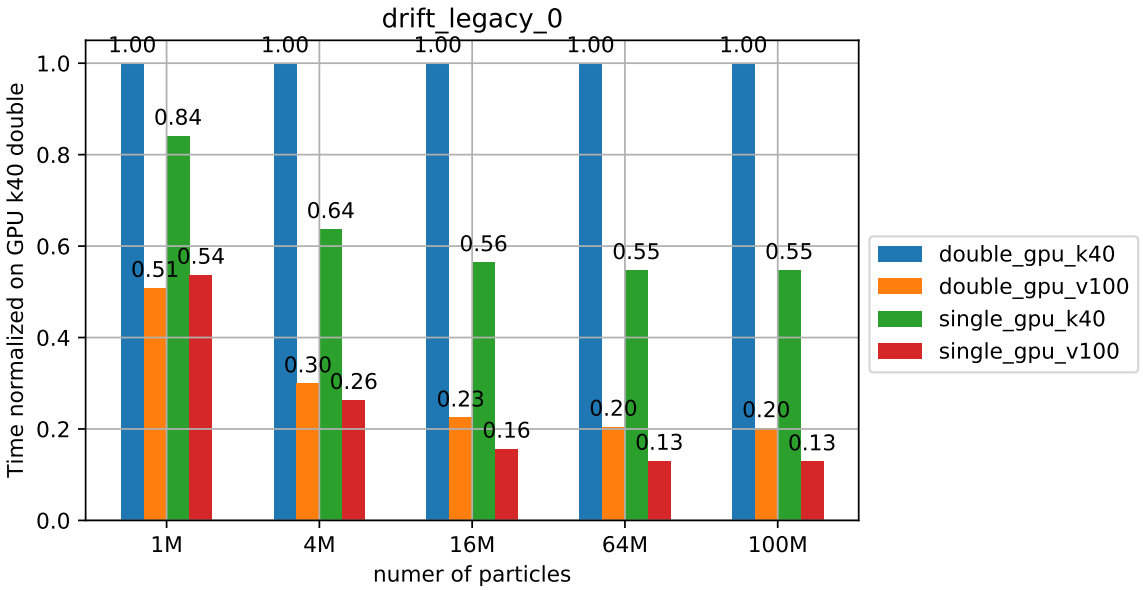


Figure 4.7: *Comparison between different GPUs for the drift function with the legacy solver with alpha order 0*

For the legacy solver, for every alpha_order values, the plots in figures 4.6 and 4.7 are almost identical. The GPU is a more than 20 times faster than the CPU, even for small sizes. Again, the V100 is 5 times faster than the k40.

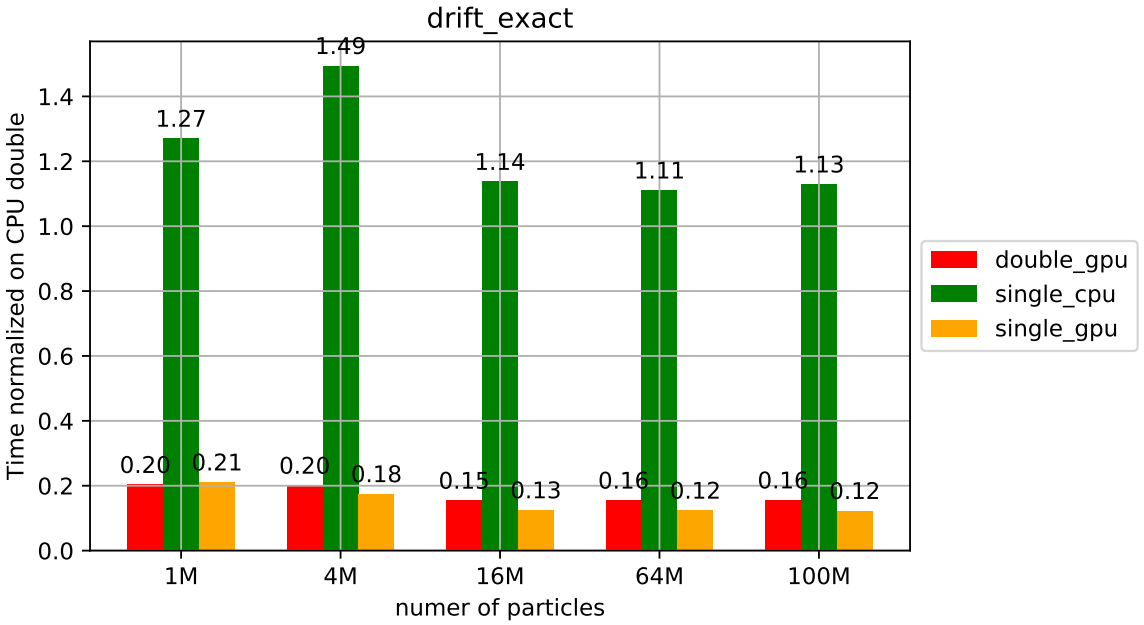


Figure 4.8: *Comparison of CPU and GPU for the drift function with the exact solver*

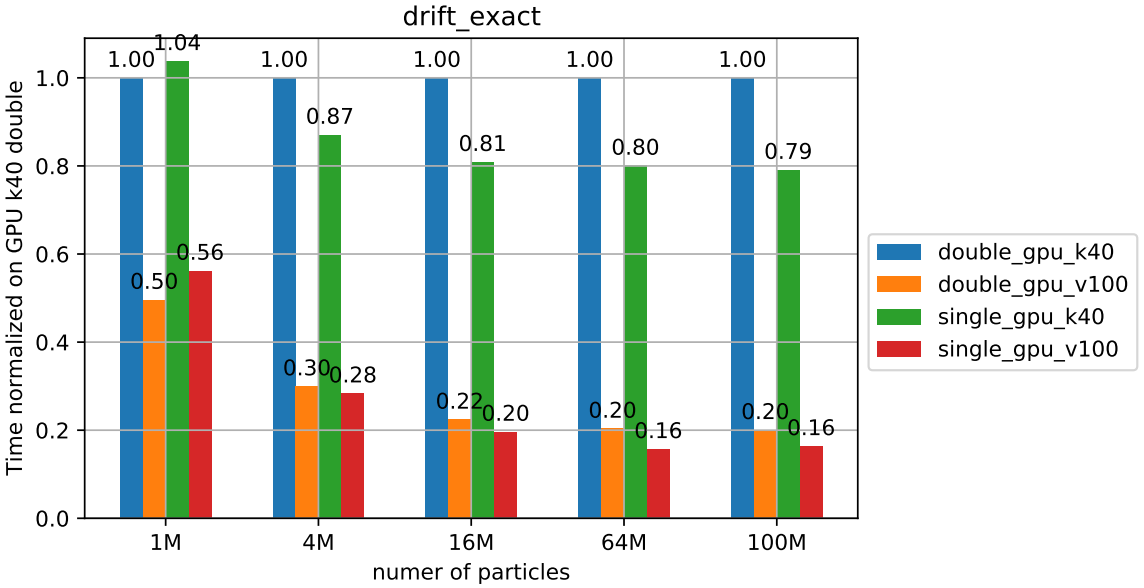


Figure 4.9: *Comparison between different GPUs for the drift function with the exact solver*

In this version, in figures 4.8 and 4.9 we can see that the comparison between V100 and k40, is almost the same as the previous versions. The k40 converges to being more than 5 times faster, than our CPU, at a small number of particles.

4.1.3 Kick Kernel

Kick function is used to update the particle energy due to the RF kick in a given RF station. The kicks are summed over the different harmonic RF systems in the station. The cavity phase can be shifted by the user via phi_offset. We can see the formula applied by the kick kernel below.

$$\Delta E^{n+1} = \Delta E^{n} + \sum_{k=0}^{n_{\mathrm{rf}}-1} eV_k^n \sin\left(\omega_{\mathrm{rf,k}}^n \Delta t^n + \phi_{\mathrm{rf,k}}^n\right) - \left(E_s^{n+1} - E_s^n\right)$$

This is the CPU source code that implements this.

```
 void kick(const double * __restrict__ beam_dt,
                                        double * __restrict__ beam_dE, const int n_rf
                                        const double * __restrict__ voltage,
                                        const double * __restrict__ omega_RF,
                                        const double * __restrict__ phi_RF,
                                        const int n_macroparticles,
                                        const double acc_kick){
int j;

// KICK
for (j = 0; j < n_rf; j++)
#pragma omp parallel for
                for (int i = 0; i < n_macroparticles; i++)
                                beam_dE[i] = beam_dE[i] + voltage[j]
                                                * fast_sin(omega_RF[j] * beam_dt[i]

// SYNCHRONOUS ENERGY CHANGE
#pragma omp parallel for
        for (int i = 0; i < n_macroparticles; i++)
                beam_dE[i] = beam_dE[i] + acc_kick;

}
```

We can see that it has a nested-loop, and a simple for-loop. The variable n_rf usually has small values, less than ten. The CUDA implementation is the following.

```
__global__ void kick(
        const double  *beam_dt,
        double  *beam_dE,
        const int n_rf,
        const double  *voltage,
        const double  *omega_RF,
        const double  *phi_RF,
        const int n_macroparticles,
        const double acc_kick
        )
{
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    double my_beam_dt;
    double sin_res;
    double dummy;
    for (int i=tid; i<n_macroparticles; i += blockDim.x*gridDim.x){
        my_beam_dt = beam_dt[i];
        for (int j=0; j<n_rf; j++){
            sincos(omega_RF[j]*my_beam_dt + phi_RF[j], &sin_res, &dummy);
            beam_dE[i] += voltage[j] * sin_res;
            }
        beam_dE[i] += acc_kick;
    }
}
```

We have used an optimization called Loop Interchange. You can see that we changed the order of the nested-loop, we have the n_macroparticles loop outside and the n_rf loop inside. This way we do not need the second for-loop.

Below we can compare the performance of our devices in these kernels. Once again we test the sizes 1, 4, 16, 64 and 100 millions.

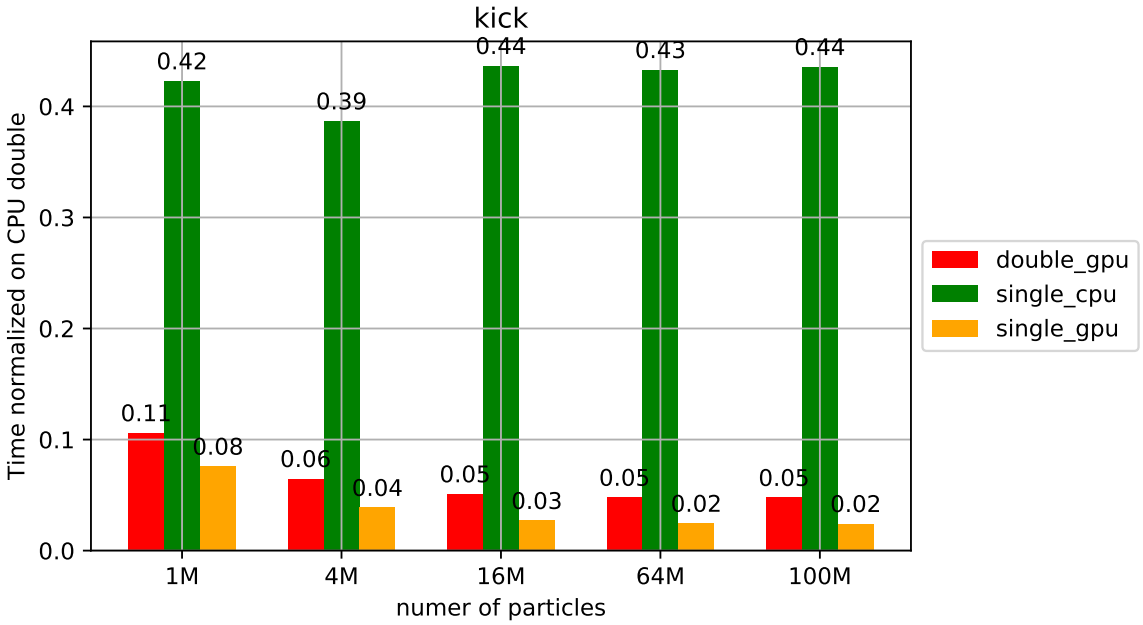


Figure 4.10: *Comparison of CPU and GPU for the kick function*

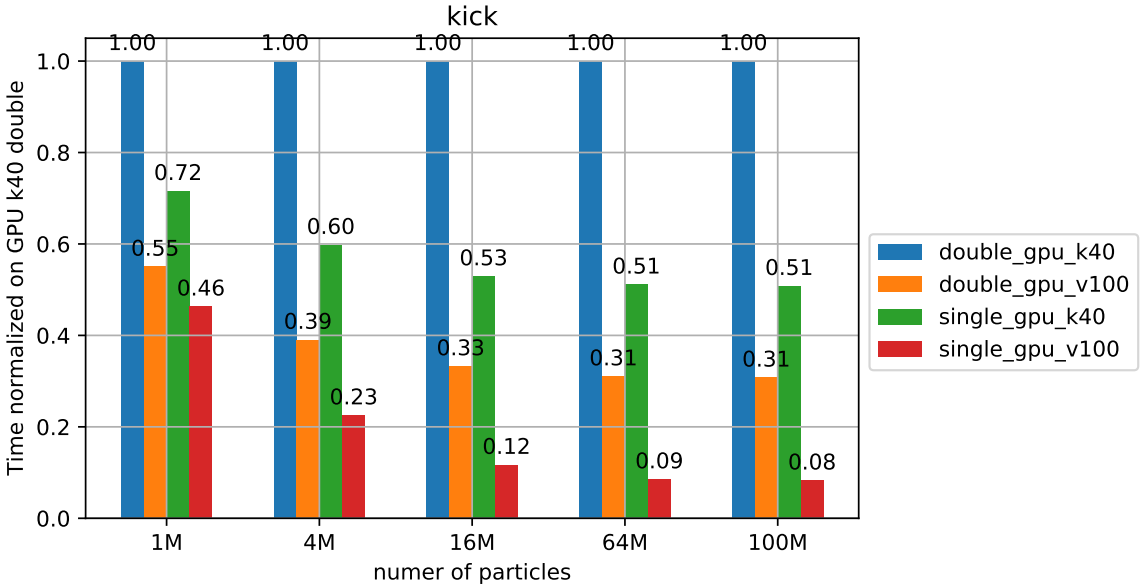


Figure 4.11: *Comparison between different GPUs for the kick function*

In figures 4.10 and 4.11, we can see that the GPU is again more than 10 times faster than the CPU. Also V100 is 3 times faster than k40 for bigger sizes, 30 times faster than our CPU in total. We can finally see that the single precision version is two times faster and more than the double precision. This happens because of sincos function which is operation heavy.

