National Technical University of Athens

School of Electrical and Computer Engineering

Computer Science Sector

Diploma Thesis

Beam Longitudinal Dynamics Simulation Code Acceleration with GPUs

Author:
Georgios Anastasios Typaldos

Supervisor:
Prof. Dimitrios Soudris

Microprocessors and Digital Systems Laboratory
School of Electrical And Computer Engineering

21 March 2023

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Beam Longitudinal Dynamics Simulation Code Acceleration with GPUs

---

Συγγραφέας: Γεώργιος Αναστάσιος Τυπάλδος

Επιβλέπων: Καθ. Δημήτριος Σούντρης

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 21η Μαρτίου 2023

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
Δημήτριος Σούντρης	Παναγιώτης Τσανάκας	Σωτήριος Ξύδης
Καθηγητής	Καθηγητής	Επίκουρος Καθηγητής

21 Μαρτίου 2023

Declaration of Authorship

Υπογραφή:

_____

Ημερομηνία:

_____

Περίληψη

Η σουίτα Beam Longitudinal Dynamics (BLonD), είναι ένα πακέτο λογισμικού ανοικτού κώδικα για την προσομοίωση της διαμήκους κίνησης σωματιδίων σε επιταχυντές. Αναπτύσσεται στο CERN από το 2014 και διαθέτει παραμετροποιήσιμη δομή, επιτρέποντας στον χρήστη να συνδυάσει ένα εύρος φυσικών φαινομένων σύμφωνα με τις απαιτήσεις της εκάστοτε μελέτης.

Στόχος της διπλωματικής εργασίας είναι η αναβάθμιση της σουίτας BLonD, αλλάζοντας την GPU υλοποίηση από τη βιβλιοθήκη PyCUDA στη CuPy, καθώς η τελευταία παρέχει διεπαφή παρόμοια με την NumPy και υποστηρίζει χαμηλού επιπέδου λειτουργίες CUDA. Αυτό έχει ως αποτέλεσμα την απλή δομή του λογισμικού και την αύξηση της συνολικής απόδοσης. Δοκιμάζονται διάφορες δομές σε επίπεδο υλικού και τεχνικές βελτιστοποίησης, όπως η ιεραρχία μνήμης σε GPU και η τεχνική thread-coarsening, για πρόσθετη αύξηση της επίδοσης. Αναπτύσσεται και αξιοποιείται επίσης ένα εργαλείο σχεδιασμού του μοντέλου roofline σε Python για την αξιολόγηση της απόδοσης των βασικών πυρήνων.

Η καινούρια έκδοση CuPy αξιολογείται χρησιμοποιώντας τρία μοντέλα GPU της NVIDIA και συγκρίνεται με πολυνηματική υλοποίηση σε AMD CPU που εκτελείται σε 16 πυρήνες. Η έκδοση CuPy υπερνικά την επίδοση σε CPU αλλά και την προηγούμενη έκδοση, επιτυγχάνοντας έως και 80 φορές καλύτερη απόδοση από τη CPU σε απαιτητικά πειράματα και ισχυρά μοντέλα GPU, έναντι 75 φορές στην έκδοση PyCUDA, ενώ ελαχιστοποιεί και τις απαιτούμενες γραμμές CUDA κώδικα από 2600 σε 350.

Λέξεις Κλειδιά: Beam Longitudinal Dynamics, High Performance Computing, CUDA, Κάρτες γραφικών, Παράλληλος Προγραμματισμός

Abstract

The Beam Longitudinal Dynamics (BLonD) suite is an open-source software package for the simulation of the longitudinal motion of particles in synchrotrons. It has been developed at CERN since 2014 and features a modular structure that allows the user to combine a variety of physics phenomena according to the study requirements.

This thesis's scope is upgrading the BLonD suite by modifying the GPU implementation to host the CuPy Python library rather than the PyCUDA library for GPU acceleration, as it provides a NumPy-like interface and low-level CUDA functionalities. This results in software simplicity, thus a better user experience, and performance enhancements, which achieve significant execution speedup. Various hardware structures and optimization techniques, such as GPU memory hierarchy and thread-coarsening, are tested for additional performance gain. A custom Python roofline model tool is also developed and utilized to assess the efficiency of main kernels.

The BLonD-CuPy implementation is evaluated using three NVIDIA GPU models and compared against a multithreaded AMD CPU implementation executed on 16 cores. The CuPy GPU version significantly surpasses the CPU and the previous PyCUDA version's performance. It achieves up to 80 CPU speedup for intensive configurations and powerful GPU models, versus a respective 75 PyCUDA speedup, while minimizing the required CUDA lines of code from 2600 to 350.

Keywords: Beam Longitudinal Dynamics, GPU, GPGPU, High Performance Computing, Parallel Programming, CUDA

Acknowledgements

This thesis marks the completion of my studies at the School of Electrical and Computer Engineering of the National Technical University of Athens.

This endeavor would not have been possible without Professor Dimitrios Soudris, who gave me the opportunity to carry out my diploma thesis under his supervision. During my research, he provided me with proper guidance and advice, and with his invaluable research experience fueled me with inspiration to accomplish this work.

I am also grateful to CERN postdoctoral fellow Konstantinos Iliakis, with whom I cooperated closely throughout this thesis. His knowledge and expertise encouraged me to significantly expand my skills in new areas while allowing me to develop my ideas. Thanks should also go to the CERN SY-RF-BR group, for giving me the chance to work on the BLonD project and the experience of collaborating with a high-end research center such as CERN.

Lastly, I would like to recognize the contribution of my friends who supported me throughout my studies and made this journey a wonderful experience. Of course, nothing would be possible without my family. Their constant support and encouragement enabled me to complete my studies and motivated me to take the next step in my academic path.

Contents

Abstract **viii**

Acknowledgements **x**

1 Εκτεταμένη Περίληψη **1**
1.1 Εισαγωγή . . . 1
1.1.1 Επισκόπηση Πρότασης . . . 1
1.2 Θεωρητικό Υπόβαθρο . . . 2
1.2.1 Σουίτα BLonD . . . 2
1.2.2 NVIDIA Κάρτες Γραφικών . . . 3
1.2.3 Μοντέλο Roofline . . . 6
1.3 Λεπτομέρειες Υλοποίησης . . . 9
1.3.1 Σχεδιασμός BLonD . . . 9
1.3.2 Εκδόσεις PyCUDA & CuPy . . . 10
1.3.3 Δεξαμενές Μνήμης CuPy . . . 12
1.3.4 Ιστόγραμμα Ακτίνας . . . 13
1.3.5 Ανάλυση Block & Grid . . . 16
1.3.6 Τεχνική Thread Coarsening . . . 17
1.3.7 Εργαλείο Roofline . . . 19
1.4 Αξιολόγηση . . . 21
1.4.1 Πειραματικός Εξοπλισμός . . . 21
1.4.2 Σύγκριση με CPU . . . 22
1.4.3 Σύγκριση με προηγούμενη έκδοση . . . 24
1.5 Συμπεράσματα και Μελλοντικές Επεκτάσεις . . . 25
1.5.1 Συμπεράσματα . . . 25
1.5.2 Μελλοντικές Επεκτάσεις . . . 26

2 Introduction **28**
2.1 CERN Accelerator complex & Beam Dynamics . . . 28
2.2 Need for HPC Beam Dynamics Simulations . . . 28
2.3 Proposal Overview . . . 29
2.4 Thesis structure . . . 30

3 Prior Art **31**
3.1 Introduction . . . 31
3.2 Beam Longitudinal Dynamics Suite . . . 31
3.3 Graphics Processing Unit . . . 32
3.3.1 General-Purpose GPUs . . . 32
3.3.2 NVIDIA Programming Model . . . 32
3.3.3 Memory Hierarchy . . . 34
3.3.4 Hardware . . . 34
3.4 CUDA in Python . . . 35
3.4.1 PyCUDA . . . 36
3.4.2 CuPy . . . 37
3.5 Roofline Model . . . 39
3.5.1 Roofline for GPUs . . . 40

4 Implementation Details **42**
4.1 Introduction . . . 42
4.2 BLonD Design . . . 42
4.2.1 PyCUDA Version . . . 43
4.2.2 CuPy Version . . . 45
4.3 Performance Optimizations . . . 46
4.3.1 Memory Pools . . . 46
4.3.2 Shared Memory . . . 47
4.3.3 Multi-Bunch Histogram . . . 48
4.4 Block & Grid size analysis . . . 50
4.5 Thread Coarsening Analysis . . . 53
4.6 Roofline analysis tool . . . 54

5 Evaluation **56**
5.1 Introduction . . . 56
5.2 CPU-GPU speedup . . . 56
5.2.1 Using different GPU models . . . 59
5.3 CPU-GPU time breakdown . . . 61
5.4 Comparison against previous version . . . 64

6 Conclusions and Future Work **66**
6.1 Conclusions . . . 66
6.2 Future Work . . . 67

Bibliography **68**

List of Figures

3.1 CUDA grid visualization 34
3.2 GPU kernel execution Hardware 35
3.3 Roofline CPU Model 40
3.4 Roofline GPU Model 41

4.1 Memory Pool Testing 47
4.2 Bunch distributions 48
4.3 GPU Shared Memory for Histogram Function 49
4.4 Kernel launch configuration testing (block size, grid size) 52
4.5 Block & Grid analysis 52
4.6 Kernel Evaluation with Roofline Model Tool 55

5.1 SPS CPU Speedup 60
5.2 PS CPU Speedup 60
5.3 LHC CPU Speedup 61
5.4 SPS Kernel Analysis 62
5.5 PS Kernel Analysis 63
5.6 LHC Kernel Analysis 63
5.7 SPS CuPy-PyCUDA comparison 65
5.8 PS CuPy-PyCUDA comparison 65
5.9 LHC CuPy-PyCUDA comparison 65

List of Tables

4.1 BLonD CuPy vs PyCUDA Version 46
4.2 Thread Coarsening Implementation 53

5.1 SPS T4 CPU Speedup 58
5.2 PS T4 CPU Speedup 58
5.3 LHC T4 CPU Speedup 58
5.4 CERN NVIDIA GPU Models 59
5.5 SPS V100 and A100 CPU Speedup 60
5.6 PS V100 and A100 CPU Speedup 60
5.7 LHC V100 and A100 CPU Speedup 61

Κεφάλαιο 1

Εκτεταμένη Περίληψη

1.1 Εισαγωγή

Στο ευρωπαϊκό κέντρο πυρηνικών ερευνών, γνωστό ως CERN, φυσικοί και μηχανικοί προσπαθούν να αποκωδικοποιήσουν τη δομή του σύμπαντος. Για τον σκοπό αυτό έχει αναπτυχθεί ο μεγαλύτερος και ισχυρότερος επιταχυντής σωματιδίων, ο μεγάλος επιταχυντής αδρονίων (LHC), ο οποίος έχει οδηγήσει σε σημαντικές επιστημονικές ανακαλύψεις όπως η παρατήρηση του μποζονίου του Higgs το 2011.

Για την προσομοίωση των διάφορων επιταχυντών χρησιμοποιείται η σουίτα Beam Longitudinal Dynamics BLonD [1], η οποία αναπτύσσεται στο CERN από το 2014. Το λογισμικό αυτό προτιμάται από πολλές επιστημονικές ομάδες παγκοσμίως, καθώς είναι σχεδιασμένο για την προσομοίωση της διαμήκους κίνησης και την παρακολούθηση των ενεργειακών και χρονικών συντεταγμένων των σωματιδίων σε επιταχυντές. Με την παραμετροποιήσιμη δομή του καλύπτει την ανάγκη για εκτενείς προσομοιώσεις, όπως περιγράφεται στην Ενότητα 1.2.1.

Οι προσομοιώσεις που πραγματοποιούνται απαιτούν σημαντική υπολογιστική ισχύ και αρκετά μεγάλο χρόνο εκτέλεσης, καθώς αποτελούνται από πολύπλοκους υπολογισμούς. Ανάλογα με τη διαθέσιμη ισχύ, μπορεί να χρειαστούν αρκετές ώρες, μέρες ή και μήνες για την εκτέλεση των πιο απαιτητικών πειραμάτων. Έτσι, το λογισμικό BLonD προσαρμόστηκε ώστε να υποστηρίζει παράλληλους επεξεργαστές αλλά και Κάρτες Γραφικών (GPU) που μειώνουν σημαντικά τον χρόνο εκτέλεσης.

1.1.1 Επισκόπηση Πρότασης

Το λογισμικό BLonD έχει υποστεί πολλαπλές τροποποιήσεις ανά τα χρόνια για να υποστηρίζει απαιτητικούς υπολογισμούς σε μικρό χρονικό εκτέλεσης. Διάφορες τεχνικές High Performance Computing (HPC) έχουν χρησιμοποιηθεί και μεγάλη

ποσότητα κώδικα έχει αναπτυχθεί ώστε να καλυφθούν οι ανάγκες προσομοίωσης. Η τελευταία έκδοση του BLonD [2] αξιοποιεί κάρτες γραφικών (GPU), με χρήση της βιβλιοθήκης PyCUDA σε Python, ώστε να επιταχύνει τους υπολογισμούς. Παρά τη σημαντική βελτίωση στην επίδοση, η έκδοση αυτή αποτυγχάνει να παρέχει αποδοτική εμπειρία χρήστη και προγραμματιστική ευκολία. Η εργασία αυτή προτείνει τη χρήση μια σύγχρονης Python βιβλιοθήκης, της CuPy [3], που επιτρέπει την ανάπτυξη μιας απλής δομής λογισμικού, ενώ εκμεταλλεύεται όλες τις δυνατότητες της GPU για να παρέχει σημαντικό κέρδος απόδοσης, όπως περιγράφεται στην Ενότητα 1.3.2. Για να αξιολογηθεί η υλοποίηση αυτή χρησιμοποιούνται βασικές μετρικές, όπως η επιτάχυνση σε σχέση με την CPU αλλά και με την έκδοση PyCUDA, οι συνολικές γραμμές κώδικα (Python και CUDA), η ποσότητα του κώδικα που επιταχύνεται, καθώς και η δομή του λογισμικού που διευκολύνει μελλοντικές τροποποιήσεις.

1.2 Θεωρητικό Υπόβαθρο

Στο κεφάλαιο αυτό παρουσιάζεται το απαραίτητο θεωρητικό υπόβαθρο για την αναβάθμιση της σουίτας BLonD όπως οι κάρτες γραφικών της NVIDIA και το μοντέλο Roofline για CPU και GPU.

1.2.1 Σουίτα BLonD

Η σουίτα Beam Longitudinal Dynamics (BLonD) [1], [4] είναι ένα πακέτο λογισμικού ανοιχτού κώδικα του CERN, για την προσομοίωση της διαμήκους κίνησης και την παρακολούθηση των ενεργειακών και χρονικών συντεταγμένων ακτίνας σωματιδίων σε επιταχυντές. Αναπτύσσεται από το 2014 και δοκιμάζεται εκτενώς για όλους τους υπάρχοντες και μελλοντικούς επιταχυντές του CERN.

Πριν την ανάπτυξη του λογισμικού BLonD, για τις προσομοιώσεις σωματιδίων στο CERN χρησιμοποιούταν η σουίτα ESME [5], η οποία αναπτύχθηκε στο Fermilab το 1984. Ωστόσο, η έλλειψη αναβάθμισης και υποστήριξης οδήγησαν σε παρόμοια λογισμικά όπως το Py-Orbit [6] και το Elegant [7], αλλά και αυτά είχαν περιορισμένες δυνατότητες προσομοίωσης. Έτσι, αναπτύχθηκε η σουίτα BLonD η οποία καλύπτει ένα ευρύ φάσμα εφαρμογών, σε επιταχυντές χαμηλών και υψηλών ενεργειών, καθώς και πληθώρα σωματιδίων όπως ηλεκτρόνια, πρωτόνια και ιόντα. Η δομή του λογισμικού επιτρέπει στους χρήστες τη ρύθμιση πολλαπλών χαρακτηριστικών της προσομοίωσης, συνδυάζοντας φυσικά φαινόμενα ανάλογα με τις απαιτήσεις του πειράματος.

Η αρχική έκδοση του BLonD γράφτηκε σε γλώσσα Python για απλότητα και διευκόλυνση της ανάπτυξης του και περιείχε ένα αναλυτικό μοντέλο δυναμικής

ακτίνων. Στην επόμενη έκδοση, τη BLonD++ [8], δημιουργήθηκε μια μαθηματική βιβλιοθήκη σε C++, η οποία υποστήριζε χρήση πολλαπλών νημάτων με το OpenMP [9] και επιτάχυνε σημαντικά τις προσομοιώσεις. Ο συνδυασμός της διεπαφής MPI [10] και του OpenMP οδήγησε στο HBLonD [11] το οποίο επωφελήθηκε σημαντικά από την απομακρυσμένη επικοινωνία διεργασιών. Τέλος, η ανάγκη για εκτενείς και απαιτητικές προσομοιώσεις δημιούργησε τον συνδυασμό του BLonD με τις Κάρτες Γραφικών (GPU) και την έκδοση CuBLonD, που περιγράφεται στην Ενότητα 1.3.2, και συνδυάζει την αρχιτεκτονική του HBLonD με έναν βελτιστοποιημένο πυρήνα σε γλώσσα CUDA για επιτάχυνση μέσω GPU. Σε αυτήν την εργασία παρουσιάζεται η ανάπτυξη αυτής της έκδοσης με μια καινούρια και ισχυρή βιβλιοθήκη, που περιγράφεται στην Ενότητα 1.3.2, ώστε να απλοποιηθεί η δομή του BLonD, επιτρέποντας την εύκολη παραμετροποίηση του κώδικα, και να επιτευχθεί καλύτερη απόδοση.

1.2.2 NVIDIA Κάρτες Γραφικών

Οι κάρτες γραφικών (GPUs) είναι από τα πιο σημαντικά υπολογιστικά εργαλεία σήμερα, με χρήση σε προσωπικό, ακαδημαϊκό και βιομηχανικό επίπεδο. Διαφέρουν από τις κεντρικές μονάδες επεξεργασίας (CPUs) καθώς είναι σχεδιασμένες να πραγματοποιούν παράλληλους υπολογισμούς με περισσότερα τρανζίστορ τα οποία αφοσιώνονται στην επεξεργασία δεδομένων. Χρησιμοποιούνται σε ένα ευρύ φάσμα εφαρμογών, όπως video rendering, παιχνίδια, και τεχνητή νοημοσύνη. Καθώς είναι ικανές να μειώσουν σημαντικά τον χρόνο εκτέλεσης απαιτητικών εφαρμογών, ξεκίνησαν να αξιοποιούνται στον βιομηχανικό και ακαδημαϊκό τομέα, υπό τον όρο ¨κάρτες γραφικών γενικού σκοπού (GPGPU)¨. Για αυτό αναπτύχθηκαν γλώσσες προγραμματισμού που βασίζονται σε παράλληλα δεδομένα (αρχιτεκτονική πολλαπλών πυρήνων), όπως οι NVIDIA CUDA [12], Brook [13], OpenCL [14] και hiCUDA [15].

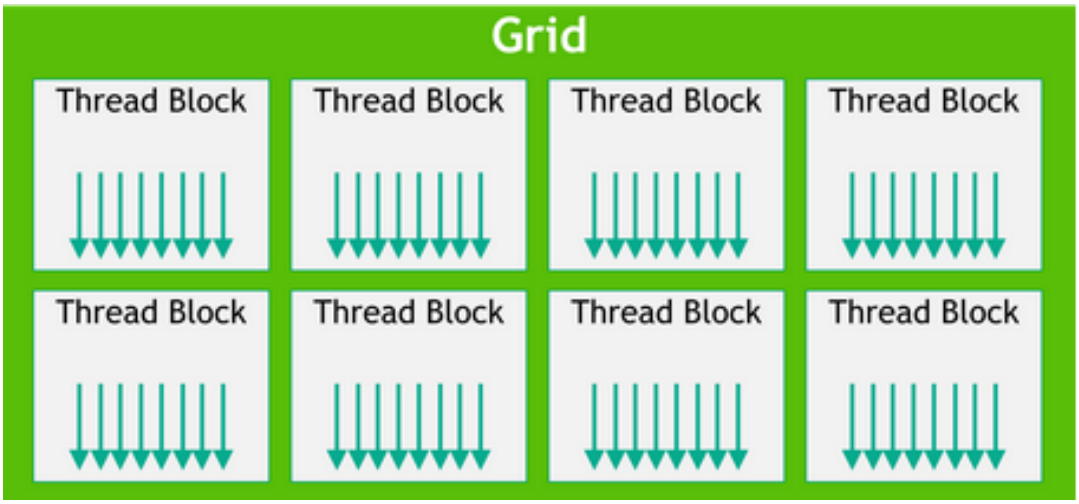
Για τον προγραμματισμό μια κάρτας γραφικών NVIDIA, της οποίας εταιρείας τα μοντέλα έχουν δοκιμαστεί εκτενώς στο λογισμικό BLonD, χρησιμοποιείται η γλώσσα CUDA C/C++, η οποία αποτελεί επέκταση της C/C++ ώστε να επιτρέπει την εκτέλεση συναρτήσεων σε πολλαπλά παράλληλα νήματα της κάρτας γραφικών. Ο χρήστης ορίζει συναρτήσεις, πυρήνες (kernels), που εκτελούνται N φορές παράλληλα από N διαφορετικά προγραμματιστικά νήματα (threads) CUDA. Ένας πυρήνας ορίζεται με το προσδιοριστικό *__global__* και καλείται όπως φαίνεται στο Listing 1.1, ο οποίος προσθέτει δύο διανύσματα A και B μεγέθους N σε ένα τρίτο διάνυσμα C, χρησιμοποιώντας 216 blocks και 1024 threads με κάθε thread να υπολογίζει πολλαπλά στοιχεία.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    for (int i = tid; i < N; i += blockDim.x * gridDim.x)
        C[i] = A[i] + B[i];
}

