View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by E-resource repository of the University of Latvia

University of Latvia
Faculty of Computing

Uldis Locans

Future Processor Hardware Architectures for the Benefit of Precise Particle Accelerator Modeling

Doctoral Thesis
Field: Computer Science
Subfield: Programming languages and systems

Scientific advisors
Dr.sc.comp., prof. Guntis Barzdins
Dr. Andreas Adelmann (PSI)
Dr. Andreas Suter (PSI)

Riga, 2017

Acknowledgements

I would like to thank my supervisors prof. Dr. Guntis Barzdins, Dr. Andreas Adelmann and Dr. Andreas Sutter for their support and guidance during this work.

I thank the co-authors Jannis Fischer, Werner Lustermann, Gunther Dissertori, Quilin Wang, Xu Xaisheng and Lukas Stingelin for their help and expertise during our interesting collaborations.

I thank my colleagues at PSI AMAS and LEM groups - it has been a real pleasure working and collaborating with all of you.

Many thanks go to my parents, family and friends for their continuous help and support trough-out this time.

This work was carried out in Paul Scherrer Institut during years 2014-2017 as a collaboration between Accelerator Modeling and Advanced Simulations (AMAS) group and Low-Energy Muon (LEM) group.

This study was financially supported by AMAS and LEM groups at PSI, as well as by Prof. Peter Arbenz group at ETH Zurich and Prof. Gunther Dissertori group at ETH Zurich.

This work was in part supported by the Latvian State research program SOPHIS

Abstract

Emerging processor architectures such as graphical processing units (GPUs) and Intel Many Integrated Cores (MICs) provide a huge performance potential for high performance computing. However developing software that uses these hardware accelerators introduces additional challenges for the developer. These challenges may include exposing increased parallelism, handling different hardware designs, and using multiple development frameworks in order to utilize devices from different vendors.

During this work the Dynamic Kernel Scheduler (DKS) was developed, to provide a software layer between the host application and hardware accelerators. DKS handles the communication between the host and the device, schedules task execution, and provides a library of built-in algorithms. Algorithms available in the DKS library will be written in CUDA, OpenCL, and OpenMP. Depending on the available hardware, the DKS can select the appropriate implementation of the algorithm.

The author used DKS to enable co-processor usage in applications such as OPAL (Object-oriented Particle Accelerator Library), musrfit and PET (Positron Emission Tomography) Image reconstruction application. These applications are developed at Paul Scherrer Institut, and ETH Zurich for particle accelerator modeling and experimental data analysis, and used by the world wide user community. The achieved results show that substantial speedups in application execution times can be achieved using co-processors compared to CPUs and with the help of DKS the process of integrating new processors in existing applications is simplified and more maintainable.

The potential of the new hardware architectures is further demonstrated by porting to CUDA application for multibunch tracking (mbtrack) developed at SOLEIL (French national synchrotron facility). This application is used at PSI for the detailed study of coupled bunch instabilities and transient beam-loading. By using the computational power of GPUs the necessary simulations can be executed on the GPU instead of a larger computing cluster that would be required otherwise.

Keywords: Hardware acceleration, GPU computing, Intel MIC, CUDA, OpenCL, OpenMP

Contents

Contents **iv**

List of Figures **viii**

List of Tables **xi**

Nomenclature **xi**

1 Introduction **1**

1.1 Research background and motivation 1

1.2 Novelty of the work 2

1.3 Thesis and research questions 4

1.4 Research methodology 5

1.5 The aim and tasks of the research 5

1.6 Main results of the thesis and approbation of the results 6

1.7 Publications of the research results 7

1.8 Outline of the thesis 9

2 Future hardware architectures **11**

2.1 GPU Accelerated Computing 11

2.1.1 GPU Architecture 11

2.1.2 CUDA 12

2.1.3 OpenCL 14

2.2 Intel Many Integrated Cores 15

2.2.1 Intel MIC architecture 15

2.2.2 OpenMP and vectorization 16

2.2.3 OpenCL . . . 17
2.3 Hardware accelerators . . . 18
2.3.1 Hardware accelerators used during this work . . . 18

3 Dynamic Kernel Scheduler **19**
3.1 Accelerator libraries . . . 19
3.1.1 ArrayFire . . . 19
3.1.2 Thrust . . . 20
3.1.3 Boost.compute . . . 22
3.1.4 VexCL . . . 23
3.1.5 ViennaCL . . . 24
3.1.6 Other libraries . . . 24
3.2 DKS concept and architecture . . . 25
3.3 DKS algorithm library . . . 28
3.3.1 DKS Base . . . 29
3.3.2 DKS and OPAL . . . 30
3.3.3 DKS and Musrfit . . . 31
3.3.4 DKS and PET Image Reconstruction . . . 31
3.4 DKS auto-tuning . . . 32
3.4.1 Auto-tuning concept . . . 32
3.4.2 DKS and auto-tuning . . . 33

4 OPAL **35**
4.1 OPAL and DKS . . . 35
4.2 FFT Poisson solver . . . 36
4.2.1 FFT Based Particle-Mesh Solver . . . 36
4.2.2 FFT-based Convolutions . . . 36
4.3 FFT Poisson solver and DKS . . . 37
4.3.1 CUDA Implementation of the Poisson Solver in DKS . . . 38
4.3.2 OpenMP Implementation of the Poisson Solver in DKS . . . 39
4.3.3 Integration of DKS in OPAL . . . 40
4.3.4 Performance Results . . . 40
4.4 Particle matter interaction . . . 41

4.4.1 The Energy Loss 42
4.4.2 Coulomb Scattering 43
4.4.3 Large Angle Rutherford Scattering 44
4.4.4 OPAL implementation 44
4.4.5 Particle drift via time integration 47
4.5 Particle matter interaction and DKS 48
4.5.1 The DKS Implementation of the Particle Matter Interaction Model 48
4.5.2 Particle matter interaction on the GPU 50
4.5.3 Particle matter interaction on the MIC 52
4.5.4 Integrating DKS in OPAL 56
4.6 Performance Results 58

5 Musrfit 61
5.1 Musrfit and DKS 61
5.2 Problem description 62
5.3 χ^2 and MLH kernels on GPU 64
5.4 User defined kernels 66
5.5 Musrfit speedups with GPUs 67

6 PET Image reconstruction and analysis 70
6.1 PET Image reconstruction basics 70
6.1.1 Image reconstruction 71
6.1.2 Image analysis 74
6.2 GPU kernels for PET image reconstruction and analysis 75
6.2.1 Forward and backward projections 76
6.2.2 Source and background calculation 78
6.3 Results 79

7 Multi-bunch tracking code - mbtrack 83
7.1 MBTRACK 84
7.1.1 Collective effects in synchrotrons 84
7.1.2 Limitations of mbtrack MPI version 84
7.1.3 Parallelization in mbtrack MPI version 85

7.2 GPU acceleration of MBTRACK 86
7.2.1 mbtrack CUDA 86
7.2.2 GPU memory management 86
7.2.3 The basic single particle transformations 87
7.2.4 Geometric ring impedance 89
7.2.5 Statistics calculations 93
7.3 Results 94
7.3.1 Verification of the mbtrack-cuda 94
7.3.2 Performance of the mbtrack-cuda 95

8 Conclusions **97**

References **99**

List of Figures

2.1 Intel MIC micro-architecture 16

3.1 Approximating pi on the device using ArrayFire. 20
3.2 A Thrust program for sorting data on the GPU 21
3.3 Example of Boost.Compute library. 22
3.4 Example of VexCL vector operations. 23
3.1 ViennaCL architecture 24
3.2 The Dynamic Kernel Scheduler. 25
3.3 The Dynamic Kernel Scheduler concept 26
3.5 Example of DKS usage for FFT 27
3.4 Architecture of the Dynamic Kernel Scheduler. The nodes shown in red are planned in the future versions of DKS, in case other vendor GPU support is required. 28
3.6 Example of setup of auto-tuning test 33

4.1 The OPAL software structure and connection to DKS 35
4.2 FFT-Poisson solver sequence diagram 38
4.3 FFT-Poisson 2D domain decomposition 39
4.4 Particle matter interaction. With the final energy $E < E_0$ and larger momenta spread due to Coulomb scattering and the large angle Rutherford scattering. 42
4.1 Struct to store one particle 45
4.2 Loop trough the particles, calculate energy loss, Coulomb scattering and large angle Rutherford scattering for each particle. 46
4.3 Calculate energy loss for a particle using Bethe-Bloch equation . . 47

4.4 Move dead particles to the end of the list and erase 48
4.5 Integration sequence diagrams 49
4.6 Particle in matter sequence diagrams, where B denotes particle bunch 50
4.5 Data structure for storing particles that are in the material on the GPU 50
4.6 CUDA kernel for MC simulation. 52
4.7 Sorting of the particles on the GPU. 53
4.8 Data structure for storing particles that are in the material on the MIC 53
4.9 Vectorization of checkHit and particle drift if it leaves material . . 54
4.10 Vectorization of energy loss calculations 55
4.11 Vectorization of coulomb scattering 56
4.12 Vectorization of Rutherford, P1, P2 and P3 are temporary arrays of size MIC_WIDTH with pregenerated random values. 57
4.13 Initializing DKS for particle matter interaction in OPAL. 57
4.14 Integrating DKS in OPAL for particle matter interaction simulations. 58

5.1 Schematic of a time differential μSR experiment 62
5.2 Flow diagram of parameter fitting with MUSRFIT using MINUIT2 and DKS 65
5.1 CUDA examples of predefined functions that can be used to create the user function. 67
5.2 Example of parsed user defined function ready for compilation. . . 67
5.3 Parameter fitting with χ^2 function running on the GPU. The time is shown for the execution of one iteration of the `minimize` command of Minuit2 69

6.1 PET imaging basic principles. 72
6.2 Cross section of the image showing LORs, with predominant direction in the x plane, and a slice of the image along this direction being processed. 77
6.1 Example code of DKS interface integrated in the host application for image reconstruction. 78

6.3 2D representation of sphere placement at the voxel position for source and background calculation. 79
6.4 Execution time for forward and backward projections. Run on Intel(R) Xeon(R) CPU E5-2690 v3 and Nvidia Tesla K40c 81
6.5 Calculation of source and background values with different sphere diameters. 82

7.1 Allocate the arrays for the data structures. 87
7.2 Kernel to apply wake potential effect to particles in a bunch. . . . 93

List of Tables

2.1 Specifications of used hardware accelerators 18

3.1 Main class functions to interact with device 30

4.1 FFT Poisson Solver results 40

4.2 OPAL degrader results 59

5.1 Parameter fitting with χ^2 function running on the GPU. The given time is for the execution of the `minimize` command of Minuit2 . . 68

6.1 Performance of image reconstruction and analysis example. 80

7.1 Comparison of mbtrack-mpi and mbtrack-cuda on 8 core machine with 1 Nvidia Tesla K40c GPU for the first test simulation. 95

7.2 Comparison of mbtrack-mpi and mbtrack-cuda on 8 core machine with 1 Nvidia Tesla K40c GPU for the second test simulation. . . 96

Chapter 1

Introduction

1.1 Research background and motivation

In recent years hardware accelerators have become increasingly popular within scientific computing. Based on the Top500 list from November 2016 [1], 86 of the top 500 supercomputers in the world are accelerator based. This includes two of the top three systems on the list: Tianhe-2 which uses Intel Xeon Phi coprocessors and Titan which uses NVIDIA K20x GPUs. GPU usage for general purpose computing has become even more important, due to the gaming industry. Almost every computer is now equipped with a GPU, but if the application is not exploiting the GPU, it is not using all the available computational power of the system. The main benefit of these new devices is the performance potential they provide. The GPUs and also Intel MICs are designed to execute massively parallel workload very efficiently when compared to CPUs, which are designed for serial task execution. Taking advantage of the these resources can substantially increase the performance of an application.

Despite the growing popularity of these devices, developing software that can take advantage of hardware accelerators can become a challenging task, especially for large existing applications. Each hardware accelerator has its own architecture and memory hierarchy which must be taken into account to gain the maximum performance out of the device. In addition to hardware differences, there are also varying methods to program these devices. NVIDIA provides the CUDA [2]

toolkit for its GPUs, both AMD and NVIDIA support the OpenCL [3] framework, and Intel allows usage of standard tools and languages to program Intel MIC processor [4], but parallelization and vectorization of the code is needed to gain the best performance. There are also OpenACC [5] and OpenMP 4.0 [6] standards that allow the targeting of hardware accelerators by expressing parallelism through compiler directives.

In this work, the Dynamic Kernel Scheduler (DKS) is presented which provides a slim software layer between the host application and the hardware accelerators. DKS separates the accelerator and framework specific code from the host application and provides a simple interface that can be implemented in the host application to offload tasks to the accelerator. DKS provides functions to handle communication and data transfer between host and device, as well as a library of functions written in CUDA, OpenCL, and OpenMP that allow the targeting of different hardware accelerators.

The ability of DKS to have implementations using different frameworks and libraries, and switch between them from the host application allows the targeting of hardware accelerators of different types and fine tuning of the code to gain the maximum performance from each device. This approach also provides more portability and software investment protection for the host application. In case a hardware architecture is no longer available, a new architecture or development framework emerges, only DKS needs to be updated.

1.2 Novelty of the work

With the increasing popularity of hardware accelerators there have been many attempts to ease the development of applications that exploit these devices. These attempts range from creation of cross-platform standards, use of compiler directives, device specific libraries and high level APIs that allow the creation of GPU/MIC code.

The OpenCL standard [3] gives the opportunity to create a cross-platform code, but while the code is guaranteed to execute on the device, as long as it supports OpenCL, the performance portability is not guaranteed. This means that different implementations of the same algorithm might be necessary to gain

the best performance from the device. In addition device specific frameworks such as CUDA and CUDA libraries for Nvidia devices, or OpenMP, vectorization and MKL libraries for Intel devices, provide advantages over OpenCL in performance and ease of development. For this reason use of only OpenCL as a device language may not be the best option in each case.

The OpenMP standard starting with version 4.0 provides the support to offload the code to the target device [7]. This would allow the creation of GPU/MIC code using OpenMP pragmas, an approach similar as used with OpenACC [5]. Unfortunately compiler support for creating device codes using OpenMP4.0 or OpenACC is limited and for this reason creating device code that would run on all the devices and platforms is not yet a feasible task.

In order to help with the code development for the GPUs, Nvidia provides a set of GPU accelerated libraries with the CUDA toolkit [8]. These libraries can be easily incorporated in the host applications to offload tasks to the device. Similar approach can be used on Intel MIC where Intel Math Kernel Library (Intel MKL) is available [9].

Several parallel vector libraries, such as Thrust [10], ArrayFire [11] and Boost-Compute [12], are available that implement parallel versions of algorithms from C++ standard template libraries. These libraries provide a collection of functions such as scan, sort, and reduce, which can be combined to implement complex algorithms.