4.1.4 Linear Interpolation Kick Kernel

This kernel is used instead of the kick kernel when the linear_interpolation variable of the tracker object is set to True. Below you can see the implementation for the CPU.

```
extern "C" void linear_interp_kick(
                    double * __restrict__ beam_dt,
                    double * __restrict__ beam_dE,
                    const double * __restrict__ voltage_array,
                    const double * __restrict__ bin_centers,
                    const double charge,
                    const int n_slices,
                    const int n_macroparticles,
                    const double acc_kick)
{


    const int STEP = 64;
    const double inv_bin_width = (n_slices - 1)
                                / (bin_centers[n_slices - 1]
                                   - bin_centers[0]);

    double *voltageKick = (double *) malloc ((n_slices - 1) *
                                             sizeof(double));
    double *factor = (double *) malloc ((n_slices - 1) * sizeof(double));

    #pragma omp parallel
    {
        unsigned fbin[STEP];

        #pragma omp for
        for (int i = 0; i < n_slices - 1; i++) {
            voltageKick[i] =  charge * (voltage_array[i + 1] -
                              voltage_array[i]) * inv_bin_width;
            factor[i] = (charge * voltage_array[i] -
                        bin_centers[i] * voltageKick[i]) + acc_kick;
        }




        #pragma omp for
        for (int i = 0; i < n_macroparticles; i += STEP) {
```

```
            const int loop_count = n_macroparticles - i > STEP ?
                                   STEP : n_macroparticles - i;

            for (int j = 0; j < loop_count; j++) {
                fbin[j] = (unsigned) std::floor(
                                        (beam_dt[i + j] - bin_centers[0])
                                        * inv_bin_width);
            }

            for (int j = 0; j < loop_count; j++) {
                if (fbin[j] < n_slices - 1) {
                    beam_dE[i + j] += beam_dt[i + j] *
                    voltageKick[fbin[j]] +
                    factor[fbin[j]];
                }
            }

        }
    }
    free(voltageKick);
    free(factor);
}
```

In this function first of all we compute two new arrays, voltageKick and factor, that we are going to use in the big for-loop. Then each core computes the corresponding bins of its particles with the floor function. Finally it performs some operations on the beam_dE based on these bins. We are doing exactly the same operations in the GPU kernel which can be seen below.

```
extern "C"
__global__ void lik_copy(
    double * __restrict__ beam_dt,
    double * __restrict__ beam_dE,
    const double * __restrict__ voltage_array,
    const double * __restrict__ bin_centers,
    const double charge,
    const int n_slices,
    const int n_macroparticles,
    const double acc_kick,
    double * __restrict__ glob_voltageKick,
    double * __restrict__ glob_factor
)
{
```

```
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    double const inv_bin_width = (n_slices - 1)
                                 / (bin_centers[n_slices - 1]
                                 - bin_centers[0]);


    for (int i = tid; i < n_slices - 1; i += gridDim.x * blockDim.x) {
        glob_voltageKick[i] = charge *
                              (voltage_array[i + 1] - voltage_array[i]) *
                              inv_bin_width;
        glob_factor[i] =     (charge * voltage_array[i] -
                             bin_centers[i] * glob_voltageKick[i])
                             + acc_kick;
    }
}
```

In the above kernel we create the two arrays voltageKick and factor just like in the CPU function. We have to do it in a different kernel because the threads of different blocks do not have a way to synchronize. As a result these arrays that are used by all threads must be calculated before the main kernel. One approach that we tried is to put these arrays in the shared memory. This way we would need only one kernel. Also accessing elements in the shared memory would be much faster than accessing them from the global memory. But there are some problems with that idea. First of all, shared memory is of limited capacity, the most recent GPUs have 49152 bytes per block, 12.288 integers or 6.144 doubles. In experiments with more bins than the number that can fit, we will again need two kernels and we will also have branch divergence, because we have two cases, either using the shared memory array or the global memory array. Since we do not have a lot of experiments with this small number of bins, we do not use this version.

Below we can see the main kernel.

```
extern "C"
__global__ void lik_comp(
    double * __restrict__ beam_dt,
    double * __restrict__ beam_dE,
    double * __restrict__ voltage_array,
    double * __restrict__ bin_centers,
    const double charge,
    const int n_slices,
    const int n_macroparticles,
    const double acc_kick,
    const double * __restrict__ glob_voltageKick,
    const double * __restrict__ glob_factor)
{
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
```

```
    double const inv_bin_width = (n_slices - 1)
                                 / (bin_centers[n_slices - 1] -
                                 bin_centers[0]);
    int fbin;
    const double bin0 = bin_centers[0];
    for (int i = tid; i < n_macroparticles; i += blockDim.x * gridDim.x) {
        fbin = floor((beam_dt[i] - bin0) * inv_bin_width);
        if ((fbin < n_slices - 1) && (fbin >= 0))
            beam_dE[i] += beam_dt[i] * glob_voltageKick[fbin] +
            glob_factor[fbin];
    }
}
```

Keywords like __restrict__ and const

In some kernels like this one, some values are being reused. In order to take advantage of that, a good strategy would be to keep them cached. In the Cuda Programming Guide, it is mentioned that: *"Data that is read-only for the entire lifetime of the kernel can also be cached in the unified L1/texture cache. When the compiler detects that the read-only condition is satisfied for some data, it will cache it. The compiler might not always be able to detect that the read-only condition is satisfied for some data. Marking pointers used for loading such data with both the const and __restrict__ qualifiers increases the likelihood that the compiler will detect the read-only condition."* To achieve that, while declaring arrays that we want to cache we are using these keywords.
In figures 4.12 and 4.13, we can compare our devices.

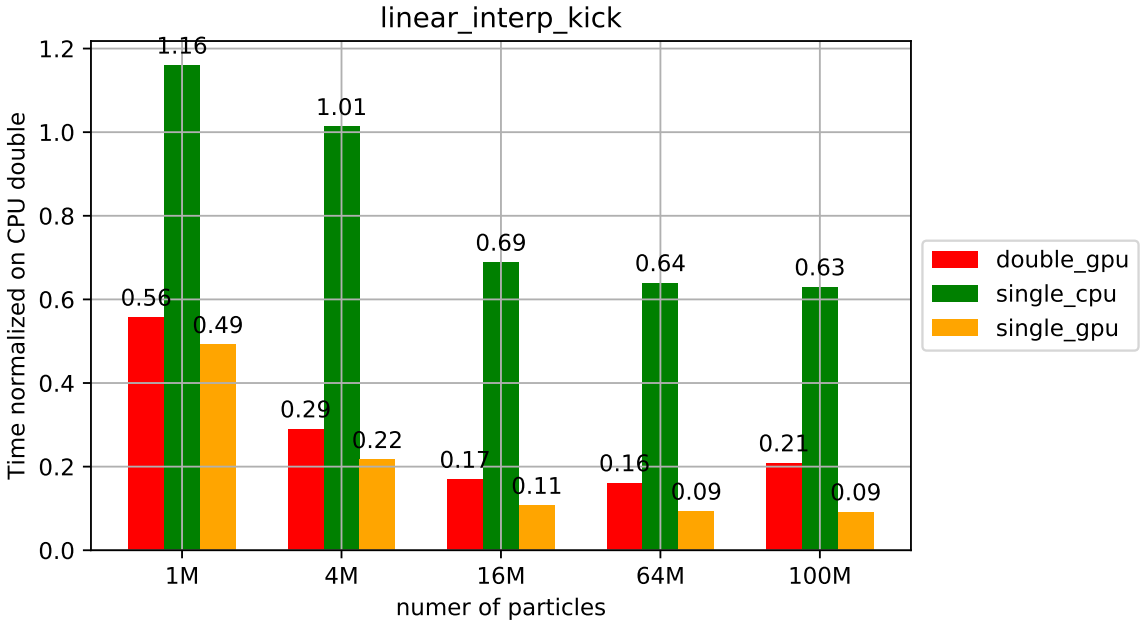


Figure 4.12: *Comparison of CPU and GPU for the linear_interpolation_kick function*

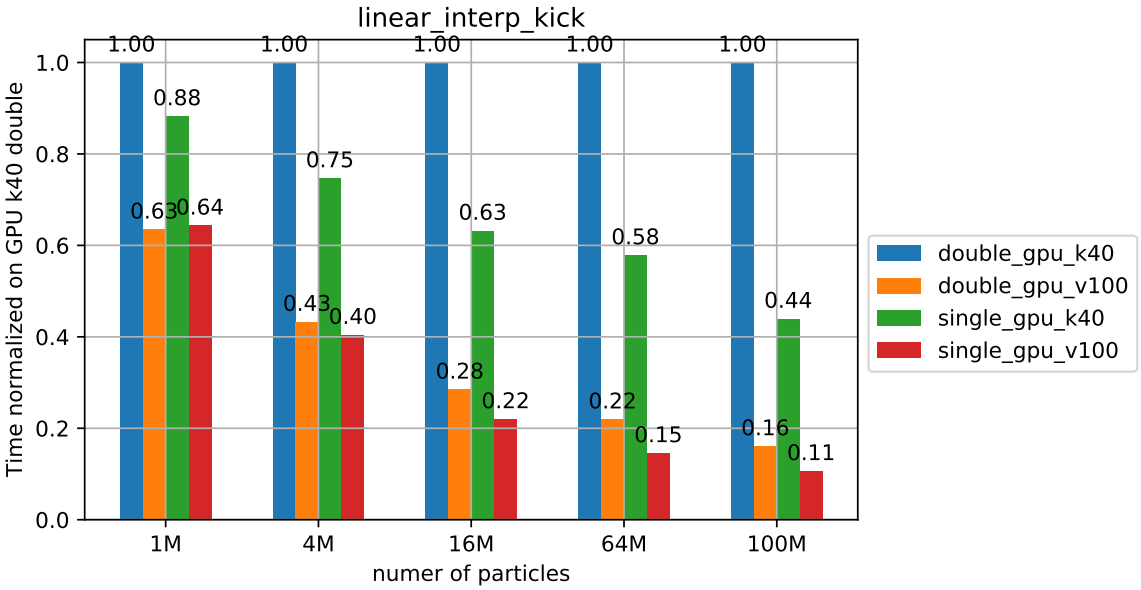


Figure 4.13: *Comparison between different GPUs for the linear_interpolation_kick function*

In this kernel the speedup of the GPU is lower because the ratio of computations to memory operations is lower. The double precision of k40 is 5 times faster than the CPU for big sizes. Also for both CPU and GPU the single precision version speedup is converging to two as we increase the size. Increasing the size means less conflicts in the bins so less time for memory operations. Finally once again the V100 is five times faster than the k40 for greater sizes, where the runtime of the kernel is dominated by computations.

4.1.5 FFTs

There are also two other functions that take a lot of time to produce results. These functions are rfft and irfft. The rfft function produces the real Fast-Fourier-Transform of an array, and the irrft produces the inverse-real Fast-Fourier-Transform of an array. These two functions have two inputs. The first input is an array, and the second is the size of the input that we want to use, with default value, the size of the first argument. These two arguments will create the input that will be passed to the rfft, and irfft functions that are provided by the module scikit-cuda and use the cufft, which is library of CUDA. So, we have some functions we have written ourselves that are called gpu_rfft and gpu_-irfft that are using scikit-cuda's rfft and irfft. Our functions are developed in a way that gpu_rfft's and gpu_irfft's results are identical to numpy's rfft and irfft.

Supposing that the first argument is input_array and second argument is n, we have the following three cases.

- n is None, we give input_array as input to the corresponding scikit-cuda function.
- n < len(input_array), we pass the input_array[:n] as input to the corresponding scikit-cuda function
- n > len(input_array), we pass the input_array padded with zeros as input to the corresponding scikit-cuda function

For these functions we have two optimizations. First of all, in order to call the functions provided by scikit-cuda, you need to create a plan. Since these plans are frequently reused, we create a dictionary and save them there. With this dictionary, we save time by not creating new plans, if they already exist. Also, there is a minor problem with the second and the third case. For these two cases, where the input of gpu_(i)rfft is different than the input of (i)rfft, we will need to allocate and delete a temporary array, the input of (i)rfft. Also the result array needs to be stored in a new array. In order to save time, we have developed a new feature, called gpu_cache, that stores temporary arrays, and reuses them. It will be described in the next subsection, but since it helps a lot these two functions you can see some results here too.

Now we will show some results of these two functions for our devices. In the following plots we compare the FFTs of the fftw library than can be used in blond, instead of numpy methods. It is also important to know that these functions usually take the histogram of dt as the first argument and a greater number than the length of histogram as the second argument.