int main()
{
    ...
    int N;
    // Kernel invocation with 216 blocks of 1024 threads
    VecAdd<<<216, 1024>>>(A, B, C, N);
    ...
}
```

Listing 1.1: Κλήση Πυρήνα CUDA [12]

Τα νήματα CUDA καθορίζονται από το διάνυσμα **threadIdx** (μίας, δύο ή τριών διαστάσεων), σχηματίζοντας ένα block νημάτων. Τα blocks, που μπορούν να περιέχουν έως και 1024 νήματα σε σύγχρονες GPUs, συνενώνονται και σχηματίζουν ένα πολυδιάστατο (μίας, δύο ή τριών διαστάσεων) πλέγμα (grid), όπως φαίνεται στο Σχήμα 1.1. Ο συνολικός αριθμός των νημάτων που εκτελούνται ισούται με τον αριθμό των νημάτων ανά block επί τον αριθμό των blocks.



Σχήμα 1.1: Πλέγμα από blocks

Στις κάρτες γραφικών της NVIDIA, τα νήματα και τα blocks μπορούν να προσπελάσουν διαφορετικά είδη μνημών. Αρχικά κάθε νήμα προσπελάζει τη δική του ιδιωτική τοπική μνήμη και καταχωρητές και μια κοινή μνήμη εμφανή σε όλα τα

νήματα του block. Έπειτα, τα νήματα προσπελάζουν πιο γενικού τύπου μνήμες. Οι τρεις κυριότερες μνήμες της GPU είναι:

Κοινή Μνήμη (Shared Memory) Μνήμη on-chip με σημαντικά υψηλότερο εύρος ζώνης και χαμηλότερη καθυστέρηση από την καθολική μνήμη. Αποτελείται από τράπεζες μνήμης (memory banks) ίσου μεγέθους, οι οποίες μπορούν να προσπελαστούν ταυτόχρονα και να εξυπηρετούν πολλαπλά αιτήματα διαφορετικών διευθύνσεων, όταν δεν υπάρχουν συγκρούσεις · διαφορετικά νήματα που προσπελάζουν την ίδια τράπεζα μνήμης.

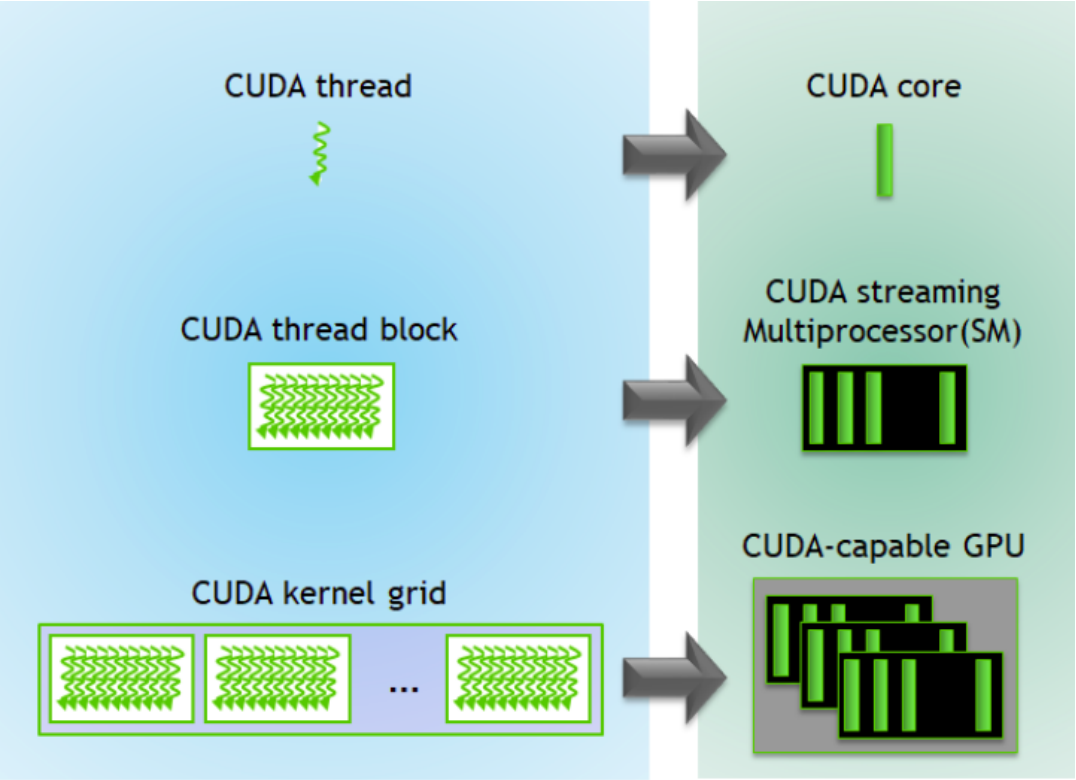
Καθολική Μνήμη (GLobal Memory) Μνήμη της συσκευής GPU η οποία προσπελάζεται μέσω συναλλαγών μνήμης μεγέθους 32, 64 ή 128 bytes. Η πρόσβαση στην καθολική μνήμη χαρακτηρίζεται από υψηλή καθυστέρηση και χαμηλό εύρος ζώνης. Για μέγιστη απόδοση θα πρέπει οι προσβάσεις στη μνήμη να είναι συνεχείς (coalesced), δηλαδή τα νήματα να προσπελάζουν συνεχόμενα blocks της μνήμης, π.χ., διπλανά νήματα που διαβάζουν διπλανά κελιά ενός πίνακα.

Διαχειρίσιμη Μνήμη (Managed Memory) Χώρος μνήμης προσπελάσιμος από την CPU και την GPU ταυτόχρονα, με κοινό χώρο διευθύνσεων. Επιτρέπει τον αποτελεσματικό διαμοιρασμό δεδομένων εξαλείφοντας την ανάγκη για αντιγραφή δεδομένων μεταξύ μνήμης CPU και GPU.

Σε επίπεδο υλικού, οι κάρτες γραφικών της NVIDIA περιέχουν επεξεργαστές με πολλαπλά νήματα (Streaming Multiprocessors-SMs). Πρόκειται για επεξεργαστές γενικού σκοπού με πυρήνες (για πράξεις κινητής υποδιαστολής μονής και διπλής ακρίβειας), εσωτερικούς καταχωρητές, κρυφές μνήμες για αποδοτική πρόσβαση στα δεδομένα και χρονο-προγραμματιστές warp. Ο όρος warp προσδιορίζει ένα group από 32 παράλληλα νήματα, που τα δημιουργεί, τα διαχειρίζεται, τα χρονοπρογραμματίζει και τα εκτελεί ένας SM. Όταν η GPU δημιουργεί ένα καινούριο πλέγμα πυρήνα (kernel grid), τα blocks νημάτων ανατίθενται στους διαθέσιμους SMs προς εκτέλεση όπως φαίνεται στο Σχήμα 1.2. Αυτή η αρχιτεκτονική ονομάζεται Μονής Εντολής, Πολλαπλών νημάτων (Single-Instruction, Multiple-Thread SIMT).

Όλα τα νήματα στο ίδιο warp εκτελούν τις ίδιες εντολές, σε οργάνωση Μονής Εντολής, Πολλαπλών Δεδομένων (Single Instruction Multiple Data SIMD) · μια εντολή ελέγχει πολλαπλά επεξεργαστικά στοιχεία. Όταν ένα block χωρίζεται σε warps από τον SM (τα νήματα 0-31 στο warp 1, τα νήματα 32-63 στο warp 2 κλπ), ο χρονο-προγραμματιστής δίνει προτεραιότητα στα έτοιμα προς εκτέλεση warps που δεν περιέχουν εξαρτήσεις δεδομένων. Αν είναι έτοιμα διαφορετικά warp, εφαρμόζεται πολιτική χρονο-προγραμματισμού. Μέγιστη απόδοση επιτυγχάνεται

όταν όλα τα 32 threads ακολουθούν το ίδιο μονοπάτι εκτέλεσης και κανένα νήμα δεν παρεκκλίνει λόγω διακλάδωσης υπό συνθήκη. Σε αυτή την περίπτωση, το warp εκτελεί κάθε μονοπάτι διακλάδωσης ξεχωριστά, απενεργοποιώντας τα νήματα που δεν παρεκκλίνουν.



Σχήμα 1.2: Εκτέλεση πυρήνα σε GPU

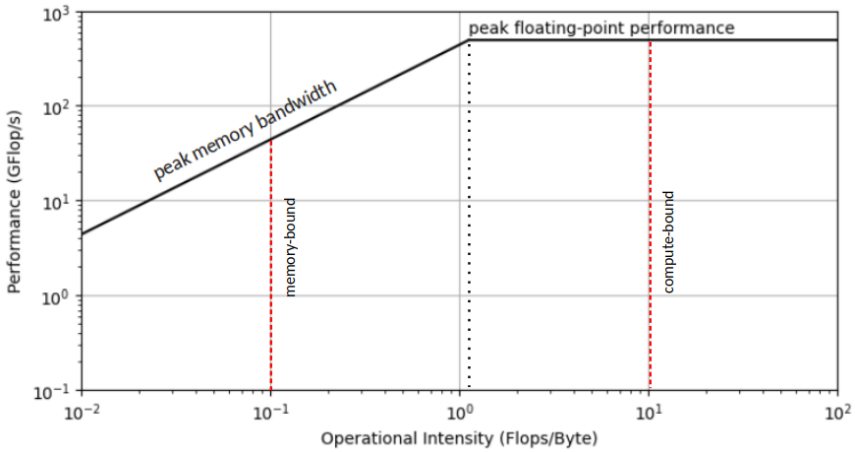
1.2.3 Μοντέλο Roofline

Το μοντέλο roofline είναι ένα εργαλείο γραφικής απεικόνισης, το οποίο συνδυάζει την απόδοση κινητής υποδιαστολής (floating point performance), την απόδοση της μνήμης και τη λειτουργική ένταση (operational intensity) [16]. Το τελευταίο μέγεθος χαρακτηρίζει την κίνηση μεταξύ των κρυφών μνημών και της μνήμης, δηλαδή το εύρος ζώνης της δυναμικής μνήμης τυχαίας προσπέλασης (DRAM) που χρειάζεται μια εφαρμογή υπό εκτέλεση. Για έναν επεξεργαστή, η μέγιστη υπολογιστική απόδοση και η απόδοση μνήμης μπορούν να βρεθούν μέσω των χαρακτηριστικών του υλικού ή με την εκτέλεση ειδικών προγραμμάτων (microbenchmarks).

Όπως φαίνεται στο Σχήμα 1.3, μια οριζόντια γραμμή απεικονίζει τη μέγιστη απόδοση κινητής υποδιαστολής του επεξεργαστή, το ανώτερο υπολογιστικό όριο για κάθε εκτελούμενο πυρήνα. Η μέγιστη απόδοση μνήμης μπορεί να υπολογιστεί διαιρώντας την απόδοση (GFlop/s) στον y-άξονα με τη λειτουργική ένταση (Flops/Byte) στον x-άξονα, που ισούται με την απόδοση μνήμης (GB/s). Η γραμμή

45° δίνει το όριο της απόδοσης μνήμης για συγκεκριμένη λειτουργική ένταση. Το σημείο τομής των δύο γραμμών δίνει τη μέγιστη υπολογιστική απόδοση και τη μέγιστη απόδοση μνήμης του επεξεργαστή. Η συνολική απόδοση μιας εφαρμογής περιορίζεται σύμφωνα με τον παρακάτω τύπο:

$$Performane(GFLOP/s) \leq min \begin{cases} Peak\ GFlop/s \\ Peak\ GB/s \times Operational\ Intensity \end{cases}$$



Σχήμα 1.3: Μοντέλο CPU Roofline

Για να χαρακτηριστεί μια εφαρμογή βάσει του μοντέλου roofline, θα πρέπει να υπολογιστούν η λειτουργική της ένταση και η απόδοση αυτής. Αν το σημείο που προκύπτει βρίσκεται δεξιά του σημείου τομής, η εφαρμογή χαρακτηρίζεται ως δεσμευμένη λόγω υπολογισμού (compute-bound), καθώς ξοδεύει τον περισσότερο χρόνο υπολογίζοντας δεδομένα. Αν βρίσκεται κάτω από την οριζόντια γραμμή, σημαίνει ότι η εφαρμογή δέχεται υπολογιστικές βελτιώσεις για καλύτερη απόδοση. Αν το σημείο βρίσκεται αριστερά του σημείου τομής, η εφαρμογή χαρακτηρίζεται ως δεσμευμένη λόγω μνήμης (memory-bound), καθώς ξοδεύει χρόνο μεταφέροντας δεδομένα από και προς τη μνήμη. Στην περίπτωση που το σημείο δε χτυπάει την κεκλιμένη ευθεία, σημαίνει ότι μπορούν να εφαρμοστούν τεχνικές για τη βελτιστοποίηση των προσβάσεων.

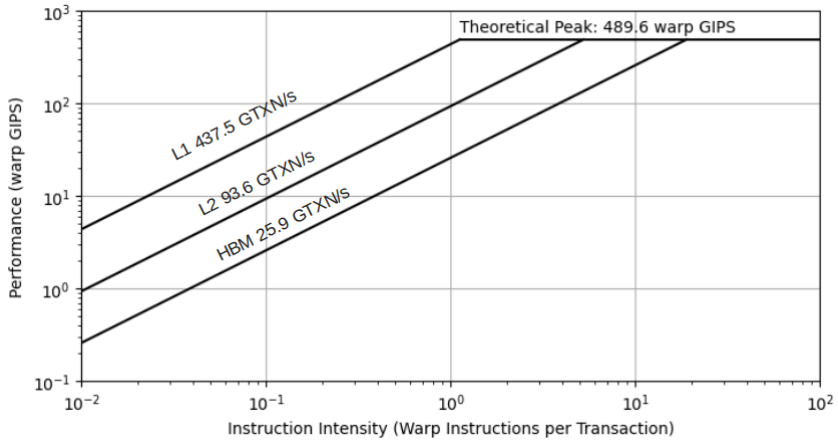
Για την εφαρμογή του μοντέλου roofline σε εφαρμογές που επιταχύνονται με κάρτες γραφικών, απαιτούνται κάποιες τροποποιήσεις εξαιτίας της διαφορετικής αρχιτεκτονικής τους. Αντί για την απόδοση κινητής υποδιαστολής, καταμετρώνται οδηγίες [17], καθώς επιτρέπουν την αναγνώριση συμφόρησης στη ροή εντολών (κλήση - αποκωδικοποίηση - έκδοση) και αξιοποίησης του αγωγού του επεξεργαστή (pipeline utilization). Για τη δημιουργία των αντίστοιχων γραμμών στο μοντέλο, πρέπει να υπολογιστεί η μέγιστη απόδοση σε εντολές ανά δευτερόλεπτο. Στην περίπτωση της κάρτας γραφικών NVIDIA Tesla V100, κάθε SM (συνολικά

80) περιέχει 4 χρονο-προγραμματιστές warp οι οποίοι εκδίδουν μία εντολή ανά κύκλο. Η μέγιστη απόδοση αυτής υπολογίζεται από τον παρακάτω τύπο:

$$Performance = 80(SMs) \times 4(warp\ scheds) \times 1(IPS) \times 1.53(GHz) = 489.6\ GIPS$$

Για την ανάλυση τον προσβάσεων στη μνήμη, χρησιμοποιείται ο όρος ¨συναλλαγή¨ (transaction) ως φυσική μονάδα [18]. Για τις κρυφές μνήμες L1, L2 και τη μνήμη υψηλού εύρους ζώνης (High Bandwidth Memory HBM) το μέγεθος της συναλλαγής είναι 32 bytes, ενώ για την κοινή μνήμη 128 bytes. Μια φόρτωση δεδομένων σε επίπεδο warp μπορεί να δημιουργήσει έως και 32 συναλλαγές. Για τη μοντελοποίηση των L1, L2 και HBM πρέπει να υπολογιστεί το αντίστοιχο εύρος ζώνης σε δισεκατομμύρια εντολές ανά δευτερόλεπτα (GTXN/s). Για την V100 τα εύρη ζώνης φαίνονται στο Σχήμα 1.4. Η απόδοση ενός πυρήνα που εκτελείται σε μια κάρτα γραφικών, μετρούμενο σε δισεκατομμύρια εντολές ανά δευτερόλεπτο (GIPS), περιορίζεται από το μέγιστο εύρος ζώνης της συσκευής (GTXN/s), την ένταση των οδηγιών (instruction intensity) και τη μέγιστη απόδοση της συσκευής σε GIPS, σύμφωνα με τον παρακάτω τύπο:

$$GIPS \leq min \begin{cases} Peak\ GIPS \\ Peak\ GTXN/s \times Instruction\ Intensity \end{cases}$$



Σχήμα 1.4: Μοντέλο GPU Roofline

Για τον χαρακτηρισμό ενός πυρήνα που επιταχύνεται με GPU απαιτούνται η ένταση οδηγιών και η απόδοση αυτού. Τα εργαλεία που χρησιμοποιούνται και οι κατάλληλες μετρικές για τον σχεδιασμό του μοντέλου roofline σε GPU, περιγράφονται στην Ενότητα 1.3.7.

1.3 Λεπτομέρειες Υλοποίησης

Στο κεφάλαιο αυτό παρουσιάζονται η προηγούμενη και η καινούρια έκδοση του BLonD καθώς και οι προσεγγίσεις που έγιναν για τη βελτιστοποίηση του λογισμικού. Δοκιμάζεται επίσης και η τεχνική thread-coarsening και παρουσιάζεται ένα εργαλείο που αναπτύχθηκε για την ανάλυση μέσω μοντέλου roofline σε Python με χρήση του εργαλείου NVIDIA profiler (nvprof).

1.3.1 Σχεδιασμός BLonD

΄Οπως περιγράφηκε στην Ενότητα 1.2.1, η σουίτα BLonD προσομοιάζει την κίνηση σωματιδίων στους επιταχυντές. Είναι γραμμένη κυρίως σε γλώσσα C++ και Python με μία εκτενής μαθηματική βιβλιοθήκη σε C++. Τα τρία βασικά στοιχεία που μοντελοποιούνται στο λογισμικό BLonD είναι:

1. Το σύγχροτρο ή ΄δαχτυλίδι΄
2. Η ακτίνα που κυκλοφορεί στον αγωγό
3. Οι κοιλότητες ραδιοσυχνοτήτων (RF cavities)

Η σουίτα BLonD χρησιμοποιεί Python για να αντιπροσωπεύσει τα παραπάνω στοιχεία και για να προσομοιάσει την περιστροφή των σωματιδίων, τα οποία περιγράφονται κυρίως από τις συντεταγμένες $(\Delta t_{(\mathrm{n})}, \Delta E_{(\mathrm{n})})$ για το χρόνο άφιξης και την ενέργεια στο RF τμήμα, με σημείο αναφοράς ένα εξωτερικό ρολόι. Ο χρήστης μπορεί να ορίσει τον αριθμό των RF σταθμών, από έναν έως δώδεκα. Σημαντικές συναρτήσεις για την παρακολούθηση των σωματιδίων είναι:

kick Ανανεώνει τη συντεταγμένη ΔE από το βήμα n στο $n+1$ με βάση τη συντεταγμένη Δt, και την αντίστοιχη ενέργεια που λαμβάνεται στον RF σταθμό.

drift Μοντελοποιεί την κίνηση της ακτίνας μεταξύ των RF σταθμών ανανεώνοντας τη συντεταγμένη Δt με χρήση της ανανεωμένης ενέργειας του σωματιδίου.

linear_interpolation_kick Αντικαθιστά τη συνάρτηση kick όταν η μεταβλητή *linear_interpolation* του αντικειμένου tracker είναι ενεργή.

induced_voltage_sum Προσθέτει τις συνεισφορές επαγόμενης τάσης.

histogram Παράγει το συνολικό προφίλ της ακτίνας.

Για μια πιο αναλυτική προσέγγιση στη δομή του BLonD, οι βασικές Python κλάσεις και τα αρχεία είναι:

Beam Κλάση που περιέχει τις παραπάνω συντεταγμένες της ακτίνας και τις ιδιότητες αυτής.

Profile Κλάση που περιέχει το beam profile και σχετικές ποσότητες συμπεριλαμβανομένου του εύρους της ακτίνας.

RingAndRFTracker Κλάση που επιτρέπει την παρακολούθηση συντεταγμένων των σωματιδίων για δεδομένο σταθμό RF και τμήμα του δαχτυλιδιού, έως τον επόμενο σταθμό. Περιέχει τη συνάρτηση *track* η οποία εφαρμόζει τις συναρτήσεις *kick* και *drift*.

butils_wrap Αρχείο που περιέχει συναρτήσεις, οι οποίες φορτώνουν τις αντίστοιχες υλοποιήσεις σε C++ από τη μαθηματική βιβλιοθήκη.

bmath Αρχείο που δημιουργεί ένα λεξικό Python (θα χρησιμοποιείται ο όρος dictionary), το οποίο περιέχει τα ονόματα των συναρτήσεων που ορίζονται στο butils_wrap και ανανεώνει το globals() dictionary ώστε να επιτρέψει γενική πρόσβαση σε αυτές τις συναρτήσεις. Επίσης, ενεργοποιεί την υλοποίηση MPI.

mpi_config Αρχείο που επιτρέπει την υλοποίηση MPI ώστε να χρησιμοποιηθούν πολλαπλές CPU ή GPU.

1.3.2 Εκδόσεις PyCUDA & CuPy

Η πρώτη έκδοση του λογισμικού BLonD για κάρτες γραφικών, που ονομάζεται CuBLonD, απαιτούσε την ανάπτυξη πυρήνων CUDA. Η ενσωμάτωση του κώδικα CUDA σε Python έγινε με χρήση των βιβλιοθηκών PyCUDA και Scikit-CUDA. Οι πυρήνες CUDA, που περιέχονται σε αρχεία **.cu**, μεταγλωττίζονται σε δυαδικά αρχεία CUDA χρησιμοποιώντας τον μεταγλωττιστή C της NVIDIA και φορτώνονται με κλήση της συνάρτησης *SourceModule* της PyCUDA. Η διαδικασία αυτή επιτρέπει απευθείας κλήσεις σε κώδικα CUDA από κάθε αρχείο Python του BLonD με σχεδόν μηδαμινό κόστος στην απόδοση.

Οι συναρτήσεις του BLonD χρησιμοποιούν πίνακες NumPy για αριθμητικούς υπολογισμούς. Για να επιταχυνθούν οι πιο απαιτητικοί υπολογισμοί με χρήση GPU, οι πίνακες αυτοί θα πρέπει να μεταφερθούν από τη CPU στη GPU. Στην PyCUDA αυτό επιτυγχάνεται μετατρέποντας όλους τους πίνακες *numpy.ndarray* σε αντικείμενα της κλάσης *GPUArray*. Για να γίνει αυτό, αναπτύχθηκε στο BLonD η κλάση *CGA*, η οποία περιέχει έναν πίνακα NumPy και έναν PyCUDA πίνακα στην GPU και τους συγχρονίζει αυτόματα (μέσω συναρτήσεων επαλήθευσης όταν πραγματοποιούνται αλλαγές στον έναν). Επιπλέον, κάθε κλάση που πρέπει να επιταχυνθεί μέσω GPU (π.χ. Beam, Profile) σχεδιάστηκε εκ νέου (π.χ. gpu_beam)

ώστε να κληρονομεί τις περισσότερες συναρτήσεις του αντικειμένου-γονέα και να υλοποιεί καινούριες συναρτήσεις που καλούν τους κατάλληλους πυρήνες CUDA, χρησιμοποιώντας πίνακες *CGA*. Σημαντικά αρχεία που τροποποιήθηκαν ή αναπτύχθηκαν είναι:

gpu_butils_wrap Περιέχει υλοποιήσεις σε PyCUDA που αντικαθιστούν αυτές του butils_wrap.

gpu_physics_wrap Περιέχει συναρτήσεις που καλούν τους αντίστοιχους πυρήνες CUDA.

bmath Τροποποιήθηκε για να αποθηκεύει ένα dictionary το οποίο φορτώνει τις συναρτήσεις από το gpu_butils_wrap ώστε το globals() dictionary να ανανεωθεί κατάλληλα.

gpu_cache Αποθηκεύει έναν χρησιμοποιημένο *GPUArray* σε ένα dictionary, και το ανακαλεί ξανά όταν μια διαφορετική συνάρτηση επιθυμεί πρόσβαση σε πίνακα ίδιων διαστάσεων, ώστε να αποφεύγονται περιττές δεσμεύσεις της GPU μνήμης.

Για να ενεργοποιηθούν οι λειτουργίες της GPU, απαιτούνται τα παρακάτω αρχεία:

__init__ Αρχείο που θέτει έναν αριθμό από blocks και νήματα ανά block, απαραίτητα για την κλήση των πυρήνων CUDA.

gpu_activation Αρχείο που ενεργοποιεί τις GPU υλοποιήσεις κλάσεων.

Η πολυπλοκότητα της PyCUDA υλοποίησης δημιούργησε την ανάγκη για μια καινούρια υλοποίηση, η οποία θα απλοποιήσει σημαντικά τη δομή του BLonD ενώ παράλληλα θα επιτρέψει περισσότερες λειτουργίες και θα διατηρήσει ή θα βελτιώσει τον χρόνο εκτέλεσης σε κάρτες γραφικών. Η βιβλιοθήκη CuPy ικανοποιεί τα παραπάνω κριτήρια, καθώς έχει διεπαφή παρόμοια με της NumPy και παρέχει υποστήριξη για πολλές λειτουργίες CUDA χαμηλού επιπέδου.

Αρχικά η δομή στην CuPy υλοποίηση είναι αρκετά απλοποιημένη, καθώς υποστηρίζονται περισσότερες απαραίτητες συναρτήσεις, οπότε δε χρειάζονται αρκετοί πυρήνες CUDA και το αρχείο *cupy_butils_wrap* περιέχει σημαντικά λιγότερες συναρτήσεις.

Η πιο σημαντική αλλαγή είναι οι τροποποιήσεις στο αρχείο *bmath* με στόχο να ενισχυθεί η ευκολία χρήσης του λογισμικού. Οι συναρτήσεις της CuPy που είναι παρόμοιες με της NumPy επιτρέπουν την εναλλαγή μεταξύ των δύο βιβλιοθηκών, με την αλλαγή της λέξης *"numpy."* σε *"cupy."* ακολουθούμενη από την επιθυμητή συνάρτηση. Στο αρχείο *bmath* δημιουργούνται δύο παρόμοια dictionaries:

_CPU_func_dict Dictionary που περιέχει τα ονόματα των συναρτήσεων του ***butils_wrap*** και επεκτείνεται με όλες τις απαραίτητες συναρτήσεις της NumPy.

_GPU_func_dict Dictionary που περιέχει τα ονόματα των συναρτήσεων του ***cupy_butils_wrap*** και επεκτείνεται με όλες τις απαραίτητες συναρτήσεις της CuPy.

Συνεπώς, πλέον δεν είναι απαραίτητη η ξεχωριστή υλοποίηση GPU των βασικών κλάσεων, ενώ κάθε τέτοια κλάση υποστηρίζει τις ακόλουθες συναρτήσεις:

to_gpu Μεταφέρει όλους τους απαραίτητους πίνακες στη GPU.

to_cpu Μεταφέρει όλους τους απαραίτητους πίνακες στη CPU.

Όταν ο χρήστης επιθυμεί να επιταχύνει μια κλάση μέσω GPU, καλεί τη συνάρτηση *to_gpu()* και στη συνέχεια χρησιμοποιούν τις ίδιες κλήσεις συναρτήσεων όπως στη βιβλιοθήκη NumPy. Αυτό επιτυγχάνεται καλώντας όλες τις συναρτήσεις μέσω των dictionaries στο αρχείο **bmath**.

Τα πλεονεκτήματα της υλοποίησης CuPy έναντι της PyCUDA συνοψίζονται στον Πίνακα 1.1. Η δομή της βιβλιοθήκης CuPy που ακολουθεί την NumPy επιτρέπει έναν εύκολα αναγνώσιμο κώδικα και αναιρεί την ανάγκη για επιπλέον αρχεία με GPU υλοποιήσεις. Έτσι, δημιουργείται απλουστευμένη δομή στο λογισμικό, ενώ ο ελαχιστοποιημένος CUDA πυρήνας παρέχει σημαντικό πλεονέκτημα απόδοσης με τις βελτιστοποιημένες συναρτήσεις του. Η νέα έκδοση GPU, με τις συναρτήσεις *to_gpu()* στις βασικές κλάσεις, ενισχύει την ικανότητα τροποποίησης του λογισμικού, καθώς ο χρήστης χρειάζεται ελάχιστες αλλαγές για να χρησιμοποιήσει την επιτάχυνση μέσω GPU. Τέλος, η υλοποίηση πολλαπλών NumPy συναρτήσεων στην CuPy επιτρέπει την επιτάχυνση επιπλέον λειτουργιών, που δεν μπορούν να επιταχυνθούν μέσω PyCUDA. Έτσι, επιτυγχάνεται επιπλέον κέρδος στη συνολική απόδοση.

Κατηγορία	PyCUDA	CuPy	Πλεονέκτημα CuPy
Δομή Πίνακα	Μοναδικός GPUarray	Δομή NumPy ndarray	Ευανάγνωστος κώδικας
Απαιτούμενα Αρχεία	Επιπλέον αρχεία GPU	Όχι επιπλέον αρχεία	Απλή δομή λογισμικού
Γραμμές CUDA	2600	350	Βελτιστοποιημένοι πυρήνες
Παραμετροποίηση	Δύσκολη	Εύκολη	Διευκόλυνση τροποποιήσεων
Επιταχυνόμενες Συν/εις	Περιορισμένες	Πλειοψηφία NumPy	Κέρδος απόδοσης

Πίνακας 1.1: Έκδοση CuPy vs PyCUDA

1.3.3 Δεξαμενές Μνήμης CuPy

Ορισμένες προσομοιώσεις του BLonD απαιτούν τη συνεχή χρήση των ίδιων πινάκων για εκτέλεση λειτουργιών, όπως υπολογισμοί FFT. Αν οι θέσεις μνήμης