There have also been attempts to create higher level APIs and abstractions to ease the creation of the GPU code [13, 14, 15]. These attempts focus on creating a more generic way of expressing the parallelism needed for the GPU code, that is later translated to CUDA or OpenCL kernels.

The DKS API does not aim to replace or replicate these efforts but rather provide a confined layer where all of these approaches can be used. Developer provided libraries together with hand tuned kernels provide the ability to create fast, optimized algorithms for each hardware accelerator. DKS algorithms for Nvidia graphics cards use handwritten CUDA kernels complemented by cuFFT, cuBLAS, cuRand and Thrust libraries, while the algorithms to target Intel MIC use OpenMP complemented by Intel's Math Kernel Library (MKL). OpenCL is used to target devices from other vendors, which in this work were AMD GPUs.

With DKS all of these approaches can be combined to create fast and optimized device code for each device, while keeping the changes to integrate these devices in the host application minimal.

As can be seen, there is no universal and effective solution to include different hardware accelerators in existing applications - for this reason many domain specific solutions are developed, for example TensorFlow and Torch libraries for deep learning applications, ArrayFire, Thrust and Boost.Compute for vector and matrix operations, and also DKS library, developed by the author, for physics simulations.

1.3 Thesis and research questions

During this work the following thesis have been proposed:

- With the use of hardware accelerators it is possible to significantly speed up the performance if physics simulation applications.
- It is possible to develop a universal library (DKS), that would allow to effectively integrate devices with different architectures in the existing physics simulation tools.

The research questions studied during this work is how to effectively develop the needed algorithms for these parallel architectures and how to develop the DKS library. DKS library needs to provide a common interface so that the application could communicate with different hardware accelerators un execute tasks on these devices. At the same time library needs to provide the best possible implementation of the algorithms for different architectures. This library needs to be easily extendable, so that other algorithms needed in the future or needed for different applications could be easily added. DKS library also needs to be extendable with different hardware accelerator support, so in the future new devices could be added without changing the interface used in the host application.

1.4 Research methodology

During this work both theoretical analysis and practical research methods were used.

Theoretical studies were used to analyze existing physics simulation tool and the algorithms used in these applications as well as attempts to use hardware accelerators in similar simulation application optimization. Additionally existing hardware accelerator libraries were analyzed to develop a concept for DKS architecture.

The practical implementation of the thesis is the development of DKS library which was also successfully integrated in multiple physics simulation and experimental data analysis applications. Additionally benchmark tests were performed to study the benefits that these devices can provide for simulation applications.

1.5 The aim and tasks of the research

The main objective of the thesis was to integrate new processor technologies such as GPUs and Intel MICs to speed up existing particle accelerator simulation software, such as OPAL and mbtrack, software for experimental data analysis such as musrfit, and PET image reconstruction.

To take advantage of the new processor architectures, algorithms that will be executed on these devices, need to be rewritten using the supported programming languages such as CUDA, OpenCL and OpenMP. To achieve the optimal performance from each device the algorithms also need to be constructed with the device architecture in mind.

To facilitate the integration of the device specific algorithms in large existing applications, DKS was developed which allows to separate all the device specific code from the host application. With the help of DKS it is possible to create device code using multiple different frameworks to target different devices and integrate this code in the host application with one simple interface. This simplifies the process of supporting multiple co-processors for the host application. DKS also eases the process of developing, maintaining and optimizing the device code.

The auto tuning framework for DKS is developed to allow DKS to execute the

GPU code with best possible launch parameters. This allows DKS to adapt to different hardware environments since GPU hardware resources, such as available memory, available cores and registers per core, vary between different devices and influence how the code should be launched for best performance.

1.6 Main results of the thesis and approbation of the results

The main result of the thesis is universal DKS library, developed by the author, that eases the integration of hardware accelerators in existing physics simulation applications. During this work DKS was used to add the option to offload the compute intensive parts of the simulations to GPU and Intel MIC for various physics simulation codes, developed and used at PSI and ETH Zurich. These codes include:

- OPAL - a framework for general particle accelerator simulations
- musrfit - a framework to analyze muSR data
- PET Image reconstruction and analysis software
- mbtrack - multibunch tracking code

DKS library was developed to be easily extendable un reusable in other applications. This library can also be extended to support other hardware accelerators.

DKS was integrated in OPAL (Object-oriented Parallel Accelerator Library) which is an open source C++ framework for general particle accelerator simulations. For acceleration of OPAL simulations using GPUs and Intel MIC devices CUDA and OpenMP implementations were created for FFT based Poisson solver and for Monte Carlo simulations of particle matter interactions. These algorithms are the ones of the most time consuming parts in some of the most widely run simulations using OPAL - ring cyclotron simulations and degrader simulations.

DKS was used in musrfit - a framework for muSR data analysis, to enable almost real time data analysis of experimental data. In musrfit parameter fitting using χ^2 and log-max likelihood minimization was offloaded to the GPUs to speed

up data analysis of muSR experiments. CUDA and OpenCL was used in DKS to create the necessary GPU algorithms and the performance was tested using various GPUs from different vendors. The results show that almost real-time data analysis performance can be achieved with a single GPU. This allows to perform the data analysis simultaneously with the experiments, guiding the experimenter to optimize the measuring program.

Using DKS and CUDA a PET image reconstruction and analysis software was optimized to allow compute intensive parts of the code to run on Nvidia GPUs. With the help of GPUs it was demonstrated that the execution time of PET image reconstruction can be decreased significantly, bringing the project closer to its goal of real time image reconstruction.

Multibunch tracking code mbtrack developed at SOLEIL (French national synchrotron facility) and used at PSI to study multibunch instabilities was ported to CUDA. This allows the code to execute on Nvidias GPUs. The results show a substantial improvement in computing time for single bunch simulations compared to CPU version. The main benefit of GPU version for multibunch simulations is the ability to run the large simulations required at PSI without the need of a computing cluster, which was required for the original version.

OPAL, musrfit and mbtrack applications accelerated by GPUs are used at PSI for particle matter interaction simulations (OPAL), studies of multi bunch instabilities for SLS-2 upgrade proposal (mbtrack) and for μSR experimental data analysis (musrfit). PET image reconstruction program was used as a prototype at ETH Zurich to demonstrate the potential of GPU acceleration and could be used in the future in order to achieve real time image reconstruction.

1.7 Publications of the research results

The list of publications by the author includes 2 papers submitted to journal "Computer Physics Communications" and three papers in the conference proceedings (ICAP2015, IPAC2017 and μSR2017). In all three papers the author of the dissertation was the main author of the publication and responsible for the design of the algorithms for parallel architectures in DKS, the GPU/MIC code development, verification and benchmark tests.

- Andreas Adelmann, Uldis Locans, Andreas Suter, The Dynamic Kernel Scheduler – Part 1, Computer Physics Communications, Volume 207, October 2016,
http://dx.doi.org/10.1016/j.cpc.2016.05.013, (Scopus).

- Uldis Locans, Andreas Adelmann, Andreas Suter, Jannis Fischer, Werner Lustermann, Gunther Dissertori, Qiulin Wang, Real-Time Computation of Parameter Fitting and Image Reconstruction Using Graphical Processing Units, Computer Physics Communications, Accepted for publication,
http://dx.doi.org/10.1016/j.cpc.2017.02.007, (Scopus).

- Uldis Locans, Andreas Adelmann, Andreas Suter, Dynamic Kernel Scheduler (DKS) - Accelerating the Object Oriented Particle Accelerator Library (OPAL), Proceedings of ICAP2015,
http://accelconf.web.cern.ch/AccelConf/ICAP2015/papers/proceed.pdf, (JACoW).

- Uldis Locans, Xu Haisheng, Andreas Adelmann, Lukas Stingelin, A GPU variant of mbtrack and its application in SLS-2, International Particle Accelerator Conference (IPAC2017).
https://doi.org/10.18429/JACoW-IPAC2017-THPAB051

- Uldis Locans, Andreas Suter, musrfit - Real Time Parameter Fitting Using GPUs, 14th International Conference on Muon Spin Rotation, Relaxation and Resonance (μSR2017)

Presentation of the work includes conference talks and poster presentations.

- Uldis Locans, Andreas Adelmann, Andreas Suter, Dynamic Kernel Scheduler (DKS) - a thin software layer between host application and hardware accelerators, Platform for Advanced Computing (PASC), Zurich, Switzerland, 2015, Poster presentation.

- Uldis Locans, Andreas Adelmann, Andreas Suter, Dynamic Kernel Scheduler (DKS) - Accelerating the Object Oriented Particle Accelerator Library (OPAL), International Computational Accelerator Physics Conference, Shanghai, China, 2016, Contributed talk.

- Uldis Locans, Xu Haisheng, Andreas Adelmann, Lukas Stingelin, A GPU variant of mbtrack and its application in SLS-2, International Particle Accelerator Conference (IPAC2017), 2017, Copenhagen, Denmark, poster.
- Uldis Locans, Andreas Suter, musrfit - Real Time Parameter Fitting Using GPUs, 14th International Conference on Muon Spin Rotation, Relaxation and Resonance (μSR2017), 2017, Sapporo, Japan, poster.

1.8 Outline of the thesis

Chapter 2 gives and overview over currently available hardware accelerator architectures, programming frameworks used to program co-processors as well as hardware and programming frameworks used in this work.

Chapter 3 describes in detail the concept and implementation of Dynamic Kernel Scheduler. This chapter focuses on the ideas of DKS and how it can be used to help host applications to utilize the co-processors.

In the 4th chapter a description of OPAL is given, as well as algorithms that were chosen for offload to the hardware accelerators. This chapter provides the information on how these algorithms are implemented in OPAL and how DKS was used to enable co-processor support. Results of benchmark tests using Nvidia GPUs and Intel Xeon Phi co-processors are reported.

Chapter 5 describes the challenges of enabling GPU support for musrfit. The runtime generation of CUDA or OpenCL code in DKS for execution on GPU is described as well as DKS integration in musrfit. Results provided in this chapter show the benefits of GPU usage for parameter fitting in musrfit compared to the currently used CPU version.

Chapter 6 gives an overview of the PET Image reconstruction and analysis algorithms used at ETH Zurich and the efforts to speed up these algorithms using GPUs. CUDA code used to execute the algorithms is described as well as DKS usage to integrate the code in the existing application. The potential of GPUs to speed up these algorithms is reported in the result section of this chapter.

Chapter 7 describes the efforts of creating a CUDA version of multibunch tracking software mbtrack developed at SOLEIL and used at PSI. This chapter

describes the motivation for the CUDA version as well as development of the CUDA algorithms.

The conclusions of the work are given in the chapter 8.

Chapter 2

Future hardware architectures

2.1 GPU Accelerated Computing

GPU accelerated computing refers to the use of GPUs for general purpose applications. The main benefit of the GPU is the massively parallel architecture which consists of thousands of cores designed to very efficiently execute multiple tasks simultaneously. GPU accelerated applications use the GPU to perform the compute intensive parts of the code while the remainder of the application is still running on the CPU.

2.1.1 GPU Architecture

In GPU accelerated computing, the GPU is usually connected to the host using a PCI-Express bus. The GPU is equipped with its own memory and data needs to be transferred from the host to the device, though some support for direct access of the host memory is available with certain restrictions. The GPU is optimized for high throughput calculations so high data bandwidth is required. This is achieved using wide data paths which allows to fetch multiple data elements in on cycle.

Each GPU consists of a number of streaming multiprocessors (compute units for AMD devices) containing multiple cores. For example Nvidia Tesla K40 GPU is comprised of 15 streaming multiprocessors and 2880 cores while AMD FirePro W9100 consists of 44 compute units and 2816 cores. Each core can execute a

single thread, but the cores are grouped in SIMT (Single Instruction Multiple Thread) fashion, which means that all cores in the same warp execute the same instruction. This has an effect on how conditional statements are handled in the GPU - some threads in the same group may be stalled if conditional operations need to be executed. In Nvidia devices these groups are called warps and consist of 32 threads while in AMD devices groups consist of 64 threads and are called wavefronts.

Unlike the host side GPUs do not have a sophisticated cache structure to improve memory performance, but there is a small software managed cache attached to each streaming multiprocessor and shared among the cores. In CUDA this is known as shared memory while OpenCL refers to it as local memory. While global memory access can take hundreds of clock cycles shared or local memory is a low-latency memory and runs close to register speeds since it is located on the streaming multiprocessor. This memory is shared among the cores on the same streaming multiprocessor and is used to communicate between the cores or for storage of data that is reused frequently because of its low latency. Each core also has it own private memory which is used for thread private variable storage. If threads are using too much private memory it will limit how many warps can be active on a single streaming multiprocessor or it is possible to spill the private memory to the slower global memory.

An important part of the GPU threading model is the context switching between warps. GPUs allow multiple warps to be active on a streaming multiprocessor and quickly switch the execution from one warp to another. This is done in order to hide the latency of global memory access. Global memory access can take hundreds of clock cycles - while one warp is waiting for data from global memory the GPU can switch the execution of the instructions to a different warp.

2.1.2 CUDA

CUDA is a parallel computing platform and programming model developed by NVIDIA to enable general purpose computing on Nvidia GPUs. The CUDA platform allows developers to create GPU kernels using programming languages like C, C++ and Fortran, which makes the programming of these devices more user

friendly.

A CUDA function written for execution on a GPU is called a kernel. When a CUDA kernel is launched all the threads are executing the same code. Each thread can be identified by a unique ID and all the threads are organized in groups, a group of threads in CUDA is called a block. When a CUDA kernel is launched the programmer needs to specify how many blocks and how many threads per block to launch. Each block is then allocated to a streaming multiprocessor. The threads allocated on each streaming multiprocessor need to share the registers available on this core, hence the number of registers required by a single thread and number of threads per block limits how many blocks can be allocated on single core. The threads in the block are grouped in warps of 32 threads, as described in the previous section, so it is usually beneficial to specify the number of threads per block as a multiple of 32. For latency hiding of global memory accesses it is also usually beneficial to have more than one warp per block. Since the GPU delivers the best performance when all the GPU resources are utilized it is beneficial to have enough threads active on a core to keep the GPU occupied and allow switching of the warps to hide memory access latency. This adds an additional challenge to the programmer since available resources differ between different GPU devices.

The CUDA toolkit includes a wide range of GPU-accelerated libraries containing algorithms and functions optimized for execution on the GPU. These libraries were incorporated in the current work to avoid rewriting algorithms that are already optimized for the GPU. During this work cuFFT, cuRand and cuBLAS libraries were used for FFT transforms, random number generation and BLAS functions. The CUDA toolkit also includes the Thrust library, which is a C++ template library for CUDA based on the Standard Template Library (STL). Thrust provides a large collection of data parallel primitives such as scan, sort, and reduce which can be combined to implement more complex algorithms. Algorithms created with Thrust can be used together with CUDA libraries and hand written CUDA C kernels. During this work CUDA C kernels were complimented with CUDA libraries and Thrust algorithms to create the necessary functionality.