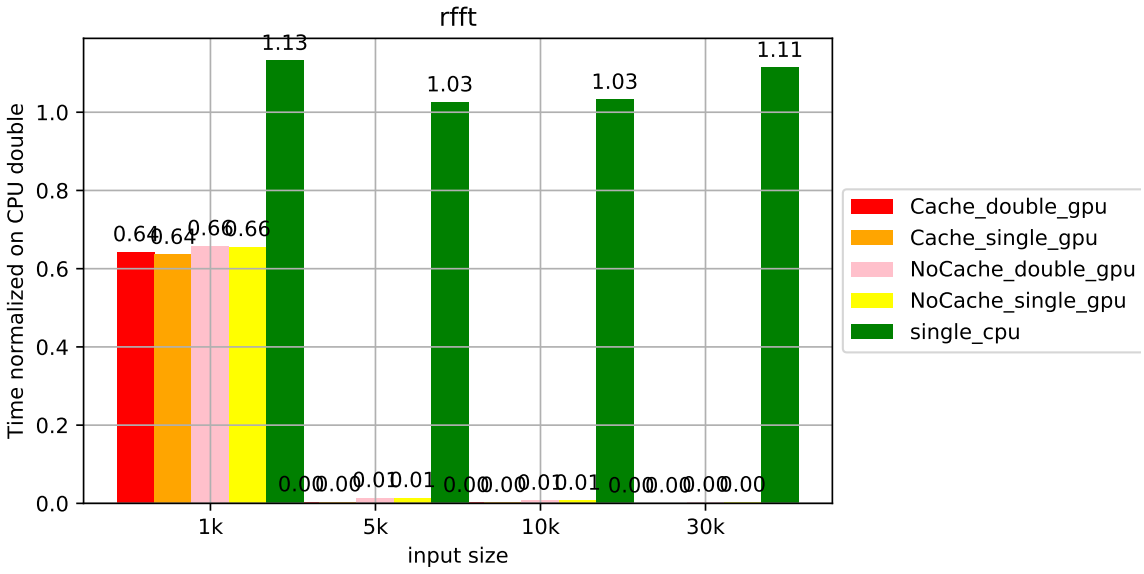


Figure 4.14: *Comparison of GPU versions for the rfft function*

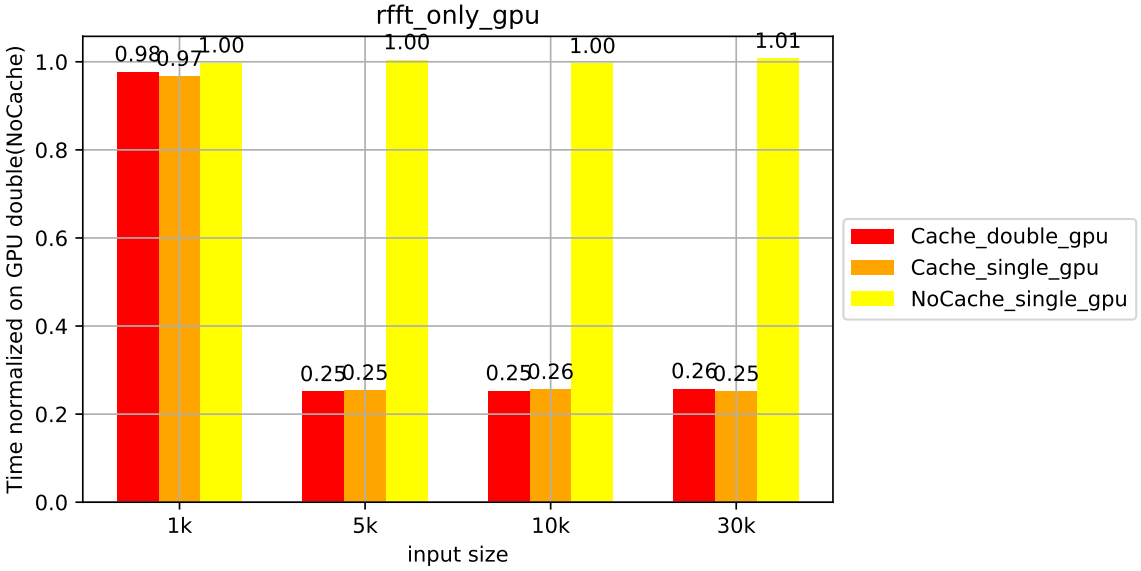


Figure 4.15: *Comparison of GPU versions for the rfft function*

In figure 4.14 plot we compare k40 with our CPU. We can see that for greater sizes, which are typically used, the speedup is more than 100. The GPU is clearly faster for these operations. This is something that can be also seen in experiments. Also, we observe that when we use our gpu_cache optimization we are four times faster than without it. Additionally, single precision is not helping in this case for both the CPU and the GPU as we see in figure 4.15.

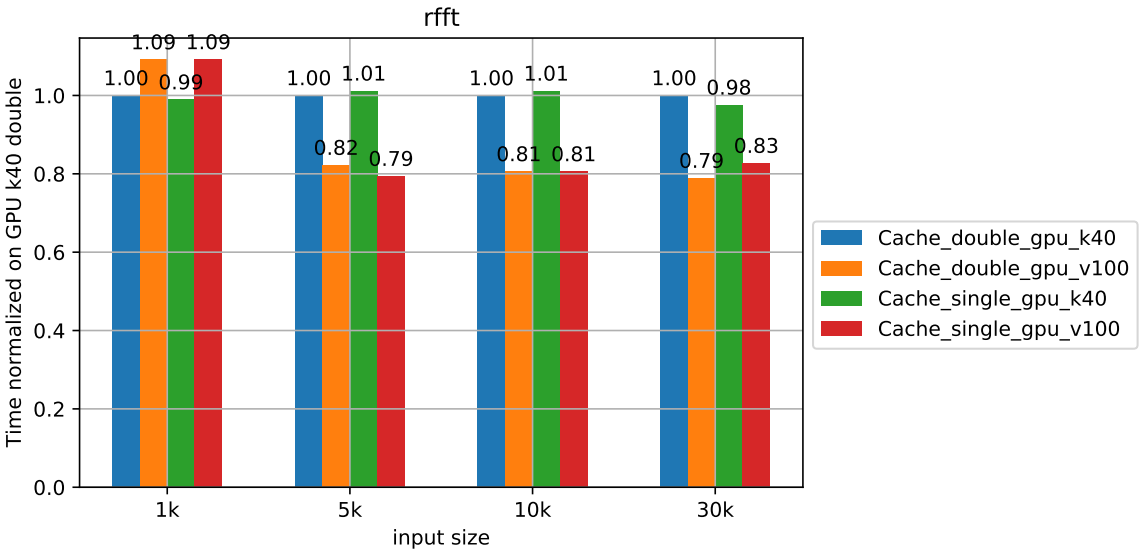


Figure 4.16: *Comparison of GPU devices for the rfft function*

Finally in figure 4.16 we can see that V100 is not boosting the performance like in the previous cases since we only have 1.25 speedup for both single and double precision. This is happening because these algorithms, have low occupancy in order to achieve good performance. As a result, V100 cant take advantage of its 80SM's and the performance boost is smaller.

All of the above apply also for the irfft function. We can see that in figures 4.17, 4.18 and 4.19.

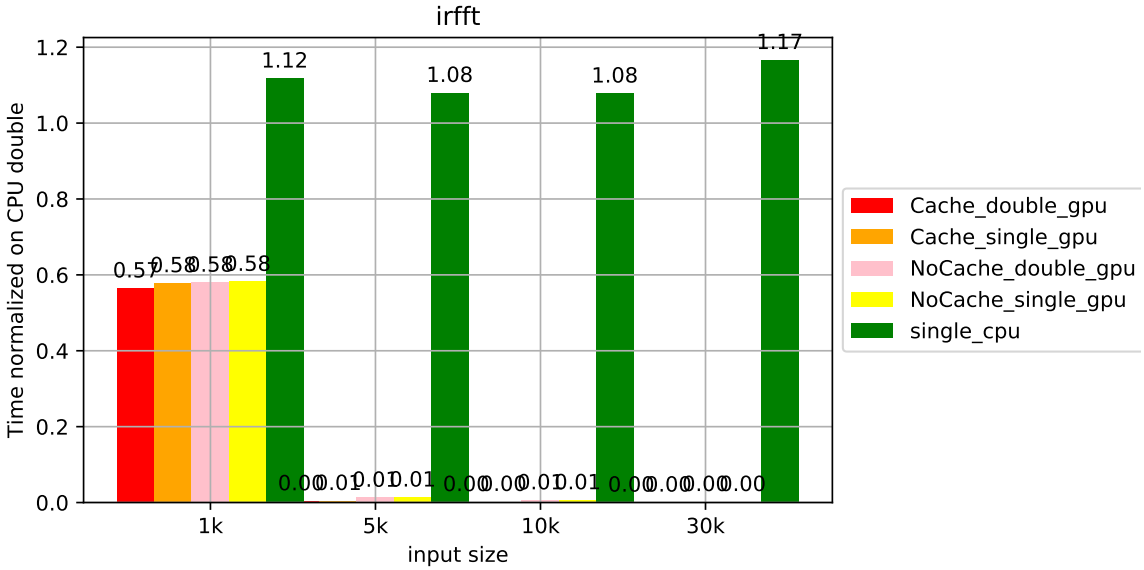


Figure 4.17: *Comparison of GPU versions for the irfft function*

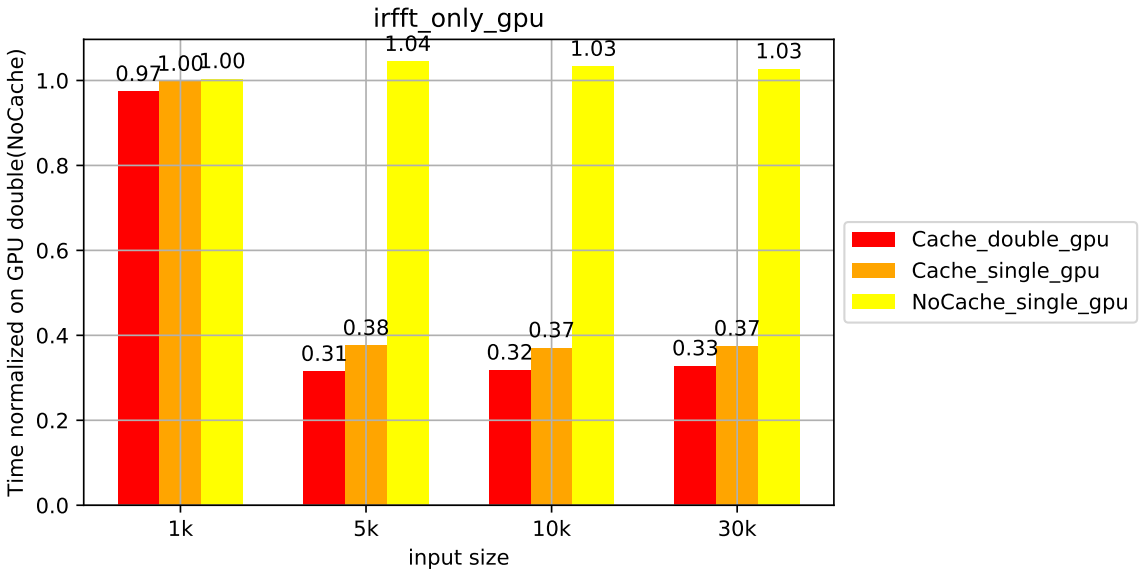


Figure 4.18: *Comparison of GPU versions for the irfft function*

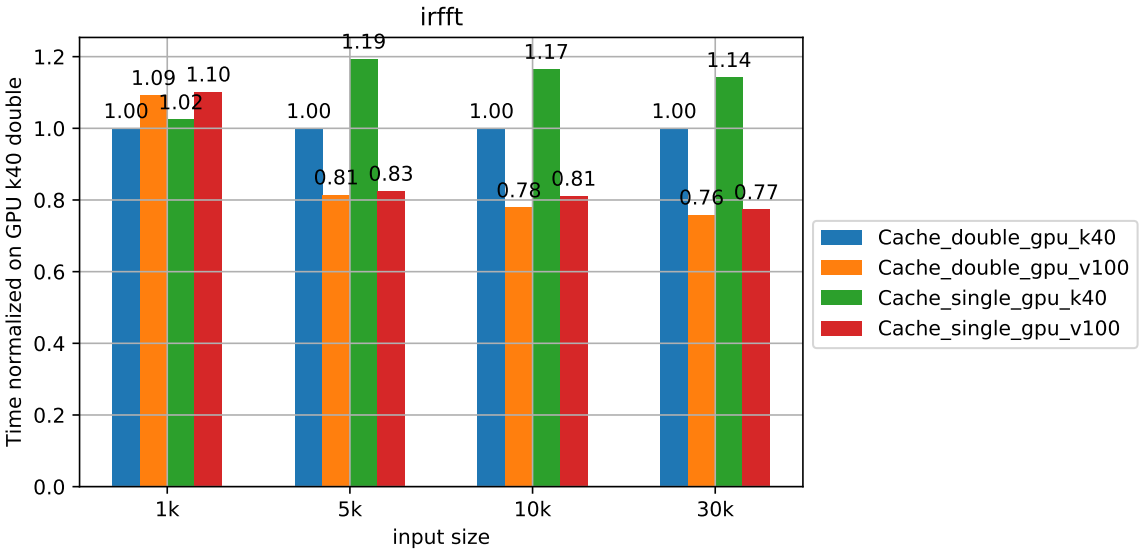


Figure 4.19: *Comparison of GPU devices for the irfft function*

4.2 Selecting Grid Parameters

In order to use our kernels, we need to select grid and block size. These two sizes will determine the occupancy of our GPU and how much work a single thread will do. In order to keep the occupancy at 100% we need to have 2048 threads in each Streaming Multiprocessor. Also each block can take at most 1024 threads for most GPUs. So a combination that is easy to use, and will work for almost every GPU is to use block size equal to 1024, and grid size equal to two times the number of SM's . This way we will have two blocks per SM, so 2048 threads per SM, maximizing its occupancy. There are a few exceptions that will not work with these parameters. One example is the optimized histogram kernel. In this kernel, it would seem good to use more blocks and smaller ones, to utilize share memory more. Although this approach does not seem to work. Something we have seen from experience, is that setting the grid size higher than the value our strategy would give, is beneficial when we have a very high number of bins. For the other kernels, just maximizing the occupancy is enough.

Our strategy is not always optimal. Actually, no combination of grid and block size can be optimal for all sizes. We can see that in figures 4.20, 4.21, 4.22 and 4.23. We have tried different sizes for each kernel and we can see that for each size the optimal combination is different.