αυτών δεσμεύονταν κάθε φορά από την αρχή, ο χρόνος εκτέλεσης θα ήταν σημαντικά μεγαλύτερος. Για να αποφευχθεί αυτή η διαδικασία δέσμευσης και αποδέσμευσης της μνήμης και οι δύο υλοποιήσεις του BLonD σε GPU εφαρμόζουν δεξαμενές μνήμης που διαχειρίζονται από το λογισμικό.

Η έκδοση PyCUDA χρησιμοποιεί μια υλοποιημένη δεξαμενή μνήμης, που ορίζεται στο αρχείο **gpu_cache** όπως ορίζεται στην Ενότητα 1.3.2. Ενεργοποιώντας το αρχείο αυτό, σημαντικές δομές της μνήμης αποθηκεύονται στην μνήμη και επιστρέφονται όταν ζητούνται, χρησιμοποιώντας την πολιτική "Least Recently Used", καθώς η δεξαμενή μνήμης διαθέτει προκαθορισμένη ποσότητα μνήμης. Με αυτόν τον μηχανισμό επιτυγχάνεται κέρδος απόδοσης περίπου 23% με 25%.

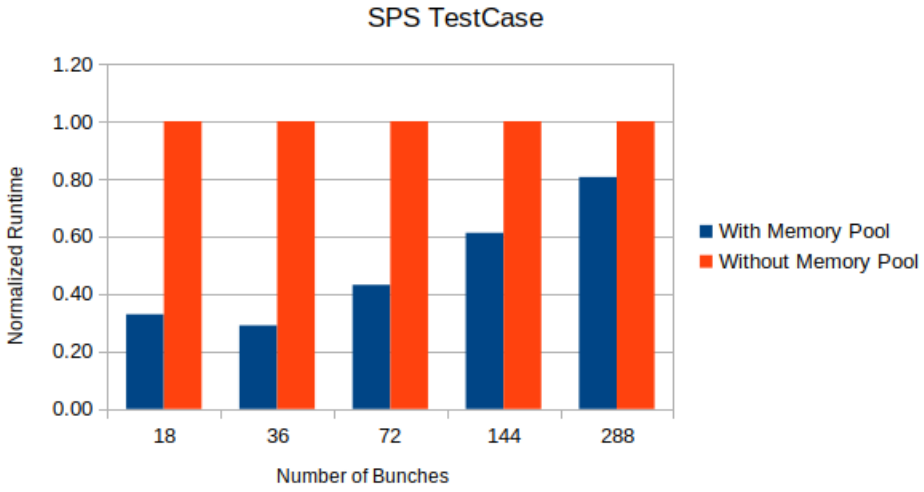
Η CuPy χρησιμοποιεί τη δική της δεξαμενή μνήμης, η οποία περιορίζει την επιβάρυνση των δεσμεύσεων μνήμης και του συγχρονισμού μεταξύ CPU και GPU. Δεν απαιτείται η υλοποίηση μιας δεξαμενής μνήμης, καθώς η CuPy διαθέτει δύο διαφορετικούς τύπους:

- Η δεξαμενή μνήμης της συσκευής (μνήμη GPU) για δεσμεύσεις μνήμης της GPU
- Η "Pinned" μνήμη (μη εναλλάξιμη μνήμη CPU) για μεταφορές μεταξύ CPU και GPU

Για να ελεγχθεί το κέρδος απόδοσης χρησιμοποιείται το SPS testcase ,που περιγράφεται στην Ενότητα 1.4, με ενεργοποιημένη και απενεργοποιημένη τη δεξαμενή μνήμης και τα αποτελέσματα παρουσιάζονται στο Σχήμα 1.5. Η δοκιμή πραγματοποιήθηκε και με τα δύο είδη μνήμης ενεργοποιημένα και στα αποτελέσματα γίνεται κανονικοποίηση με βάση την εκτέλεση με απενεργοποιημένες τις δεξαμενές μνήμης. Είναι φανερό ότι οι δεξαμενές μνήμης της CuPy παρέχουν σημαντικό κέρδος στην απόδοση. Με αυξανόμενο αριθμό δεσμών στο πείραμα, απαιτούνται περισσότερες δεσμεύσεις στη μνήμη και μεταφορές, οπότε η αποθήκευση στην κρυφή μνήμη μεγάλων πινάκων μπορεί να αυξήσει σημαντικά την αποτελεσματικότητα της εκτέλεσης.

1.3.4 Ιστόγραμμα Ακτίνας

Ένας σημαντικός CUDA πυρήνας του BLonD είναι ο **histogram**, ο οποίος είναι υπεύθυνος για την παραγωγή του beam profile χρησιμοποιώντας τις συντεταγμένες **dt** ως είσοδο. Ο πυρήνας αυτός αξιοποιεί την κοινή μνήμη της CUDA, δεσμεύοντας ένα beam profile πίνακα για κάθε block (private) μέσω ατομικών ενεργειών (atomic operations). Έπειτα αθροίζονται όλα στη global μνήμη για το συνολικό beam profile. Για να το κάνει αυτό, χωρίζει την ακτίνα σε ίσα μέρη (slices per bunch). Οι δύο εκδόσεις του **histogram** είναι:

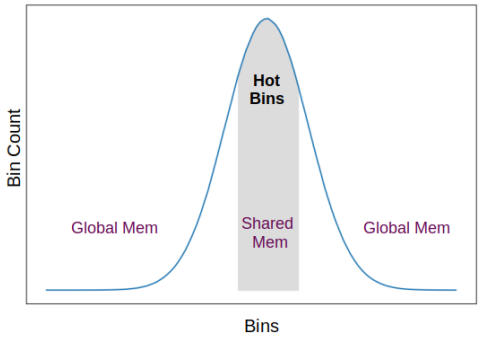


Σχήμα 1.5: Έλεγχος Δεξαμενών Μνήμης CuPy

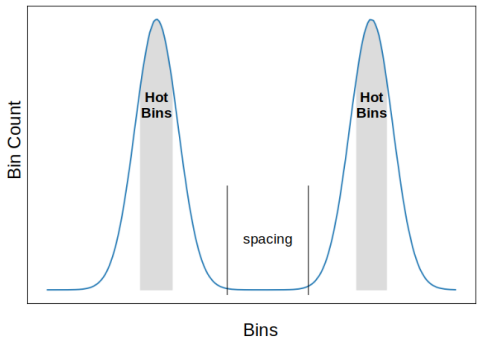
simple histogram Η έκδοση αυτή χρησιμοποιείται όταν ο πίνακας beam profile χωράει στην κοινή μνήμη

hybrid histogram Η έκδοση αυτή χρησιμοποιείται όταν μόνο ένα μέρος του πίνακα beam profile χωράει στην κοινή μνήμη. Καθώς η ακτίνα σωματιδίων εκδηλώνει Gaussian κατανομή, μόνο τα πιο σημαντικά ("hottest") σημεία γύρω από το κέντρο αποθηκεύονται στην κοινή μνήμη, όπως φαίνεται στο Φιγυρε 1.6α΄.

Αξιοποιώντας την κοινή μνήμη επιτυγχάνεται κέρδος απόδοσης έως και 51% [2] σε απαιτητικές προσομοιώσεις.



(α΄) Κατανομή μονής δέσμης



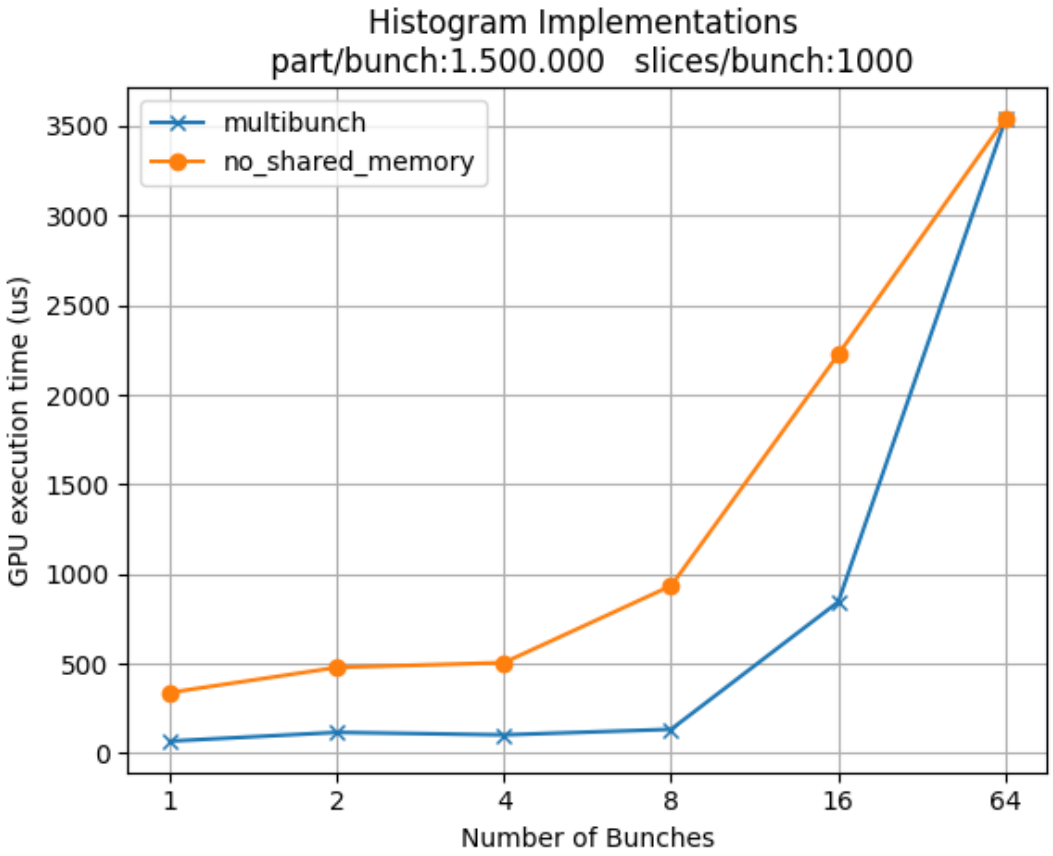
(β΄) Κατανομή διπλής δέσμης

Οι παραπάνω υλοποιήσεις, παρόλο που επιταχύνουν σημαντικά τους υπολογισμούς, αποτυγχάνουν να χειριστούν αποτελεσματικά τις περιπτώσεις πολλαπλών δεσμών της ακτίνας, αφού δε συμπεριλαμβάνουν το κενό (spacing) μεταξύ των δεσμών και τη διαφορετική κατανομή των σημαντικών σημείων. Για αυτό προτείνεται μια πιο γενική υλοποίηση, η οποία αξιοποιεί τις παρακάτω παραμέτρους:

- spacing: Ο κενός χώρος μεταξύ των δεσμών
- padding: Padding των σημείων
- n_bunches: Αριθμός δεσμών

Η κατανομή πολλαπλών δεσμών φαίνεται στο Σχήμα 1.6β΄. Το νήμα που εκτελεί την υλοποίηση αυτή ανακαλύπτει τη δέσμη που ανήκει το κάθε σημείο (**bunch_no**) και το κεντρικό σημείο, αξιοποιώντας το κενό μεταξύ των δεσμών (**total_bunch_space**. Έπειτα υπολογίζει τα όρια και αποθηκεύει τα αντίστοιχα σημεία στην κοινή μνήμη. Μετά τον συγχρονισμό των νημάτων, οι ίδιες παράμετροι χρησιμοποιούνται για τον υπολογισμό του σωστού δείκτη ώστε να αντιγραφούν οι τιμές αυτές από την κοινή στη global μνήμη.

Η απόδοση της νέας αυτής υλοποίησης φαίνεται στο Σχήμα 1.7. Σε αυτό, συγκρίνεται με μια υλοποίηση που δε χρησιμοποιεί την κοινή μνήμη, χρησιμοποιώντας 1.500.000 σωματίδια και 1000 slices per bunch. Είναι φανερό ότι η multi-bunch υλοποίηση επιδεικνύει καλύτερη απόδοση όσο οι δέσμες αυξάνονται, ενώ ταυτόχρονα παρέχει ακριβέστερα αποτελέσματα από τις προηγούμενες υλοποιήσεις.



Σχήμα 1.7: Κοινή μνήμη GPU για Συνάρτηση Histogram

1.3.5 Ανάλυση Block & Grid

΄Οπως αναφέρθηκε στην Ενότητα 1.2.2, ένας πυρήνας CUDA εκτελείται από πολλαπλά νήματα τα οποία φυσικά ανατίθενται σε πυρήνες CUDA. Τα νήματα αυτά ομαδοποιούνται σε blocks τα οποία ανατίθενται σε έναν SM για εκτέλεση. Μια ομάδα από blocks σχηματίζει ένα πλέγμα (grid) και η κάρτα γραφικών εκτελεί τον πυρήνα [19]. Για να εκτελεστεί ένας πυρήνας, απαιτείται ο ορισμός των δύο παρακάτω παραμέτρων:

- Grid Size: Αριθμός από block στο πλέγμα
- Block Size: Αριθμός νημάτων σε ένα block

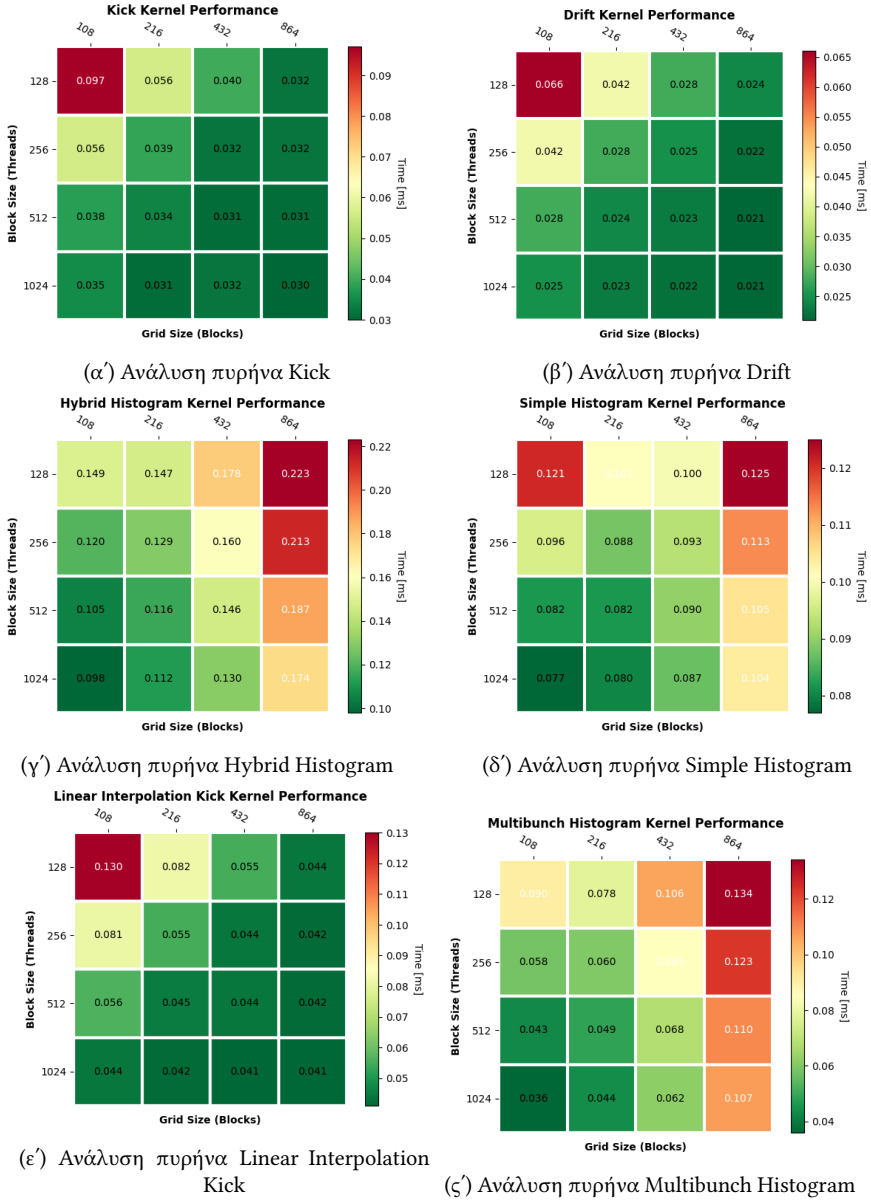
Στο BLonD οι διάφοροι πυρήνες CUDA καλούνται με χρήση της συνάρτησης *RawModule()*. Αυτό απαιτεί τον ορισμό μεγέθους block και grid, ώστε η GPU να εκτελέσει τον πυρήνα. Για την περαιτέρω ανάλυση της απόδοσης των blocks και του grid στους βασικούς πυρήνες, πραγματοποιήθηκαν πειράματα σε κάρτα γραφικών NVIDIA A100, που περιγράφεται στην Ενότητα 1.4.1. Συγκεκριμένα δοκιμάστηκαν οι πυρήνες *hybrid/simple/multibunch histogram, linear_interpolation_kick, kick* και *drift* με διαφορετικές παραμέτρους, ώστε να βρεθεί το βέλτιστο μέγεθος blocks και grid για καθέναν. ΄Οπως φαίνεται στο Σχήμα 1.8, όλοι οι πυρήνες επιτυγχάνουν ελάχιστο χρόνο εκτέλεσης για μέγεθος block τα 1024 νήματα, δηλαδή το μέγεθος "Max Threads per Block". Το αποτέλεσμα αυτό δε συμβαίνει για ίδιο μέγεθος grid σε κάθε πυρήνα, οπότε καθένας μπορεί να παραμετροποιηθεί ώστε να έχει βέλτιστη επίδοση.

Για να απλοποιηθεί η διαδικασία αυτή, ορίζεται ένας αριθμός και για τις δύο τιμές, με βάση τα αποτελέσματα των διάφορων πυρήνων:

- Grid Size = 2 * No. SMs
- Block Size = Max Threads per Block

΄Ολοι οι πυρήνες επομένως εκτελούνται με μέγεθος grid ίσο με τον διπλάσιο αριθμό των διαθέσιμων SMs και μέγεθος block ίσο με το μέγιστο δυνατό αριθμό των νημάτων που υποστηρίζει η κάθε GPU. Οι τιμές αυτές τίθενται ως συμβιβασμός με βάση την παραπάνω ανάλυση. Στο Σχήμα 1.9α΄ δοκιμάζονται διάφορα μεγέθη block σε κάρτα γραφικών NVIDIA Tesla A100, και το αποτέλεσμα είναι βέλτιστη επίδοση για 1024 (Max Threads per Block) νήματα. Στο Σχήμα 1.9β΄ δοκιμάζονται διάφορα μεγέθη grid στην ίδια κάρτα γραφικών, και το αποτέλεσμα είναι βέλτιστη επίδοση για 432 blocks (2*No. SMs), ενώ η διαφορά με τα 216 (2 * No. SMs) blocks είναι αμελητέα. Οπότε, τα προκαθορισμένα μεγέθη που αναφέρονται παραπάνω επιλέγονται ώστε να προσφέρουν αποτελεσματική επίδοση. Ωστόσο, εάν κάποιος

χρήστης θέλει να μεταβάλει τις παραμέτρους αυτές, μπορεί να ορίσει τις μεταβλητές περιβάλλοντος **GPU_BLOCKS** και **GPU_THREADS** για να ορίσει το μέγεθος του grid και του block αντίστοιχα.



(α′) Ανάλυση πυρήνα Kick

(β′) Ανάλυση πυρήνα Drift

(γ′) Ανάλυση πυρήνα Hybrid Histogram

(δ′) Ανάλυση πυρήνα Simple Histogram

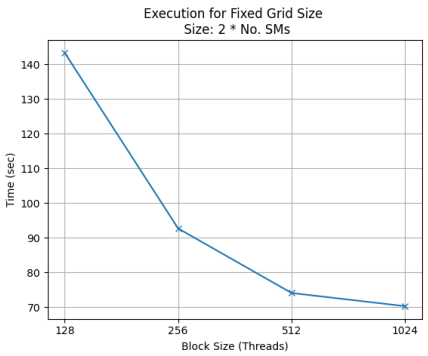
(ε′) Ανάλυση πυρήνα Linear Interpolation Kick

(ς′) Ανάλυση πυρήνα Multibunch Histogram

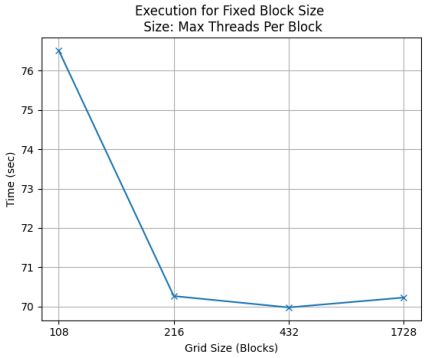
Σχήμα 1.8: Παράμετροι εκτέλεσης πυρήνων (block size, grid size)

1.3.6 Τεχνική Thread Coarsening

Μια μέθοδος βελτιστοποίησης για την ενίσχυση της απόδοσης των πυρήνων CUDA είναι η thread coarsening [20] [21], η οποία συγχωνεύει δύο ή περισσότερα παράλληλα νήματα, μειώνοντας τον συνολικό αριθμό τους αυξάνοντας τη δουλειά ενός μοναδικού thread. Οι δύο βασικές τεχνικές thread coarsening είναι:



(α′) Δοκιμή με σταθερό μέγεθος grid



(β′) Δοκιμή με σταθερό μέγεθος block

Thread-Level Σε αυτήν την τεχνική παραμένει σταθερός ο συνολικός αριθμός των blocks (grid size), αλλά ο συνολικός αριθμός των νημάτων μειώνεται, καθώς κάθε block λειτουργεί με λιγότερα νήματα.

Block-Level Σε αυτήν την τεχνική παραμένει σταθερός ο αριθμός των νημάτων ανά block, αλλά ο συνολικός αριθμός των blocks (grid size) μειώνεται ανάλογα με τον παράγοντα coarsening.

Για να υποστηρίζει ένας πυρήνας την τεχνική αυτή, πρέπει να υποστεί ορισμένες τροποποιήσεις. Αυτές φαίνονται στον Πίνακα 1.2 και εξασφαλίζουν ότι η εκτέλεση αυτού παράγει ορθό αποτέλεσμα, ενώ πραγματοποιούνται συνεχόμενες (uncoalesced) προσβάσεις στη μνήμη. Αυτό οφείλεται στο βήμα (stride) **S**, το οποίο οδηγεί τα διπλανά νήματα ενός block να προσπελάζουν διπλανά κελιά μνήμης, ώστε να μη δημιουργούνται διαμάχες. Μεταβάλλοντας τον παράγοντα coarse (**C**), επιτυγχάνεται μεγαλύτερη μείωση των διαθέσιμων νημάτων και blocks.

CUDA function	**After thread-level coarse**	**After block-level coarse**
threadIdx	$\lfloor \frac{threadIdx}{S} \rfloor \cdot S \cdot C$ $+(threadIdx) mod(S) + i \cdot S$	threadIdx
blockIdx	blockIdx	$C \cdot blockIdx + i \cdot S$
gridDim * blockDim	gridDim * blockDim	$C \cdot gridDim \cdot blockDim + i \cdot S$
blockDim	$C \cdot blockDim$	blockDim
gridDim	gridDim	$C \cdot gridDim$
blockIdx * blockDim + threadIdx	rewrite using def: $blockIdx * blockDim + threadIdx$	

Πίνακας 1.2: Υλοποίηση Thread Coarsening,με Stride S, Coarsening Factor C, και i ως Index του Coarsened Thread ή του Thread Block ώστε $0 \leq i < C$

Για να ελεγχθούν αν αυτές οι μέθοδοι ενισχύουν την απόδοση βασικών πυρήνων του BLonD, οι πυρήνες *linear_interpolation_kick* και*kick* εκτελέστηκαν σε κάρτα γραφικών NVIDIA A100. Παρόλο που οι πυρήνες του BLonD είναι ήδη βελτιστοποιημένοι ώστε να ελαχιστοποιούνται οι προσβάσεις στη μνήμη, οι παραπάνω

πυρήνες δοκιμάζονται με την τεχνική thread-level coarsening. Ως αναφορά χρησιμοποιήθηκε εκτέλεση με 1024 threads ανά block και grid με 216 blocks. Οι πυρήνες εκτελέστηκαν με 1.500.000 σωματίδια, 192 δέσμες και 192.000 slices, με τα παρακάτω αποτελέσματα:

Χρόνος πυρήνα(μs):		kick	linear inter kick
coarse, block size	-, 1024	32	42
	2, 512	32	42
	2, 256	34	45
	4, 512	36	45
	4, 256	37	46

΄Οπως φαίνεται, η τεχνική αυτή δεν προσφέρει ιδιαίτερη απόδοση στους συγκεκριμένους πυρήνες. Αρχικά, οι πυρήνες αυτοί είναι βελτιστοποιημένοι ώστε να επιδεικνύουν μέγιστη επίδοση από πλευράς σχεδιασμού. Συνεπώς, η κάρτα γραφικών δεν προσφέρει απόδοση καθώς δεν οφείλεται σε ικανότητα υπολογισμού. Δεύτερον, οι περιορισμένοι πόροι ενός SM (όπως καταχωρητές και κοινή μνήμη) οδηγούν στη μείωση του occupancy με τον συνδυασμό πολλαπλών νημάτων ενός block στην τεχνική thread-level coarsening. Η ίδια συμπεριφορά προκύπτει και στη block-level coarsening, όπου κάθε block αναλαμβάνει αυξημένο φορτίο και επομένως αξιοποιεί περισσότερους πόρους.

1.3.7 Εργαλείο Roofline

Για την αξιολόγηση της αποδοτικότητας των πυρήνων αναπτύχθηκε ένα εργαλείο roofline σε Python. Η προτεινόμενη μεθοδολογία [17], αξιοποιεί τις παρακάτω μετρικές που συλλέγονται με χρήση του nvprof [22] ή του ncu [23]:

- inst_executed_thread: αριθμός εντολών που εκτελούνται
- gld/gst_transactions: αριθμός global συναλλαγών της L1
- shared_load/store_transactions: αριθμός shared συναλλαγών της L1
- l2_read/write_transactions: αριθμός συναλλαγών της L2
- dram_read/write_transactions: αριθμός συναλλαγών της HBM

Χρησιμοποιώντας τις παραπάνω μετρήσεις υπολογίζονται η ένταση των εντολών (instruction intensity) σε εντολές warp ανά συναλλαγή (Warp Instructions per Transaction) και η απόδοση του πυρήνα σε δισεκατομμύρια εντολές warp ανά δευτερόλεπτο (Warp Giga Instructions per Second) για δεδομένη κρυφή μνήμη:

- $instruction\_intensity = \frac{inst\_executed\_thread/32}{No\ transactions}$

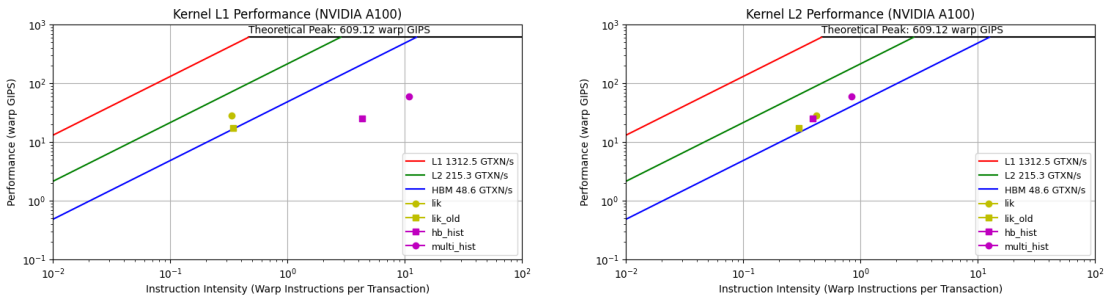
- $performance = \frac{inst\_executed\_thread/32}{10^9 * run\_time}$

Για την κρυφή μνήμη L1, ο αριθμός των συναλλαγών ισούται με τις συνολικές global και shared συναλλαγές. Για τις L2 και HBM, ο αριθμός των συναλλαγών ισούται με το σύνολο των συναλλαγών L2 και HBM αντίστοιχα.

Το εργαλείο χρησιμοποιείται για την αξιολόγηση βασικών πυρήνων του BLonD, όπως φαίνεται στο Σχήμα 1.10, όπου συγκρίνονται οι παλιές με τις βελτιστοποιημένες εκδόσεις των πυρήνων *linear_interpolation_kick* και *histogram*. Στον πρώτο πυρήνα μεταβλήθηκε το μοτίβο πρόσβασης στη μνήμη, ώστε να μειωθούν οι μεταφορές και να αυξηθεί η επίδοση. Για την προσομοίωση χρησιμοποιήθηκε η κάρτα γραφικών NVIDIA A100. Η μέγιστη απόδοση των L1, L2 και HBM καθώς και το μέγιστο IPS που φαίνονται, αντλήθηκαν από το εγχειρίδιο της GPU [24]. Εμπειρικά δεδομένα μπορούν να βρεθούν χρησιμοποιώντας micro-benchmarks [25]

Στο Σχήμα 1.10α΄, παρουσιάζεται η απόδοση των πυρήνων στην L1 cache. Αρχικά, ο νέος πυρήνας *multibunch_histogram* επιδεικνύει καλύτερη πρόσβαση στη μνήμη και καλύτερη επίδοση από τον *hybrid_histogram*, ενώ και οι δύο είναι compute-bound, καθώς πραγματοποιούν προσβάσεις στην κοινή μνήμη της L1 cache (σύγκριση με κόκκινη γραμμή). Δεύτερον, ο πυρήνας *linear_interpolation_kick* έχει καλύτερη υπολογιστική επίδοση μετά τις βελτιστοποιήσεις, ενώ και οι δύο εκδόσεις είναι memory-bound λόγω των προσβάσεων στην κύρια μνήμη. Στο σχήμα Σχήμα 1.10α΄, παρουσιάζεται η απόδοση των πυρήνων στην L2 cache. Όλοι οι πυρήνες επιδεικνύουν συμπεριφορά memory-bound (σύγκριση με πράσινη γραμμή), καθώς η L2 cache εξυπηρετεί τις προσβάσεις στην κοινή μνήμη, με τους βελτιστοποιημένους πυρήνες να έχουν καλύτερη επίδοση.

Οπότε, με το εργαλείο "GPU-Roofline-Python" που είναι διαθέσιμο στο GitHub [26], αξιολογήθηκαν σημαντικοί πυρήνες του BLonD μετά από βελτιστοποιήσεις. Τα αποτελέσματα υποδεικνύουν ότι οι τροποποιήσεις αυτές ωφέλησαν την εκτέλεση του BLonD.



(α΄) Μοντέλο Roofline για L1 cache

(β΄) Μοντέλο Roofline για L2 cache

Σχήμα 1.10: Αξιολόγηση Πυρήνων με το εργαλείο Roofline

1.4 Αξιολόγηση

Για την αξιολόγηση της νέας έκδοσης CuPy του BLonD χρησιμοποιούνται τρία testcases τα οποία αξιοποιούν διαφορετικά σύγχροτρα του CERN και επιδεικνύουν ένα ευρύ φάσμα λειτουργιών. Από το μικρότερο στο μεγαλύτερο μηχάνημα, τα testcases είναι τα ακόλουθα:

1. **Proton Synchrotron (PS)** Το testcase αυτό χρησιμοποιεί το δεύτερο σύγχροτρο της αλυσίδας εγχυτήρων του LHC, με περιφέρεια 628m και ισχύ επιτάχυνσης πρωτονίων ενέργειας έως 26 GeV και 18 δέσμες ταυτόχρονα.

2. **Super Proton Synchrotron (SPS)** Το SPS επιταχύνει τα σωματίδια που λαμβάνονται από το PS μέχρι τα 450 GeV. Έχει περιφέρεια 7 χιλιομέτρων, είναι ένα από τα μεγαλύτερα μηχανήματα και μπορεί να δεχθεί έως και τέσσερις παρτίδες των 72 δεσμών.

3. **Large Hadron Collider (LHC)** Ο LHC είναι ο μεγαλύτερος και ισχυρότερος επιταχυντής σωματιδίων στον κόσμο, με περιφέρεια 27 χιλιομέτρων και ενέργεια σύγκρουσης 13TeV ή περισσότερο. Μπορεί να δεχτεί έως και 2808 δέσμες σωματιδίων από το SPS.

1.4.1 Πειραματικός Εξοπλισμός

Για την εκτέλεση των πειραμάτων σε CPU χρησιμοποιείται πολυνηματική υλοποίηση ώστε να μειώσει τον συνολικό χρόνο εκτέλεσης. Επίσης, αξιολογούνται δεδομένα μονής και διπλής ακρίβειας, καθώς το BLonD παρέχει τη δυνατότητα αυτή. Το μοντέλο CPU που χρησιμοποιείται είναι το **EPYC 7302** της AMD με τις ακόλουθες προδιαγραφές:

- Αριθμός Πυρήνων : 16
- Αριθμός Νημάτων : 32
- Βασικό Ρολόι (GHz) : 3.0

Η υποδομή του CERN προσφέρει πρόσβαση σε μοντέλα GPU για την εκτέλεση πειραμάτων. Παρακάτω παρουσιάζεται ένας συγκριτικός πίνακας αυτών:

Μοντέλο NVIDIA Tesla	Turing T4	Volta V100S	Ampere A100
Έκδοση CUDA	7.5	7.0	8.0
Αριθμός Πυρήνων CUDA	2560	5120	6912
Αριθμός SMs	40	80	108
Μέγεθος Μνήμης (GB)	16	32	40
Βασικό Ρολόι (MHz)	585	1245	765
Αρ. Τρανζίστορ (Εκατ/μυρια)	13.600	21.100	54.200
Επιφάνεια ψηφίδας (mm^2)	545	815	826

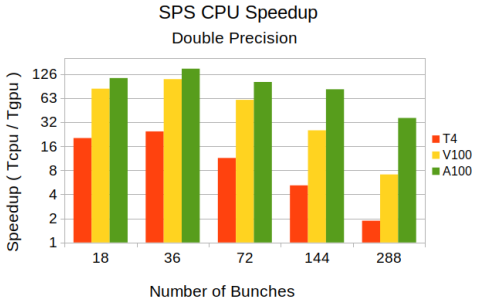
1.4.2 Σύγκριση με CPU

Στην ενότητα αυτή, παρουσιάζονται γραφήματα στα οποία φαίνεται η επιτάχυνση που επιτυγχάνει η έκδοση CuPy ($T_{\text{CPU}}/T_{\text{GPU}}$) στα διάφορα μοντέλα GPU έναντι της εκτέλεσης σε 16 πυρήνες CPU με ένα thread ανά πυρήνα, για τα τρία πειράματα που αναφέρθηκαν παραπάνω. Οι παράμετροι που χρησιμοποιούνται για την εκτέλεση των πειραμάτων παρουσιάζονται στον παρακάτω πίνακα:

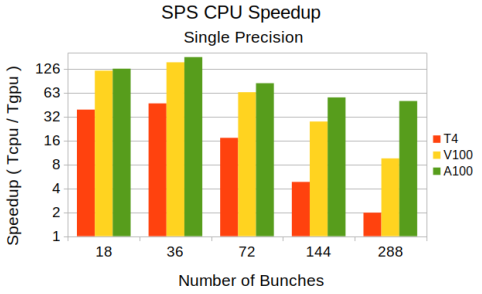
Configuration	PS	SPS	LHC
particles/bunch	1, 2, 4, 8, 16M	1M	1,5M
bunches	21	18, 36, 72, 144, 288	12, 24, 48, 96, 192
slices/bunch	256	1408	1000
turns	10,000		
precision	Double, Single		