CUDA kernels are usually stored in .cu files and compiled using a Nvidia CUDA Compiler (nvcc). NVCC splits the program in two parts - the host part

compiled by the general C, C++ or FORTRAN compiler, and the GPU part compiled by nvcc for execution on the GPU. During this work in addition to nvcc compiler to create the GPU code Nvidias runtime compilation library (NVRTC) was used as well. NVRTC allows to create GPU kernels from a source code stored in character string format. This feature was used to create the GPU kernels at runtime when full kernel functionality is unknown before the user input.

2.1.3 OpenCL

The Open Computing Language (OpenCL) is a framework for developing applications that can execute on heterogeneous platforms. The OpenCL execution model consists of a host device (typically CPU) and a target device that executes the OpenCL kernel. The target device can be any device that supports the OpenCL framework CPUs, GPUs, Intel MICs, FPGAs or others. In this chapter we focus on the use of OpenCL for the GPU programming.

OpenCL programs are written in a C-like language with extensions for parallel programming such as memory fence operations and barriers. Using OpenCL it is possible to run data and task parallel applications. Each OpenCL kernel is called a *work-item* and is identified by its own *id*. The *global work-size* determines the total number of *work-items* which can be split into *work-groups*. *Work-items* inside the *work-group* can communicate trough local memory and can be synchronized using barrier and fence operations.

An OpenCL application begins by querying the available platforms and devices. There can be any number of OpenCL platforms available on a single system allowing to run the application on a wide range of devices. The CUDA toolkit provides the Nvidia OpenCL platform allowing applications to target Nvidia GPUs, the AMD OpenCL SDK gives access to the AMD platform for targeting AMD CPUs and GPUs, and Intel OpenCL Runtime provides Intel OpenCL platform to target Intel's devices. There are other platforms available to target different devices, but the three previously mentioned were used in this work to target CPUs, GPUs and Intel MICs.

Once the devices, which will be used by the application, are identified a context is created with one or multiple of these devices. Contexts are used by the OpenCL

runtime for managing objects such as command-queues, memory, program and kernel objects and for executing kernels on one or more devices specified in the context. Many operation are performed with the respect to a given context, but there are also operation that are device specific. For example program compilation and kernel execution are performed for a specific device. A command queue is associated with each device in the context and all the work executed on the device is scheduled trough this command queue. Many OpenCL programs follow the same pattern. After selecting the platform and devices to use create a context, allocate memory, create device-specific command queues, perform data transfer, execute the kernels and read the data back to host memory.

2.2 Intel Many Integrated Cores

Intel MIC is a new technology developed by Intel specifically for HPC. The first Intel MIC architecture co-processor is the Intel Xeon Phi Knights corner. These co-processors combine many Intel PC cores on a single chip and allow developers to develop application for these accelerators using standard C, C++ or FORTRAN source code. Intel MIC co-processors run a Linux operating system and can be used to run separate applications (native mode) or as a part of heterogeneous systems (offload mode), where CPU offloads part of the task to execute on this device, similar as in the case when GPUs are used in HPC applications.

2.2.1 Intel MIC architecture

The Intel Xeon Phi co-processor runs its own Linux operating system and it is connected to the host CPU trough the PCI Express (PCIe) bus. The use of the Linux operating system allows a virtualized TCP/IP connection to be implemented trough the PCIe bus, which allows to access the co-processor as a network node. This allows users to connect to co-processor trough secure shell and directly run individual jobs on it (native mode). Intel MIC also supports heterogeneous applications where part of the code executes on the host while part executes on the co-processor [16].

Each core of the Intel MIC is equipped with a private L2 cache that is kept fully

coherent by a global-distributed tag directory. The memory controllers provide an interface to the global memory available on the co-processor while the PCIe interconnect provides access to the memory on the host. All these components are connected via the ring interconnect as shown in figure 3.1.

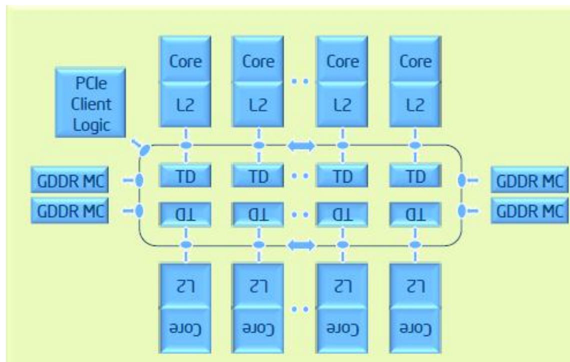

Figure 2.1: Intel MIC micro-architecture [16].

Intel MIC cores are designed to handle high throughput parallel workloads while being power efficient. Each core can support up to 4 hardware threads and contains a vector processing unit (VPU). The VPU features a 512-bit SIMD instruction set and can execute 16 single-precision (SP) or 8 double-precision (DP) operations per cycle. There is also support for Fused Multiply-Add (FMA) instructions which allows to double the SP and DP instructions executed per cycle [16].

During this work the first Intel Xeon Phi processor *Knights Corner* was used. This processor was used in a co-processor mode where host offloads the compute intensive parts of the calculation to the Xeon Phi.

2.2.2 OpenMP and vectorization

One of the main benefits of the Xeon Phi co-processors is the ability to use standard programming tools and languages to program these devices. It is possible to add OpenMP-like pragmas to C/C++ or Fortran code to mark regions of the code that should be offloaded and executed on the Xeon Phi. The offload pragmas are detected by compiler and the code is compiled for execution on the device, the code to transfer the data between the devices is also generated automatically

although the programmer can influence the data transfer with additional clauses in the pragmas.

To efficiently use all the available cores on the Intel MIC the code region that is offloaded to the co-processor needs to be parallelized. During this work OpenMP is used to parallelized the code for Intel MIC. To fully utilize the computing resources on each core it is essential to take advantage of the VPU. The vectorization of the code is done by the compiler, but additional OpenMP pragmas are available to assist the compiler in vectorizing the code. In most cases some restructuring of the code is necessary to allow the compiler to vectorize the code and optimize the benefits of the VPU. These optimization usually focuses on creating a favorable memory access patterns to improve the performance and removing memory dependencies to enable vectorization.

2.2.3 OpenCL

Intel Xeon Phi processors can also be programmed using OpenCL standard, but while OpenCL is a portable programming model, than can be run across multiple platforms, it does not guarantee performance portability. Since the GPUs and Intel MIC have different hardware architectures the OpenCL code designed for GPUs is not guaranteed to perform well on the Intel MIC.

One of the biggest differences of Intel MIC and GPU is the absence of programmable shared local memory. While GPUs relay on the programmable shared memory to optimize memory accesses, Intel MICs have a fully coherent cache hierarchy, similar to the one found on CPUs, that automatically speeds up memory accesses. Another difference is the number of threads launched by the device, while GPUs rely on hardware scheduling of many tiny threads executed in a SIMT fashion, Intel MIC uses the operating system to schedule medium sized threads.

Since OpenCL is not able to provide performance portability, multiple programming models are used throughout this work. To program Intel MIC devices OpenMP with offload pragmas was chosen, while OpenCL was used mainly to target GPUs from vendors other than Nvidia.

2.3 Hardware accelerators

2.3.1 Hardware accelerators used during this work

During this work Nvidia and AMD GPUs were used as well as Intel MIC co-processor. All the co-processors used during this work are summarized in table 2.1. Some tasks during this work were designed specifically for Nvidia GPUs while other focused on all the available devices.

To compare the achieved results on the devices bench-marking results are presented trough-out the work comparing the performance of the various accelerators and to the performance of the CPU. The CPU performance on a single core was usually take as a baseline to compare the potential speedup. In applications where multicore implementations are available, the performance of applications was also measured using all the available CPU cores, for a more complete comparison of the advantages of hardware accelerators.

Table 2.1: Specifications of used hardware accelerators

Name	Memory	Memory bandwidth	Processing Power DP
Nvidia Tesla K20c	5GB	208GB/sec	1.17 Tflops
Nvidia Tesla K40c	12GB	288GB/sec	1.43 Tflops
AMD Radeon R9 390x	8GB	384GB/sec	0.739 Tflops
Intel Xeon Phi 5110p	8GB	320GB/sec	1.01 Tflops

Chapter 3

Dynamic Kernel Scheduler

3.1 Accelerator libraries

Many frameworks and libraries have been proposed to tackle the issue of integrating hardware accelerators in large scale applications and ease the code development for these new devices.

3.1.1 ArrayFire

ArrayFire is an open source matrix library for rapid development of general purpose GPU (GPGPU) computing and parallel computing applications. ArrayFire provides fine tuned functions for linear algebra, convolutions, reductions, and FFT's as well as signal processing, image processing, statistics, and graphics libraries [17]. ArrayFire is hardware neutral and contains CUDA, OpenCL and C back-ends to support Nvidia GPUs, AMD GPUs/APUs and Intel Xeon Phi co-processors [11].

ArrayFire functions operate on matrix objects (arrays) which can contain floating point values (single or double precisions), real or complex values, and boolean data. The arrays used in computations are multidimensional and can be manipulated with arithmetic operations and ArrayFire functions as shown in the example 3.1. A parallel for loop GFOR is also provided by ArrayFire, which allows to execute many instances of independent routines in a data-parallel fashion [17].

ArrayFire uses Just-In-Time compilation (JIT) to generate the device code

```
1 //sample 40 million points on the GPU
2 array x = randu(20e6), y = randu(20e6);
3 array dist = sqrt(x * x + y * y);
4
5 //the ratio of how many fell in the unit circle
6 float num_inside = sum<float>(dist < 1);
7 float pi = 4.0 * num_inside / 20e6;
```

Code example 3.1: Approximating pi on the device using ArrayFire.

on-the-fly and optimize memory transfers to maximize throughput. At runtime ArrayFire aggregates arithmetic operations and function calls for a single variable instance in a Abstract Syntax Tree (AST) data structure. A single kernel is created on the fly to evaluate all the functions in a AST [17].

In many cases using just ArrayFire is not enough to provide all the necessary functionality that an application needs. ArrayFire can be added to an existing CUDA or OpenCL application, or custom CUDA or OpenCL kernels can be created to supplement ArrayFire's functionality. For CUDA devices ArrayFire manages its own memory and operates in its own stream, while for OpenCL devices ArrayFire creates its own context and command queue. ArrayFire also creates custom IDs for the used devices. ArrayFire API provides a set of functions to interface ArrayFire with custom code. These functions mostly focus on selecting the correct stream/context and synchronizing the execution of the code, as well as gaining access to memory objects on the device [11].

3.1.2 Thrust

Thrust is a CUDA parallel template library based on the C++ Standard Template Library (STL). Thrust uses the high level interface to implement high performance parallel applications and is fully interoperable with the rest of CUDA software ecosystem [10].

With Thrust the developer describes the algorithm using a collection of highly optimized functions, that map efficiently on the targeted Nvidia device architecture. Thrust provides an abstract interface to fundamental algorithms such as scan, sort and reduction. With the help of C++ templates Thrust makes these algorithms generic and allows them to be executed with custom user defined data

types and operators [18].

```
1  //generate 16M random numbers on the host
2  thrust::host_vector<int> h_vec(1 << 24);
3  thrust::generate(h_vec.begin(), h_vec.end, rand);
4
5  //transfer data to device
6  thrust::device_vector<int> d_vec = h_vec;
7
8  //sort data on the device
9  thrust::sort(d_vec.begin(), d_vec.end());
10
11 //transfer data back to the host
12 thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
```

Code example 3.2: A Thrust program for sorting data on the GPU

Thrust program operates on vector containers, which can be stored on the host side or on the device side. An example of thrust application to sort an array of data is shown in 3.2. All the details of the device launch parameters such as grid and block size, the details of memory management and CUDA kernels to launch are handled by Thrust and completely hidden from the developer [18].

Thrust functions are derived from four fundamental parallel algorithms - for each, reduce, scan, and sort. These algorithms are generic in both the data type that needs to be processed and the operations to applied to the data [18].

Important feature of any GPU library is interoperability with external GPU code, since in most cases no library provides a complete list of functions needed by the applications. Thrust is implemented with CUDA C/C++ and is interoperable with user defined CUDA kernels and external CUDA libraries. Thrust vectors can be used by CUDA kernels and external libraries by extracting a raw pointer to the data residing in a Thrust vector, and raw pointers to GPU memory can be wrapped in Thrust containers to be used by the Thrust algorithms [18].

The main disadvantage of Thrust library is that it is written only in CUDA and can only target Nvidia devices. Since Thrust is included in the CUDA toolkit the best use of this library is to complement CUDA applications to avoid rewriting algorithms that are already optimized in Thrust. During this work Thrusts sort, reduce and scan algorithms were used to complement CUDA kernels and speed up the code development.

3.1.3 Boost.compute

Boost Compute is a C++ library for GPU computing platforms based on OpenCL. Boost Compute library is made up of multiple layers. The top layer provides a C++ wrapper over the OpenCL API, to help manage the OpenCL objects such as devices, kernels and command queues. On top of the core layer Boost Compute implements the C++ Standard Template Library (STL) providing common containers (such as vector and array) and algorithms (such as transform, reduce, sort) [12]. An example of using Boost Compute library is shown in code example 3.3. The example includes the use of both layers of the library - the core layer for device management and the algorithm layer to perform transformation of the input data.

```
1  // get default device and setup context
2  compute::device device = compute::system::default_device();
3  compute::context context(device);
4  compute::command_queue queue(context, device);
5
6  // generate random data on the host
7  std::vector<float> h_vector(10000);
8  std::generate(h_vector.begin(), h_vector.end(), rand);
9
10 // create a vector on the device
11 compute::vector<float> d_vector(h_vector.size(), context);
12
13 // transfer data from the host to the device
14 compute::copy(h_vector.begin(), h_vector.end(), d_vector.begin(), queue);
15
16 // calculate the square-root of each element in-place
17 compute::transform(d_vector.begin(),
18                    d_vector.end(),
19                    d_vector.begin(),
20                    compute::sqrt<float>(),
21                    queue
22                    );
23
24 // copy values back to the host
25 compute::copy(d_vector.begin(), d_vector.end(), h_vector.begin(), queue);
```

Code example 3.3: Example of Boost.Compute library.

Similarly to the Thrust library Boost Compute allows the use of custom user defined data types and operators to be used in the algorithms provided by the library. In addition Boost Compute is designed to be able to operate together with the OpenCL API which allows Boost Compute to be combined with custom OpenCL kernels and external OpenCL libraries [12].

The main advantage of Boost Compute over Thrust is the use of OpenCL as the backend, which allows to target different device types. While the disadvantages are the need for a third party library (Thrust is supplied with the CUDA toolkit) and the inability to inter operate applications written using CUDA.