Histogram

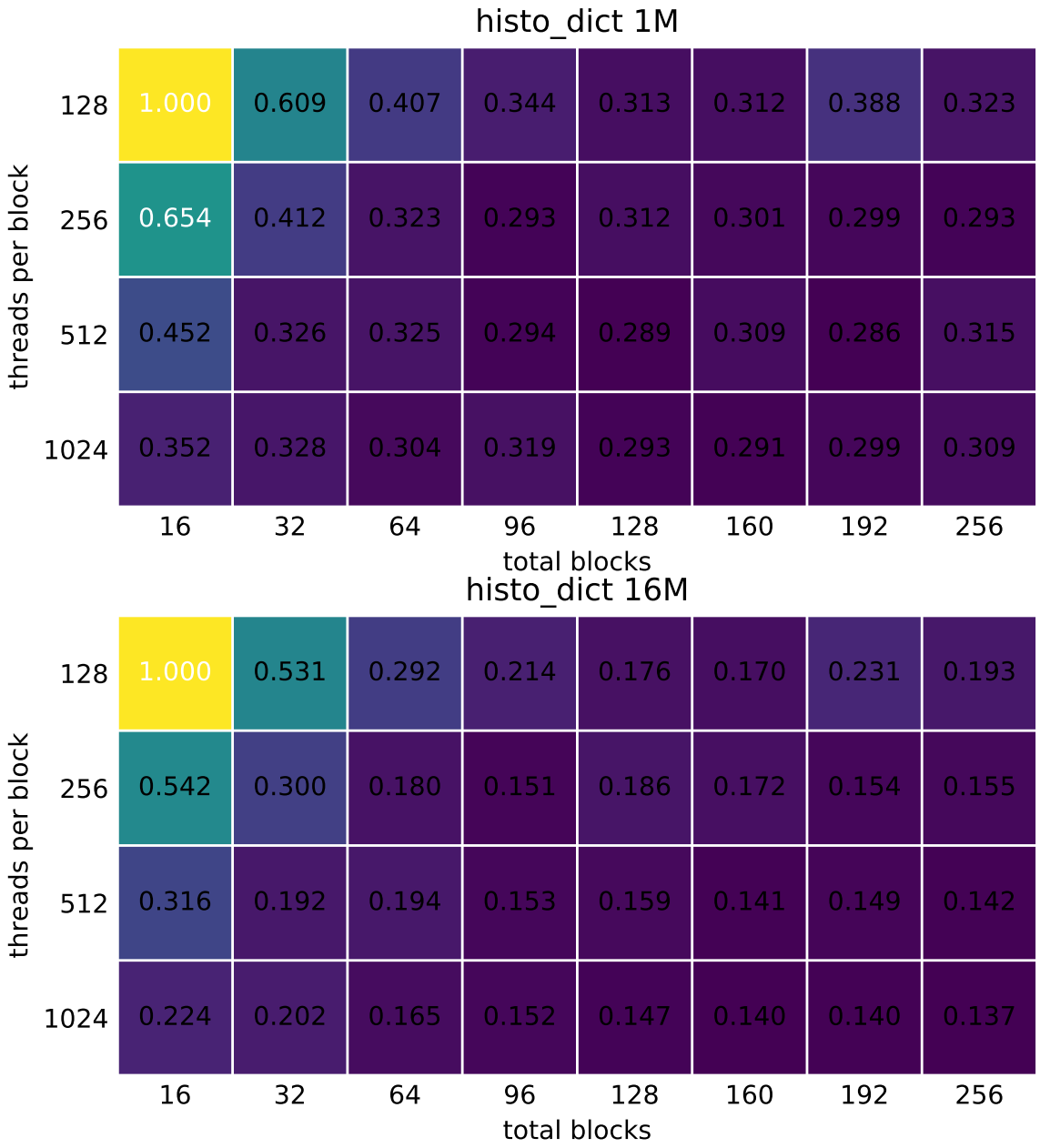


Figure 4.20: *Histogram Grid Parameters*

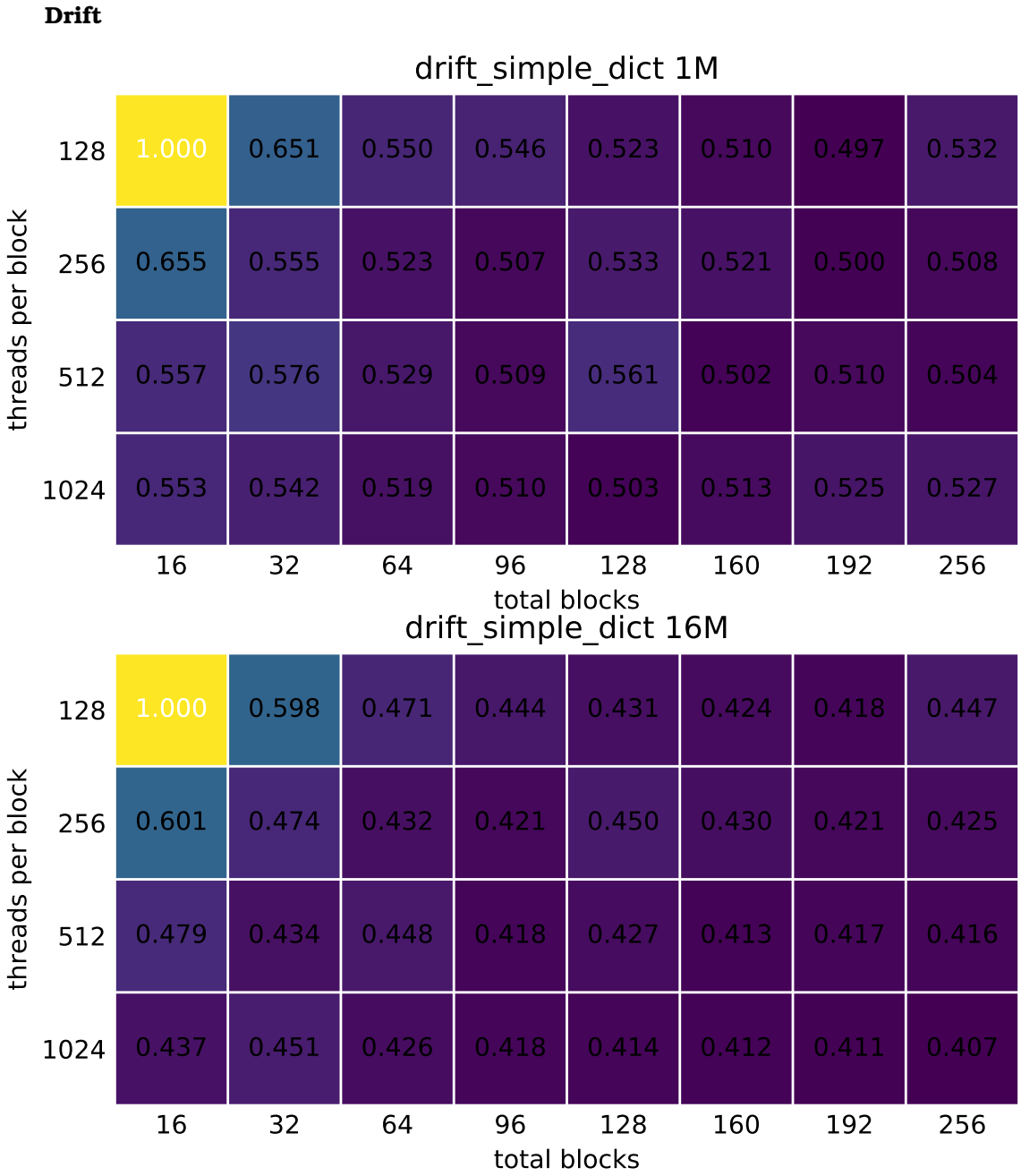


Figure 4.21: *Drift Grid Parameters*

Kick

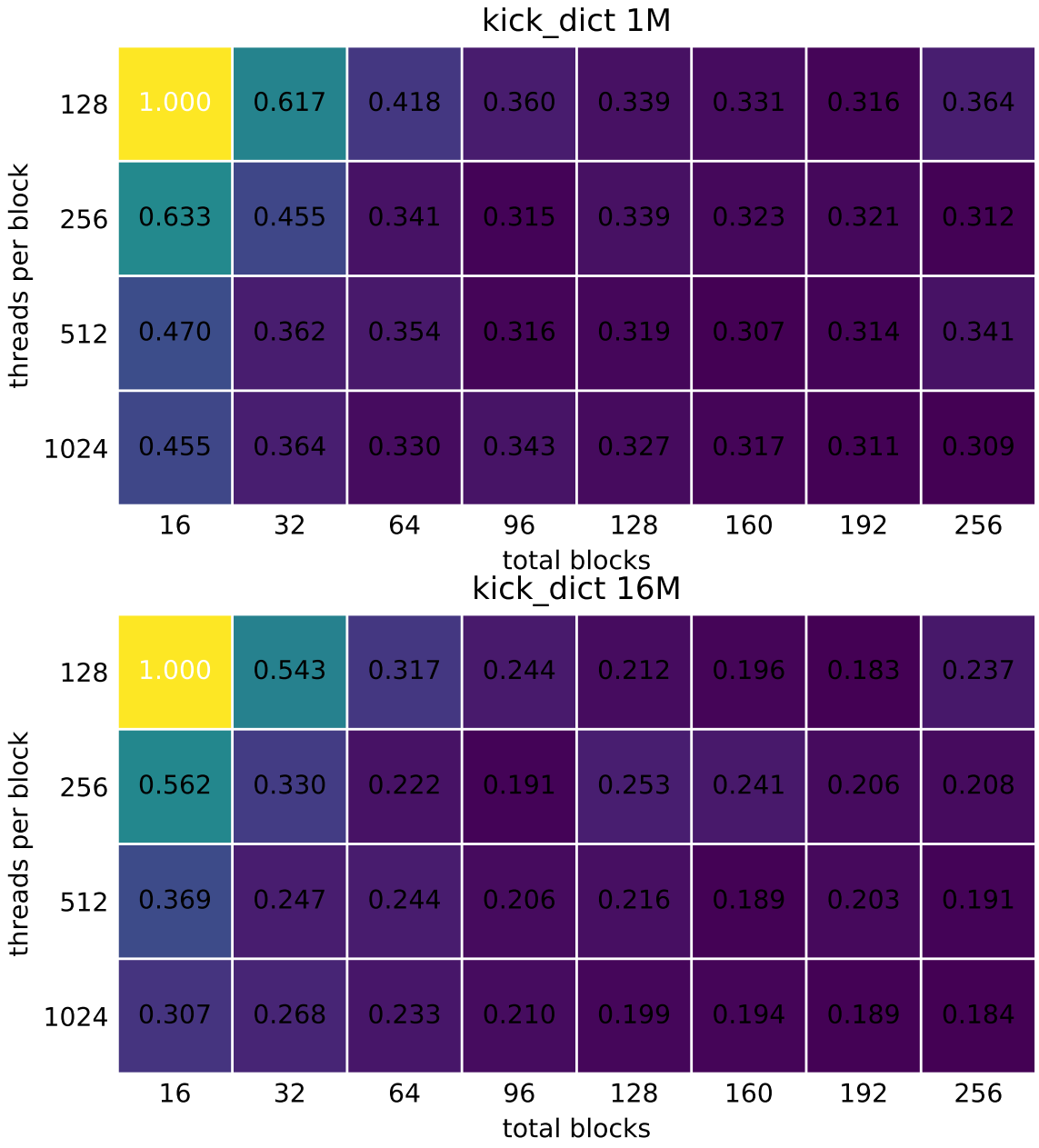


Figure 4.22: *Kick Grid Parameters*

Linear Interpolation Kick

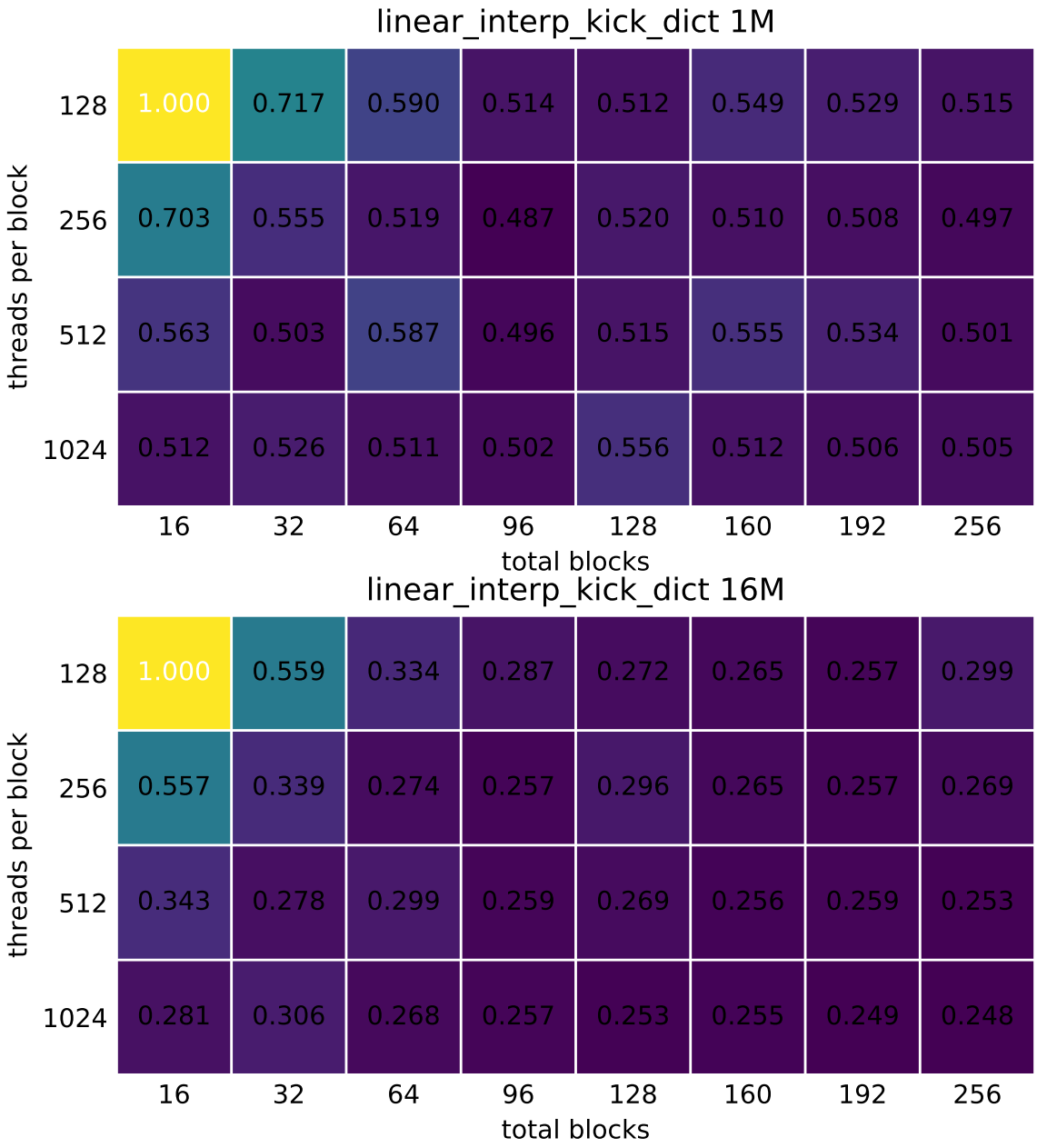


Figure 4.23: *Linear Interpolation Kick Grid Parameters*

In the above plots, we can see that even for the same kernel, there is no optimal configuration. From what we have seen in practice, using 30 blocks and 1024 threads is really good in experiments. There is only one exception, the histogram kernel in one of our benchmarks. For this case only, if we use more blocks, we can see better performance.

4.3 Gpu_Cache

Like described before, in the ffts, we have developed a feature to remove allocation and deletion time for temporary arrays. So in order to allocate a new array, we do not use the method of pycuda *gpuarray.empty()*, but a function like *gpu_cache.get_gpuarray(key)*. The key has information about the type of the array, the size, and the object that is going to use it. Also there is one string that helps to distinguish arrays of the same object that have the same size and type. To implement this we use a dictionary that maps keys to gpuarrays, and this way, we keep them in scope. This is an example of how we use our gpu_cache:

```
def assign_values(obj, value_for_a, value_for_b, new_size):
    obj.gpuarray_a = gpu_cache.get_gpuarray(('np.float64',
                                             new_size, id(obj), 'a'))
    obj.gpuarray_b = gpu_cache.get_gpuarray(('np.float64',
                                             new_size, id(obj), 'b'))
```

In the example above we have a method that changes the values of gpuarray_a and gpuarray_b fields of an object. First of all let's suppose that this function is called with new_size equal to 20. Since neither ('np.float64', new_size, id(obj), 'a') nor ('np.float64', new_size, id(obj), 'b') exist in our cache, they are being created with *gpuarray.empty()*, that is provided by pycuda. After a second call , these gpuarrays would be out of scope and deleted, but with the dictionary of our cache we keep them in our memory. So if a third call is made with the new_size equal to 20, just like the first time, the arrays from the cache are going to be reused. We are using the id of an obj to distinguish arrays with the same characteristics that belong to different objects, but if an array is temporary and different objects can use it we put a default value like 0. In this example you can also see the strings 'a' and 'b' that make our cache understand that these arrays are different.