Τα αποτελέσματα για τα SPS, PS και LHC φαίνονται στα Σχήματα 1.11, 1.12 και 1.13 αντίστοιχα, με την περίπτωση του SPS να παρουσιάζεται σε λογαριθμικό άξονα για ευκολία απεικόνισης. Σε όλα παρατηρείται η σημαντική αύξηση της επίδοσης με χρήση ισχυρών μοντέλων GPU. Ειδικά η NVIDIA A100 προσφέρει έως και 175 φορές καλύτερο χρόνο εκτέλεσης από τη CPU στην περίπτωση του SPS για 36 δέσμες και δεδομένα μονής ακρίβειας. Αυτό οφείλεται στην υπολογιστική πολυπλοκότητα του συγκεκριμένου πειράματος λόγω απαιτητικών παραμέτρων και μεγάλων δεδομένων εισόδου. Ο συνδυασμός πολλαπλών SMs και αυξημένης χωρητικότητας μνήμης, επιτρέπει στα μοντέλα V100 και A100 να υπολογίζουν αποδοτικά μεγάλο όγκο δεδομένων, παρόλο που στο SPS παρατηρείται μια πτώση στην επιτάχυνση με αύξηση των δεσμών, λόγω των απαιτητικών παραμέτρων που επιβάλλουν μεγάλη πίεση ειδικά στην κοινή μνήμη της GPU μέσω της συνάρτησης histogram. Επιπλέον, όλα τα μοντέλα GPU εμφανίζουν καλύτερη επίδοση στην

περίπτωση δεδομένων μονής ακρίβειας, κάτι που οφείλεται στις ταχύτερες μονάδες υπολογισμού FP32 έναντι FP64.

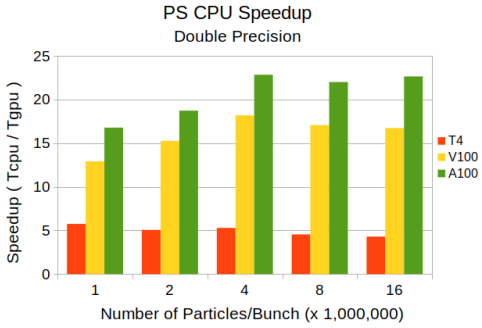


(α′) Επιτάχυνση για δεδομένα διπλής ακρίβειας

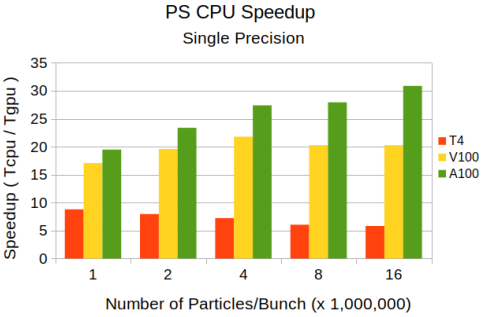


(β′) Επιτάχυνση για δεδομένα μονής ακρίβειας

Σχήμα 1.11: Επιτάχυνση του SPS

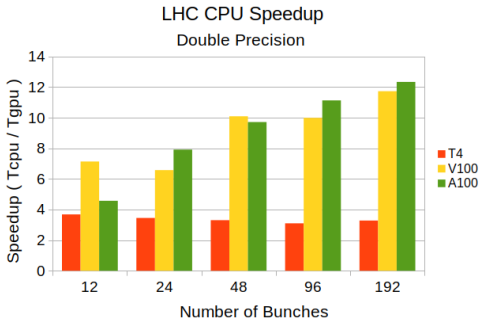


(α′) Επιτάχυνση για δεδομένα διπλής ακρίβειας

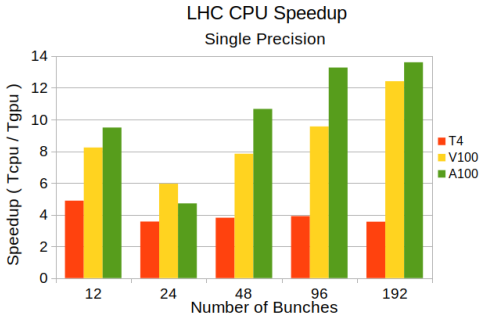


(β′) Επιτάχυνση για δεδομένα μονής ακρίβειας

Σχήμα 1.12: Επιτάχυνση του PS



(α′) Επιτάχυνση για δεδομένα διπλής ακρίβειας



(β′) Επιτάχυνση για δεδομένα μονής ακρίβειας

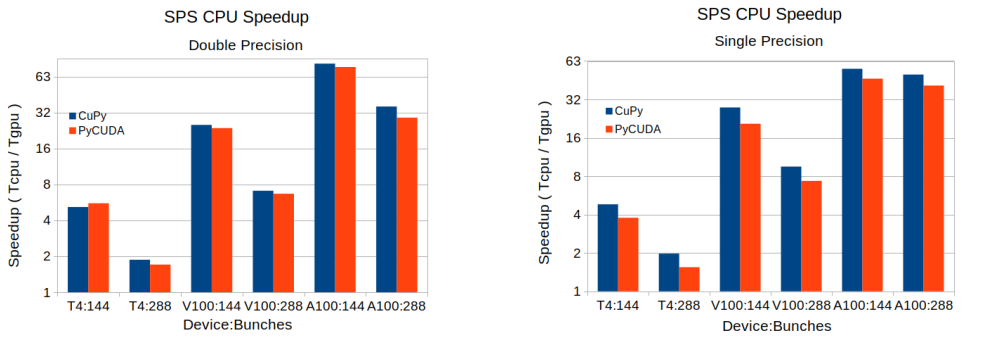
Σχήμα 1.13: Επιτάχυνση του LHC

1.4.3 Σύγκριση με προηγούμενη έκδοση

Στην ενότητα αυτή, η έκδοση CuPy του BLonD συγκρίνεται με την προηγούμενη υλοποίηση PyCUDA. Για να ελαχιστοποιηθούν τα συνολικά πειράματα, συγκρίνονται τα δύο μεγαλύτερα configurations κάθε πειράματος για δεδομένα μονής και διπλής ακρίβειας. Επομένως, εκτελούνται τα ακόλουθα πειράματα:

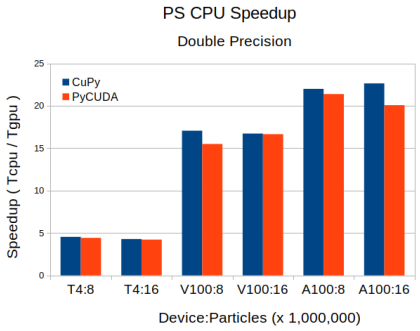
Configuration	PS	SPS	LHC
particles/bunch	1M, 16M	1M	1,5M
bunches	21	18, 288	12, 192
slices/bunch	256	1408	1000
turns	10,000		
precision	Double		

Τα αποτελέσματα για τα SPS, PS και LHC φαίνονται στα Σχήματα 1.14, 1.15 και 1.16 αντίστοιχα, με την περίπτωση του SPS να παρουσιάζεται σε λογαριθμικό άξονα για ευκολία απεικόνισης. Η CuPy υλοποίηση επιδεικνύει σημαντικά καλύτερη απόδοση στα περισσότερα configurations. Λόγω της δομής της, η CuPy επιτρέπει την επιτάχυνση περισσότερου κώδικα, άρα και μικρότερους χρόνους εκτέλεσης. Ακόμη και σε απαιτητικά πειράματα, όπως τα SPS και LHC, η έκδοση CuPy BLonD καταφέρνει να ξεπεράσει την έκδοση PyCUDA. Για το SPS, η CuPy επιτυγχάνει μέγιστο 34% περισσότερη επιτάχυνση από την έκδοση PyCUDA στην εκτέλεση V100:144:single. Αυτό αυξάνεται σε 30% για το PS στην εκτέλεση A100:8:single, ενώ για το LHC γίνεται 26% στην εκτέλεση V100:96:double. Οπότε, παράλληλα με την απλοποίηση της δομής του λογισμικού και την αυξημένη προγραμματιστική ευκολία, η υλοποίηση CuPy επιτυγχάνει καλύτερα αποτελέσματα από την προκάτοχο της υποδεικνύοντας την επίτευξη των σημαντικών στόχων της αναβάθμισης αυτής.

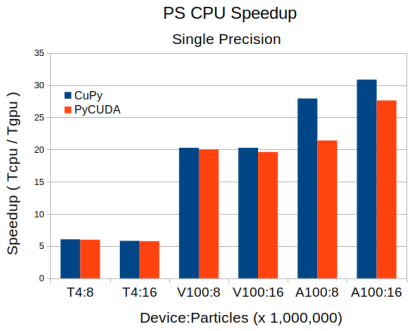


(α′) Σύγκριση για δεδομένα διπλής ακρίβειας (β′) Σύγκριση για δεδομένα μονής ακρίβειας

Σχήμα 1.14: Σύγκριση με PyCUDA του SPS

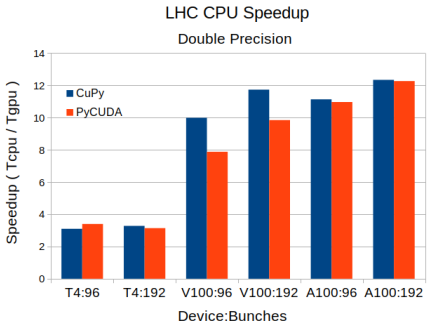


(α′) Σύγκριση για δεδομένα διπλής ακρίβειας

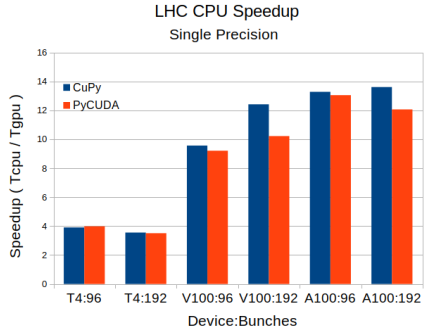


(β′) Σύγκριση για δεδομένα μονής ακρίβειας

Σχήμα 1.15: Σύγκριση με PyCUDA του PS



(α′) Σύγκριση για δεδομένα διπλής ακρίβειας



(β′) Σύγκριση για δεδομένα μονής ακρίβειας

Σχήμα 1.16: Σύγκριση με PyCUDA του LHC

1.5 Συμπεράσματα και Μελλοντικές Επεκτάσεις

1.5.1 Συμπεράσματα

Η εργασία αυτή παρουσιάζει την αναβάθμιση του λογισμικού BLonD [1]. Με στόχο την αποτελεσματικότητα, την απόδοση και μια ενισχυμένη εμπειρία χρήστη, η σουίτα BLonD τροποποιήθηκε ώστε να αξιοποιεί τη βιβλιοθήκη CuPy σε Python για επιτάχυνση με χρήση GPU [3], η οποία αντικατέστησε τη βιβλιοθήκη PyCUDA [27]. Η αλλαγή αυτή δημιούργησε μια πιο απλουστευμένη δομή στο λογισμικό και αύξησε τη συνολική απόδοση.

Η αναβάθμιση απαιτούσε τη δοκιμή διαφόρων δομών υλικού και λογισμικού, όπως τις δεξαμενές μνήμης της CuPy και τον τρόπο εκτέλεσης των πυρήνων CUDA, που μπορούν να ωφελήσουν την απόδοση του BLonD. Επίσης, εξετάστηκε η τεχνική thread-coarsening και αναπτύχθηκε ένα εργαλείο για την αναπαράσταση του μοντέλου roofline ώστε να αξιολογηθούν διάφοροι πυρήνες του BLonD.

Τέλος, η νέα υλοποίηση σε CuPy αξιολογήθηκε συγκρίνοντας ορισμένα απαιτητικά testcases σε τρία διαφορετικά μοντέλα NVIDIA GPU έναντι 16 πυρήνων CPU. Αυτό ανέδειξε το σημαντικό πλεονέκτημα σε απόδοση της έκδοσης CuPy

και της χρήσης GPU για επιτάχυνση λογισμικού. Παράλληλα με την προγραμματιστική ευκολία και την απλή δομή του λογισμικού που προσφέρει, η έκδοση CuPyεπέδειξε καλύτερη επίδοση από την έκδοση PyCUDA για όλα τα πειράματα που πραγματοποιήθηκαν.

Η υλοποίηση CuPy κατάφερε να αλλάξει σημαντικά τη δομή της σουίτας BLonD. Ένα εργαλείο που χρησιμοποιείται από επιστήμονες στο CERN αλλά και παγκοσμίως, μπορεί πλέον να αξιοποιήσει αποτελεσματικά κάρτες γραφικών για να επιταχύνει την εκτέλεση του και να μειώσει σημαντικά τον απαιτούμενο χρόνο εκτέλεσης. Εκτός αυτού, η δομή του λογισμικού είναι απλή και εύκολα τροποποιήσιμη, οπότε μπορεί να καλύψει κάθε διαφορετική ανάγκη.

1.5.2 Μελλοντικές Επεκτάσεις

Παρόλο που έγιναν σημαντικές αλλαγές στη σουίτα BLonD, υπάρχει ακόμη αρκετός χώρος για περαιτέρω βελτιστοποιήσεις. Μερικές από αυτές είναι:

- Η ενσωμάτωση πολλαπλών GPU μέσω MPI μπορεί να οδηγήσει σε μειωμένους χρόνους εκτέλεσης. Εκτός από την παρούσα υλοποίηση MPI σε CPU, στην οποία πολλαπλές CPU μοιράζονται τον υπολογιστικό φόρτο ώστε να αυξήσουν την επιτάχυνση της εφαρμογής, πολλαπλές GPU μπορούν επίσης να λειτουργήσουν παράλληλα. Χρησιμοποιώντας τη διεπαφή MPI, αρκετές GPU μπορούν να μοιραστούν εξίσου τον υπολογιστικό φόρτο και να επιτύχουν καλύτερη επίδοση από μία GPU.

- Η χρήση διαφορετικών GPUs στις προσομοιώσεις είναι μια καλή λύση, στις περιπτώσεις που διάφορα μοντέλα GPU είναι διαθέσιμα. Αν για παράδειγμα υπάρχουν τα μοντέλα NVIDIA V100 και A100 προς χρήση, το λογισμικό BLonD θα πρέπει να χειριστεί αποτελεσματικά αυτά τα δύο ώστε να επιταχύνει τους υπολογισμούς.

- Μια "έξυπνη" υβριδική υλοποίηση του λογισμικού BLonD, με CPU και GPU να μοιράζονται τους έντονους υπολογισμούς, μπορεί να έχει σημαντικά πλεονεκτήματα. Ενώ η έκδοση CuPy επιτυγχάνει στο να χρησιμοποιεί τη GPU για τους πιο απαιτητικούς υπολογισμούς, όπως πράξεις FFT, επιπλέον βελτιώσεις μπορούν αυξήσουν την αποτελεσματικότητα της εκτέλεσης. Όπως φάνηκε και στα διάφορα testcases, ορισμένοι πυρήνες CUDA απαιτούν μεταφορές μνήμης, οι οποίες αυξάνουν τον χρόνο εκτέλεσης αφού πρέπει να κληθούν δεδομένα από τη CPU. Οπότε, οι memory-bound πυρήνες μπορούν να εκτελεστούν σε περιβάλλον CPU ενώ οι compute-bound μπορούν να εκτελεστούν αποδοτικά σε GPU μέσω της διεπαφής OpenMP.

- Η δοκιμή των μοντέλων GPU της AMD αποτελεί επίσης το επόμενο βήμα στην ανάπτυξη του BLonD. Τα τελευταία χρόνια η AMD καταλαμβάνει όλο και μεγαλύτερο μέρος στην παγκόσμια αγορά, επομένως η δοκιμή της απόδοσης του BLonD στα μοντέλα αυτής είναι απαραίτητη. Καθώς ο στόχος του BLonD είναι να καλύπτει ένα μεγάλο εύρος αναγκών, είναι σημαντική η ανάλυση της συμπεριφοράς του λογισμικού σε διάφορα συστήματα, όπως οι GPUs της AMD που θα παρέχουν πληροφορίες για τη μελλοντική εξέλιξη της σουίτας BLonD.

Chapter 2

Introduction

2.1 CERN Accelerator complex & Beam Dynamics

At the European Organization for Nuclear Research, also known as CERN, physicists and engineers strive to understand how the universe works on a fundamental level. To achieve that, they have developed the world's largest and most powerful particle accelerator, the Large Hadron Collider (LHC), which has provided great scientific achievements, including the Nobel-prize-winning discovery of the Higgs boson in 2011. Beam dynamics is the field of physics that studies particle motion in synchrotrons and can be divided into longitudinal and transverse beam dynamics, which focus on the longitudinal and on the transverse particle motion respectively.

While several longitudinal beam dynamics simulation tools exist, since 2014 the BLonD suite [1] has been developed at CERN and is currently used by labs worldwide. It is designed for the simulation of the longitudinal motion and the tracking of energy and time coordinates of beam particles in synchrotrons, and with its modular structure, it covers the need for intensive custom simulations, as described in Sections 3.2 and 4.2.

2.2 Need for HPC Beam Dynamics Simulations

Beam dynamics simulations could be very demanding and require a vast amount of execution time, as they are comprised of complex computations. Depending on the processing power, this could range from several hours to weeks or even months. Single computer nodes are unable to cope with such demanding computational needs, thus, High-Performance Computing (HPC) methods need to be employed, to enable the cooperation of multiple nodes, both CPUs and GPUs, to calculate large-scale simulations in a reasonable amount of time.

For this reason, CERN has dedicated significant funds to continuously upgrading its computer ecosystem with the latest-generation CPUs and GPUs and providing

them to users for testing and development. In an endeavor to explore a vast range of physics phenomena, CERN conducts a diverse research program covering multiple physics topics, like the Standard Model, supersymmetry, exotic isotopes and cosmic rays. These experiments are computationally intensive, for the aforementioned reasons, thus, they are accelerated with the use of powerful GPU models. Due to the same characteristics of BLonD simulations, the suite is a good fit for GPU acceleration.

2.3 Proposal Overview

During the years, BLonD has undertaken significant modifications, to support intensive calculations in shorter execution times. As described in section 4.2, several HPC techniques have been applied and a vast amount of code has been developed to accommodate simulation needs. The latest BLonD version [2] harnesses GPU models, by using the Python PyCUDA library [28] to accelerate computations. However, despite the significant performance advantage achieved with the GPU, this implementation fails to provide an efficient user experience and programming ease. This thesis proposes the employment of a modern Python library called CuPy [3], that provides a simple software structure while leveraging all the GPU acceleration capabilities to provide significant performance gain, as described in sections 3.4.2 and 4.2.2. To evaluate this implementation, fundamental metrics are utilized, namely, the achieved speedup compared to CPU acceleration and the PYCUDA version, the total line of codes needed (both Python and CUDA), the amount of accelerated code, and the software simplicity that facilitates future development.

2.4 Thesis structure

The rest chapters of this thesis are organized as follows:

- In chapter 3, HPC-related concepts and background knowledge, necessary for BLonD development, are presented
- In chapter 4, the BLonD modifications are presented and described; the techniques used for the software upgrade and the tools created for its performance analysis.
- In chapter 5, execution results of intensive BLonD testcases in the new CuPy version are presented. The achieved GPU speedup is discussed and a comparison with the previous version is made.
- In chapter 6, a brief description of the work done in this thesis is displayed, and brief proposals for future BLonD development are provided.

Chapter 3

Prior Art

3.1 Introduction

In this chapter, topics essential to this thesis are presented. First, the BLonD simulation suite is described in Section 3.2, followed by an overview of the NVIDIA GPU technology in Section 3.3. Second, the PyCUDA and CuPy libraries for CUDA programming in Python are briefly presented in Section 3.4. Finally, in Section 3.5, the Roofline Model and its GPU alternative are presented.

3.2 Beam Longitudinal Dynamics Suite

The Beam Longitudinal Dynamics simulation suite BLonD [1], [4] is an open-source software package for the simulation of the longitudinal motion and the tracking of energy and time coordinates of beam particles in synchrotrons. Since 2014, BLonD has been continuously developed, thoroughly benchmarked, and applied for every existing and future synchrotrons of CERN. Before the development of the BLonD suite, the simulations at CERN have been performed using the ESME [5] suite, a longitudinal beam dynamics code developed at Fermilab in 1984. However, it lacked constant development and support and therefore became obsolete. Similar simulators, namely Py-Orbit [6] and Elegant [7], have been used but also lacked a variety of simulation features. The BLonD suite covers the need for a wide range of applications, from low to high-energy synchrotrons, from electrons over protons to ions, and from space-charge to synchrotron-radiation dominated regimes. With its modular structure, it provides users with many specification options, enabling the combination of physics phenomena according to the experiment's requirements.

The original version was written in Python to enable simplicity and rapid development, and supported a detailed beam dynamics model. For the following version, BLonD++ [8], a C++ computational backend was introduced, which supported multi-threading with OpenMP [9] and significantly increased the simulation speedup.

The combination of MPI [10] with OpenMP resulted in HBLonD [11] that greatly profited from remote process communication. Finally, the need for intensive calculations in minimum execution time, led to the integration of GPUs in BLonD's structure and the CuBLonD version, presented in Section 4.2, which combines the HBLonD architecture with an optimized CUDA core for GPU acceleration. The work presented in this thesis concerns the upgrade of the GPU implementation with a modern and powerful Python library, described in Section 3.4.2, in order to simplify the software structure enabling effortless customization and achieving greater execution speedup.

3.3 Graphics Processing Unit

The Graphics Processing Unit (GPU) is currently one of the most significant tools in computer science and technology, for personal, academic, and industrial use. The GPU differs from the CPU, as it is designed for highly parallel computations with more transistors devoted to data processing rather than caching and flow control. Originally it was introduced in the 1970s as a programmable processing unit for graphics rendering (under the term graphics processor unit), and later rebranded from NVIDIA with the introduction of GeForce 256 Graphics Processing Unit. Nowadays, GPUs are used in a wide range of applications, including graphics and video rendering, gaming, and artificial intelligence. As GPUs are capable of significantly reducing the workload of intensive problems, a new term has been introduced, namely, General-Purpose GPUs.

3.3.1 General-Purpose GPUs

Realizing the potential of GPUs, computer scientists strived to extend their use outside the scope of traditional computer graphics. With the evolution of GPU programmability, General-Purpose GPUS (GPGPUs), are becoming increasingly popular for application acceleration in the industrial domain with the spike of machine learning and high-performance computing applications. GPU architecture has evolved into a flexible and unified many-core architecture, and many non-graphics data-parallel languages have emerged, such as NVIDIA CUDA [12], Brook [13], OpenCL [14], and hiCUDA [15].

3.3.2 NVIDIA Programming Model

There are two major GPU manufacturers, NVIDIA and AMD, with distinct GPU architectures and programming models. BLonD has been thoroughly tested on NVIDIA GPUs, but testing AMD models is a future goal of the suite's development.

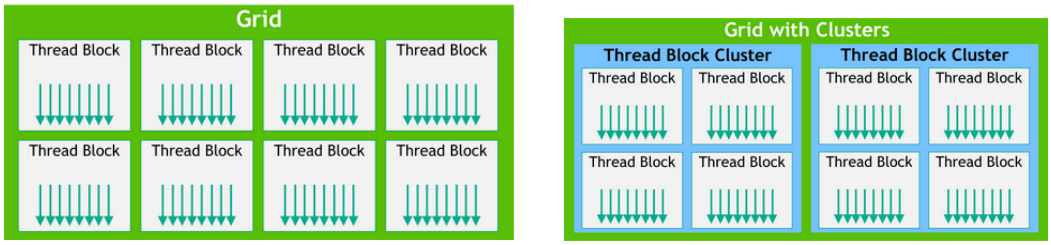
To program an NVIDIA GPU, CUDA C/C++ is used, which extends the C/C++ programming language (.cu file extension) to accommodate function execution on multiple parallel GPU threads. The user defines C++ functions, called **kernels**, which are executed N times in parallel by N different CUDA programming threads. A kernel is defined using the *__global__* specifier and called with the syntax shown in Listing 3.1, which adds two N-size vectors A and B into a third vector C, using 216 blocks and 1024 threads, with each thread calculating multiple unique elements.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    for (int i = tid; i < N; i += blockDim.x * gridDim.x)
        C[i] = A[i] + B[i];
}

int main()
{
    ...
    int N;
    // Kernel invocation with 216 blocks of 1024 threads
    VecAdd<<<216, 1024>>>(A, B, C, N);
    ...
}
```