3.1.4 VexCL

VexCL is a vector expression template library for OpenCL/CUDA. The aim of this library is to reduce the amount of boilerplate code needed to develop algorithms for GPUs. The library provides intuitive notation for vector arithmetic, reductions, sparse matrix vector products and other algorithms [19].

VexCL provides OpenCL, CUDA and Boost.Compute backends to generate GPU code to target devices of different types. VexCL uses intuitive notations for vector operations and functions. In order to be used in the same expression all the vectors must have the same size and must be allocated on the same device. If these criteria are met VexCL generates GPU kernel code from the vector expression which is executed at runtime. VexCL expressions can combine device vectors and scalars with arithmetic, logic and bitwise operators, and also with built-in OpenCL/CUDA functions. An example of launching VexCL vector operations on the GPU is shown in code example 3.4. In addition VexCL contains parallel implementations of some the most widely used vector algorithms such as reduce, scan, sort and others [19].

```
1  const size_t n = 1024 * 1024;
2
3  // Get compute devices supporting double precision:
4  vex::Context ctx(vex::Filter::DoublePrecision);
5
6  // Prepare input data, transfer it to the device(s):
7  std::vector<double> c(n);
8  vex::vector<double> A(ctx, a), B(ctx, b), C(ctx, n);
9
10 // Launch compute kernel:
11 C = A + B;
12
13 // Get result back to host:
14 vex::copy(C, c);
```

Code example 3.4: Example of VexCL vector operations.

In cases where vector operations and VexCL algorithms are not enough cus-

tom kernels can be defined in VexCL to perform a user defined functions. The custom kernels must be defined using CUDA or OpenCL code depending on which backend is selected for the use of VexCL [20].

3.1.5 ViennaCL

The Vienna Computing Library (ViennaCL) is a linear algebra library for parallel computations on many-core architectures such as GPUs, Intel MICs, and many core CPUs. The library is written in C++ and uses CUDA, OpenCL, and OpenMP to target the devices [21].

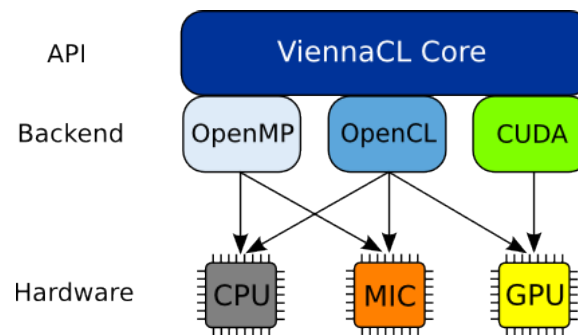

Figure 3.1: ViennaCL architecture [21].

The algorithms in the ViennaCL library primarily focus on common linear algebra operations (BLAS levels 1, 2 and 3) and also provides iterative solvers for large systems of equations [21]. ViennaCL API follows existing programming and interface conventions established with uBLAS, which is part of the Boost library. This allows ViennaCL to be easily integrated in existing applications that use BLAS libraries with minimal code changes [22].

3.1.6 Other libraries

Many drop in libraries are available for the GPU and Intel Xeon Phi, that provide functions that are highly optimized for these devices and can be used to complement the custom kernels and ease the development process. The Nvidia CUDA toolkit includes several libraries that provide highly optimized algorithms and functions that can be used to offload tasks to the GPU [8]. Libraries such as

cuFFT, cuBLAS and cuRand are used throughout this work. Intel Math Kernel Library (Intel MKL) provides optimized functions for Intel Xeon Phi processors [9]. The Intel MKL library was used in this work for FFT and random number generation for the Intel MIC code. AMD Compute Libraries (ACL) provide a similar set of libraries as the CUDA toolkit including clFFT, clBLAS and clRNG [23].

3.2 DKS concept and architecture

Developing applications that can take advantage of different types of hardware accelerators will usually require use of multiple development frameworks and may benefit from multiple of the libraries described in the section 3.1. The Dynamic Kernel Scheduler was developed to unify the code development for hardware accelerators using the most appropriate tools, as shown in figure 3.2. This allows to ease the code development and tuning for specific devices, and helps with the integration of new devices in existing applications [24].

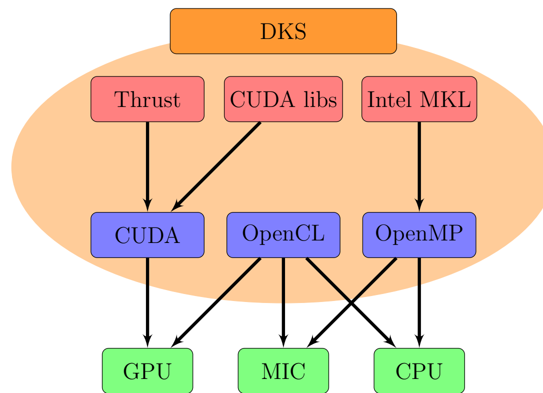

Figure 3.2: The Dynamic Kernel Scheduler.

The Dynamic Kernel Scheduler (DKS) is a slim software layer between the host application and the hardware accelerator frameworks, as depicted in Figure 3.3. The aim of DKS is to allow the creation of fast fine tuned kernels using device specific frameworks such as CUDA, OpenCL, OpenACC and OpenMP and accelerator libraries such as Thrust, Nvidia CUDA libraries, Intel MKL or others. On

top of that, DKS allows the easy use of these kernels in host applications without providing any device or framework specific details. This approach facilitates the integration of different types of devices in the existing applications with minimal code changes and makes the device and the host code a lot more manageable.

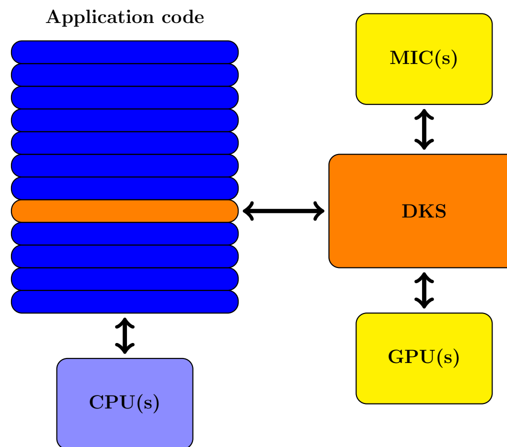

Figure 3.3: The Dynamic Kernel Scheduler concept

The architecture of DKS can be split in three main parts:

1. The first part provides communication functions that handle memory allocation and data transfer to and from the device. All the memory management is left to the user. This way the data transfers and memory allocation can be scheduled only when necessary. DKS also supports GPU streams such that asynchronous data transfer and kernel execution can be implemented when possible.

2. The second part of DKS consists of a function library, which contains algorithms written in CUDA, OpenCL, and OpenMP to target different devices. DKS can switch between implementations based on the hardware that is available. Writing functions using multiple frameworks results in extra work, but provides the opportunity to fine tune kernels for each device architecture for maximum performance. It also allows the targeting of systems containing different types of devices. The different implementations of the code are always separated so the code is still easy to manage. Additionally if a host

application is targeted at a specific system, implementations that are not needed can be omitted.

3. The third part of DKS is the auto-tuning functionality. The aim of auto-tuning is to select the appropriate implementation of the algorithm and change the launch parameters according to the devices that are available on the system in order to gain the maximum performance. The auto-tuning functionality relies on knowledge of device architecture and benchmark tests that can be run on the system before running the application.

```
1  //allocate memory on device and write data
2  void *mem_ptr;
3  mem_ptr = dks.allocateMemory<Complex_t>(DATA_SIZE, NULL);
4  dks.writeData<Complex_t>(mem_ptr, DATA_ARRAY, DATA_SIZE);
5
6  //execute FFT or IFFT
7  if (direction == 1)
8    dks.callFFT(mem_ptr, DIMENSIONS, DIM_SIZE);
9  else
10   dks.callIFFT(mem_ptr, DIMENSIONS, DIM_SIZE);
11
12 //read data and free memory
13 dks.readData<Complex_t>(mem_ptr, DATA_ARRAY, DATA_SIZE);
14 dks.freeMemory<Complex_t>(mem_ptr, DATA_SIZE);
```

Code example 3.5: Example of DKS usage for FFT

Code example 3.5 shows DKS usage inside a host application to perform a fast Fourier transform. The host application has full control over the memory allocation and data transfer to the device, but there are no device specific details in the host code. DKS evaluates the calls made by host application and chooses the appropriate device to use, and algorithm implementation, to run the code on the selected accelerator.

The Dynamic Kernel Scheduler is split into separate modules. Each module contains function implementations using different frameworks. The base class for each module contains functions which handles the device management, memory management, and data transfer, this base class can be extended to cover all the necessary algorithm specific functions. The base class of DKS receives all the

calls from the host application and decides which device specific implementation should be used to run the code on the device. Figure 3.4 shows the architecture of the latest version of DKS, for each module base class can be easily extended to include other algorithms and the base class of DKS can be extended to include other modules to handle different development frameworks.

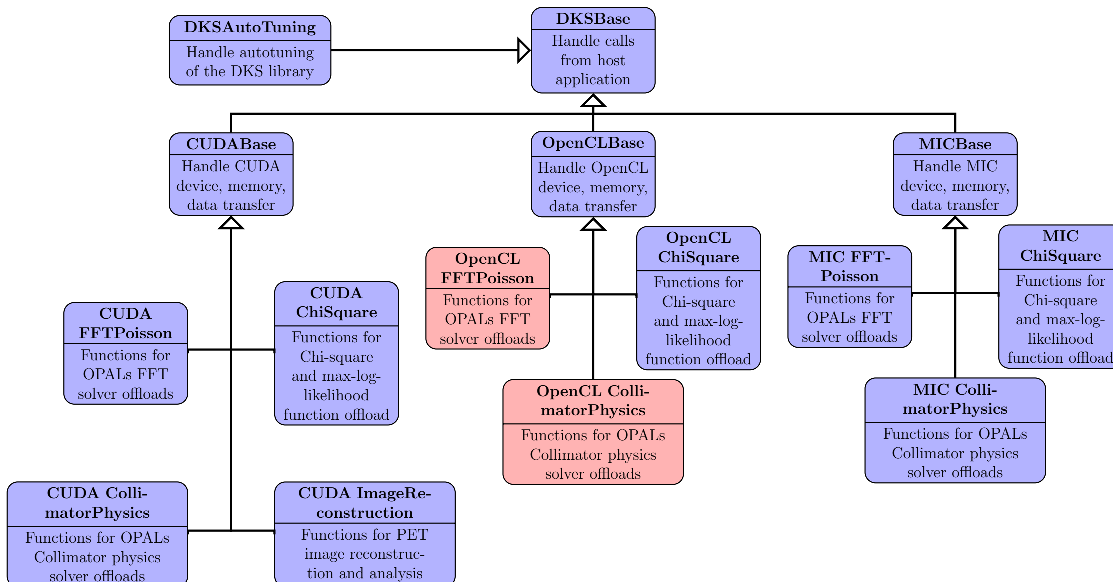

Figure 3.4: Architecture of the Dynamic Kernel Scheduler. The nodes shown in red are planned in the future versions of DKS, in case other vendor GPU support is required.

3.3 DKS algorithm library

The DKS algorithm library contains algorithm implementations written using CUDA, OpenCL and OpenMP to allow host applications to offload the compute intensive calculations to the co-processor. Before the algorithm is implemented in DKS the host application is profiled to find the compute intensive parts of the CPU code and if the algorithm can benefit from parallelization on the co-processor a suitable device code is created in DKS. The algorithms in DKS are specifically tailored to match the implementations in the host applications so the offloaded code could recreate the results of the original code.

The hardware accelerators that will be used are dependent on the host application. Some applications are targeting a wider range of hardware and therefore algorithms are implemented using various frameworks, while some applications target only Nvidia GPUs and only CUDA implementations of the offloaded code are necessary.

The main benefit of the DKS algorithm library is the separation of the device code from the host code. This allows for easier development and maintenance of the device and host code. During this work algorithms form OPAL, musrfit and Image Reconstruction applications were implemented in the DKS and integrated back to host applications allowing these applications to offload the computations to co-processors with limited changes to the original code.

The specific algorithms implemented in the DKS algorithm library are shown in figure 3.4 and are described in the following sections. These sections contain detailed explanations of the algorithms needed by the host applications, how these algorithms were adapted for the massively parallel devices and the integration of DKS back in the host application. In the future the DKS library can be extended to include other algorithms for different applications and it can also be extended to support other hardware accelerators not used during this work (for example FPGAs).

3.3.1 DKS Base

The base part of DKS library provides the basic functions needed to communicate with the hardware accelerators. A list of available base class functions is shown in table 3.1, this list shows the most used functions to communicate with the devices. The base class is implemented using CUDA, OpenCL and OpenMP allowing DKS to target various hardware accelerators. Using the base class host application can query the available devices and get the device information as well as select the device that should be used. Base class also allows host application can perform memory management and data transfers, including gathering and scattering the data when multiple CPU cores share the same GPU memory region. For GPU devices the base class also handles the creation of streams and synchronization functions, that allow overlapping memory transfers and GPU kernel execution.

Table 3.1: Main class functions to interact with device

Function	**Description**
setDevice	Set device used by DKS
getDevices	Get information about available devices
getDeviceCount	Get number of available devices
pushData	Allocate memory and transfer data to device
pullData	Transfer data from device and free memory
allocateMemory	Allocate memory on the device
registerHostMemory	Page lock allocated host memory
unregisterHostMemory	Unregister page locked memory
writeDataAsync	Write data to the device
readDataAsync	Read data from the device
gather3DDataAsync	Gather 3D data from multiple mpi processes to one memory region
scatter3DDataAsync	Scatter 3D data to multiple MPI processes from one device memory region
freeMemory	Free memory allocated on device
sendPointer	Send pointer to device memory from one MPI process to another
receivePointer	Receive pointer to device memory from another MPI process
closeHandle	Close handle to device memory created by receive-Pointer
createStream	Create stream for asynchronous kernel execution and data transfer on GPUs
syncDevice	Wait till all tasks running on device are completed

3.3.2 DKS and OPAL

DKS OPAL module provides CUDA and OpenMP implementations for algorithms that allow OPAL to offload FFT Poisson solver and particle matter interaction simulations. Using DKS Base module OPAL can schedule memory management and data transfers to the devices and using the OPAL module schedule the kernel execution on the device. For the FFT Poisson solver gather and scatter data transfers are used with the GPU to allow multiple CPU cores to share one GPU.

For the FFT Poisson solver DKS contains a FFT module, that allows to perform the Fast Fourier transformations on the accelerator. For GPU devices DKS uses cuFFT library provided by Nvidia, while for Intel MIC DKS uses

Intels MKL library. Additionally the FFT Poisson solver requires the calculation of Greens function and element by element multiplication of arrays. DKS contains optimized CUDA and OpenMP implementations of these algorithms. Detailed description of the algorithm and the created GPU codes is given in section 4.3.