A feature of our gpu_cache is its capacity. We can specify the capacity of our cache, and if the addition of a new array makes the cache surpass its limit, the dictionary will be cleared, and then our new array will be allocated and saved to it. Someone can choose to not specify the capacity, and the cache will be cleared when the GPU can't allocate a new array.

The gpu_cache is used mostly for the FFTs, where we need new temporary arrays that will not continue to exist after their usage. In small experiments, the FFTs have a big percentage of time, so improving their performance with the gpu_cache is important. Below we can see how the gpu_cache optimization reduces the duration of some experiments. To show the improvement, we have tested some examples and we present the results below in figures 4.24, 4.25 and 4.26.

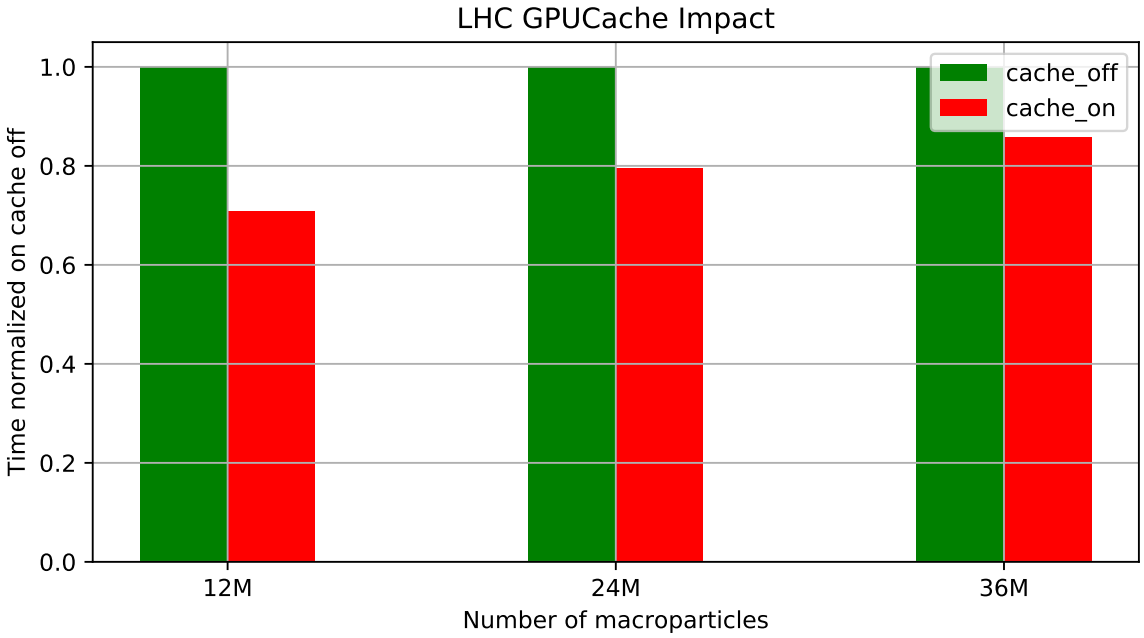


Figure 4.24: *GPU cache on LHC*

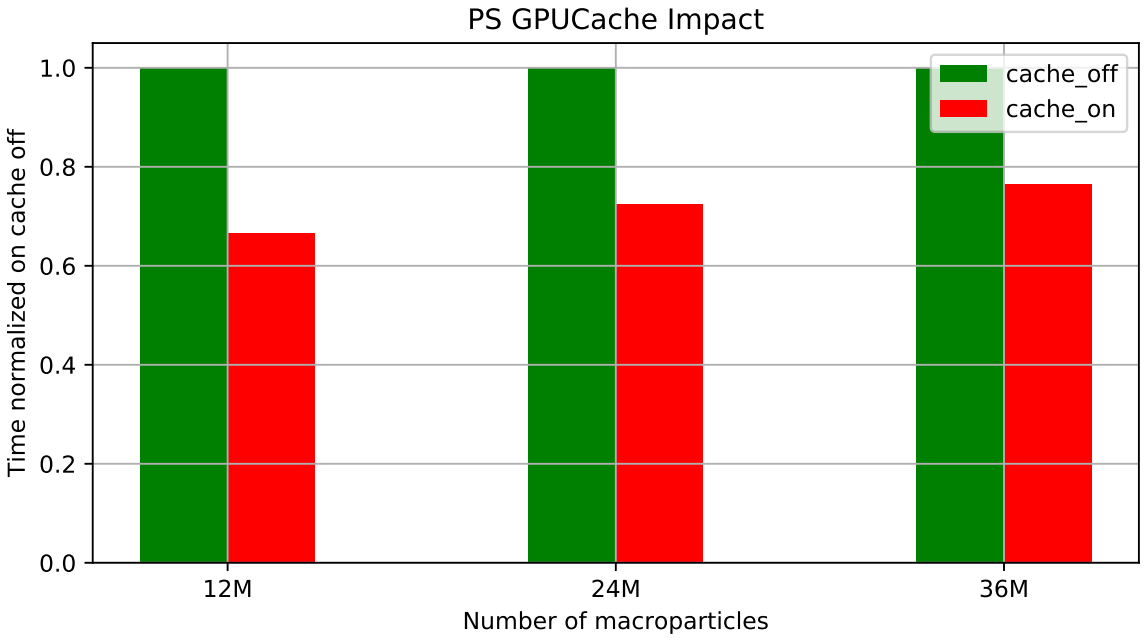


Figure 4.25: *GPU cache on PS*

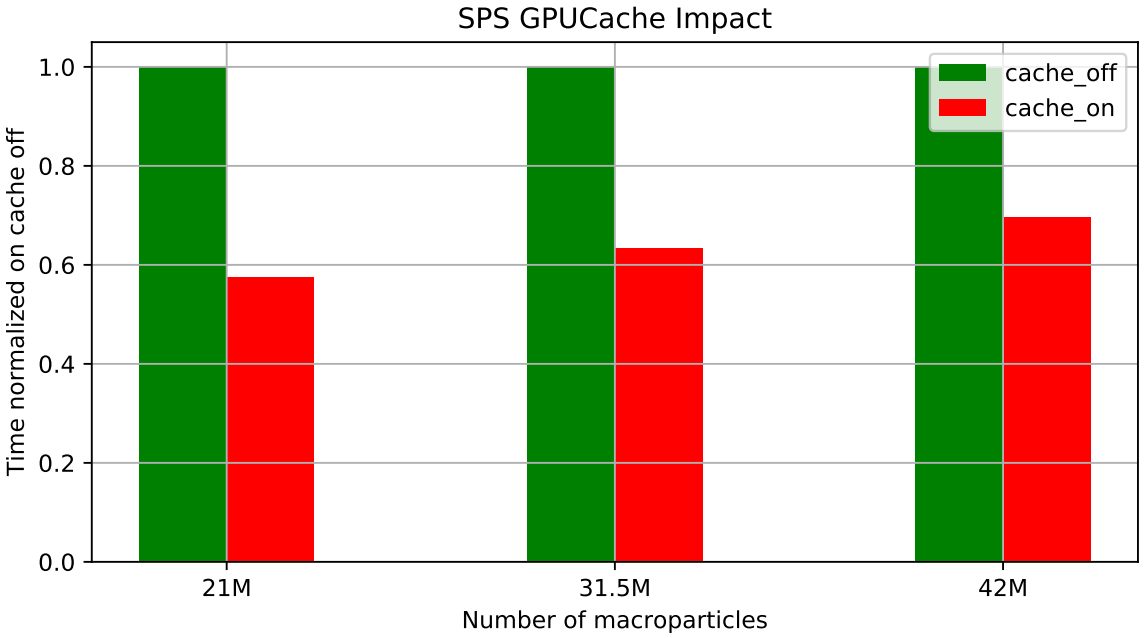


Figure 4.26: *GPU cache on SPS*

From the results we can see that gpu_cache is a really nice feature to have, especially for small experiments. For every experiment we reduce the time to less than 80% of the initial. Like we mentioned before, for simulations with more particles, where the runtime is dominated by linear_interpolation_kick and drift kernel, the performance boost we get with this optimization drops. If with the number of particles, we also increase the number of bins, then we still expect to see a good speedup, since the time of the ffts is dependent on the number of bins.

Chapter 5

Enabling GPU

In this chapter, after we describe the basic components of a BLonD simulation with more details, we will explain how we use the GPU to accelerate simulations. Also we will describe the CGArray object that really helps physicists to develop their simulations, without having to know how to use the GPU.

5.1 Description of BLonD simulation

We will try to mention and explain some of the basic BLonD classes, and their attributes.

Beam

This class is one of the most basic classes of the simulation. The most important attributes are the dE and the dt arrays. There is also an id array, which has information on which particles are lost. A particle is considered lost, if its coordinates do not meet some requirements. It also has some methods to check these requirements and adjust the id array. Finally, we have methods there, that compute the mean value and the std of our dt and dE coordinates.

Profile

In this class, we have information and methods about the histogram of dt. It gets a Beam object as a parameter, so it can access the dt array. The basic method of this class is slice, that computes the histogram of dt, that is saved in the attribute n_macroparticles. It has some parameters for this method like cut_right and cut_left, that set the boundaries of the histogram. One other attribute is the bin_centers array, that has the centers of each bin. Some other methods that we have here are the beam_profile_derivative that computes the derivative of the histogram and the beam_spectrum_generation that computes the rfft of the histogram.

Tracker

A lot of operations during the main loop are happening with methods of this class. The most important method of this class is the track method, that has two parts. The first part, that we call it pre_track has some functionalities that are used to update some

values, used for the next part, which we call it track_only. In this next part, we apply kick or linear_interpolation_kick and drift function, to our beam.

TotalInducedVoltage

A lot of classes inherit from this class. Each one, has its own functionality, but all of them are updated from the histogram, and have methods and attributes that are used in the tracking process. These objects use a lot the FFTs function from their methods, and in small experiments, the time consumed by the CPU for these methods, is a big percentage of the runtime (up to 60%).

RFStation

This class contains some arrays as attributes, that are also used in the tracking process. We have no important methods in this class. Only its arrays, that are used by almost every other class.

bmath.py file

This python file is mostly used to apply optimizations. It has a lot of useful features. First of all it has a global dictionary, that is used to map strings to methods. This way, we can change a method implementation, while calling the method with the same name. Also, in this file we have a class that let us to switch to single or double precision and methods that let the user enable the GPU and the GPU_cache.

5.2 GPU Corresponding Classes

The objects that were described in the previous section have some attributes, arrays or constant values, and some methods. In order to use GPU acceleration, the methods that consume time, must be ported in the GPU. In order to do that, we have created a new directory under blond, which is called gpu, that implements that and its structure will be explained below.

Each file, that describes a class of the CPU implementation, that is used in the GPU has a corresponding copy in this folder. For example for the beam.py file there is a gpu_beam.py under the gpu directory. Such files are: beam, beamfeedback, impedance, profile, rf_parameters and tracker. We also have two more files, butils_wrap and physics_wrap with some core functions of blond. Additionally there is an __init__.py file, that we use to specify the grid parameters that will be used for our kernels. We have a file for the gpu_cache implementation and one file for the CGA class which we will see later, and finally one file called gpu_activation that will also be explained later.

We will explain what is happening in these files with an example. We will use the beam file and the Beam object. We described before the the Beam object has some attributes like dE and dt, and some methods like the mean value of these two arrays. In order for this class to be ported in the GPU we need to have some copies of these arrays in the GPU and also equivalent methods, that will use these GPU arrays. In order to do that in the

gpu_beam.py file, there is a class called GpuBeam that inherits from the Beam class, and has some methods overridden. The question is what happens with the attributes arrays, that have to be in the GPU but can also be requested or even changed by CPU functions. To support these needs, we have developed a class, called CGArray, derived from Cpu Gpu Array, that supports exactly that, having a copy of an array both in the GPU and the CPU.

This happens in the gpu_activation file. For every object we have a use_gpu method that we add to our classes. For example, for the Beam class we have use_gpu_Beam method. This method updates the array to the CGA object, and also updates the __-class__ attribute to GpuBeam.

5.2.1 CGA class

The need for this CGA class arose, when we saw that physicists are doing operations on arrays that we only kept in the GPU, in their main loops. A typical example is the following.

```
for i in range(iterations):
    profile.track()
    tracker.track()
    profile.bin_centers += 0.25
```

In this example, the bin_centers array is supposed to exist in the GPU, so operations like that would not be supposed to work. In order to avoid that from happening we let two arrays to exist, a CPU array with the name of bin_centers, and one array that exist in the GPU with the name dev_bin_centers. These two arrays are linked in a CGA object. This means that when requested, either for CPU or for GPU, a valid copy will be given.

In order to use CGA array we have to do two things. First of all in the use_gpu activation function of an object we need to add a line to create the CGA object.

```
def use_gpu_beam(self):
    from ..gpu import gpu_beam as gb
    from ..gpu.gpu_cpu_array import CGA

    if self.__class__ == gb.GpuBeam:
        return

    self.dE_obj = CGA(self.dE)

    self.dt_obj = CGA(self.dt)

    self.id_obj = CGA(self.id, dtype2=bm.precision.real_t)
    self.__class__ = gb.GpuBeam
```

You can see in lines 8,19 and 12, that we are creating 3 CGA objects. To use them you add some properties in the class that you want, just like in the following example.

```
class GpuBeam(Beam):

    @property
    def dE(self):
        return self.dE_obj.my_array

    @dE.setter
    def dE(self, value):
        self.dE_obj.my_array = value

    @property
    def dev_dE(self):
        return self.dE_obj.dev_my_array

    @dev_dE.setter
    def dev_dE(self, value):
        self.dE_obj.dev_my_array = value
```

In this piece of code you can see what you need to do, to use the dE_obj that was created earlier. We have some properties, getters and setter for both the CPU and the GPU. You can see that the dE_obj has two arrays as attributes, my_array which is the CPU array, and dev_my_array, which is the GPU array.

The CGA class has some really important functionalities. First of all both the CPU and the GPU copy can be valid or invalid. That means that if we modify for example the GPU copy, the CPU one will be invalid until we request it. If we request it, it will be validated, and then it will be returned to us. We can invalidate it with three ways.

```
# supposing the profile object has a CGA n_macroparticles array

# First way, we set the n_macroparticles array to a new one
profile.n_macroparticles = np.array([1,2,3,4, 5, 6, 7, 8])

# Second way, we set a slice of the array
profile.n_macroparticles[:5] += 3

# Third way, we invalidate it manually
profile.n_macroparticles_obj.invalidate_gpu()
```

So when we alter a copy, in GPU or in the CPU, the other copy is invalid, until we either request it or alter it. Now the CGA class has some more features that we want for our case.