Listing 3.1: CUDA Kernel Example [12]

CUDA threads are defined by the thread index, the **threadIdx** vector, forming a one, two, or three-dimensional block of threads; a thread block. Thread blocks, which can contain up to 1024 threads on modern GPUs, are grouped together forming a one, two, or three-dimensional grid of thread blocks, as seen in Figure 3.1a. Each block is respectively defined by the block index, the **blockIdx** vector, and its dimensions can be obtained with the **blockDim** vector. The total number of threads executed is equal to the number of threads per block times the number of blocks. New NVIDIA GPU models which come with compute capability 9.0 (e.g. NVIDIA H100 Tensor Core GPU for data centers), support optional Thread Block Clusters that are made up of thread blocks, as seen in Figure 3.1b.



(A) Grid of Thread Blocks (B) Grid of Thread Block Clusters

FIGURE 3.1: CUDA grid visualization [12]

3.3.3 Memory Hierarchy

In an NVIDIA GPU, threads and thread blocks can access different types of memory. First, each thread accesses its own private local memory and registers and a shared memory visible to every thread in a block. Second, threads can access more general types of memory. The three most important addressable memory spaces include:

Shared Memory On-chip memory with much higher bandwidth and lower latency than global memory. It is divided into equally-sized memory banks, that can be accessed simultaneously and can service multiple distinct address requests, where no bank conflicts (different threads accessing the same memory bank) are present.

Global Memory Device memory (GPU memory) accessed via 32, 64, or 128-byte memory transactions. Access to device memory is characterized by high latency and low bandwidth. For maximum global memory throughput data accesses should be coalesced, meaning that threads request data in continuous memory blocks, e.g., adjacent threads reading adjacent array elements.

Managed Memory Memory space accessible from both the CPU and GPU coherently, with a common address space. It enables effective data sharing by eliminating the need for copying data between CPU and GPU memory.

3.3.4 Hardware

From the hardware perspective, GPUs contain a scalable array of multithreaded Streaming Multiprocessors (SMs). An SM is a general-purpose processor containing execution cores (for single, double-precision floating-point operations and special function units), internal registers, and caches for efficient data accesses (e.g. L1 cache and shared memory) and warp schedulers. The term "warp" defines a group of 32 parallel threads, that an SM creates, manages, schedules, and executes. As the GPU invokes a kernel grid, the thread blocks are assigned to available SMs for

execution, as seen in Figure 3.2. This SM architecture is called Single-Instruction Multiple-Thread (SIMT).

All threads in the same warp execute the same instruction in a Single Instruction Multiple Data (SIMD) organization; a single instruction controls multiple processing elements. As a block is divided into warps by the SM (threads 0-31 to warp 1, threads 32-63 to warp 2, etc.), the warp scheduler prioritizes the ready-to-execute warps that contain no data dependencies. If multiple warps are ready, a scheduling policy assigns the next fetched instruction and kernel execution continues similarly. Maximum warp efficiency is achieved when all 32 threads follow the same execution path and no thread diverges via a conditional branch. In that case, the warp executes each branch path taken, while disabling threads that did not diverge.

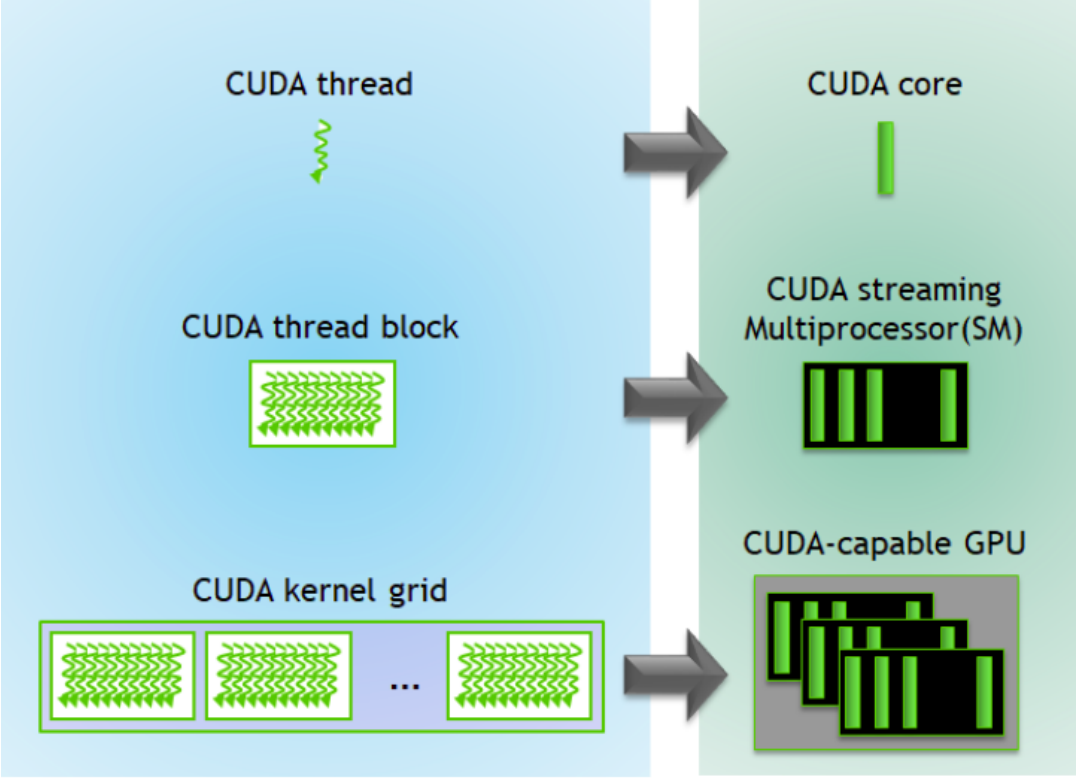


Figure 3.2: GPU kernel execution Hardware

3.4 CUDA in Python

CUDA programming in C/C++ can be complex, especially for individuals unfamiliar with the structure of a C/C++ program. Many scientific frameworks and simulation suites are developed by scientists who, in most cases, are more experienced with the Python language as it accommodates a simple programming structure and is used for fields like machine learning and scientific computing. Therefore, CUDA wrappers have been developed to tackle this problem and provide an easy and efficient

alternative. They successfully communicate with the GPU device, utilizing low-level CUDA commands, while enabling the use of the Python language to interact with the CUDA backend. For the BLonD suite, the wrappers used are PyCUDA and CuPy, which are described in the following sections.

3.4.1 PyCUDA

PyCUDA is a Python programming environment for CUDA [28]. Based on the CUDA driver API, it enables efficient GPU code acceleration through a convenient programming interface. Its basic advantages over similar CUDA API wrappers are:

- Complete access to CUDA driver API
- Automatic management of resources (object cleanup)
- Automatic error checking
- Speed provided by C++ base layer
- Integration with NumPy

PyCUDA offers several classes that make CUDA programming convenient, with the most important being:

SourceModule Class that creates a Module from CUDA source code.

GPUArray An array structure that stores its data and performs its computations on the computing device.