In order to perform Monte Carlo simulations for particle matter interaction, DKS contains CUDA and OpenMP codes for energy loss and Coulomb scattering calculations. For random numbers DKS uses Nvidias cuRand library and Intels MKL VSL library. In addition DKS provides functions to move dead particles to the end of the array, this is done by a loop on the Intel MIC and using Thrust library to sort the array on the GPU. A detailed description of the algorithm and the implemented CUDA and OpenMP methods in DKS is given in section 4.4.

3.3.3 DKS and Musrfit

DKS Musrfit module contains functions to perform χ^2 and max-log likelihood calculations as well as provides the mechanism to create and compile the GPU code at runtime. Musrfit targets GPUs to accelerate the calculations, therefore DKS contains kernel implementations using CUDA and OpenCL to allow targeting of Nvidia and AMD devices. The OpenCL implementation also allows using the CPU and potentially Intel MIC devices for acceleration, although the code is optimized specifically for GPU architectures. A Detailed description of the DKS Musrfit module is given in chapter 5.

3.3.4 DKS and PET Image Reconstruction

The DKS PET image reconstruction module targets the Nvidia GPUs and contains the CUDA implementation of forward and backward projections. These functions allow to perform all the time consuming parts of the reconstruction on Nvidia GPUs. The algorithms and their implementation are described in detail in chapter 6.

3.4 DKS auto-tuning

Automatic performance tuning, or auto-tuning, describes a process of automatic selection of runtime parameters to achieve the best possible performance. Performance can be measured in multiple ways like execution time, power consumption etc. In DKS we are focusing on the execution time so this measure of performance will be used in the auto-tuning framework.

3.4.1 Auto-tuning concept

Performance of the code that is running on massively parallel processors like GPUs can be affected by the number of launch parameters like the number of threads per block and the number of blocks. The optimal launch parameters are changing between GPU kernels since each function requires a different amount of GPU resources. The optimal launch parameters also change between different devices since each device has different resources available.

The selection of these parameters can be left to the programmers, but this way it is usually based on trial and error tests, knowledge of hardware and the specifics of the algorithm. It is not transferable to other algorithms and more importantly it is not transferable to other devices. Auto-tuning frameworks aim to solve this problem by automatically finding the optimal parameters.

The simplest method of auto-tuning is to perform a exhaustive search, in which all the possible parameter configurations are tested [25]. This approach is guaranteed to give the best possible solution, but is only suitable if the parameter space is small. Another similar approach is line search where all the possible parameter configurations are tested one parameter at the time. The results of this approach are dependent on the order in which the parameters are tested and it is up to the developer to provide the best search order for each case [26]. In general auto-tuning frameworks where the details of the algorithm are not taken into account during the parameter optimization process heuristic search algorithms are used to find the best configurations. There are many search based algorithms that can be used, but the most widely used are hill climbing, simulated annealing and genetic algorithm [27, 28, 29].

3.4.2 DKS and auto-tuning

There are several strategies when auto-tuning of an application can be performed. They can be divided in three main categories:

- Compile time - auto-tuning is performed during installation of the application by analyzing the source code, executing the kernels and taking into account the knowledge of the hardware information that is available on the system;
- Offline - auto-tuning is performed after installation, but before the application is run;
- Online - auto-tuning is performed continuously during runtime allowing application to adapt to changes in kernel parameters [30].

In DKS we are focusing on the offline auto-tuning by using previously created benchmark tests. The benchmark tests provide the function that needs to be tested and the parameters that should be adjusted. The parameters to be configured for each function need to be set by developer. The limits of the parameters may be also set by the developer or these limits may be extracted from device properties, depending on the device that is installed on the system (like max threads per block on the GPU, or max threads for OpenMP, etc.).

```
1  //create the function to be timed, with the necessary parameters
2  std::function<int()> f = std::bind(&ChiSquareRuntime::launchChiSquare,
       chiSq, fitType, mem_data, mem_err, length, numpar, numfunc, nummap,
       timeStart, timeStep, result);
3
4  //add the function to the auto-tuning framework
5  autoTuning->setFunction(f, "launchChiSquare");
6
7  //set the adjustable parameters and their ranges
8  autoTuning->addParameter(&chiSq->blockSize_m, minThreads, maxThreads,
       step, "BlockSize");
9  autoTuning->addParameter(&chiSq->numBlocks_m, minBlocks, maxBlocs, step,
       "NumBlocks");
10
11 //perform the parameter search
12 autoTuning->hillClimbing(restarts);
```

Code example 3.6: Example of setup of auto-tuning test

An example of auto-tuning test setup is shown in code example 3.6. The auto-tuning framework executes the function varying the parameters in order to find the parameter set that gives the best (fastest) execution time. The developer is responsible of providing the function for the auto-tuning with the necessary parameters, parameters that should be auto-tuned and there ranges, and the search method that should be used. After the benchmark tests are done a small database is created that stores the launch configurations for each kernel on the tested devices. This database is used by DKS at run-time to determine the kernel launch parameters. If the auto-tuning tests are not performed the DKS uses default parameters, that are set by the developer.

Chapter 4

OPAL

4.1 OPAL and DKS

OPAL (Object Oriented Particle Accelerator Library) is a parallel, open source C++ framework for general particle accelerator simulations which includes 3D space charge, short range wake fields, and particle matter interaction. OPAL is based on IPPL (Independent Parallel Particle Layer) which adds parallel capabilities. Main functions inherited from IPPL are structured rectangular grids, fields, parallel FFT, and particles with the respective interpolation operators. Other features are expression templates, and massive parallelism (up to 65000 processors) which allows it to tackle the largest problems in the field.

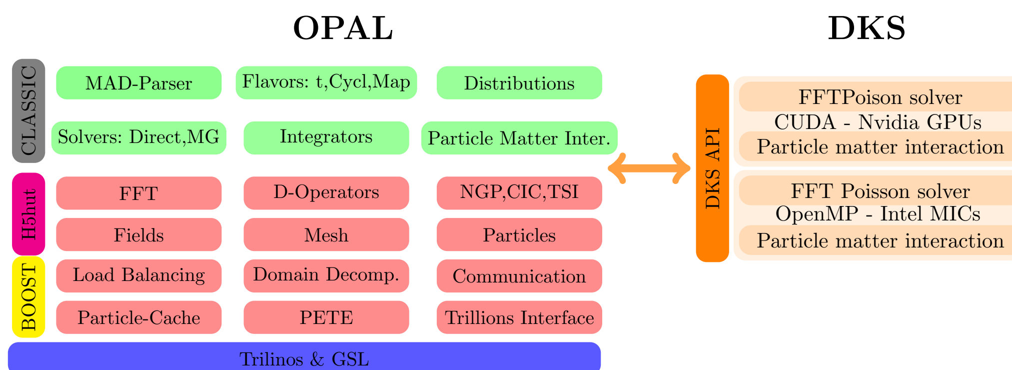

Figure 4.1: The OPAL software structure and connection to DKS

4.2 FFT Poisson solver

4.2.1 FFT Based Particle-Mesh Solver

The Particle-Mesh (PM) solver is best described in the book by R.W. Hockney & J.W. Eastwood [31]. Instead of calculating the mutual interaction of a large number of particles, in the PM solver one discretises the computational domain $\Omega := [-a_x, a_x] \times [-a_y, a_y] \times [-a_z, a_z]$ into a regular mesh of $M_x \times M_y \times M_z$ grid points. The beam sizes a_x, a_y, a_z are usually time dependent. Other geometries are possible but not discussed here. The mesh sizes h_x, h_y, and h_z are allowed to change independently over time to assure a particle fitted grid. An essential part of any self-consistent electrostatic beam dynamics code is the Poisson solver. From the – time to solution – point of view, we observe that in the order of 1/3 of the computational time is spent in this algorithm,

In many of the physics applications, the bunch can be considered as small compared to the transverse size of the surrounding beam pipe ($\partial\Omega$). If this is the case the conducting walls can be neglected and, we can solve an open boundary problem. Here we follow the method of Hockney [32] and compute the potential on a grid of size $2^3 M_x M_y M_z$. The calculation then is making use of Fast Fourier Transform (FFT) techniques, with a computational effort scaling as $\mathcal{O} 2^3 M_x M_y M_z \log_2(2^3 M_x M_y M_z)$ [31, 33, 32].

4.2.2 FFT-based Convolutions

Given a charge density ρ, we search for the scalar potential ϕ by solving Poisson's equation

$$\nabla^2 \phi = -\rho/\epsilon_0, \tag{4.1}$$

subject to $\phi = 0$ at $\partial\phi \to \infty$, i.e. in an unbound domain. If we know the Green's function $G(x, x', y, y', z, z')$, then the solution

$$\phi(x, y, z) = \int\int\int dx'dy'dz' \rho(x', y', z') G(x, x', y, y', z, z')$$

is the convolution of the a source charge at (x', y', z') and G. In our case of an isolated charge distribution, we get

$$\phi(x,y,z) = \int\int\int dx'dy'dz'\rho(x',y',z')G(x-x',y-y',z-z'), \tag{4.2}$$

with

$$G(u,v,w) = \frac{1}{\sqrt{u^2+v^2+w^2}}.$$

We now discretise Eq. (4.2) on the previous mentioned Cartesian grid

$$\phi_{i,j,k} = h_x h_y h_z \sum_{i'=1}^{M_x}\sum_{j'=1}^{M_y}\sum_{k'=1}^{M_z} \rho_{i',j',k'} G_{i-i',j-j',k-k'}. \tag{4.3}$$

The two scalar fields $\rho_{i,j,k}$ and $G_{i-i',j-j',k-k'}$ are now defined on the grid and we efficiently obtain the solution of Eq. (4.3) using Fourier techniques by

$$\phi_{i,j,k} = h_x h_y h_z \ \mathrm{FT}^{-1}\{(\mathrm{FT}\{\rho_{i,j,k}\}) \otimes (\mathrm{FT}\{G_{i,j,k}\})\},$$

with $\otimes$ denoting the Hadamard product. The notation FT{.} for the forward FFT and $\mathrm{FT}^{-1}\{.\}$ for the inverse FFT is used.

4.3 FFT Poisson solver and DKS

The algorithm implemented in OPAL to solve the Poisson equation 4.1 is sketched in algorithm 1. To allow OPAL to offload the FFT Poisson solver to the GPU or Intel MIC the algorithm is implemented in DKS using CUDA and OpenMP. Since OPAL parallelizes the computations using MPI, the CUDA version allows multiple MPI nodes to share one GPU device for, the development of the OpenMP version was discontinued after poor performance obtained by the single core implementation.

Algorithm 1: FFT Poisson Solver

Host: Setup environment on the device;
Host: Allocate memory on the device for ρ and G;
Device: Compute integrated Greens function G on the device;
Device: FFT: $G \rightarrow \hat{G}$;
Host: Transfer ρ to the device;
Device: FFT: $\rho \rightarrow \hat{\rho}$;
Device: multiply complex fieds $\hat{\phi} = \hat{\rho} \cdot \hat{G}$;
Device: IFFT: $\hat{\phi} \rightarrow \phi$;
Host: Free memory on the device;

4.3.1 CUDA Implementation of the Poisson Solver in DKS

For use on NVIDIA GPUs the FFT Poisson solver is implemented in DKS in double precision using CUDA. It uses NVIDIA's cuFFT library to perform the 3D FFT, separate kernels to calculate the Greens function and, perform the multiplication on the GPU. CUDA streams are used to overlap the transfer of the ρ field to the GPU and the calculation of the Greens function. The sequence diagram in Figure 4.2 shows the steps executed for the FFT Poisson solver on the host and GPU.

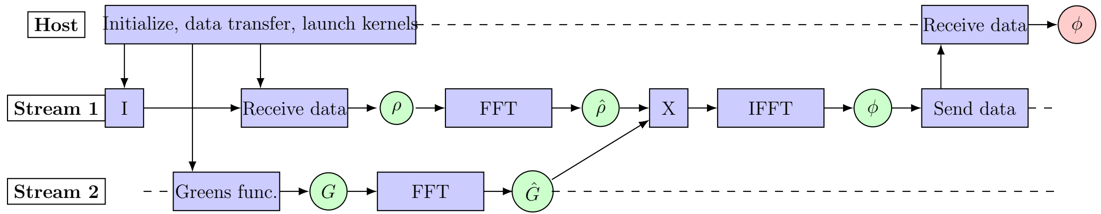

Figure 4.2: FFT-Poisson solver sequence diagram

OPAL allows to parallelize the simulation over multiple CPU cores using MPI. When the simulation runs over multiple cores each core receives its own part of the computational domain. One example of the domain decomposition is shown in figure 4.3. Since the FFT needs to be run on the whole domain only one process initializes the CUDA kernels, while other processes only provide the data to the GPU and read the results.

In order to allow multiple CPU cores to write to the same memory space

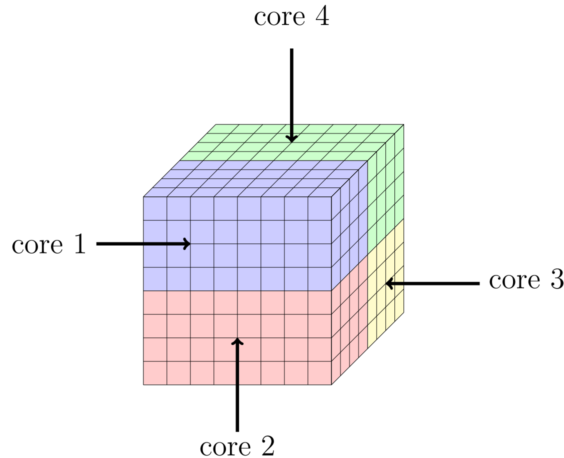

Figure 4.3: FFT-Poisson 2D domain decomposition

CUDA inter-process communications (IPC) are used. The master process allocates the necessary memory on the GPU and using CUDA IPC shares the pointer to this memory with other processes using MPI. Once all the MPI processes have finished writing the data to the GPU master process launches the FFT calculations. While the data transfer is performed the master process launches the kernels to calculate the Greens function on the GPU, which can be performed in parallel of data transfer.

4.3.2 OpenMP Implementation of the Poisson Solver in DKS

To take advantage of Intel MIC devices the FFT Poisson solver was implemented using OpenMP and the Intel Math Kernel Library. Intel MIC was used in the offload mode and Greens function and multiplication of the complex fields was parallelized on the device using OpenMP. To perform the FFT on the device the Intel MKL library was used which provides the FFT optimized for MIC devices.

The algorithm flow is kept the same as in the CUDA case and the only change in OPAL that is necessary to switch between GPU and MIC is to set the correct device. The first results using Intel MIC showed that the performance of the FFT Poisson solver on the devices is around 3 times slower that the performance on a single CPU core and more than 40 times slower than the performance of the

GPU. The bad performance was due to the fact that the 3D FFT provided by Intel MKL library performed slower than the cuFFT on the GPU and also slower than the FFT implementation in OPAL. Because of the poor performance of the FFT on Intel MIC the FFT Poisson solver was not developed further for Intel MIC Knights corner and was not included in the final benchmarks [34].