We can use different types for the copies. For example, in the profile class, the n_macroparticles array in real for the CPU and integer for the GPU. Also the 2d arrays that we have in the CPU are 1d in the GPU. We need to take that in mind when we are developing methods for our GPU objects, since we are using the GPU copies for them.

Finally, to sum things up, CGA class is a way to have synchronized arrays in the CPU and in the GPU without adding overhead. This will help physicists to develop their simulation file without having to think what is in the GPU and what is not. Also this can help in case some methods are not developed for the GPU, but they use arrays that also reside in the GPU.

5.2.2 Enabling GPU from the mainfile

The first thing that we have to do, to use the GPU using the PyCuda module is the initialization of the GPU. In order to do that in a pretty way, we have developed a class called GPUDev. When we initialize the GPU a new GPUDev object is created. We initialize the GPU with the following piece of code:

```
import blond.utils.bmath as bm
bm.use_gpu(gpu_id)
```

This command initializes the context of the GPU. Also it updates the dictionary of the bmath file with our GPU functions. For example the drift function is updated from the CPU one to the GPU one. Also the file activation is being imported, and so the use_gpu_xxx methods are added to their corresponding classes. Finally we direct PyCuda to a cubin file, that we have from compiling our Cuda kernels. This cubin file has all the functions that are requested from PyCuda. After we have initialized our GPU, we need to call the use_gpu method for the objects that we plan to use in our main loop. An example follows.

```
import blond.utils.bmath as bm
bm.use_gpu()
beam.use_gpu()
tracker.use_gpu()
rf_station.use_gpu()
profile.use_gpu()
```

These lines are everything that you need to do in order to use the GPU. And in fact, you can skip a few lines. Some objects have other object as their attributes. For example, the Beam object is an attribute of the tracker object. Since tracker will also call the use_gpu for the beam we can skip line 3. The same applies for rf_station and the profile object.

Chapter 6

Benchmarks

In this chapter, we will see the final results of our GPU implementation. To perform the benchmarks we used the Greek Supercomputer ARIS. The cluster that we have been given access to, had in total 44 nodes, each node with two CPUs Intel Xeon E5-2660v3, two GPUs NVidia K40 and 64GB of RAM. You can see the specs of our devices in tables 6.1, 6.2 and 6.3. In this cluster we performed three experiments. LHC, PS and SPS, with 5 different configurations for sizes and nodes. First of all we will describe briefly these experiments, and afterwards we will see and discuss the results.

Intel Xeon E5-2660v3 Specs	
Cores	10
Threads	20
Cache	25 MB
Frequency	2.60 GHz

Table 6.1: *Intel Xeon Specifications*

Nvidia K40 vs Nvidia V100		
	k40	v100
Architecture	Kepler	Volta
Process Size	28 nm	12 nm
Memory Size	12 GB	32 GB
Memory Bus	384 bit	4096 bit
Bandwidth	288.4 GB/s	897 GB/s
SM count	15	80
Cores	2880	5120
L1 cache (per SM)	16KB	128KB
L2 cache	1636KB	6MB
Cuda Version	3.5	7.0

Table 6.2: *Comparison between k40 and v100*

Peak Performance of Device (TFLOPS)			
Operation	Intel Xeon E5-2660v3	Nvidia k40	Nvidia V100
FP32	2.1	5.04	14
FP64	1.05	1.68	7

Table 6.3: *The specification of our devices*

6.1 Experiments

First of all, these experiments have a few things in common. In all of them, we let the main loop run for ten thousand turns. All of them have some common objects too. They have a beam, a profile, a tracker and a TotalInducedVotage object. All of them have the linear_interpolation_feature activated, and because of that they are not using the kick function but the linear_interpolation_kick. Apart from that, we test different particle sizes for them as well as number of bunches and number of slices for the histogram. The LHC and SPS experiments also have one more feature, the beam-feedback object, that performs each round a reduction on the histogram and some operations on it. These operations, when done in the GPU, change the results of some simulations, so they are performed in the CPU.

6.2 MPI

In the plots that will follow, we have used two versions. The first one is MPI-over-OpenMP for our CPU benchmarks. The second is MPI-over-Cuda. These two versions both perform the same MPI operations. In the beginning of the main loop, the beam particles is being distributed among workers. Then for every turn, the workers compute their particles histogram, and perform an all-reduce operation, to get the global histogram. After that, all of them do the induced_voltage_sum operation. You can see that process in figure 6.1.

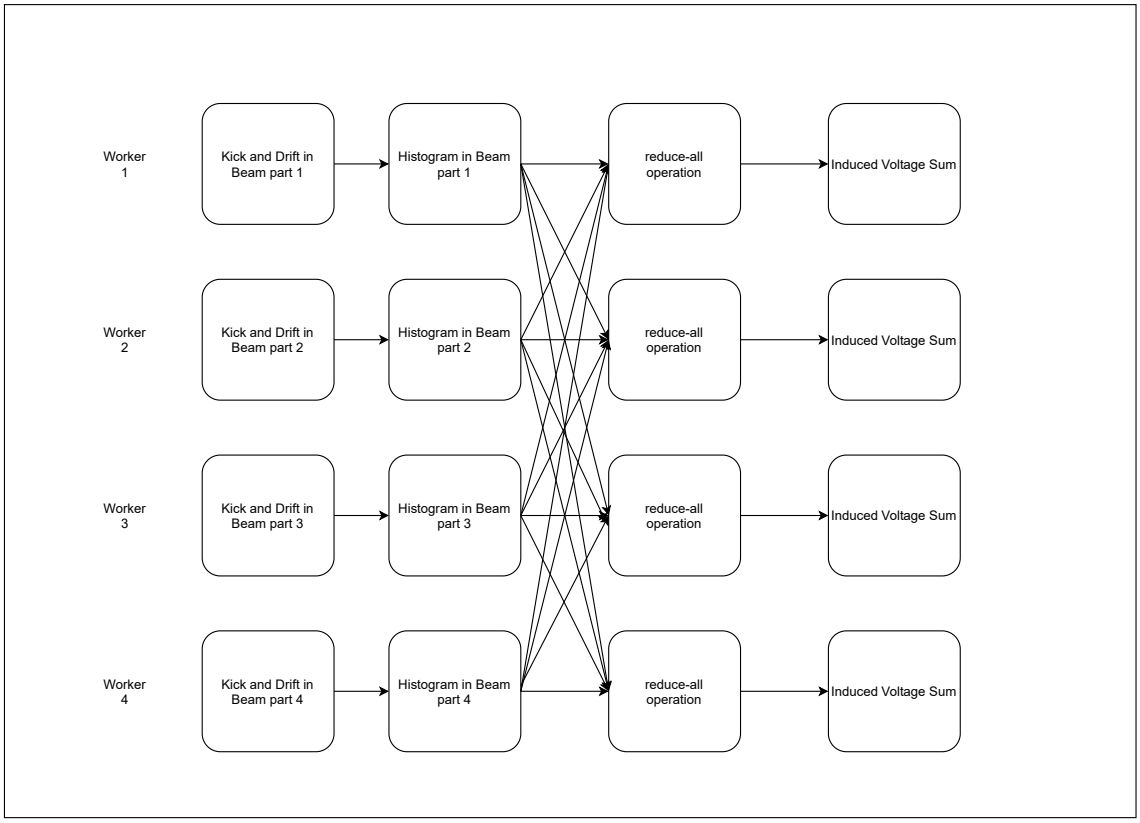


Figure 6.1: *MPI workers 1 turn example*

6.2.1 Classification Of Operations

In order to be able to explain what happens, we classify the operations in 3 different categories.

- **comp**: operations that their time is dependent from the number of particles.
- **serial**: operations that their time is independent from the number of particles.
- **comm**: operations that are happening due to the need for communication.

So with the above classification, in the MPI version, if we increase the number of nodes and keep the size of our simulation the same, we reduce the comp time for each worker. The serial time will remain the same. The communication time will probably increase, since with more workers, we need more synchronization time for the all-reduce operation.

6.2.2 CPU vs GPU

In this subsection we will see the results of our GPU implementation compared with the CPU implementation. We have tried four different configurations, that are described in figures 6.4, 6.5. The results of our benchmarks can be viewed in figures 6.2, 6.3, 6.4.

	LHC	PS	SPS
particles per node	18 million	21 million	18 million

Table 6.4: *Particles per node for experiments*

	CPU-BASE	CPU-TP	GPU-1PN	GPU-2PN
workers per node	2	2	1	2
task parallelism	off	on	off	off

Table 6.5: *Configurations Description*

Task parallelism is used in the serial operations like the FFTs. For example, in a node, we let one worker perform the first serial operation and the second performs the other one. In this way we save some computation time, but we add some communication time. For GPU this feature is not good, since the time saved is less than the communication time added plus the time that it takes to do the GPU to CPU memory operation.

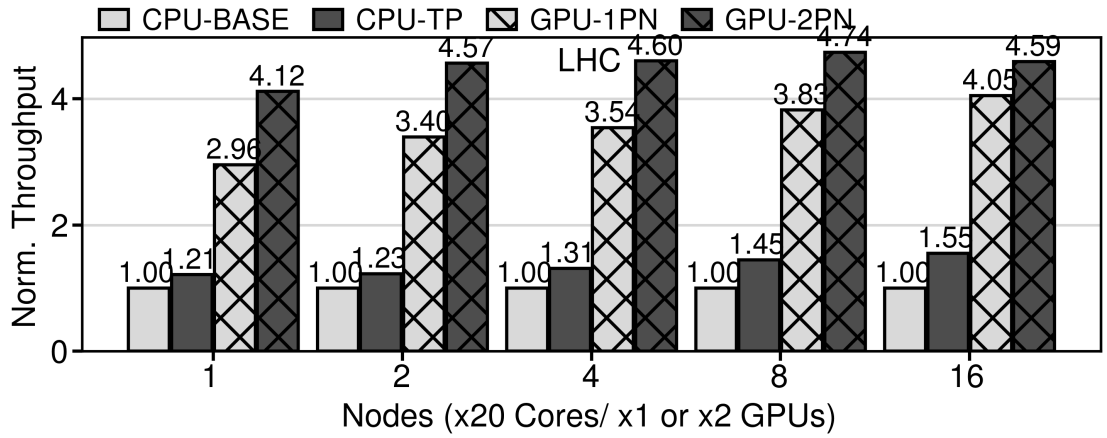


Figure 6.2: *LHC: CPU vs GPU*

In figure 6.2 we see that the GPU-1PN is at least three times faster than the CPU-BASE. If we move from GPU-1PN to GPU-2PN we see a difference because the comp operations of one node are now distributed to two workers. Since comp operations are not the only things that we perform, we do not expect two see a two times speedup, which we do not. The difference between GPU-1PN and GPU-2PN is getting worse as we increase the number of nodes. For example, using 16 nodes means that instead of having 16 workers, we have 32.

For task-parallelism, we can see that the CPU-TP gets better as we increase the number of nodes. The reason for that is that FFTs, which are being performed really slow in the CPU, are more demanding when having more bunches, since more bunches mean more slices for the histogram that is the input to the FFTs.

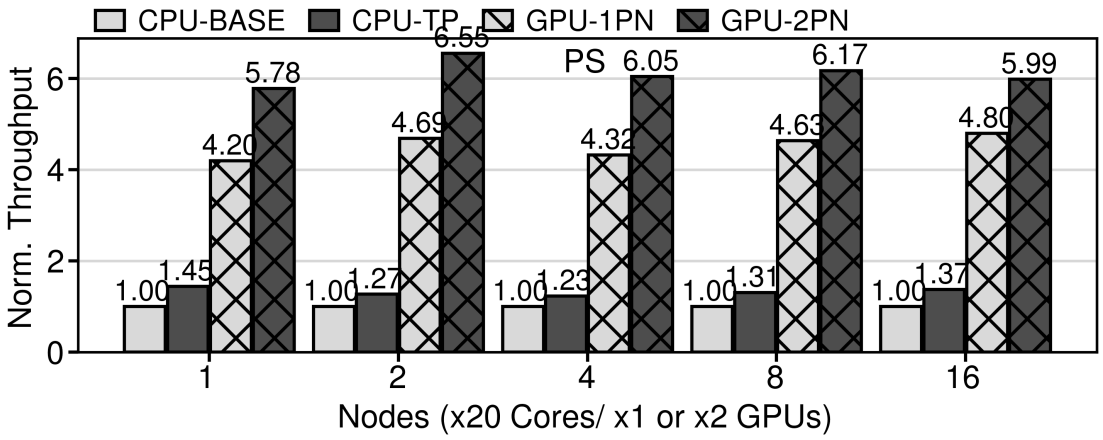


Figure 6.3: *PS: CPU vs GPU*

For the PS experiments, the results in figure 6.3 are very close to the results of LHC. The GPU performs better in this experiment, our GPU-1PN is more than 4 times faster than the CPU-BASE and our GPU-2PN is more than 6 times faster. Once again, the difference between GPU-1PN and GPU-2PN is getting smaller as we increase the nodes.

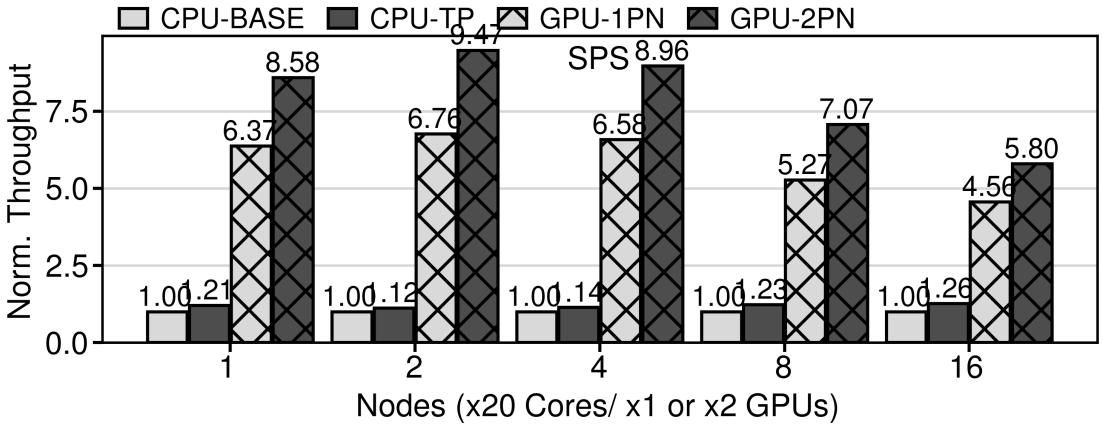


Figure 6.4: *SPS: CPU vs GPU*

Finally, in figure 6.4 we see the results of the SPS experiment. This experiment is the best in terms of CPU and GPU comparison. The configuration with 1, 2 and 4 nodes are faster more than 6 times than the CPU-BASE and the GPU-2PN is more than 8 times, close to 9. The problem we have here is that the scaling is pretty bad. As we increase the number of nodes, the runtime is dominated by comm operations, because the nodes need to communicate with each other.