An example to demonstrate the capabilities of PyCUDA is matrix multiplication, shown in Listing 3.2. The *SourceModule* class is used to compile a *CUDA C* kernel; in this case, the kernel doubles each element of the given array. The *GPUArray* class offers a simpler method of multiplying a NumPy array, as shown in Listing 3.3:

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
import numpy

mod = SourceModule("""
  __global__ void doublify(float *a)
  {
    int idx = threadIdx.x + threadIdx.y*4;
    a[idx] *= 2;
  }
  """)

a = numpy.array(([1.1,2.2],[3.3,4.4])).astype(numpy.float32)
```

```
a_gpu = cuda.mem_alloc(a.nbytes)
cuda.memcpy_htod(a_gpu, a)

func = mod.get_function("doublify")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
```

Listing 3.2: PyCUDA SourceModule

```
import pycuda.gpuarray as gpuarray
import pycuda.autoinit
import numpy

a = numpy.array(([1.1,2.2],[3.3,4.4])).astype(numpy.float32)
a_gpu = gpuarray.to_gpu(a)
a_doubled = (2*a_gpu).get()
```

Listing 3.3: PyCUDA GPUArray

Apart from the convenience that the *GPUArray* class offers, it supports a wide variety of NumPy-array arguments, such as *shape, size* and *dtype*, and functions under the categories of Trigonometric, Exponential, Reductions, etc. All of these can be found in the official documentation [27] and their use is beyond the scope of this thesis.

3.4.2 CuPy

CuPy [3] is an array library for GPU-accelerated computing with Python. It is versatile and efficient as it supports multiple operations with a simple programming interface. CuPy also utilizes CUDA Toolkit libraries like cuBLAS to provide extended usage of GPU architecture. Its greatest advantages over the PyCUDA environment are the following:

- Highly compatible with NumPy & SciPy
- Drop-in replacement to NumPy
- Most of NumPy/SciPy modules supported
- Extensive low-level CUDA support
- Thorough Documentation

CuPy implements a subset of the NumPy interface. Its basic array component, the *ndarray* class, is used for creating arrays that are allocated directly on the current device (the default GPU). The NumPy-identical syntax makes CuPy programming accessible even to inexperienced Python users. The CuPy equivalent of the previous matrix multiplication example is presented in listings 3.4 and 3.5:

```
import cupy as cp
import numpy as np

mod = cp.RawModule("""
  __global__ void doublify(float *a)
  {
    int idx = threadIdx.x + threadIdx.y*4;
    a[idx] *= 2;
  }
  """)

a_gpu = cp.array(([1.1,2.2],[3.3,4.4])).astype(np.float32)

func = mod.get_function("doublify")
func(args=(a_gpu), block=(4,1,1), grid=(4,1,1))

a_cpu = cp.asnumpy(a_gpu)
```

Listing 3.4: CuPy RawModule

```
import cupy as cp
import numpy as np

a_gpu = cp.array(([1.1,2.2],[3.3,4.4])).astype(np.float32)
a_doubled = 2*a_gpu
a_cpu = cp.asnumpy(a_doubled)
```

Listing 3.5: CuPy ndarray

Apart from the programming simplicity that CuPy offers, a significant advantage over PyCUDA is that the direct allocation of an array on the GPU device greatly reduces execution times as host-to-device memory transfers are not needed. The corresponding result can remain in the GPU until a CPU calculation is needed or until the data need to be transferred to another CPU function.

3.5 Roofline Model

Application optimization is an important step in software development. In order to ensure that the requested application achieves maximum performance on the target architecture (CPU or GPU), performance analysis is required. The Roofline Model is a visual tool that combines floating-point performance, memory performance, and operational intensity [16]. The latter characterizes traffic between caches and memory, i.e., the DRAM bandwidth needed by a running application. For a specific processor, peak computational and memory performance can be obtained by hardware specifications or microbenchmarks.

As seen in Figure 3.3, a horizontal line depicts the processor's peak floating point performance, the upper computational bound for any kernel. The peak memory performance can be calculated by dividing the achieved performance (GFlop/s) in the y-axis by the operational intensity (Flops/Byte) in the x-axis, which equals memory performance (GB/s). This line at a 45-degree angle gives the memory performance bound for a specific operational intensity. The intersection of these two lines provides the processor's peak computational and memory performance. The total performance of an executed application is limited according to the following formula:

$$Performance(GFLOP/s) \leq min \begin{cases} Peak\ GFlop/s \\ Peak\ GB/s \times Operational\ Intensity \end{cases}$$

To characterize an application using the Roofline model, its operational intensity and performance need to be calculated. If the resulting point lies to the right of the intersection point, then the application is compute-bound, meaning that the application spends most of its running time calculating data. Lying beyond the horizontal peak performance line means that the computational algorithm can be enhanced for better execution. If the resulting point lies to the left of the intersection point, then the application is characterized as memory-bound, meaning that it spends time transferring data to/from memory. In case the point does not hit the slanted part of the roof, then memory optimization techniques can be applied.

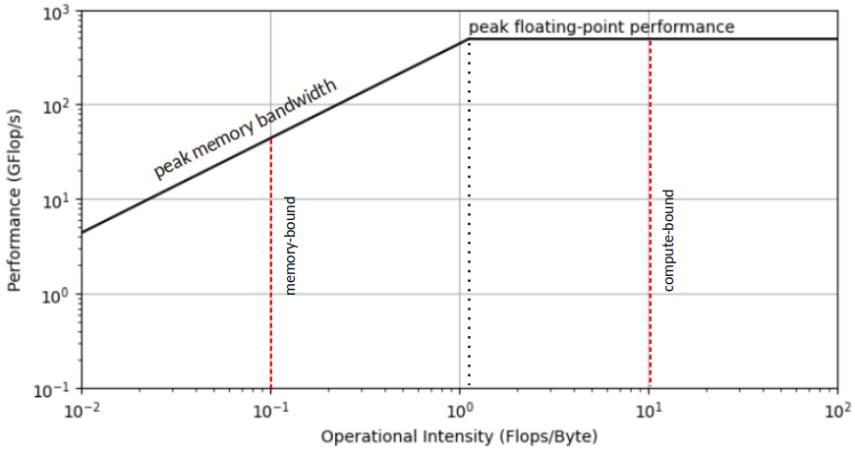


FIGURE 3.3: Roofline CPU Model

3.5.1 Roofline for GPUs

Applying the Roofline model for GPU-accelerated applications [17], requires some modifications, due to the different nature of GPU architecture. Instead of floating-point performance, instruction counting is used, as it enables the identification of fetch-decode-issue bottlenecks and pipeline utilization. Further analysis is possible for a GPU Roofline, considering global and shared memory accesses and also thread predication, which can have a significant impact on the total performance. However, for a simple analysis, the maximum performance can be solely used, as in the CPU Roofline.

To create the ceilings for a specific GPU, the peak performance in instructions per second needs to be calculated. For the NVIDIA Tesla V100 GPU (described in Section 5.2.1), each SM (80 in total) contains four warp schedulers that can dispatch one instruction per cycle. Therefore, the peak performance is calculated as follows:

$$Performance = 80(SMs) \times 4(warp\ scheds) \times 1(IPS) \times 1.53(GHz) = 489.6\ GIPS$$

To analyze memory accesses, the "transaction" is used as the natural unit [18]. For global/local memory, namely the L1 cache, the L2 cache, and the HBM memory, the transaction size is 32 bytes, while for the shared memory it is 128 bytes. A warp-level load may create up to 32 transactions. To model the ceilings for L1, L2 cache, and HBM, the respective bandwidth in billions of transactions per second (GTXN/s) needs to be calculated. For the V100, the bandwidths are visible in Figure 3.4. A kernel's performance in the GPU, measured in billions of instructions per second (GIPS), is limited by the peak machine bandwidth (GTXN/s), the instruction intensity, and the maximum GIPS machine performance, according to the formula:

$$GIPS \leq min \begin{cases} Peak\ GIPS \\ Peak\ GTXN/s \times Instruction\ Intensity \end{cases}$$

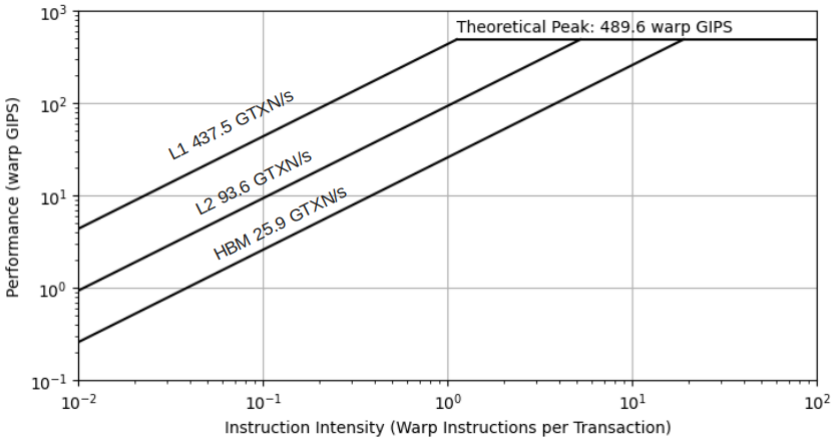


Figure 3.4: Roofline GPU Model

To estimate the efficiency of a GPU-accelerated kernel, its instruction intensity and performance need to be calculated. The tools needed, and the metrics required for employing the GPU roofline model, are described in Section 4.6.

Chapter 4

Implementation Details

4.1 Introduction

The purpose of this chapter is to describe BLonD's previous and current GPU implementations and to present optimization details and approaches. In Section 4.2.1 the previous PyCUDA implementation, its software structure, and main modules are presented. In Section 4.2.2 the new CuPy version, its main features, and advantages over the previous version are demonstrated. Performance optimizations in the CuPy version are exhibited in Section 4.3. In the following sections, optimization approaches regarding CUDA kernel execution (Section 4.4) and the thread coarsening technique (Section 4.5) are described, and a tool developed for roofline analysis using Python introduced Section 4.6).

4.2 BLonD Design

As described in Section 3.2, BLonD suite simulates the longitudinal motion of particles in synchrotrons. It is mainly written in C++ and Python programming languages, with an extensive C++ mathematical library. The three main components modeled in BLonD [2] are:

1. The synchrotron or 'ring'
2. The beam circulating in the beam pipe
3. The Radio-Frequency (RF) cavities

While real synchrotrons use bunches with trillions of particles, to reduce memory footprint BLonD simulator uses macro-particles (mentioned as particles) that represent multiple real particles. The simulations' complexity scales linearly with the number of particles used, which typically range from a few to hundreds of millions.

BLonD's frontend uses Python modules to represent the above-mentioned components and to simulate the rotation of particles, which are mainly described using the coordinates $(\Delta t_{(n)}, \Delta E_{(n)})$ for arrival time and energy at the RF cavities, with respect to an external reference clock. The user can define the number of RF cavities, ranging from one to a dozen. Important functions for particle tracking are:

kick Updates the ΔE coordinate from time step $n\ to\ n+1$ based on the Δt coordinate, and the RF voltage energy kicks received in the corresponding RF station.

drift Models the beam motion between the RF stations by updating the Δt coordinate using the updated energy of the particle.

linear_interpolation_kick Replaces the kick kernel when the variable *linear_interpolation* of the tracker object is set.

induced_voltage_sum Sums all the induced voltage contributions.

histogram Generates the global beam profile.

To provide an insight into the BLonD structure, the main classes and modules are presented:

Beam Class that contains the aforementioned beam coordinates and the beam properties.

Profile Class that contains the beam profile and related quantities, including beam spectrum and profile derivative.

RingAndRFTracker Class that enables particle coordinate tracking for a given RF station and the ring section until the next station. Contains the *track* function which applies the *kick* and the *drift* functions.

butils_wrap Module that contains Python functions, which load the respective C++ implementations from the mathematical library.

bmath Module that creates a dictionary, which holds the names of the functions defined in butils_wrap and updates the globals() dictionary to enable universal access to these functions. It also activates the MPI implementation.

mpi_config Module that enables the MPI implementation in order to use multiple CPUs or GPUs.

4.2.1 PyCUDA Version

The first GPU version of BLonD, CuBLonD, required the development of CUDA kernels. The integration of native CUDA code in Python was achieved with the

PyCUDA and Scikit-CUDA libraries. The CUDA kernels, contained in **.cu** files, are compiled to CUDA binary (**.cubin**) files using the Nvidia C compiler, and loaded using the *SourceModule* described in Section 3.4. This process enables direct calls to native CUDA code from every Python frontend module, with almost zero performance overhead.

BLonD functions utilize NumPy arrays for numerical calculations. In order to accelerate the most intensive calculations using a GPU, the arrays should be transferred accordingly to the GPU. As PyCUDA defines the *GPUArray* class, all required instances of *numpy.ndarray* should be transformed into instances of the *GPUArray*. To tackle this issue, the *CGA* class was developed, which contains both a NumPy and a PyCUDA GPU array and syncs them automatically (through validation functions when data on the CPU or the GPU are modified). Apart from this, each class that needs to be GPU accelerated (e.g. *Beam* and *Profile*), was remade as a different one (e.g. *gpu_beam*), which inherits most functions from the parent class, but it also implements new functions that call CUDA kernels, using *CGA* arrays. Important modules that were modified or developed are:

gpu_butils_wrap Contains PyCUDA functions and kernels to substitute the butils_wrap module.

gpu_physics_wrap Contains functions that call the respective CUDA kernels.

bmath Modified to store a GPU dictionary that loads functions from the gpu_butils_wrap for the globals() dictionary to be updated accordingly.

gpu_cache Stores a used *GPUArray* into a custom dictionary, and retrieves it again when a different function wishes to access a same-dimension array, in order to avoid unnecessary GPU memory allocations.

For the GPU functionalities to be enabled, the following modules are required:

__init__ Module that sets a default number of blocks and threads per block, needed for CUDA kernel calls.

gpu_activation Module that activates the GPU implementations of various classes.

To sum up, the main characteristics of the PyCUDA version are:

- 9 files for GPU implementation of basic objects
- init and activation files
- about 2600 lines of CUDA code

4.2.2 CuPy Version

The complexity of the PyCUDA implementation highlighted the need for a new approach, which would simplify the software structure while enabling more functionalities and maintaining or enhancing the GPU speedup. The CuPy library satisfies these requirements, since it has a NumPy-like interface and provides extensive low-level CUDA support.

First, the software structure in the CuPy version is greatly simplified. This is due to the variety of NumPy functions that CuPy supports, thus, eliminating the need for supplementary functions in the ***cupy_butils_wrap*** module and even CUDA kernels. More supported functions also translate to more code being accelerated, therefore enhancing simulations in general.

The most important change in the CuPy version was the modifications done in the ***bmath*** module. The goal was to enhance the user-friendliness of the BLonD suite in order for everyone to be able to understand the main modules and modify them to their needs. The identical NumPy functions that CuPy utilizes, enable simple interchange between the two libraries with just a change of the keyword "**numpy.**" to "**cupy.**" followed by the respective function. Therefore, in the ***bmath*** module, two seemingly similar dictionaries are created:

CPU_func_dict Dictionary that holds the names of the ***butils_wrap*** functions and is expanded with all needed NumPy functions and callables.

GPU_func_dict Dictionary that holds the names of the ***cupy_butils_wrap*** functions and is expanded with all needed CuPy functions and callables.

As a result of these changes, the GPU implementation of basic classes is no longer required, nor is the activation module. Every such class now supports the two following functions:

to_gpu Transfers all necessary arrays to GPU (NumPy to CuPy arrays).

to_cpu Transfers all necessary arrays to CPU (CuPy to NumPy arrays).

Should a user need to accelerate a class using the GPU, they call the *to_gpu()* function and then use the same function calls as with the NumPy library. This is achieved by calling every function through the dictionaries in the ***bmath*** module, as explained above.

Overall, the advantages of the CuPy version over the PyCUDA are stated in Table 4.1. The built-in NumPy-like *ndarray* of the CuPy library enables an easily readable code and removes the need for additional files for GPU implementations. This creates a simplified software structure, while the minimized CUDA core provides a

significant performance gain with optimized kernels. The new GPU version, with the *to_gpu* functions supported by the main modules, enhances the software's customizability, as the user needs to implement minimum changes to a CPU-targeted code, in order to use a GPU model for acceleration. Finally, the software implementation of a wide range of NumPy functions in the CuPy library enables the acceleration of many additional operations, unable to be accelerated with PyCUDA. Thus, an additional performance gain is achieved.

Category	PyCUDA	CuPy	CuPy Advantage
Array structure	Custom GPUarray	NumPy-like ndarray	Easily readable code
Required Files	Additional GPU files	No extra files	Simple software structure
CUDA lines	2600	350	Optimized Kernels
Customizability	Difficult	Easy	Facilitates future development
Accelerated Code	Limited functions	Most NumPy functions	Performance gain

Table 4.1: BLonD CuPy vs PyCUDA Version

4.3 Performance Optimizations

4.3.1 Memory Pools

Several BLonD simulations require the continuous use of the same array to conduct operations, such as FFT calculations. Should these array memory regions be reallocated for every code reuse of a same-size array, the execution time would be significantly higher. To avoid this allocation/deallocation process, both GPU versions of BLonD utilize a software-managed memory pool.

The PyCUDA version uses a custom memory pool, defined in the ***gpu_cache*** file described in Section 4.2.1. By activating this module, important memory structures are cached and returned upon request, using the Least Recently Used policy, as the memory pool hosts a fixed amount of memory. With this mechanism, an average of 23% to 35% performance gain is achieved.

CuPy uses its own memory pool by default, which mitigates the overhead of memory allocation and CPU/GPU synchronization. A custom implementation is not required for BLonD, as two different types of CuPy memory pools exist:

- The Device memory pool (GPU memory) for GPU memory allocations
- The Pinned memory pool (non-swappable CPU memory) for CPU to GPU transfers

To test the performance gain, the ***SPS*** testcase (described in Section 5.2) has been executed with and without the memory pool, and the results are presented in Figure 4.1. The testing was executed with both, device and pinned, memory pools enabled, and the results are normalized to the execution with the disabled memory pools. It is clear that CuPy memory pools offer significant performance gains. With an increasing number of bunches, more memory allocations and transfers are required, therefore memory allocation and caching of large arrays could greatly increase execution efficiency.

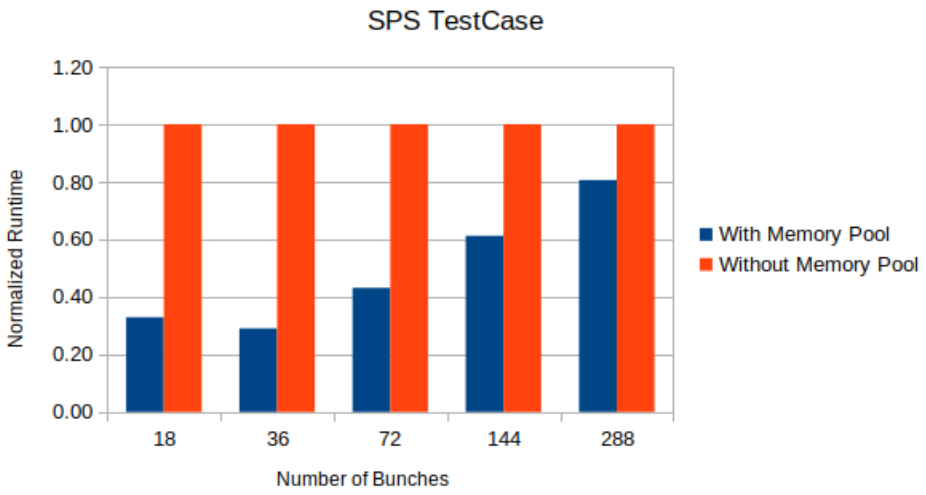


FIGURE 4.1: Memory Pool Testing

4.3.2 Shared Memory

Accessing the same data multiple times from different threads could be significantly time-consuming. Therefore, to avoid costly global memory accesses, CUDA offers the **shared memory** structure; on-chip memory that provides almost 100x lower latency than uncached global memory accesses. Shared memory is allocated per thread block, so every thread in the same block can access data in it. CUDA offers the two following types of shared memory:

Static Shared Memory It is used when the size of the shared memory array is known at compile time. To declare such an array, the *__shared__* primitive is used inside the CUDA kernel.

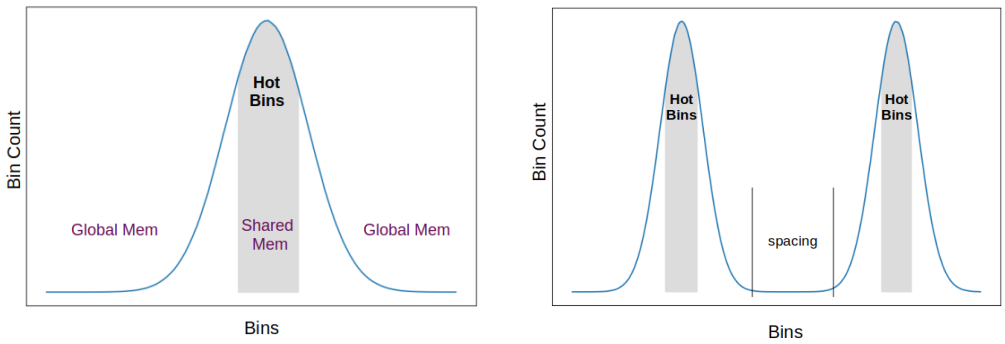
Dynamic Shared Memory It is used when the size of the shared memory array is not known at compile time. The required memory allocation size is specified with an additional kernel call parameter, and the *extern __shared__* primitive is used inside the CUDA kernel.

The shared memory structure is also used in BLonD. Specifically, the **histogram** kernel is responsible for generating the beam profile by using the **dt** coordinates as input. This kernel allocates a thread-block private beam profile in the shared memory by using atomic operations, which would be much more time-consuming in global memory. Then these beam profiles are reduced to generate the global beam profile. There are two versions of the **histogram** kernel:

simple histogram This version is used when the beam profile fits in the shared memory

hybrid histogram This version is used when only a portion of the beam profile fits in the shared memory. As the beam displays a Gaussian distribution, only the most important ("hot") bins around the center are stored in the shared memory, as shown in Figure 4.2a.

By utilizing the shared memory, a performance gain of up to 51% [2] can be achieved in extensive simulations.



(A) Single bunch distribution with "hot bins" in the shared memory

(B) Distribution with 2 bunches with multiple "hot bins" in the shared memory

FIGURE 4.2: Bunch distributions

4.3.3 Multi-Bunch Histogram

The above histogram implementations, although they accomplish to significantly improve the needed computations, they fail to efficiently handle the case of multiple bunches. The implementations, as they are, do not take into consideration the spacing between bunches and the different placement of the most significant bins.

For this reason, a more general histogram implementation is proposed, which utilizes the following input parameters:

- spacing: The "empty" space between bunches
- padding: The optional value padding for the bins
- n_bunches: The total number of bunches

The shape of a multi-bunch distribution can be seen in Figure 4.2b. A thread executing the multi-bunch histogram implementation discovers the bunch number of each bin (**bunch_no**) and the central bin value by utilizing the total space between bunches (**total_bunch_space**). Then, it calculates the boundaries and stores the respective bins to the shared memory. After thread synchronization, the same parameters are used to calculate the right index in order to copy these values from the shared to the global memory. The *multibunch_histogram* implementation can be seen in Listing 4.1, while the other implementations can be found in the BLonD GitHub repository [4]. Although the multi-bunch histogram implementation requires additional testing before being incorporated in BLonD, its features are meant to enhance calculation performance, as it stores only the right bins in the shared memory and overall displays the histogram procedure more accurately, as it takes into account the existence of multiple particle bunches. Its performance advantage can be seen in Figure 4.3, in which it is compared for increasing bunches against a version with no shared memory, using 1,500,000 particles and 1000 slices per bunch. It is clear that the multi-bunch histogram displays better performance, thus, this histogram version manages to produce more accurate results, while enhancing the performance with the use of GPU shared memory.

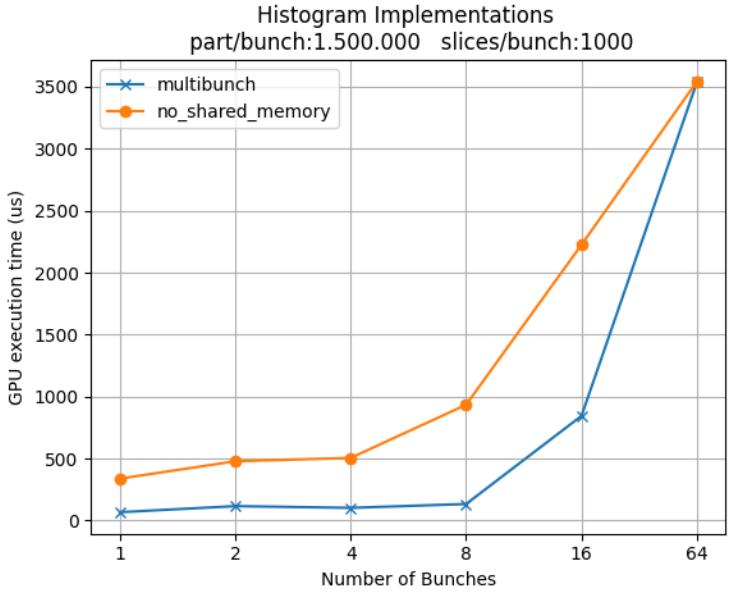


FIGURE 4.3: GPU Shared Memory for Histogram Function