4.3.3 Integration of DKS in OPAL

OPAL uses the DKS interface to initiate the Greens function computation, FFT and the multiplication of the complex fields on the GPU. If multiple CPU cores are used, the computation is only initiated by one core while other cores just provide the data and read the results back from the GPU.

4.3.4 Performance Results

To test OPAL's performance, we use a RingCyclotron example with a similar problem setup as reported in [35, 36]. The test system consists of a host with two Intel Xeon e5-2609 v2 processors and a Nvidia Tesla K20c or Tesla K40c. On the host, 8 CPU cores are available. The first simulations where run using only the 8 CPUs available on the host. However in the second case, DKS is used to offload the FFT Poisson solver to the GPU.

Table 4.1: FFT Poisson Solver results

FFT size	DKS	Total time (s)	OPAL speedup	FFTPoisson time (s)	FFTPoisson speedup
64x64x32	no	324.98		22.53	
	K20c	**311.17**	**×1.04**	**7.42**	**×3**
	K40c	**293.7**	**×1.10**	**7.32**	**×3**
128x128x64	no	434.22		206.73	
	K20c	**262.74**	**×1.6**	**32.15**	**×6.5**
	K40c	**245.08**	**×1.8**	**25.87**	**×8**
256x256x128	no	2308.05		1879.84	
	K20c	**625.37**	**×3.6**	**202.63**	**×9.3**
	K40c	**542.73**	**×4.2**	**160.87**	**×11.7**
512x512x256	no	3760.46		3327.14	
	K40c	**716.86**	**×5.2**	**302.49**	**×11**

Table 4.1 shows the results of these test runs for multiple problem sizes. The results show that offloading the FFT Poisson solver to the GPU can provide a substantial speedup even when we have multiple CPU cores sharing one accelerator. The limiting factor for the performance of the FFT Poisson solver is the data transfer from the host side to the device. Since data needs to be moved to and from GPU at every time step, for the largest problem size reported in the benchmark tests, data transfer can take up to 55% of the total simulation time. Another limiting factor is the performance of the FFT transform. FFT is a memory bound algorithm and is able to reach only a fraction (about 10% was observed on our test system) of the devices peak performance. Since for the Poisson solver time to perform FFT takes up to 80% of all time spent for calculations, the speed of the solver depends severely on the speed of the FFT.

4.4 Particle matter interaction

One of the features in OPAL is the ability to perform Monte Carlo simulations of the particle beam interaction with matter. A fast charged particle moving through the material undergoes collisions with the atomic electrons and loses energy. In addition, particles are also deflected from their original trajectory due to the Coulomb scattering with nuclei, as shown in figure 4.4. The energy loss in OPAL is calculated using the Bethe-Bloch formula and the change of the particle trajectory is simulated using Multiple Coulomb Scattering and Single Rutherford Scattering [37, 38, 39]. All the computation is done in double precision. At every time step when the particle beam is inside a material, the following steps are executed:

- calculate the energy loss of the beam,
- delete the particle if the particle's kinetic energy is smaller than a given threshold,
- apply Coulomb and Rutherford scattering to the beam.

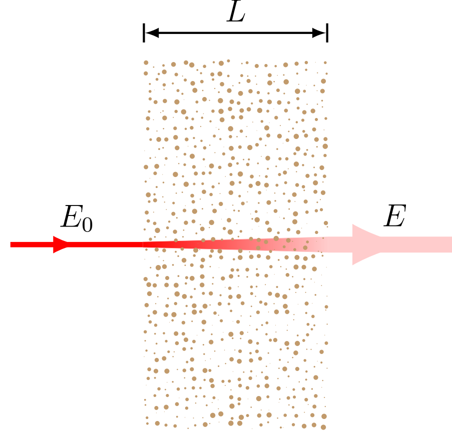

Figure 4.4: Particle matter interaction. With the final energy $E < E_0$ and larger momenta spread due to Coulomb scattering and the large angle Rutherford scattering.

4.4.1 The Energy Loss

The energy loss is calculated using the following Bethe-Bloch equation:

$$-dE/dx = \frac{Kz^2 Z}{A\beta^2}\left[\frac{1}{2}\ln\frac{2m_e c^2\beta^2\gamma^2 T_{max}}{I^2} - \beta^2\right], \tag{4.4}$$

where Z is the atomic number of absorber, A is the atomic mass of absorber, m_e is the electron mass, and z is the charge number of the incident particle. K is defined as $4\pi N_A r_e^2 m_e c^2$, where r_e is the classical electron radius, N_A is the Avogadro's number, and I is the mean excitation energy. β and γ are the kinematic variables. Lastly T_{max} is the maximum kinetic energy that can be imparted to a free electron in a single collision. It is defined as,

$$T_{max} = \frac{2m_e c^2\beta^2\gamma^2}{1 + 2\gamma m_e/M + (m_e/M)^2}, \tag{4.5}$$

where M is the incident particle mass.

For relatively thick absorbers, the number of collisions is large, and therefore the energy loss distribution is Gaussian in form. For non-relativistic heavy

particles, the spread, σ_0, of the Gaussian distribution is calculated by:

$$\sigma_0^2 = 4\pi N_A r_e^2 m_e^2 c^4 \rho \frac{Z}{A} \Delta s, \tag{4.6}$$

where ρ is the density and s is the thickness of the material.

4.4.2 Coulomb Scattering

The Coulomb scattering is treated as two independent events: the multiple Coulomb scattering and the large angle Rutherford scattering.

Using the distribution given in [40], the multiple- and single-scattering distributions can be written as:

$$P_M(\alpha)d\alpha = \frac{1}{\sqrt{\pi}} e^{-\alpha^2} d\alpha, \tag{4.7}$$

$$P_S(\alpha)d\alpha = \frac{1}{8\ln(204Z^{-1/3})} \frac{d\alpha}{\alpha^3}, \tag{4.8}$$

where $\alpha = \frac{\theta}{\langle\Theta^2\rangle^{1/2}} = \frac{\theta}{\sqrt{2}\theta_0}$. The transition point between multi and single scattering occurs at the angle $\theta = 2.5\sqrt{2}\theta_0 \approx 3.5\theta_0$, where the value of θ_0 is the scattering angle from Moliere's theory and is defined as,

$$\theta_0 = \frac{13.6MeV}{\beta cp} z\sqrt{\Delta s/X_0}[1 + 0.038\ln(\Delta s/X_0)], \tag{4.9}$$

where p is the momentum, Δs is the step size, and X_0 is the radiation length.

To perform a Monte Carlo simulation for the multiple Coulomb scattering two independent Gaussian random variables (z_1 and z_2) are created with mean zero and variance one. The new position and momentum can then be calculated by:

$$x = x + \Delta s p_x + z_1 \Delta s \theta_0/\sqrt{12} + z_2 \Delta s \theta_0/2, \tag{4.10}$$

$$p_x = p_x + z_2\theta_0. \tag{4.11}$$

The values for the $y - p_y$ plane are calculated with the very same Monte-Carlo algorithm.

4.4.3 Large Angle Rutherford Scattering

Only a small percentage of particles undergo large angle Rutherford scattering. This percentage is given by:

$$\chi_{single} < \frac{\int_{2.5}^{\infty} P_S(\alpha)d\alpha}{\int_0^{2.5} P_M(\alpha)d\alpha + \int_{2.5}^{\infty} P_S(\alpha)d\alpha} = 0.0047. \tag{4.12}$$

The process to define if a particle undergoes a Rutherford scatter is as follows:

- A random number ξ_1 between 0 and 1 is generated. If and only if this random number is smaller than χ_{single} the particle undergoes single Rutherford scattering. The value of χ_{single} does not change significantly for different materials, hence a fixed value of $\chi_{single} = 0.0047$ is used, in order to avoid unnecessary computation.
- A second random variable ξ_2 between 0 and 1 is generated to calculate the angle, the particle rotates about.
- The third and last random number ξ_3 determines the direction of the rotation:

$$\theta_{Ru} = \begin{cases} +2.5\sqrt{\frac{1}{\xi_2}}\sqrt{2}\theta_0 & \text{if} \quad \xi_3 < 0.5 \\ -2.5\sqrt{\frac{1}{\xi_2}}\sqrt{2}\theta_0 & \text{if} \quad \xi_3 > 0.5. \end{cases} \tag{4.13}$$

4.4.4 OPAL implementation

This chapter describes how the particle matter interaction is implemented in OPAL. It gives a brief description on how particles are stored and how simulations for particle matter interaction are performed.

Particles are stored in structure as shown in code example 4.1, where $Vector_t$ is a defined vector data type that holds 3 double values. All the particles in the beamline are stored in a list called *bunch*. Once the particle enters the material it is moved from *bunch* to the $locParts_m$ list which holds all the particles that are currently in the material. Important variables that are needed for particle matter interaction calculations are *Rincol*, which contains the position of the particle,

and *Pincol* which contain momentum of the particle. Also used is the *label* which, for the CPU case can have two values, 0 (particle alive) or -1 (particle dead). For random values that are necessary for this Monte Carlo simulation OPAL uses GSL random number generator.

```
typedef struct {
  int label; unsigned localID;
  Vector_t Rincol; Vector_t Pincol;
  long IDincol; int Binincol;
  double DTincol; double Qincol;
  long LastSecincol;
  Vector_t Bfincol; Vector_t Efincol;
} PART;

std::vector<PART> locParts_m;
```

Code example 4.1: Struct to store one particle

OPAL code for executing particle matter interaction is shown in example 4.2. The function *checkHit* checks if the particle is still inside the material. If *checkHit* returns true and particle is inside the material *EnergyLoss* calculates the amount of energy each particle loses. Each calculation of *EnergyLoss* requires two random numbers of Gaussian distribution. After *EnergyLoss*, its momentum is updated and Coulomb scattering is simulated, if the particle loses to much energy it is considered 'dead' and removed from the simulation. If *checkHit* returns false, then the particle has exited the material, OPAL lets the particle drift during this time-step and then adds the particle back to the *bunch*.

The Coulomb scattering is treated as two independent events: the multiple Coulomb scattering and the large angle Rutherford scattering. Every particle undergoes the Coulomb scattering, but only a small percentage of particles undergo large angle Rutherford scattering.

To perform the Monte Carlo simulation for the multiple Coulomb scattering two independent Gaussian random variables are created, with mean zero and variance one: z_1 and z_2. The main parameters for evaluating the effect of the multiple Coulomb scattering are the new spatial coordinate y_{plane} and the new angle θ_{plane} defined by:

```
for(int i = 0; i < locParts_m.size(); ++i) { //loop trough all particles
  bool pdead = false;
  Vector_t &R = locParts_m[i].Rincol;
  Vector_t &P = locParts_m[i].Pincol;
  double Eng = (sqrt(1.0  + dot(P, P)) - 1) * m_p;

  if(checkHit(R,P,dT_m, deg, coll)) { //chekc if particle is in material
    EnergyLoss(&Eng, &pdead, dT_m); //calc energy loss

    if(!pdead) { //check if particle is still alive
      double tot =  sqrt((m_p + Eng) * (m_p + Eng) - (m_p) * (m_p)) / m_p;
      P = P * tot / sqrt(dot(P, P));
      CoulombScat(R, P, dT_m); //calc Coulomb scattering
      locParts_m[i].Rincol = R; //update R and P values in the list
      locParts_m[i].Pincol = P;
    } else { //if particle dead set label to -1
      locParts_m[i].label = -1.0;
    }
  } else { //if particle exits material drift and move out of the list
    locParts_m[i].Rincol += dT_m * Physics::c * P / sqrt(1.0+dot(P, P));
    addBackToBunch(bunch, i);
  }
}
```

Code example 4.2: Loop trough the particles, calculate energy loss, Coulomb scattering and large angle Rutherford scattering for each particle.

$$y_{plane} = \frac{z_1 \theta_0 \Delta s}{\sqrt{12}} + \frac{z_2 \theta_0 \Delta s}{2} \tag{4.14}$$

and

$$\theta_{plane} = z_2 \theta_0 \tag{4.15}$$

Once the new spatial coordinate and the new angle have been evaluated, the reference system of the particle is adjusted in OPAL to the new direction of motion with the angle Ψ_{yz} [41]. The new coordinates of the particle are:

$$y = y + \Delta s p_y + y_{plane} \cdot \cos \Psi_{yz} \tag{4.16}$$

$$z = z - y_{plane} \cdot \sin \Psi_{yz} + \Delta s p_z \tag{4.17}$$

Large angle Rutherford scattering is done for only a small percentage of particles, this percentage is given by equation 4.12.

A random number ξ_1 between 0 and 1 is generated. If and only if this random number is smaller than χ_{single} the particle undergoes single Rutherford scattering. The value of χ_{single} does not change significantly for different materials, hence a fixed value of $\chi_{single} = 0.0047$ is used, in order to avoid unnecessary computation. A second random variable ξ_2 between 0 and 1 is generated to calculate the angle, the particle rotates about. The third and last random number ξ_3 determines the direction of the rotation. Code that performs Coulomb scattering and single Rutherford scattering on a single particle is shown in example 4.3.

```
1  //Coulomb scattering
2  double z1 = gsl_ran_gaussian(rGen_m,1.0);
3  double z2 = gsl_ran_gaussian(rGen_m,1.0);
4  double thetacou = z2 * theta0;
5
6  while(fabs(thetacou) > 3.5 * theta0) {
7    z1 = gsl_ran_gaussian(rGen_m,1.0);
8    z2 = gsl_ran_gaussian(rGen_m,1.0);
9    thetacou = z2 * theta0;
10 }
11 updatePositionAndMomentum();
12
13 //Rutherford Scattering
14 if(gsl_rng_uniform(rGen_m) < 0.0047) {
15   double P3 = gsl_rng_uniform(rGen_m);
16   double thetaru = 2.5 * sqrt(1 / P3) * sqrt(2.0) * theta0;
17   if(gsl_rng_uniform(rGen_m) > 0.5)
18     thetaru = -thetaru;
19   updatePositionAndMomentum();
20 }
```

Code example 4.3: Calculate energy loss for a particle using Bethe-Bloch equation

To manage the particle array after calculation of the energy loss and Coulomb scattering, dead particles are moved to the end of the list and deleted. The CPU code for performing this action is shown in code example 4.4. Particles which are dead have label -1, so they are moved to the end of the list.