We can confirm all of the above if we look at the figures 6.5, 6.6 and 6.7 The intra is about serial operations, like the FFTs and the beam_phase. The comm is about the synchronization time we have to pay for communication. The part of the bars that can not be seen is about the computation time.

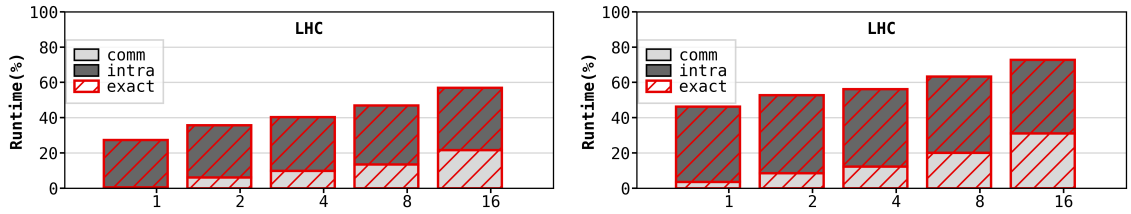


Figure 6.5: *LHC: On the left we have GPU-1PN and on the right we have GPU-2PN*

In figure 6.5 we can see the time breakdown for the LHC experiment. The comm time is getting bigger as we increase the number of nodes. We actually expected that kind of behavior. The intra time is also increased from GPU-1PN to GPU-2PN. The reason for this is that the operations that we are doing in the CPU are now performed by 10 cores and not by 20.

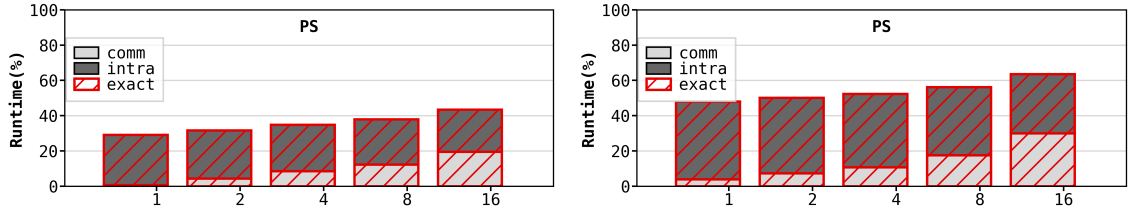


Figure 6.6: *PS: On the left we have GPU-1PN and on the right we have GPU-2PN*

In figure 6.6 we can see the time breakdown for the PS experiment. The behavior of comm time is almost identical to LHC. The same applies to the intra time. Its relation with the comp time, which is the part of the runtime that remains, is stable. What we mean by this, is that their ratio is almost the same for every number of nodes. In the GPU-1PN their ration is close to 25% and in GPU-2PN it is close to 50%.

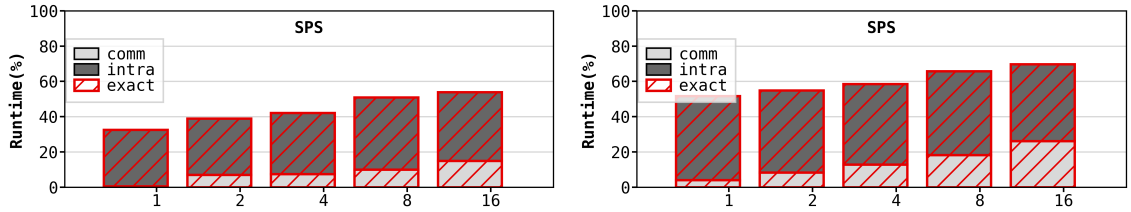


Figure 6.7: *SPS: On the left we have GPU-1PN and on the right we have GPU-2PN*

The time breakdown of the SPS experiment can be seen in figure 6.7. Just like in the PS and the LHC the runtime is being dominated by intra and comm time as we increase the number of nodes.

6.2.3 Approximation Methods

In order to boost the performance of our MPI version when increasing the number of nodes, we have used some approximation methods. These approximation methods are used mainly to skip the reduce-all operation of the workers, and reduce the communication time. These methods are the following. It may seem that these methods lead to

the loss of some accuracy, but as we will see, because of some characteristics of our distribution the loss is very small, and the experiments can be considered valid.

- rds: with this approximation method, we choose to perform the histogram operation every n-turns. The number of n is small (for our benchmarks we have tried 2 and 3). The hypothesis behind this approximation method is that the histogram does not change drastically after a small number of turns, and so we can use the previous one. This way we reduce the numbers we perform the reduce-all operation, but also the serial operations. This happens because if the histogram stays the same, the values produced by it also remain the same. So we can skip them. So the time for serial operations is reduced to $100 * \frac{1}{n}\%$

- srp: we consider that the parts of the beam that are distributed to workers, follow the same distribution, since this process is done randomly. So, with this approximation technique, we suppose that the global histogram can be computed, if we scale the local one. For example, if we have two workers, each one in order to compute his histogram need to divide the local histogram with the percentage of particles it possesses. In this example he needs to divide every value of the local histogram with 0.5. With this approximation we save the communication time, because the workers do not need to perform the reduce-all operation at all.

- f32: the third approximation method is to try float32 numbers instead of float64. There is a mechanism developed that let the user decide if he wants to use float32 or float64 precision. With this approximation we expect to see reduction in the comp time. We expect the drift and the kick function to reduce their runtime, close to 50% but for the other operations we expect to remain almost the same.

6.2.4 Weak Scaling Approximation Plots

These benchmarks run for the approximation methods. We wanted to see how much they improve the runtime when used. So for each configuration above, we tested the following methods.

- exact: f64 precision, no approximation method used.

- srp-3: f64 precision, we used the srp method, and updating the histogram evey 3 turns.

- rds: f64 precision, we used the rds method.

- f32-exact: f32 precision, no approximation method used.

- f32-srp-3: f32 precision, we used the srp method, and updating the histogram evey 3 turns.

- f32-rds: f32 precision, we used the rds method.

Now we will see how each experiment was benefited from these approximation methods, starting with the LHC in 6.8.

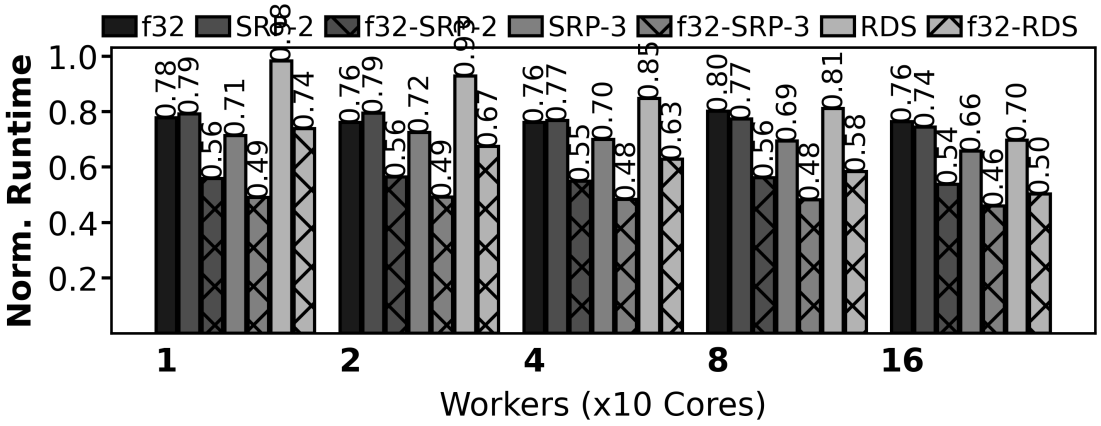


Figure 6.8: *LHC Approximation Speedup*

The f32 version, seems to achieve a speedup more than 1.3 ($\frac{1}{0.76}$). The reason for that, is that not all the operations are faster because of the f32 precision. The FFTs and the histogram function, just like we saw earlier, are not faster than their corresponding f64 versions. So we only get speedup from linear_interpolation_kick and drift.

The SRP-3 reduces the runtime of the FFTs and also the synchronization time. The speedup it offers, is getting better as we increase the number of nodes. Having more nodes results in more communication time, and because of that, SRP-3 is more effective.

Next we have f32-SRP-3, that is a combination of the two previous techniques. Its results, match the addition of the previous results. This is because the the performance we gain by the f32 on some serial operations, that are also benefited by the SRP-3. This is can be confirmed by the numbers. What we expect is:

$$time\_saved\_by\_f32 + time\_saved\_by\_srp \simeq time\_saved\_by\_both$$

What really happens is what we expected,

$$(1 - 0.70) + (1 - 0.73) \simeq (1 - 0.44)$$

$$(1 - 0.76) + (1 - 0.70) \simeq (1 - 0.44)$$

$$(1 - 0.70) + (1 - 0.70) \simeq (1 - 0.44)$$

$$(1 - 0.76) + (1 - 0.68) \simeq (1 - 0.42)$$

$$(1 - 0.70) + (1 - 0.65) \simeq (1 - 0.40)$$

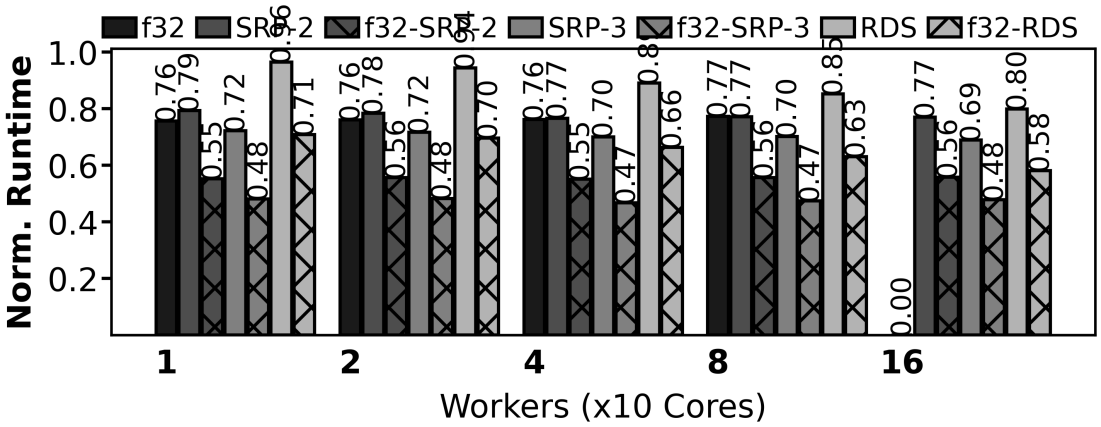


Figure 6.9: *PS Approximation Speedup*

In figure 6.9, everything we discussed about the f32, the SRP and the f32-SRP can also be applied. The RDS optimization completely removes the communication part of the MPI processes. Now they only need to communicate in the beginning, to distribute the workload, and in the end to gather the beam. We can see, that its impact gets bigger, as we increase the number of nodes. This is something that we expected, since this approximation is more effective when we have more communication. The same applies to LHC.

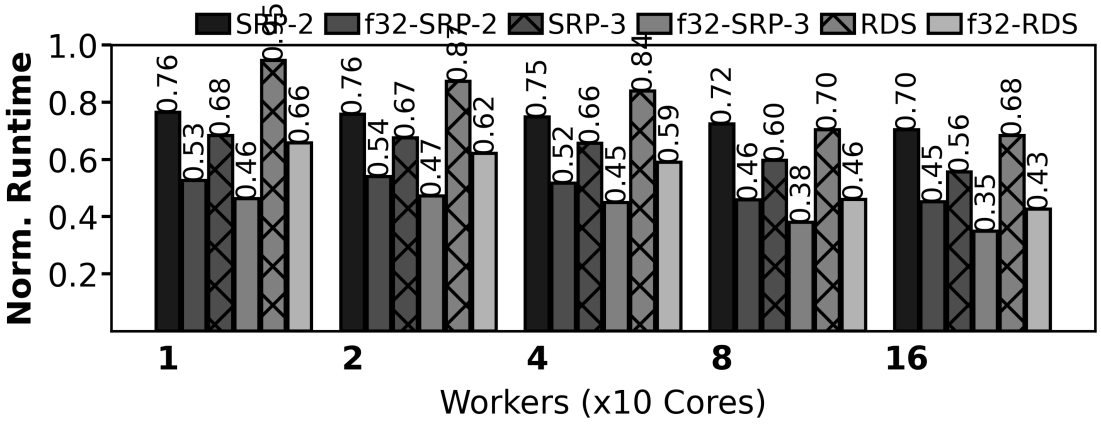


Figure 6.10: *SPS Approximation Speedup*

All of our previous observations, also apply to our SPS benchmarks and the plot in figure 6.10. The f32-RDS approximation is better from both the f32 and RDS approximations. These two are independent, so we expect the time saved by f32-RDS to be the sum of the time saved by each of them when applied alone, or at least a bit lesser than it. This also applies to the previous benchmarks.

In figures 6.11, 6.12 and 6.13 we can see the weak scaling plots for these methods.