```
multibunch_histogram(...){
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);
int const n_slices_per_bunch = int(n_slices / n_bunches);
int const bunch_capacity = min(int(capacity / n_bunches),
    n_slices_per_bunch);

const int total_bunch_space = n_slices_per_bunch + spacing -1;
int  bunch_no, index;
float center, low;

for (int i = tid; i < n_macroparticles; i += blockDim.x * gridDim.x
    ) {
     target_bin = floor((input[i] - cut_left) * inv_bin_width);
     bunch_no = target_bin / total_bunch_space;

     center = padding + bunch_no*total_bunch_space + (
    n_slices_per_bunch-1)/2.0;

     low = center - bunch_capacity/2.0;

     if (target_bin < 0 || target_bin >= n_slices)
         continue;
     if (target_bin > low && target_bin < low + bunch_capacity) {
         index = bunch_no*bunch_capacity + target_bin - low;
         atomicAdd(&(block_hist[index]), 1);
     }
     else
         atomicAdd(&(output[target_bin]), 1);
}
__syncthreads();
for (int i = threadIdx.x; i < capacity; i += blockDim.x) {
     index = padding + i/bunch_capacity*total_bunch_space + i%
    bunch_capacity;
     atomicAdd(&output[index], (double) block_hist[i]);
     }
}
```

Listing 4.1: multibunch histogram

4.4 Block & Grid size analysis

As discussed in Section 3.3, a CUDA kernel is executed by multiple CUDA threads, which are physically assigned to CUDA cores. These threads are grouped into thread blocks that are assigned to a SM for execution. A group of such blocks form a CUDA kernel grid, and the GPU executes the requested kernel [19]. In order for a

kernel to be executed by a specified GPU, the user should define the below parameters:

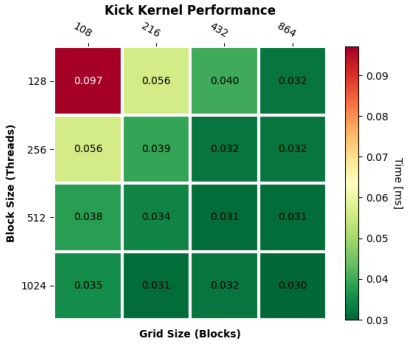
- Grid Size: Number of Blocks in the Grid
- Block Size: Number of Threads in a Block

In BLonD the various CUDA kernels are called using the CuPy *RawModule()* function. This requires the declaration of block and grid size, in order to be able to execute the kernel on the GPU device. To further analyze block and grid performance for the most important kernels, extensive experiments were performed in an NVIDIA A100 GPU model, described in Section 5.2.1. Specifically, the *hybrid/simple/multibunch histogram, linear_interpolation_kick, kick* and *drift* kernels were tested for various configurations, in order to obtain the optimum grid & block size for each. As seen in Figure 4.4, all kernels achieve minimum execution time for a block size of 1024 threads, namely, the maximum threads per block. It is important to notice, that this result does not occur for the same grid size in every kernel, therefore each kernel could be customized to use the best individual configuration.

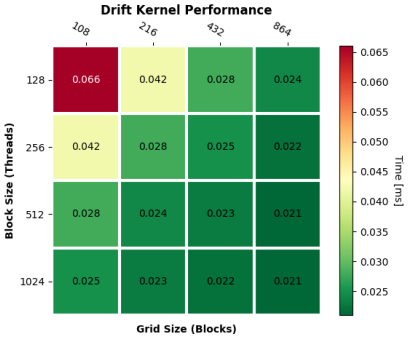
To simplify this procedure for the GPU implementation, a default number for both values is defined, considering the average performance results of the tested kernels:

- Grid Size = 2 * No. SMs
- Block Size = Max Threads per Block

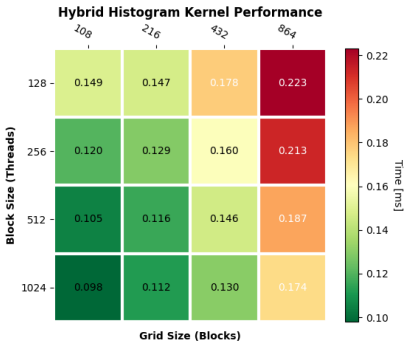
All CUDA kernels are therefore executed with a grid size equal to double the amount of available SMs and a block size of maximum possible threads in a block for the specified GPU. These values have been assigned as a compromise to the previous results. In Figure 4.5a, several block sizes are tested for an NVIDIA Tesla A100 GPU and the LHC testcase described in Section 5.2, and the result is minimum execution time in the case of 1024 (Max Threads per Block) threads. In Figure 4.5b, several grid sizes are tested for the same GPU model, and the result is minimum execution time for 432 (4*No. SMs) blocks in the grid, while the difference with 216 (2*No. SMs) blocks is negligible. Thus, the default sizes mentioned above are used, as they provide efficient performance. However, a user could define the environment variables **GPU_BLOCKS** and **GPU_THREADS** for specifying the grid and block size respectively.



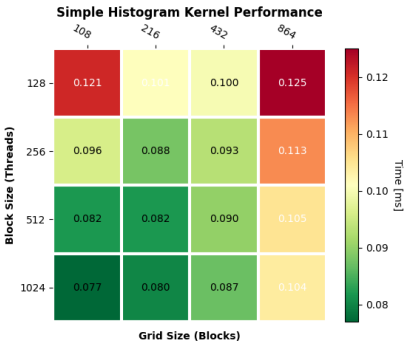
(A) Kick Kernel Analysis



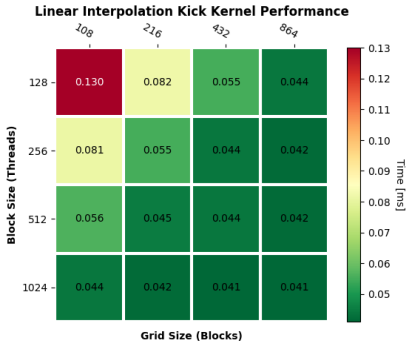
(B) Drift Kernel Analysis



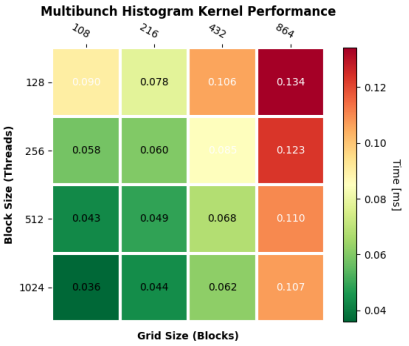
(C) Hybrid Histogram Kernel Analysis



(D) Simple Histogram Kernel Analysis

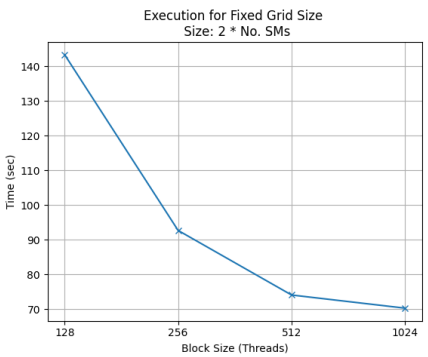


(E) Linear Interpolation Kick Kernel Analysis

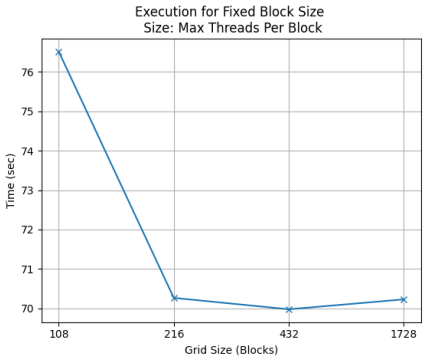


(F) Multibunch Histogram Kernel Analysis

FIGURE 4.4: Kernel launch configuration testing (block size, grid size)



(A) Testing with fixed grid size



(B) Testing with fixed block size

FIGURE 4.5: Block & Grid analysis

4.5 Thread Coarsening Analysis

An optimization method for enhancing kernel performance is thread coarsening [20] [21], which merges two or more parallel threads, increasing the work of a single thread and reducing the total number of threads. The two basic thread coarsening techniques are:

Thread-Level In this technique the total number of thread blocks (grid size) remains constant, but the total number of threads is decreased, as each block performs a task with fewer threads.

Block-Level In this technique, the number of threads per block remains constant, but the total number of thread blocks (grid size) is reduced by the coarsening factor.

For a kernel to support this technique, several modifications need to be made. These changes, shown in Table 4.2, ensure that the kernel execution produces the right result while enabling uncoalesced memory accesses. That is due to the use of the stride **S**, which ensures that adjacent threads in the same block access adjacent memory banks, thus, they do not cause conflicts. By modifying the coarse factor **C**, further reductions in threads or blocks are employed.

CUDA function	After thread-level coarse	After block-level coarse
threadIdx	$\lfloor \frac{threadIdx}{S} \rfloor \cdot S \cdot C$ $+(threadIdx) mod(S) + i \cdot S$	threadIdx
blockIdx	blockIdx	$C \cdot blockIdx + i \cdot S$
gridDim * blockDim	gridDim * blockDim	$C \cdot gridDim \cdot blockDim + i \cdot S$
blockDim	$C \cdot blockDim$	blockDim
gridDim	gridDim	$C \cdot gridDim$
blockIdx * blockDim + threadIdx	rewrite using def: $blockIdx * blockDim + threadIdx$	

TABLE 4.2: Thread Coarsening Implementation, with Stride S, Coarsening Factor C, and i as the Index of the Coarsened Thread or Thread Block Such That $0 \leq i < C$

To test if this method accomplishes better performance when used on main BLonD kernels, the *linear_interpolation_kick* and *kick* CUDA kernels were executed on a **NVIDIA A100** GPU card. Although BLonD kernels are already optimized for minimum uncoalesced and redundant memory accesses, the aforementioned kernels were tested with the thread-level coarsening method. For the reference version of the kernel, the default number of 1024 threads per block and a grid of 216 blocks are used, as discussed in Section 4.4. Testing the kernels with 1,500,000 particles, 192 bunches, and 192,000 slices, the following results were obtained:

Kernel GPU runtime (μs):		kick	linear_interp_kick
coarse, block size	-, 1024	32	42
	2, 512	32	42
	2, 256	34	45
	4, 512	36	45
	4, 256	37	46

It is clear, that thread coarsening does not enhance the performance of these specific kernels. As said, the CUDA kernels are fine-tuned in order to be maximum optimized in terms of software principles. Consequently, even the A100 GPU model (specifications described in Section 5.2.1) cannot provide additional performance, as it is not a matter of computational ability. In addition, the limited resources of an SM (e.g. registers, shared memory) result in an occupancy reduction with the combination of multiple threads of a single block in thread-level coarsening. The same behavior will occur in block-level coarsening, as each thread block undertakes an increased workload and therefore utilizes more resources.

4.6 Roofline analysis tool

To evaluate kernel efficiency, a roofline model tool in Python is constructed. This tool analyzes a proposed kernel and constructs the roofline model described in Section 3.5, by using the nvprof [22] or the Nsight Compute [23] profiling tools, which give access to various GPU metrics. For the proposed methodology [17], the following metrics are collected:

- Number of instructions executed by the kernel
- Total number of global transactions for L1
- Total number of shared transactions for L1
- Total number of L2 transactions
- Total number of HBM transactions

Using the above measurements, the instruction intensity (Warp Instructions per Transaction) and the kernel performance (Warp Giga Instructions per second) for a given cache are calculated as follows (the instructions are scaled to warp level):

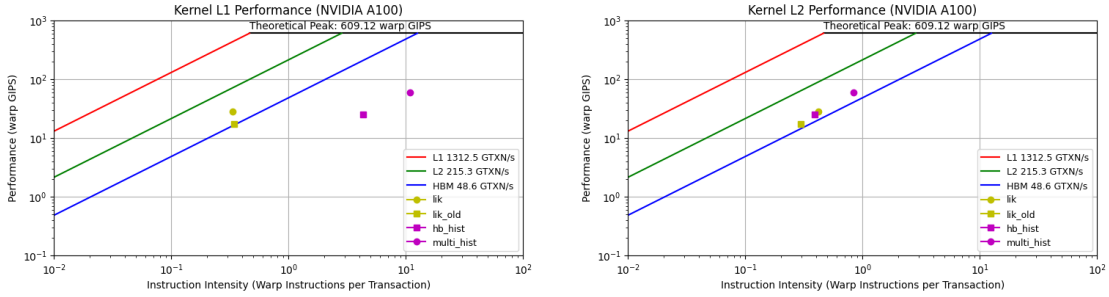
- $instruction\_intensity = \frac{No\ Instructions/32}{No\ transactions}$
- $performance = \frac{No\ Instructions/32}{10^9 * run\_time}$

For the L1 cache, the number of transactions is the total of the global and shared transactions. For the L2 cache and the HBM, the number of transactions is the total of the L2 and HBM transactions, respectively.

The roofline tool is used to evaluate main BLonD kernels, as shown in Figure 4.6, where the old versions of the *linear_interpolation_kick* and *histogram* kernels are compared against the optimized ones. The memory access pattern has been modified in the former kernel to reduce memory transfers and increase performance. For the simulation, the NVIDIA Tesla A100 GPU is utilized (described in Section 5.2.1). The ceilings for L1, L2 cache, HBM, and maximum (warp-based) IPS are sourced from the respective manual [24]. Empirical bandwidths can be calculated by using micro-benchmarks [25].

In Figure 4.6a, the L1 performance of the kernels is visualized. Firstly, the new *multibunch_histogram* kernel displays better memory accesses and computational performance than the *hybrid_histogram*, while both versions are compute-bound, as they access the fast on-chip L1 memory (compared to the red line). The *linear_interpolation_kick* kernel performs better computationally after the optimizations, while both versions are memory-bound due to main memory accesses. In Figure 4.6b, the L2 performance of the kernels is visualized. All kernels display memory-bound behavior (compared to the green line), as the L2 cache serves main memory accesses. The optimized versions display better memory performance, while the new *linear_interpolation_kick* kernel is also slightly better computationally.

Therefore, with the implemented "GPU-Roofline-Python" tool that can be found on GitHub [26], important BLonD kernels are evaluated after optimizations have been performed. The results indicate that the various modifications benefited the BLonD execution.



(A) Roofline Model for L1 cache

(B) Roofline Model for L2 cache

FIGURE 4.6: Kernel Evaluation with Roofline Model Tool

Chapter 5

Evaluation

5.1 Introduction

In this chapter, the new GPU implementation is evaluated through various benchmarks. In Section 5.2, three GPU models are compared with the CPU multicore BLonD implementation. In Section 5.3, the most intensive kernels are presented. Finally, in Section 5.4, an evaluation between the previous PyCUDA and the current CuPy version is made.

5.2 CPU-GPU speedup

To evaluate the new BLonD CuPy version, three testcases will be used. They utilize different synchrotrons of CERN's accelerator complex and display a wide range of beam dynamic features. From the smallest to the largest machine, these testcases are the following:

1. Proton Synchrotron (PS) This testcase utilizes the second synchrotron of the LHC injector chain, with a circumference of 628 m and proton acceleration power of up to 26 GeV energy and 18 bunches simultaneously.

2. Super Proton Synchrotron (SPS) The SPS accelerates the particles received from PS up to 450 GeV. It has a circumference of 7 km, is one of the largest machines worldwide, and can receive up to four batches of 72 bunches.

3. Large Hadron Collider (LHC) The LHC is the world's largest and most powerful particle collider, with a circumference of 27 km and collision energy of 13 TeV or more. It can receive up to 2808 bunches from the SPS.

For the experiments, an **AMD EPYC 7302 CPU** with the following specifications is used:

- No. of Cores : 16
- No. of Threads : 32
- Base Clock (GHz) : 3.0

The CERN infrastructure offers access to different GPUs. For the initial comparison between CPU and GPU, the **NVIDIA Tesla Turing T4** with the following specifications is used:

- Compute Capability : 7.5
- No. of CUDA cores : 2560
- No. of SMs : 40
- Memory Size (GB) : 16
- Base Clock (MHz) : 585

To reduce the overall runtime, the multicore CPU implementation is utilized with 16 cores (one thread per core). As BLonD offers double and single data precision, both options are evaluated. The configurations for every testcase are shown below:

Configuration	PS	SPS	LHC
particles/bunch	1, 2, 4, 8, 16M	1M	1,5M
bunches	21	18, 36, 72, 144, 288	12, 24, 48, 96, 192
slices/bunch	256	1408	1000
turns	10,000		
precision	Double, Single		

The results for the SPS experiment are shown in Table 5.1. The CPU execution time is increased significantly with increasing bunches, while the T4 GPU model offers a great performance gain. The highest speedup achieved is 46 times faster for the GPU execution (less than 2 minutes versus 1 hour & 27 minutes for the CPU) in the configuration of 36 bunches. The T4 GPU model computes more efficiently single precision (8.141 TFLOPS FP32 performance) than double precision numbers (254.4 GFLOPS FP64 performance), thus the single precision experiments are faster in more intensive configurations. The results for the PS and LHC experiments are shown in Tables 5.2 and 5.3 respectively. As expected, increasing particles per bunch result in greater execution times. For the PS, the highest speedup achieved is approximately 9 times faster for the GPU execution in the first configuration of a million particles, whereas for the LHC it is achieved for 12 bunches, where the GPU completes the experiment approximately 5 times faster than 16 CPU cores.

bunches	Precision	CPU Time (sec)	T4 Time (sec)	Speedup ($T_{\text{CPU}}/T_{\text{GPU}}$)
18	Double	3616.927	180.571	20.030
	Single	3217.148	83.398	38.576
36	Double	5954.564	245.765	24.229
	Single	5217.851	112.978	46.185
72	Double	5938.683	526.216	11.286
	Single	3416.51	199.896	17.091
144	Double	8408.6	1633.64	5.147
	Single	3721.924	777.365	4.788
288	Double	8457.229	4531.399	1.866
	Single	6399.739	3245.915	1.972

Table 5.1: SPS T4 CPU Speedup

part/bunch	Precision	CPU Time (sec)	T4 Time (sec)	Speedup ($T_{\text{CPU}}/T_{\text{GPU}}$)
1 M	Double	409.719	71.37	5.741
	Single	412.849	46.944	8.794
2 M	Double	618.958	122.369	5.058
	Single	604.652	75.982	7.958
4 M	Double	1168.952	221.087	5.287
	Single	986.164	136.466	7.226
8 M	Double	1919.643	422.578	4.543
	Single	1569.958	259.572	6.048
16 M	Double	3546.448	826.91	4.289
	Single	2968.963	510.627	5.814

Table 5.2: PS T4 CPU Speedup

bunches	Precision	CPU Time (sec)	T4 Time (sec)	Speedup ($T_{\text{CPU}}/T_{\text{GPU}}$)
12	Double	241.256	65.571	3.679
	Single	211.495	43.311	4.883
24	Double	410.255	118.987	3.448
	Single	312.337	87.578	3.566
48	Double	737.531	223.167	3.305
	Single	593.996	156.022	3.807
96	Double	1419.19	458.113	3.098
	Single	1178.703	301.553	3.909
192	Double	2890.337	881.349	3.279
	Single	2289.068	643.614	3.557

Table 5.3: LHC T4 CPU Speedup

The above speedup results indicate a significant performance enhancement with the use of the GPU implementation. An important note is that by increasing the intensity of the testcases, whether by experimenting with more bunches or more particles, the GPU speedup seems to be decreasing, especially in the SPS and PS testcases. This happens, as the experiments become memory intensive, as seen

in Section 5.3, and although the T4 manages to compensate for the computational needs, it cannot completely eradicate the memory (global and shared) requirements.

5.2.1 Using different GPU models

Apart from the T4 model, the CERN infrastructure offers two additional NVIDIA GPUs for conducting experiments, with their specifications shown in Table 5.4. To test the performance of these GPUs, the same experiments and configurations are used. The different GPU models are compared with the T4 model and the CPU.

NVIDIA Tesla Model	Turing T4	Volta V100S	Ampere A100
Compute Capability	7.5	7.0	8.0
No. of CUDA cores	2560	5120	6912
No. of SMs	40	80	108
Memory Size (GB)	16	32	40
Base Clock (MHz)	585	1245	765
Transistors (Million)	13,600	21,100	54,200
Die area (mm^2)	545	815	826

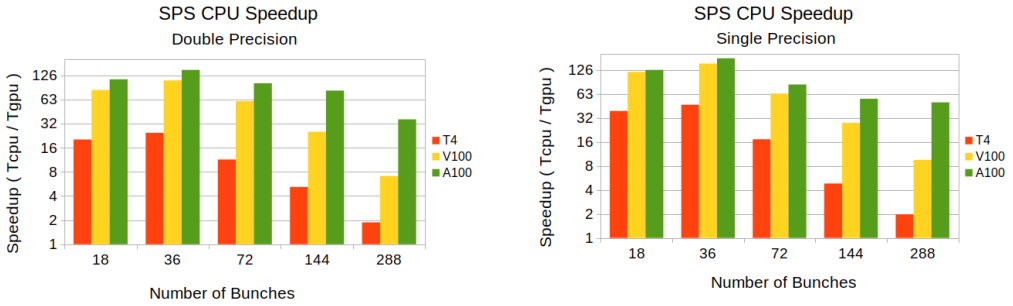