4.4.5 Particle drift via time integration

In addition to particle matter interactions particle drift via time integration was also offloaded to the accelerator. The time integration in OPAL is implemented

```
1  if (locParts_m.size() > 0) {
2    unsigned long i = 0;
3    unsigned long end = locParts_m.size() - 1;
4    while (i < end) {
5      if (locParts_m[i].label == -1) {
6        std::swap(locParts_m[i], locParts_m[end]);
7        --end;
8      } else {
9        ++ i;
10     }
11   }
12   if (locParts_m[end].label != -1)
13     ++end;
14   if (end < locParts_m.size())
15     locParts_m.erase(locParts_m.begin() + end, locParts_m.end());
16 }
```

Code example 4.4: Move dead particles to the end of the list and erase

using Boris-Buneman method and described in detail in [42]. From all the steps of Boris-Buneman method only particle push, which updates the position of the particle at every time step was offloaded to the device.

Algorithm 2: Push step of time integration.

Input : Particle position R
Input : Particle momentum P
Output: New particle position R
for $i \leftarrow 0$ **to** N **do**
  $R_i \leftarrow R_i + 0.5 * P_i \div \sqrt{1.0 + \|P_i\|}$
end

The algorithm for particle push is fairly simple and is shown in 2. Since the position of each particle can be updated independently the algorithm can be easily parallelized for GPU and MIC architectures.

4.5 Particle matter interaction and DKS

4.5.1 The DKS Implementation of the Particle Matter Interaction Model

For particle matter interactions, DKS has CUDA and OpenMP implementations of all the algorithm steps described above. This allows the computation of the

energy loss, the Coulomb scattering, and the Rutherford scattering to be offloaded to the GPU or Intel MIC. On top of particle matter interaction, DKS is also able to offload to the accelerators the transport of particles before and after the material using a time integration scheme. The sequence diagram for the integration is shown in Figure 4.5, where stream1 updates the particle position, while stream2 updates the local coordinate system.

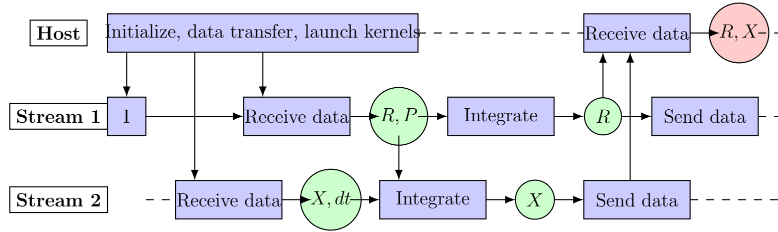

Figure 4.5: Integration sequence diagrams

To increase the performance, the data transfer is minimized as much as possible and particles that are drifting before or after the material are kept on the device. They are updated only when there are a some particles returning from the material or there has been an MPI update to balance the workload between MPI processes. Pinned host memory and streams are used with the GPU version to increase the date transfer speed, and overlap the data transfer and kernel execution for the particle drift.

Particles that are in the material are also kept in the device memory. NVIDIAs cuRAND and Intels MKL VSL libraries are used to generate random numbers to determine the necessary distributions for energy loss and scattering. NVIDIA's Thrust library is used to sort and count the particles on the GPU in order to manage the particles that need to come out of the material, but also to exclude the dead particles from Monte-Carlo simulations. Because of the high complexity of the algorithm, the CUDA version uses shared device memory for variable storage to reduce the register pressure of the kernels in order to achieve higher GPU occupancy. Structure of arrays data layout is used to store all the particles in order to allow Intel compiler to better vectorize the code for the Xeon Phi coprocessor. The sequence diagram of the particle in matter simulations on the CPU and the device is shown in Figure 4.6.

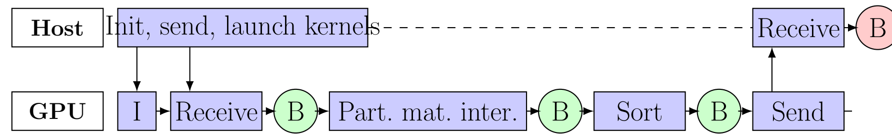

Figure 4.6: Particle in matter sequence diagrams, where B denotes particle bunch

4.5.2 Particle matter interaction on the GPU

This chapter describes the CUDA kernels used to offload the particle matter interaction code to Nvidias GPUs. There are three main consideration when creating the CUDA algorithm for the OPALs Monte Carlo code:

- Data structure - a new data structure is created to store the particle data that is needed for Monte Carlo simulation;
- CUDA kernel - CUDA kernel is created that performs the Monte Carlo simulation on a single particle;
- Thrust sort - sort the particle array to remove dead particles or particles that are coming out if the material.

The data structure used on the GPU is similar to the one used on the CPU version, but only elements that are needed for the simulation are moved to the GPU, the rest of the particle data stays in the host memory. This is done in order to decrease the amount of GPU memory used, the time needed for data transfer and to create a better data access patterns for the CUDA kernel. The CUDA data structure to store the particle is shown in code example 4.5. The rest of the particle data is stored in structure 4.1 and the are linked using the *localID*.

```
1 typedef struct {
2   int label;
3   unsigned localID;
4   double3 Rincol;
5   double3 Pincol;
6 } CUDA_PART;
```

Code example 4.5: Data structure for storing particles that are in the material on the GPU

Each CUDA kernel executes the MC simulation for one particle. The shared memory on the GPU is used to store the parameters that are reused throughout the simulation. These parameters include the size dimensions of the degrader and the properties of the material. The shared memory is also used to store the position of the particle while the momentum of the particle is stored in the private memory of each thread. The shared memory usage allows to access the particle position with low latency while reducing the number of registers needed by each thread. For random numbers cuRand library is used. The library is initialized at the beginning of the simulation and each thread receives its own random number state initialized with a different seed. Libraries *curand_normal_double* and *curand_uniform_double* functions are used to get the random numbers with the distribution as required by the algorithm.

All of the steps of the algorithm are implemented in a single kernel call. The pseudo code for the kernel call is shown in the code example 4.6. The kernel contains some conditional statements so some divergence in the threads of the same warp will be observed mostly towards the end of the simulation when particles start leaving the material or loose too much energy. Since most of the time for the MC simulation is spent when all the particles are inside the material this divergence of the threads will not influence the performance of the algorithm too much. In order to avoid warp divergence at the end of the material additional sort by the position and energy of the particle would be necessary after every iteration, which would be more inefficient than the effects of some warps having divergent threads.

To sort the particles the Thrust library is used. The particles are sorted by *label* in descending order, so particles that are moving out of the material are at the end of the array. Before the sort is performed the particles that are moving out of the material are counted and sort is performed only if such particles are found. The pseudo code to perform the particle sort on the GPU is shown in code example 4.7.

```
__shared__ double p[NUMPAR]; //shared memory variables
__shared__ double3 R[BLOCK_SIZE];
if (tid < NUMPAR)
  p[tid] = par[tid]; //load parameters to shared memory
__syncthreads();

R[tid] = data.Rincol[idx]; //load the data from global memory
double3  P = data.Pincol[idx];
curandState  s = state[idx];
double sq = sqrt(1.0 + dot(P, P)); //calculate constants
bool pdead = false;

if (checkHit(R[tid].z, p)) { //check if particle is in material
  double Eng = (sq - 1) * M_P;
  energyLoss(Eng, pdead, s, p); //calculate energy loss
  if (!pdead) { //check if particle not dead
    double ptot = sqrt((M_P + Eng) * (M_P + Eng) - (M_P * M_P)) / M_P;
    sq = sqrt(dot(P, P));
    P = P * ptot / sq;
    coulombScat(R[tid], P, s, p); //simulate Coulomb scattering
    data.Pincol[idx] = P;
  } else {
    data.label[idx] = -1; //set particle to be deleted
  }
} else {
  R[tid] = R[tid] + p[DT_M] * C * P / sq; //drift particle
  data.label[idx] = -2; //set particle to be moved out of material
}
data.Rincol[idx] = R[tid];
state[idx] = s;
```

Code example 4.6: CUDA kernel for MC simulation.

4.5.3 Particle matter interaction on the MIC

This chapter describe steps taken to offload the simulation of particle matter interaction on the Intel MIC co-processor. Four major points are emphasized to improve the performance of the code for the MIC architecture.

- Data size - not all the data that is stored for each particle is needed for this Monte Carlo simulation, so only the necessary data needs to be transferred to the MIC to decrease the data transfer times;
- Data layout - array of structures layout was changed to structure of arrays and all of the data was aligned properly for the MIC in order to improve the vectorization of the code;
- Parallelization - CPU version of the particle matter interaction was designed to run on single core and it is possible to split the whole simulation to

```
//wrap mem_ptr with thrust device ptr
thrust::device_ptr<CUDA_PART_SMALL> dev_ptr( (CUDA_PART*)mem_ptr);

//count -2 and -1 particles
compare_particle_small comp;
comp.set_threshold(0);
numaddback = thrust::count_if(dev_ptr, dev_ptr + numparticles, comp);

//sort particles
if (numaddback > 0)
  thrust::sort(dev_ptr, dev_ptr + numparticles, comp);
```

Code example 4.7: Sorting of the particles on the GPU.

multiple nodes or cores using MPI. For the MIC version it was important to parallelize the code so it is able to take advantage of all the cores available on the MIC. For this reason OpenMP was used;

- Vectorization - to take advantage of Intel MIC vector processing unit it is important to vectorize the code, for this reason the data layout and alignment was changed, and also OpenMP pragmas where inserted to let the compiler know which code is safe to vectorize.

```
typedef struct {
  int *label;
  unsigned *localID;
  double *rx, *ry, *rz;
  double *px, *py, *pz;
} PART_SoA_DKS;
```

Code example 4.8: Data structure for storing particles that are in the material on the MIC

The data layout of the particle was changed from array of structures to struct of arrays, and since only label, position and momentum of the particle is used for calculations these are the only values transferred to the co-processor. In addition *localID* of the particle was stored in order to move the particle back to the *bunch* correctly once it exits the material. The data structure used on the MIC is shown in example 4.8. The rest of the particles data is kept on the host side and when particle is moved back to *bunch* it is retrieved using *localID*. Another small change from the CPU version was the introduction of a new label for particles in

addition to 0 (particle alive and in material) and -1 (particle dead) label -2 was added for particles which are alive and leaving the material.

```
#pragma omp for
for (int ii = 0; ii < totalpart; ii += MIC_WIDTH) {
  //vectorize main loop
  #pragma vector aligned
  #pragma simd
  for (int i = ii; i < ii + MIC_WIDTH; i++) {
    if ( !checkHit(rz[i], par) ) {
      double sq = sqrt(1.0 + dot(px[i], py[i], pz[i]));
      rx[i] = rx[i] + dT_m * C * px[i] / sq;
      ry[i] = ry[i] + dT_m * C * py[i] / sq;
      rz[i] = rz[i] + dT_m * C * pz[i] / sq;
      label[i] = -2;
    }
  }
}
```

Code example 4.9: Vectorization of checkHit and particle drift if it leaves material

The first part of the simulation that was vectorized was checking if the particle is still in the material. The outer loop traverses the particles with the stride MIC_WIDTH and splits all the particles across the threads of the Xeon Phi. The inner loop is vecotrized using the simd instruction and the compiler is also informed about the proper alignment. This loop checks if the particle is still in the material. If the particle has left the material drift the particle and change the label of the particle to -2.

The MKL VSL random number generator was used for Monte Carlo simulations. To vectorize energy loss calculations the same strip mining technique as shown if code example 4.9 was used. The code loops trough all the particle with a stride MIC_WIDTH, generates 2*MIC_WIDTH random values (two random values per particle) for the EnergyLoss function, and then the inner loop is vectorized using simd instruction, as shown in the code example 4.10. Since all the memory is aligned properly *#pragma vector aligned* is used to inform the compiler about the correct alignment. In the CPU version of the code random values are generated directly in the EnergyLoss function, but for the MIC all the necessary random values are generated beforehand and passed to this function. This is one more reason to use strip mining technique to parallelize and vectorize the loop - by using strides we limit the necessary memory for temporary storage of pregenerated random values. If after the energy loss the particle is still alive its

```
1  //array of size 2*WIDTH for storing random values for the energyloss function
2  double randv[2*MIC_WIDTH] __attribute__((aligned(64)));
3
4  //for loop trough particles if label == 0 eneregy loss and if pdead update
       label to -1
5  #pragma omp for
6  for (int ii = 0; ii < totalpart; ii += MIC_WIDTH) {
7    //create array of rand values (2 per thread)
8    vdRngGaussian (GAUSSIAN, stream , 2*MIC_WIDTH ,randv, 0.0, 1.0);
9    #pragma vector aligned
10   #pragma simd
11   for (int i = ii; i < ii + MIC_WIDTH; i++) {
12     double sq = sqrt(1.0 + dot(px[i], py[i], pz[i]));
13     double Eng = (sq - 1) * M_P;
14
15     if (label[i] == 0) //calc energy loss
16       energyLoss(&Eng, &dEdx, randv, i - ii);
17     if (Eng > 1e-4 && dEdx < 0) { //update momentum if particle alive
18       double ptot = sqrt((M_P + Eng) * (M_P + Eng) - (M_P * M_P)) / M_P;
19       sq = sqrt(dot(px[i], py[i], pz[i]));
20       px[i] = px[i] * ptot / sq;
21       py[i] = py[i] * ptot / sq;
22       pz[i] = pz[i] * ptot / sq;
23     }
24     if (Eng < 1e-4 || dEdx > 0) //update label if particle dead
25       label[i] = -1;
26   }
27 }
```

Code example 4.10: Vectorization of energy loss calculations

momentum gets updated, if the particle is dead after the energy loss calculations its label gets set to -1.

To vectorize the Coulomb scattering the same technique is used where possible. In parts of the code number of iterations per loop is dependent on the generated random values, these loops can not be vectorized so they are calculated separately and temporary arrays are used to hold intermediate values between loops. The outer loop is the same as in previous cases and goes trough the particles with the stride MIC_WIDTH and splits the work to multiple OpenMP threads. Inside this outer loop for calculating the Coulomb scattering the code is split into multiple loops vectorizing the ones that are possible. This is demonstrated in the code example 4.11.