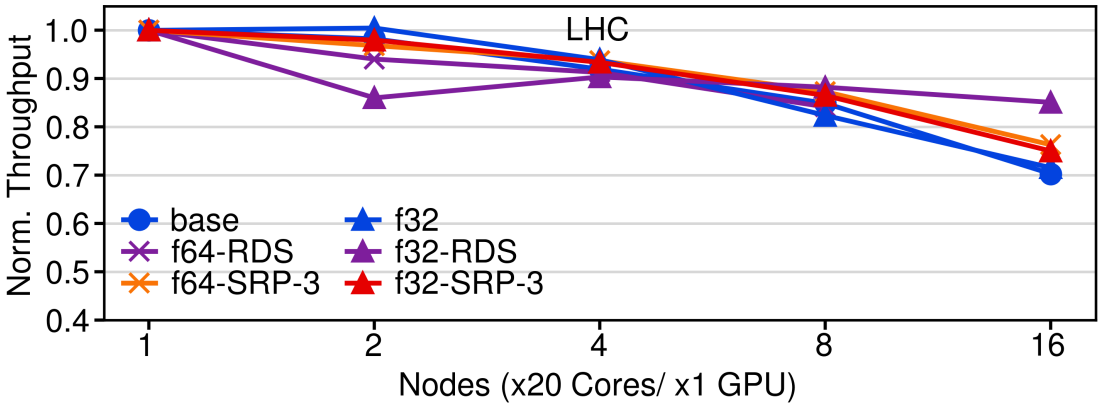


Figure 6.11: *LHC Weak Scaling*

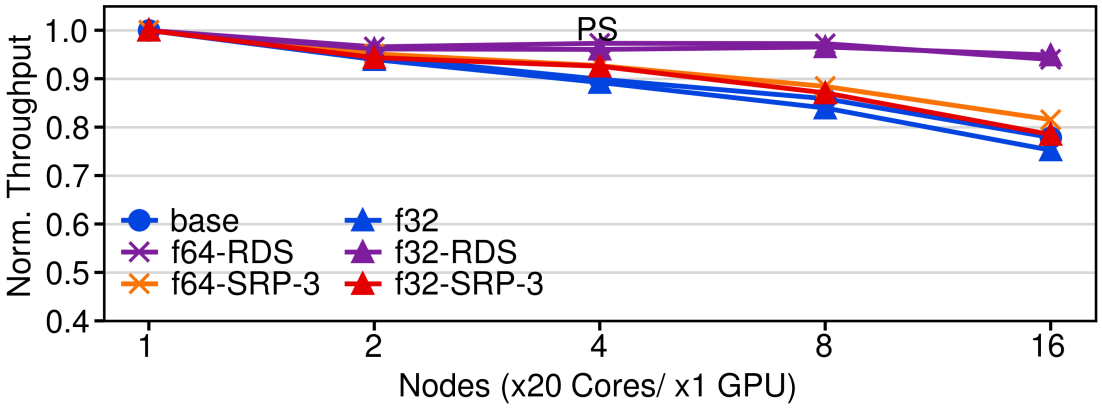


Figure 6.12: *PS Weak Scaling*

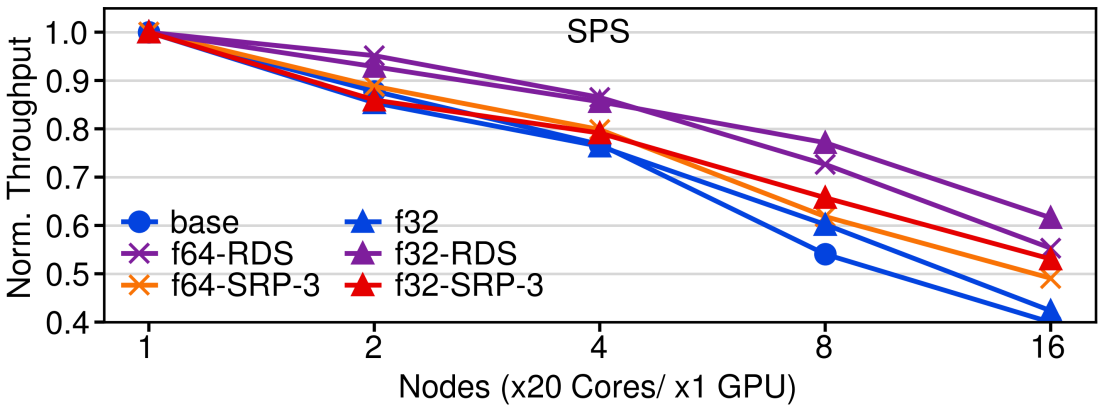


Figure 6.13: *SPS Weak Scaling*

From these three plots, LHC and PS scale pretty good. Having 16 nodes reduce the throughput to 80% of the initial value for the PS. Additionally, with the RDS approximation method, it scales even better, with the throughput being not less than 90%. The LHC benchmark has a similar throughput. The base version reaches 70% of the initial

throughput, and the f32-RDS version reaches 85%. SPS behavior is different. It scales really bad, compared to the other two. The base version's throughput drops to 40% of the initial, and the f32-RDS drops to 60%.

6.3 K40 vs V100

In this section, we want to check if our implementation is future proof. In order to do that we use the Nvidia v100, a much more powerful GPU than k40, to confirm that we get better results than k40. In tables 6.1, 6.2 and 6.3, we can see the specs of V100. We used a cluster of CERN for our V100 benchmarks. We used 1 node with 1 worker. You can see the results in figures 6.14, 6.15 and 6.16.

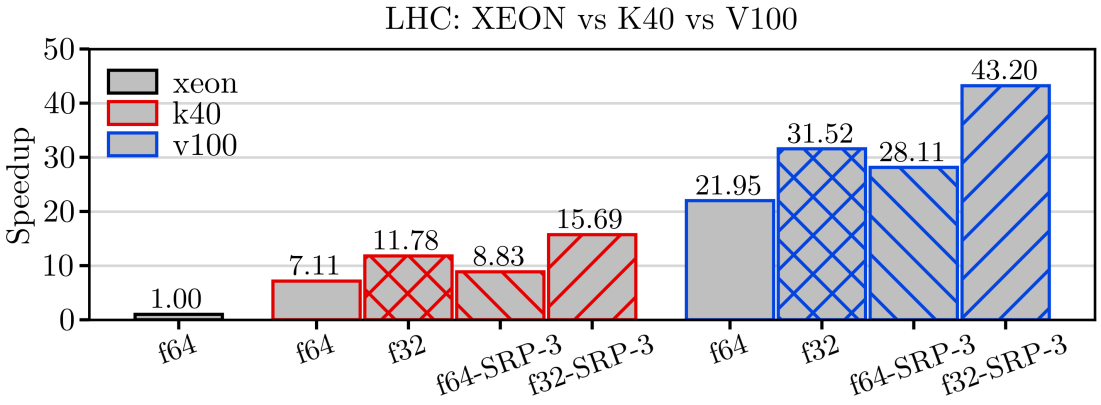


Figure 6.14: *LHC Weak Scaling*

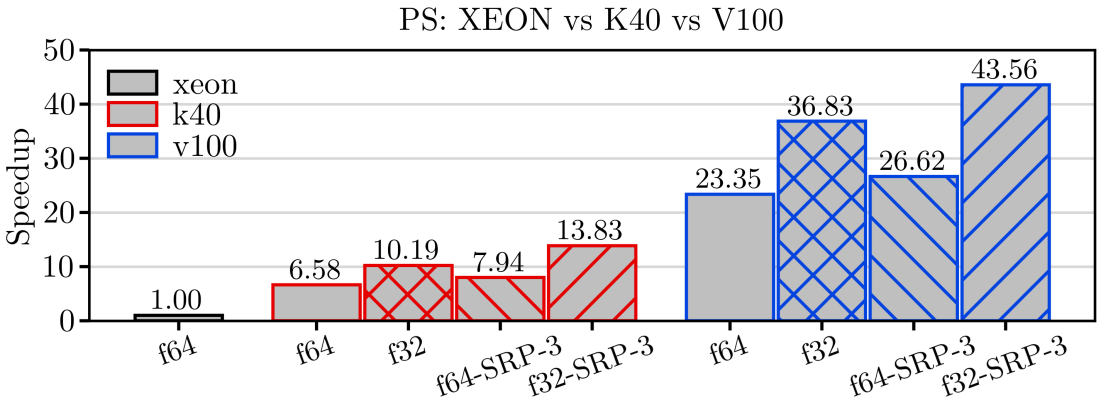


Figure 6.15: *PS Weak Scaling*

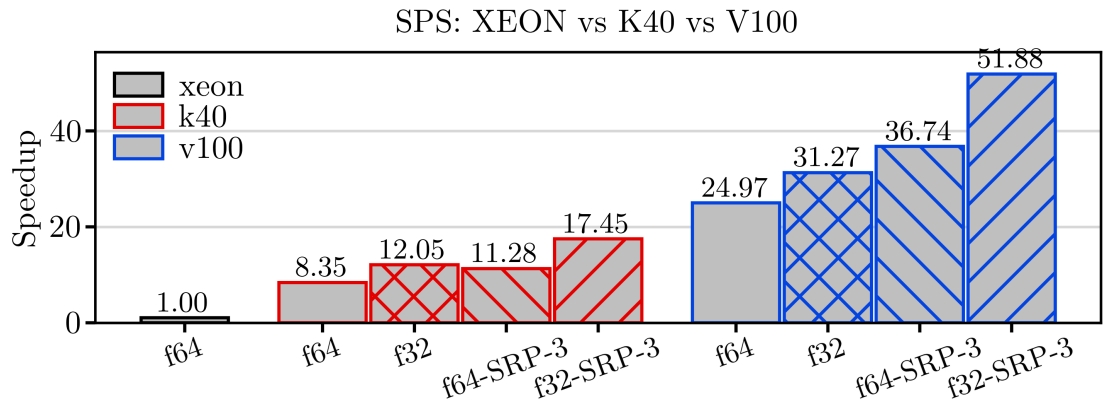


Figure 6.16: *SPS Weak Scaling*

We can see that v100 is for every configuration faster than the k40, more than three times. Also for the base version v100 is more than 20 times faster than our CPU. With these last plots, its pretty clear that if we use a better and more powerful GPU for our simulations, the simulations will produce results faster.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis, we described how we implemented the acceleration of BLonD, with the use of a GPU. First of all, we created the GPU kernels for the functions that dominate most of the time. By doing benchmarks with some data created by us, we tried to optimize these kernels, in order to get better results. After that, in order to eliminate most of the memory operations between GPU and CPU, we also implemented some small methods for the GPU.

Additionally, we demonstrated how a user can enable the GPU from his main file, and what happens when he does. That is, each class has a corresponding GPU class with its suitable methods, and the arrays transformed to CGA objects. After finishing the implementation for one worker simulation, we also created some other methods, to enable MPI to use the GPU as a worker, either with other GPUs as workers, or with CPUs as well.

After implementing all of the above, we compared our new version with the CPU version with the use of the supercomputer ARIS, with the use of three benchmarks, LHC, PS and SPS. From these benchmarks, we saw that the GPU is on average more than five times faster than the CPU, and in order to solve the problem of losing throughput when increasing the number of nodes, we used some approximation methods. With these methods, we manage to gain some performance especially, when using a lot of nodes.

Finally, in order to make sure that our implementation is future proof, we used a new and more powerful GPU, the NVidia v100 to run some benchmarks. We did that without changing anything, and the v100 proved to be at least three times faster than the k40. That leads to the assumption that when using new and better hardware, you get better results.

7.2 Future Work

Apart from these, there is still some room for improvement. Below, we propose some ideas for future work that could lead to even greater and faster simulations.

- In our current version, the initialization of the Beam, happens in only one node, and is then distributed to the other nodes. So the Beam must be able to fit in this

node's main memory. So we are limited by the main memory of one node. So if we manage to implement the initialization with MPI, we could use the main memory of all workers, and the Beam could be even bigger.

- In our benchmarks, we tried to use one CPU worker with one GPU worker. The problem was that the GPU was faster, and the GPU worker had to wait for the CPU worker. Because of that, the synchronization time was dominating the runtime. We tried to overcome this issue with the use of a dynamic load balance schema , that assigned suitable workload to each worker, in order to minimize the synchronization time. Still, the performance was worse, and using only one GPU worker was better. It would be good to try our benchmarks in a cluster with a CPU as powerful as the GPU, and repeat this process.

Bibliography

[1] F Tecker. *Longitudinal beam dynamics*. *arXiv preprint arXiv:1601.04901*, 2016.

[2] Lyndon Evans και Philip Bryant. *LHC machine*. *Journal of instrumentation*, 3(08):S08001, 2008.

[3] Joël Repond, Konstantinos Iliakis, Markus Schwarz, Elena Shaposhnikova, G Papotti, D Quartulo, H Timko και others. *Simulations of Longitudinal Beam Stabilisation in the CERN SPS with BLonD*. *Proc. ICAP'18*, σελίδες 197–203, 2018.

[4] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone και James C Phillips. *GPU computing*. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[5] Tianyi David Han και Tarek S Abdelrahman. *hiCUDA: High-level GPGPU programming*. *IEEE Transactions on Parallel and Distributed systems*, 22(1):78–90, 2010.

[6] Leonardo Dagum και Ramesh Menon. *OpenMP: an industry standard API for shared-memory programming*. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[7] Konstantinos Iliakis, Helga Timko, Sotirios Xydis και Dimitrios Soudris. *BLonD++ performance analysis and optimizations for enabling complex, accurate and fast beam dynamics studies*. *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, σελίδες 123–130, 2018.

[8] William Gropp, William D Gropp, Ewing Lusk, Anthony Skjellum και Argonne Distinguished Fellow Emeritus Ewing Lusk. *Using MPI: portable parallel programming with the message-passing interface*, τόμος 1. MIT press, 1999.

[9] Konstantinos Iliakis, Helga Timko, Sotirios Xydis και Dimitrios Soudris. *Scale-out beam longitudinal dynamics simulations*. *Proceedings of the 17th ACM International Conference on Computing Frontiers*, σελίδες 29–38, 2020.

[10] Thomas Paine, Hailin Jin, Jianchao Yang, Zhe Lin και Thomas Huang. *Gpu asynchronous stochastic gradient descent to speed up neural network training*. *arXiv preprint arXiv:1312.6186*, 2013.

[11] Tor M Aamodt, Wilson Wai Lun Fung και Timothy G Rogers. *General-purpose graphics processor architectures*. *Synthesis Lectures on Computer Architecture*, 13(2):1–140, 2018.

[12] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov και Ahmed Fasih. *PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation*. *Parallel Computing*, 38(3):157–174, 2012.

[13] Georges Aad, B Abbott, J Abdallah, AA Abdelalim, Abdelmalek Abdesselam, B Abi, M Abolins, H Abramowicz, H Abreu, E Acerbi και others. *Search for stable hadronising squarks and gluinos with the ATLAS experiment at the LHC*. *Physics Letters B*, 701(1):1–19, 2011.

[14] CMS Collaboration, S Chatrchyan, G Hmayakyan, V Khachatryan, AM Sirunyan, W Adam, T Bauer, T Bergauer, H Bergauer, M Dragicevic και others. *The CMS experiment at the CERN LHC*. *JInst*, 3:S08004, 2008.

[15] Giorgio Apollinari, I Béjar Alonso, Oliver Brüning, M Lamont και Lucio Rossi. *High-luminosity large hadron collider (HL-LHC): Preliminary design report*. Τεχνική Αναφορά με αριθμό, Fermi National Accelerator Lab.(FNAL), Batavia, IL (United States), 2015.

[16] *BLonD Web Page*. https://blond.web.cern.ch.

[17] *BLonD Github Page*. https://github.com/blond-admin/BLonD.

[18] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu και Yale N Patt. *Improving GPU performance via large warps and two-level warp scheduling*. *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, σελίδες 308–317, 2011.

[19] Nicolas Brunie, Sylvain Collange και Gregory Diamos. *Simultaneous branch and warp interweaving for sustained GPU performance*. *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, σελίδες 49–60. IEEE, 2012.