Table 5.4: CERN NVIDIA GPU Models

The results for the SPS experiment are shown in Table 5.5. In Figure 5.1, it is clear that the A100 GPU model offers a significant performance boost. Its speedup reaches up to 175 for the configuration of 36 bunches and single data precision. The combined enhancement of multiple SMs and an advanced memory capacity enables the A100 and V100 models to efficiently compute large datasets, although they still exhibit a decreasing achieved CPU speedup. The single data precision configurations achieve greater speedups, as the FP32 performance is better (16.35 TFLOPS and 19.49 TFLOPS for V100 and A100 respectively) than the FP64 (8.177 TFLOPS and 9.746 TFLOPS for V100 and A100 respectively). The total speedup is decreasing with increasing bunches, as the SPS testcase has the most intensive configurations of the three. It is executed for a max of 288 bunches with 1408 slices per bunch, thus, inducing stress to the shared memory, as described in Section 5.3.

The results for the PS experiment are shown in Table 5.6. As in the SPS testcase, Figure 5.2 shows the superiority of the V100 and A100 models versus the T4 model. The A100 achieves an increasing CPU speedup for increasing particles per bunch, which reaches up to 30 times for the configuration of 16 million particles and single data precision. The same performance advantage of the V100 and A100 models is depicted in Figure 5.3, for the LHC experiment, with the respective results shown in Table 5.7. Both achieve an increasing CPU speedup with increasing bunches, which reaches up to 12.4 for the V100 single data precision configuration of 192 bunches and up to 13.6 for the same configuration on the A100 model.

bunches	Prec	V100 (sec)	T_{T4}/T_{V100}	T_{CPU}/T_{V100}	A100 (sec)	T_{T4}/T_{A100}	T_{CPU}/T_{A100}
18	D	43.781	4.124	82.614	32.304	5.590	111.965
	S	27.062	3.082	118.881	25.594	3.258	125.699
36	D	54.852	4.481	108.557	40.642	6.047	146.513
	S	34.537	3.271	151.080	29.746	3.798	175.414
72	D	99.094	5.310	59.930	59.332	8.869	100.092
	S	53.604	3.729	63.736	41.346	4.835	82.632
144	D	336.453	4.855	24.992	103.587	15.771	81.174
	S	136.025	5.715	27.362	67.813	11.463	54.885
288	D	1201.46	3.772	7.039	237.536	19.077	35.604
	S	677.919	4.788	9.440	129.438	25.077	49.443

TABLE 5.5: SPS V100 and A100 CPU Speedup

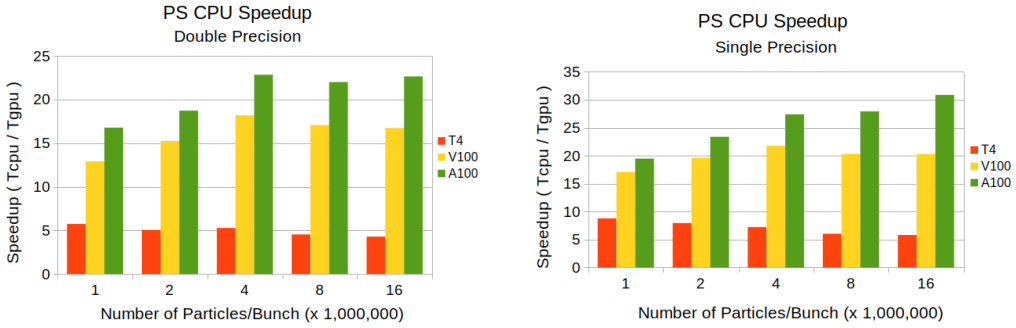


(A) Achieved Speedup for Double Precision (B) Achieved Speedup for Single Precision

FIGURE 5.1: SPS CPU Speedup (logarithmic y-axis)

part/bunch	Prec	V100 (sec)	T_{T4}/T_{V100}	T_{CPU}/T_{V100}	A100 (sec)	T_{T4}/T_{A100}	T_{CPU}/T_{A100}
1 M	D	31.732	2.249	12.912	24.423	2.922	16.776
	S	24.195	1.940	17.063	21.199	2.214	19.475
2 M	D	40.578	3.016	15.254	33.06	3.701	18.722
	S	30.873	2.461	19.585	25.859	2.938	23.383
4 M	D	64.294	3.439	18.181	51.203	4.318	22.830
	S	45.257	3.015	21.790	36.002	3.791	27.392
8 M	D	112.492	3.757	17.065	87.284	4.841	21.993
	S	77.441	3.352	20.273	56.21	4.618	27.930
16 M	D	212.017	3.900	16.727	156.638	5.279	22.641
	S	146.453	3.487	20.272	96.208	5.308	30.860

TABLE 5.6: PS V100 and A100 CPU Speedup

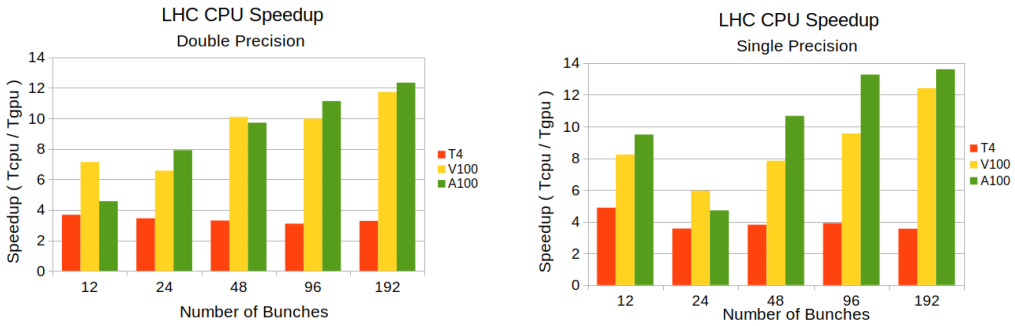


(A) Achieved Speedup for Double Precision (B) Achieved Speedup for Single Precision

FIGURE 5.2: PS CPU Speedup

bunches	Prec	V100 (sec)	T_{T4}/T_{V100}	T_{CPU}/T_{V100}	A100 (sec)	T_{T4}/T_{A100}	T_{CPU}/T_{A100}
12	D	33.814	1.939	7.135	52.768	1.243	4.572
	S	25.682	1.686	8.235	22.275	1.944	9.495
24	D	62.428	1.906	6.572	51.848	2.295	7.913
	S	52.47	1.669	5.953	66.293	1.321	4.711
48	D	73.066	3.054	10.094	75.899	2.940	9.717
	S	75.733	2.060	7.843	55.682	2.802	10.668
96	D	142.106	3.224	9.987	127.507	3.593	11.130
	S	123.219	2.447	9.566	88.781	3.397	13.277
192	D	246.389	3.577	11.731	234.249	3.762	12.339
	S	184.367	3.491	12.416	168.208	3.826	13.609

TABLE 5.7: LHC V100 and A100 CPU Speedup



(A) Achieved Speedup for Double Precision (B) Achieved Speedup for Single Precision

FIGURE 5.3: LHC CPU Speedup

5.3 CPU-GPU time breakdown

In this section, specific kernels in the same experiments are benchmarked to analyze the testcase's behavior. These include the **drift**, **linear_interpolation_kick** and **induced_voltage_sum** kernels, described in Section 4.2. The **profile** function, which utilizes the histogram kernels described in 4.3.2, is also measured. The above kernels exhibit the following computational characteristics:

- drift : Addition and multiplication core
- linear_interpolation_kick : Memory accesses
- profile : Shared memory use & atomic operations
- induced_voltage_sum : FFT operations

The configurations for the experiments are shown below:

Configuration	PS	SPS	LHC
particles/bunch	1M, 16M	1M	1,5M
bunches	21	18, 288	12, 192
slices/bunch	256	1408	1000
turns	10,000		
precision	Double		

Every machine (SPS, PS, LHC) uses a different impedance model, which is programmed to simulate its real details and imperfections. In the SPS experiment, due to the machine's characteristics, particle acceleration causes a significant amount of induced voltage to arise, as the beam interacts with the machine and the induced power causes an additional voltage. This is clear in Figure 5.4, as the *induced_voltage_sum* function requires the greatest execution time for many configurations, especially for the CPU with significantly lower computational capabilities than the GPUs. The *profile* and *linear_interpolation_kick* functions, which perform memory operations, consume most of the GPU's time for larger configurations, as greater shared memory and random accesses are requested. Finally, the *drift* computational kernel, can be easily handled by more powerful GPU models.

In the PS experiment, which uses a smaller particle configuration, the various GPU models can better accommodate the *profile* function in the shared memory, as shown in Figure 5.5. Thus, the *drift* and *linear_interpolation_kick* functions require most of the execution time. On the contrary, the LHC experiment uses a larger particle configuration, thus, the *profile* function requires significant execution time, as shown in Figure 5.6, because it stresses the GPU shared memory. Its impedance model reduces the amount of induced voltage produced, whereas the large particle input creates an increased execution time for the *drift* and *linear_interpolation_kick* kernels.

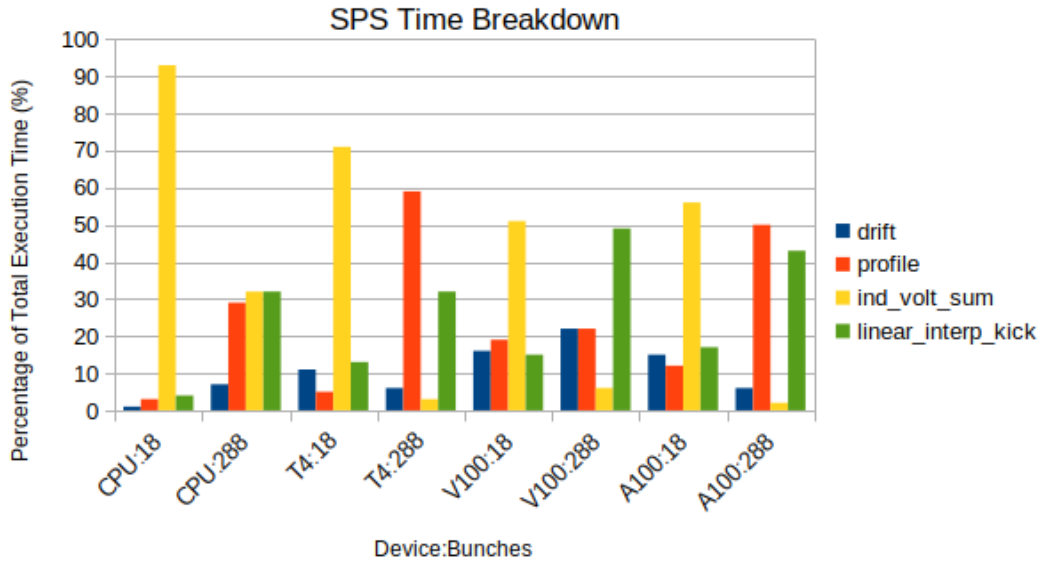


Figure 5.4: SPS Kernel Analysis

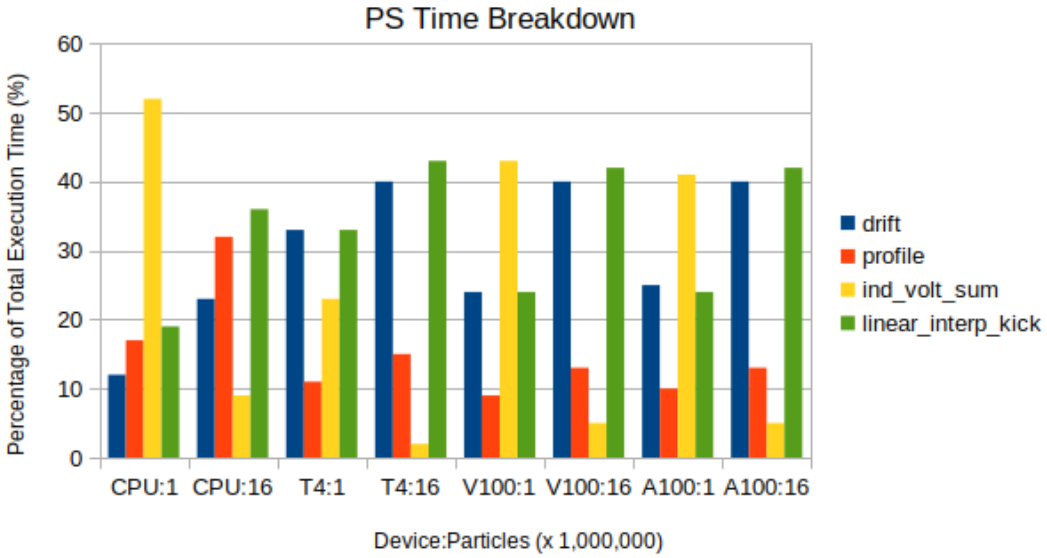


FIGURE 5.5: PS Kernel Analysis

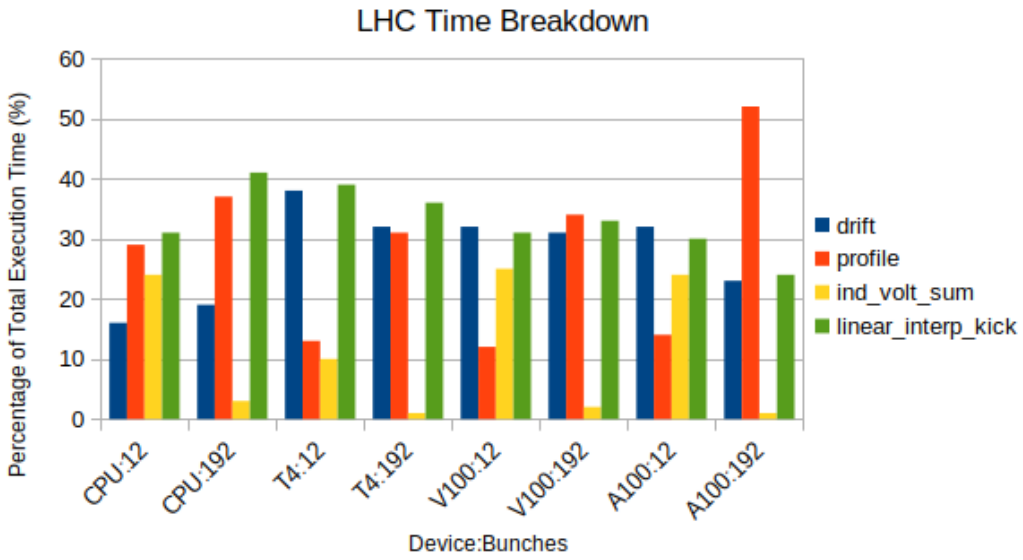


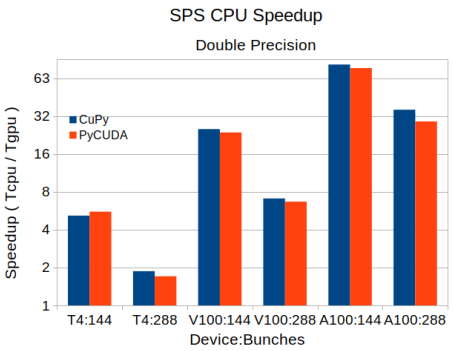
FIGURE 5.6: LHC Kernel Analysis

5.4 Comparison against previous version

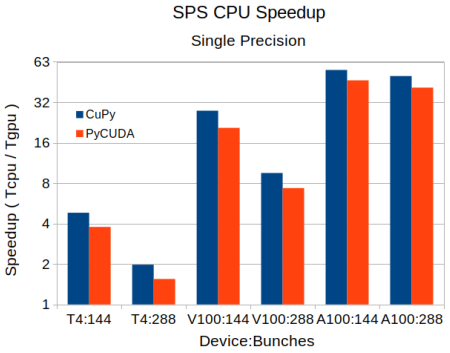
In this section, the CuPy implementation is compared against the previous PyCUDA version, by calculating the CPU speedup for every GPU model. To simplify the calculations, the two largest configurations of every testcase are used for double and single data precision. Therefore, the following experiments are benchmarked:

Configuration	SPS	PS	LHC
particles/bunch	1M	8M, 16M	1,5M
bunches	144, 288	21	96, 192
slices/bunch	1408	256	1000
turns	10,000		
precision	Double, Single		

The results for the SPS, the PS, and the LHC testcases are shown in Figures 5.7, 5.8 and 5.9 respectively. The CuPy implementation displays a significant performance advantage in all configurations. As described in Section 4.2.2, CuPy enables more code acceleration than PyCUDA, thus, it results in shorter execution times. Even in intensive experiments like the SPS and the LHC, the CuPy GPU version manages to surpass the PyCUDA version. For the SPS testcase, CuPy achieves a maximum of 34% greater speedup than PyCUDA for the V100:144:single configuration. This is increased to 30% for the PS testcase and A100:8:single configuration, while for the LHC testcase it is 26% for the V100:96:double configuration. In every testcase, the CuPy implementation offers greater performance gain, thus, minimizing even more the total execution time. Therefore, the CuPy version manages to perform better than the PyCUDA version, while using a simplified user interface and software structure, indicating that the most important goals of this software upgrade were accomplished.

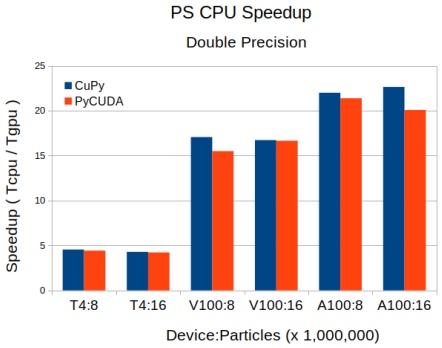


(A) Achieved Speedup for Double Precision

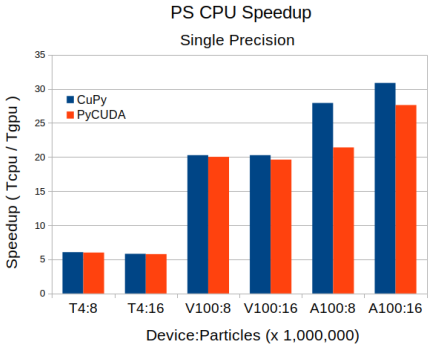


(B) Achieved Speedup for Single Precision

FIGURE 5.7: SPS CuPy-PyCUDA comparison (logarithmic y-axis)

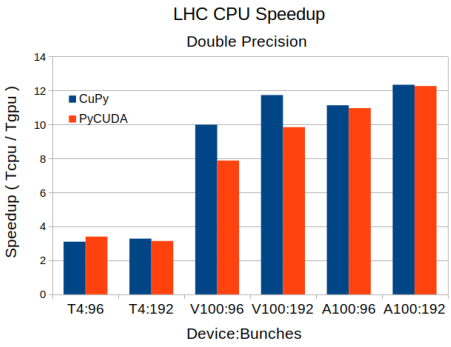


(A) Achieved Speedup for Double Precision

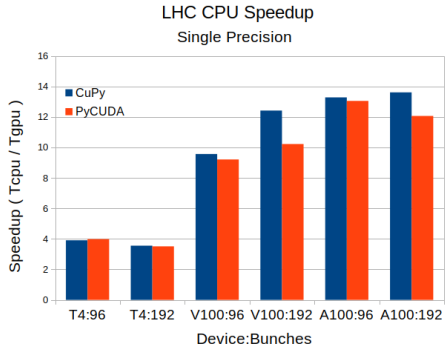


(B) Achieved Speedup for Single Precision

FIGURE 5.8: PS CuPy-PyCUDA comparison



(A) Achieved Speedup for Double Precision



(B) Achieved Speedup for Single Precision

FIGURE 5.9: LHC CuPy-PyCUDA comparison

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis presents the upgrade of the BLonD software [1]. Inspired by the need for efficiency, performance, and an enhanced user experience, the GPU version of the BLonD suite was modified to utilize the CuPy [3] Python library for GPU acceleration to substitute the PyCUDA [28] library. This enabled a simpler software structure, thus easily modifiable software according to user needs, and performance enhancement.

The upgrade process required testing of various hardware and software structures, like CuPy memory pools, shared memory, and CUDA kernel execution process, which add additional performance gain to BLonD simulations. The thread coarsening technique was also examined for possible execution benefits, and a roofline model tool in Python was designed to test intensive BLonD kernels.

Finally, the new CuPy version was evaluated on three NVIDIA GPU models and a 16-core CPU. This showcased the significant performance advantage of the CuPy implementation and the use of GPUs for software acceleration. While enabling easily modifiable code and a simpler file structure, therefore offering a better user experience, the CuPy version also exhibited better performance than the PyCUDA.

The CuPy implementation managed to efficiently alter the structure of the BLonD simulation suite. A tool that is used by many scientists at CERN and worldwide, can now efficiently utilize GPU models for performance acceleration in order to drastically reduce the required execution time. On top of this, the software structure is now simple and can be easily customized even by inexperienced programmers, as it hosts minimum Python and CUDA code, therefore it can be customized for every need, which is one of the main reasons for this thesis's work.

6.2 Future Work

Although major modifications were accomplished for the BLonD suite, there is still room for further improvements in its continuous development cycle. These future improvements include:

- Multi-GPU integration with MPI could result in reduced execution times. Apart from the current MPI CPU implementation, in which multiple CPU cores share the execution load to significantly increase application speedup, GPUs could also function in parallel. By utilizing the MPI interface, several GPU nodes could also share the total execution load and achieve even greater speedups than that of a single GPU.
- Heterogeneous GPU simulations could also be a viable solution, in the case of different GPU models available. If an NVIDIA V100 and an A100 model are available for use, the BLonD software should efficiently use these models to accelerate its simulations.
- A smart hybrid BLonD execution with the CPU and the GPU sharing the intensive work is possible to have significant performance advantages. While the CuPy version succeeds in using the GPU for the most intensive calculations, such as FFT operations, further improvements could increase execution efficiency. As seen in the result evaluation section, several CUDA kernels require many memory transfers, which increase execution time as data need to be fetched from CPU. Therefore, memory-bound operations could be performed in a CPU environment and compute-bound ones could be executed efficiently by GPUs, using the OpenMP interface.
- Testing of AMD GPU models is also the next step in BLonD's development. Over the last few years, AMD rises in the global GPU market and many systems rely on AMD models; thus testing how the BLonD suite performs with its GPU models is necessary. As BLonD's goal is to cover a wide variety of needs, it is essential to know how the software performs under different hardware systems, so using AMD GPUs would provide important insight to further evolve the BLonD suite.

Bibliography

[1] *CERN BLonD Simulation Suite Website*. URL: http://blond.web.cern.ch (visited on 03/18/2023).

[2] Konstantinos Iliakis et al. "Enabling Large Scale Simulations for Particle Accelerators". In: *IEEE Transactions on Parallel and Distributed Systems* 33.12 (2022), pp. 4425–4439. DOI: 10.1109/TPDS.2022.3192707.

[3] *CuPy website*. URL: https://cupy.dev (visited on 03/18/2023).

[4] *CERN BLonD Simulation Suite Code Repository*. URL: https://github.com/blond-admin/BLonD (visited on 03/18/2023).

[5] J. MacLachlan. "Particle tracking in E-ϕ space for synchrotron design and diagnosis". In: *Proc. Int. Conf. Appl. Accelerators Res. Ind.* 24.11 (1992).

[6] Shishlo A. et al. "The particle accelerator simulation code PyORBIT". In: *Procedia Comput. Sci.* 51 (2015), pp. 1272–1281.

[7] M. Borland. "ELEGANT: A flexible SDDS-compliant code for accelerator simulation". In: *Argonne Nat. Lab., IL, USA* (2000). DOI: 10.2172/761286.

[8] Konstantinos Iliakis et al. "BLonD++: Performance Analysis and Optimizations for Enabling Complex, Accurate and Fast Beam Dynamics Studies". In: (2018), pp. 123–130. DOI: 10.1145/3229631.3229640.

[9] Rohit Chandra et al. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[10] Message P Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. USA, 1994.

[11] Konstantinos Iliakis et al. "Scale-out Beam Longitudinal Dynamics Simulations". In: *Proceedings of the 17th ACM International Conference on Computing Frontiers*. 2020, pp. 29–38. DOI: 10.1145/3387902.3392616.

[12] *NVIDIA CUDA Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 03/18/2023).

[13] Ian Buck et al. "Brook for GPUs: Stream computing on graphics hardware". In: *ACM Trans. Graph.* 23 (Aug. 2004), pp. 777–786. DOI: 10.1145/1186562.1015800.

[14] *Open Computing Language (OpenCL)*. URL: https://www.khronos.org/opencl (visited on 03/18/2023).

[15] Tianyi David Han and Tarek S. Abdelrahman. "hiCUDA: High-Level GPGPU Programming". In: *IEEE Transactions on Parallel and Distributed Systems* 22.1 (2011), pp. 78–90. DOI: 10.1109/TPDS.2010.62.

[16] Williams S., Waterman A., and Patterson D. "Roofline:An Insightful Visual Performance Model for Multicore Architectures". In: *Commun. ACM* 52.4 (2009).

[17] Nan Ding and Samuel Williams. "An Instruction Roofline Model for GPUs". In: *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2019, pp. 7–18. DOI: 10.1109/PMBS49563.2019.00007.

[18] Charlene Yang, Thorsten Kurth, and Samuel Williams. "Hierarchical Roofline analysis for GPUs: Accelerating performance optimization for the NERSC-9 Perlmutter system". In: *Concurrency and Computation: Practice and Experience* 32.20 (2020), e5547. DOI: https://doi.org/10.1002/cpe.5547.

[19] *NVidia Developer: The CUDA Programming Model*. URL: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model (visited on 03/18/2023).

[20] Nicolai Stawinoga and Tony Field. "Predictable Thread Coarsening". In: *ACM Trans. Archit. Code Optim.* 15.2 (2018). DOI: 10.1145/3194242.

[21] Alberto Magni, Christophe Dubach, and Michael F.P. O'Boyle. "A large-scale cross-architecture evaluation of thread-coarsening". In: *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2013, pp. 1–11. DOI: 10.1145/2503210.2503268.

[22] *NVIDIA Profiler User's Guide*. URL: https://docs.nvidia.com/cuda/profiler-users-guide (visited on 03/18/2023).

[23] *NVIDIA Nsight Compute CLI*. URL: https://docs.nvidia.com/nsight-compute/NsightComputeCli (visited on 03/18/2023).

[24] *NVIDIA A100 Tensor Core GPU Architecture*. NVIDIA. 2020.

[25] *GPU Microbenchmark*. URL: https://github.com/accel-sim/gpu-app-collection (visited on 03/18/2023).

[26] *My personal Github Repository*. URL: https://github.com/Giotyp (visited on 03/18/2023).

[27] *PyCUDA website*. URL: https://documen.tician.de/pycuda (visited on 03/18/2023).

[28] Andreas Klöckner et al. "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation". In: *Parallel Computing* 38.3 (2012), pp. 157–174. ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.09.001.