Rutherford scattering is vectorized in the same way as previous loops as it is shown in 4.12, where P1, P2 and P3 are temporary arrays of size MIC_WIDTH with random values generated with vdRndGaussian.

```
double z1[MIC_WIDTH] __attribute__((aligned(64)));
double z2[MIC_WIDTH] __attribute__((aligned(64)));
double thetacou[MIC_WIDTH] __attribute__((aligned(64)));

//vectorize generation of necessary number of random values and calculation
    of thetacou
vdRngGaussian(GAUSSIAN, stream, MIC_WIDTH, z1, 0.0, 1.0);
vdRngGaussian(GAUSSIAN, stream, MIC_WIDTH, z2, 0.0, 1.0);
#pragma vector aligned
#pragma simd
for (int i = ii; i < ii + size; i++) {
  int idx = i - ii;
  thetacou[idx] = z2[idx] * theta0[idx];
}
//unknown number of iterations, cannot vectorize
for (int i = ii; i < ii + MIC_WIDTH; i++) {
  int idx = i - ii;
  if (label[i] == 0) {
    while(fabs(thetacou[idx]) > 3.5 * theta0[idx]) {
      vdRngGaussian(GAUSSIAN, stream, 1, &z1[idx], 0.0, 1.0);
      vdRngGaussian(GAUSSIAN, stream, 1, &z2[idx], 0.0, 1.0);
      thetacou[idx] = z2[idx] * theta0[idx];
    }
  }
}
#pragma vector aligned
#pragma simd
for (int i = ii; i < ii + size; i++) {
  if (label[i] == 0)
    updataPositionAndMomentu();
}
```

Code example 4.11: Vectorization of coulomb scattering

For particle management after each iteration all the particles that are dead or that are moving out of the material (label<0) are counted using OpenMP parallel reduction and these particles are moved to the end of the arrays. The particle movement is done in a single serial pass over particles. All the particles with label<0 are then transferred back to the host side, particles with label=-1 are deleted while particles with label=-2 are moved back to the *bunch*.

4.5.4 Integrating DKS in OPAL

When a new degrader object gets created in OPAL, DKS gets initialized and memory management is performed. The pseudo code for the initialization and memory management is shown in code example 4.13. The initialization consists of setting the device and framework to use, allocating memory for the particles and parameters, and initializing the random number generator.

```
1  #pragma vector aligned
2  #pragma simd
3  for (int i = ii; i < ii + MIC_WIDTH; i++) {
4    if (label[i] == 0) {
5      if(P1[idx] < 0.0047)
6        double thetaru = 2.5 * sqrt (1 / P3[idx] ) * sqrt (2.0) * theta0 ;
7        if (P2[idx] > 0.5)
8          thetaru = - thetaru ;
9        updataPositionAndMomentu();
10   }
11 }
```

Code example 4.12: Vectorization of Rutherford, P1, P2 and P3 are temporary arrays of size MIC_WIDTH with pregenerated random values.

```
1  DKSBase dksbase; //initialize DKS
2  dksbase.setAPI(api_name);
3  dksbase.setDevice(device_name);
4  dksbase.initDevice();
5
6  //allocate memory for particle array and parameters
7  par_ptr = dksbase.allocateMemory<double>(numpar, ierr);
8  mem_ptr = dksbase.allocateMemory<PART_DKS>((int)size, ierr);
9
10 //init random number generator for each thread
11 dksbase.callInitRandoms(size);
12
13 //get all material properties and transfer parameter array to GPU
14 dksbase.writeDataAsync<double>(par_ptr, params, numpar);
```

Code example 4.13: Initializing DKS for particle matter interaction in OPAL.

To offload the actual calculations of particle matter interaction on the co-processor OPAL uses DKS interface to invoke the functions described in sections 4.5.2 and 4.5.3. Before the calculations begin the particles entering the material are removed from *bunch* and transferred to the GPU using DKS communication interface. After the particle transfer the kernels are invoked and when necessary data is transferred back to the CPU side. This process is illustrated in the pseudo code in 4.14.

If OPAL is running without DKS the version reverts to the original CPU implementation described in section 4.4.4.

```
//remove particles that are going in material from bunch to dksParts_m
copyFromBunchDKS(bunch);

//write particles to GPU if any are going to material
dksbase.writeDataAsync<PART_DKS>(mem_ptr, &dksParts_m[0],
                                 dksParts_m.size(), -1, numparticles);

//execute CollimatorPhysics kernel and sort on GPU
if (numparticles > 0) {
  dksbase.callCollimatorPhysics(mem_ptr, par_ptr, numparticles);
  dksbase.callCollimatorPhysicsSort(mem_ptr, numparticles, numaddback);
}

//read particles from GPU if any are comming out of material
if (numaddback > 0) {
  dksbase.readData<PART_DKS>(mem_ptr, &dksParts_m[0], numaddback,
                             numparticles - numaddback);

  //add particles back to the bunch or delete
  for (unsigned int i = 0; i < dksParts_m.size(); ++i) {
    if (dksParts_m[i].label == -2)
      addBackToBunchDKS(bunch, i);
    else
      stoppedPartStat_m++;
  }
}
```

Code example 4.14: Integrating DKS in OPAL for particle matter interaction simulations.

4.6 Performance Results

To test the OPAL and DKS Monte Carlo simulations, an example was run where particles are shot through a $L = 1$ cm thick graphite slab. This mimics a degrader device used in the proton therapy. Timings were obtained for the integration of the equation of motion, before and after the material, as well as when the particles are moving through the material. The system setup used during the degrader benchmark tests is similar to the setup used for the FFT Based Particle-Mesh (PM) Solver tests, but the Nvidia Tesla K20c GPU is replaced with the Intel Xeon Phi 5110p co-processor. The CPU and GPU benchmarks are done using 1 and 8 CPU cores, in the cases where GPU is used all 8 cores share a single GPU. The benchmarks for Intel MIC were done using a single CPU core together with a Intel MIC co-processor, the setup with multiple CPU cores sharing a single Intel MIC card was abandoned during the work, because of the poor performance. The Intel MIC benchmarks also stop after 10^6 particles because of memory limitations.

Table 4.2 shows the benchmark results for the particle matter interaction and

the integration using various number of particles. The speedup of the particle transport through the material is ×64 to ×201 compared to the one core of the host processor, while the integration is able to achieve a speedup of around ×8 to ×13 on the GPU. The Intel MIC on the other hand shows a speedup of ×41 for the degrader and ×5 for the integration compared to the host.

Table 4.2: OPAL degrader results

Particles	Cores	DKS	Degrader time (s)	Degrader speedup	Integration time (s)	Integration speedup
10^5	1	no	15.84		3.45	
	8	no	**3.95**	**x4**	**0.53**	**x6.6**
	1	K40	**0.18**	**x86**	**0.26**	**x13**
	8	K40	**0.24**	**x64**	**0.41**	**x8**
	1	MIC	**1.02**	**x15**	**1.37**	**x2.5**
10^6	1	no	163.22		35.20	
	8	no	**42.18**	**x3.8**	**4.76**	**x7**
	1	K40	**1.26**	**x128**	**2.71**	**x13**
	8	K40	**0.96**	**x169**	**2.67**	**x13**
	1	MIC	**3.92**	**x41**	**7.05**	**x5**
10^7	1	no	1757.31		352.54	
	8	no	**454.01**	**x3.8**	**46.66**	**x7.5**
	1	K40	**14.94**	**x117**	**26.65**	**x13**
	8	K40	**8.74**	**x201**	**26.05**	**x13.5**

The limiting factor for GPU/MIC performance for the integration is the data movement. This operation requires data to be sent to the device and received from the device, at every time step. In the GPU case kernel execution can be completely overlapped with the data movement, and thus only limited by the memory bandwidth of our device.

The limiting factor for particle movement through the material is global memory access times. Each execution of the kernel requires a load of position and momentum vectors as well as the state of the random number generator for the thread. When the kernel finishes position, momentum, random number state and possibly particle state needs to be written back to global memory. If there are any dead particles, or particles that are coming out of the material, these particles need to be removed from the bunch on the accelerator. This requires a sorting of the particles which also requires a lot of memory movement on the device and

limits the performance.

Chapter 5

Musrfit

5.1 Musrfit and DKS

MINUIT2 is a C++ library allowing a multi-parameter minimization of a user-defined function [43]. It is a re-implementation of the FORTRAN library MINUIT [44], a very popular minimization package used by high energy physicists. In addition to minimization algorithms, it contains methods for analyzing the solutions and can estimate the parameter error correlation matrix. These combined capabilities are very difficult to find in other existing minimisers. Its drawback for inexperienced users is that the user-defined function needs to be implemented, compiled, and linked. This is a common practice in high energy physics, but is less common in the solid state physics community. Therefore, for the μSR community, the MUSRFIT framework [45] has been developed. This framework eases the analysis of muon spin rotation, relaxation, and resonance (μSR) experiments by allowing the user to define all the relevant input parameters and functions for MINUIT2 in a scripting manner. We will describe the problem for the specific needs of μSR, however the problem and the described solution is much more generic.

5.2 Problem description

The schematic of a time differential μSR experiment is shown in Figure 5.1. During an experiment, $\sim$ 100% polarized positive muons (μ^+) are implanted in a solid sample where they rapidly thermalise ($\sim$ 10 ps) without noticeable polarization loss.

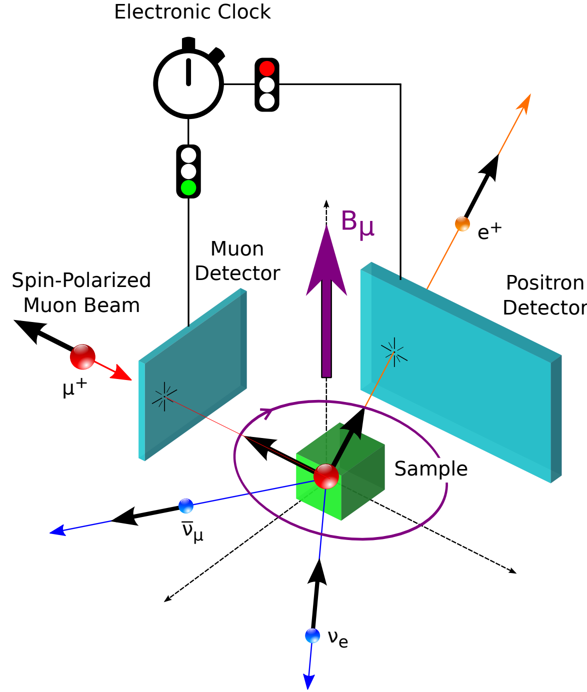

Figure 5.1: Schematic of a time differential μSR experiment

After the implantation the spin evolution of the muon ensemble is measured as a function of time. The evolution can be monitored by using the fact that the parity violating muon decay is highly anisotropic. During the decay an easily detectable positron is emitted preferentially along the direction of the μ^+ spin. The time differential μSR spectrum takes the form:

$$N^j(t, \vec{P}) = N_0^j e^{-t/\tau_\mu}[1 + A^j(\vec{p}^j, t)] + N_{\text{bkg}}^j, \tag{5.1}$$

where the time is measured in discrete steps $t = n \cdot \Delta t$ [$n \in \mathbb{N}_0$, Δt the time resolution] and j indexes the positron detectors. The "physics" of the system under consideration is described by the function $A^j(\vec{p}, t)$. More details about the

function $A^j(\vec{p}, t)$ can be found in Ref. [46]. The muon lifetime is given by τ_μ and N_0 gives the scale of the positron count. Lastly the constant N^j_{bkg} originates from uncorrelated background events. For a given positron histogram, j, the optimal parameter set

$$\vec{P}^j = \{N^j_0, N^j_{\text{bkg}}, \vec{p}^j\} \tag{5.2}$$

needs to be determined, where $\vec{p}^j$ describes the parameters for each positron histogram. Depending on the level of statistics of the positron histograms, the parameter set, $\vec{P}$, is determined by minimizing the χ^2 function:

$$\chi^2(\vec{P}) = \sum_j \sum_n \frac{[d^j_n - N^j(t, \vec{P}^j)]^2}{(d^j_{n,\text{err}})^2}, \tag{5.3}$$

where d^j_n are the measured data points of the jth positron detector. The theory describing the data is given by Eq. (5.1), and $d^j_{n,\text{err}}$ is the estimated error of d^j_n ($d^j_{n,\text{err}} = \sqrt{d^j_n}$ for the Poisson distributed positron events).

For data sets with rather limited statistics, Eq. (5.3) is not leading to satisfactory results. In this case the max log-likelihood (MLH) function

$$\mathcal{L}(\vec{P}) = 2 \cdot \sum_j \sum_n \begin{cases} \left[N^j(n \cdot \Delta t, \vec{P}^j) - d^j_n\right] + d^j_n \log\left[\dfrac{d^j_n}{N^j(n \cdot \Delta t, \vec{P}^j)}\right], & d^j_n > 0 \\ \left[N^j(n \cdot \Delta t, \vec{P}^j) - d^j_n\right] & d^j_n \leq 0 \end{cases} \tag{5.4}$$

should be maximized, which leads to a much better estimate of $\vec{P}$.

With the improvements in detector technologies, it is possible to achieve higher time resolution (smaller Δt) during the experiments. This is leading to increasing sizes of data sets that need to be analyzed, and the associated minimization/-maximization times are increasing drastically.

To perform the parameter fitting, Musrfit uses the Minuit2 library. Musrfit contains the implementations of Eqs. (5.3) and (5.4) while the minimization/maximization process is executed by Minuit2. The main, and most time consuming, part of the parameter fitting is the calculations embedded in Eqs. (5.3) and (5.4) respectively. Offloading these calculations to the GPU could lead

to a significant improvement in the total time needed to perform a parameter fit which would allow to perform real time data analysis. In the following discussion we will use χ^2 synonymous for $\mathcal{L}(\vec{P})$ fits.

What does 'real time' data analysis in the context of μSR mean and why is it important? Muon Spin Rotation/Relaxation/Resonance (μSR) is a spectroscopic, accelerator based technique where measurement slots are awarded through a highly competitive proposal system. In the best case a researcher is granted a beam time slot twice a year. During these short beam periods (typically 2-4 days), all the necessary measurements need to be performed. The material classes studied by μSR are often showing a very reach and complex physics and hence it is initially hard to judge what will be the best measuring strategy in terms of available external parameters, like temperature, field, pressure etc. The online modeling of the data is crucial to conclude on an optimal measurement program. However, for some μSR instruments, currently the parameter fitting time which is needed in this modeling process is comparable to the actual measurement time. This makes it very hard to come to a clever and decisive decision how to use the beam time. Wrong decisions will force researcher to re-apply for beam time which is a waste of resources. Therefore it is crucial to reduce the fitting time to a level allowing to come to the right conclusions during online analysis. To exemplify the above stated, the numbers for the HAL-9500 instrument at the Paul Scherrer Institut can help. A typical measurement time for a given field and temperature is 2-4 hours. Robust fitting results are only available after about half of the measurement time. A single fit with the current version of MUSRFIT which utilizes OpenMP takes about 15 min. During the first day of data taking various fitting models need to be applied and refined. This means that the full online analysis takes longer than the measurement. This is drastically improved by the DKS solution which brings the fitting times down to about 20 sec allowing to find the appropriate fitting model needed to guide the experiment successfully.

5.3 χ^2 and MLH kernels on GPU

To ease the process of adding GPU support to MUSRFIT, the Dynamic Kernel Scheduler (DKS) [24] was used. All the device specific code is developed in DKS

and Musrfit only receives a simple interface that it can use to invoke the task execution on the GPU. DKS uses CUDA or OpenCL to create the GPU code. CUDA is used to target Nvidia GPUs while the OpenCL implementation is used to target devices from other vendors (Intel, AMD).

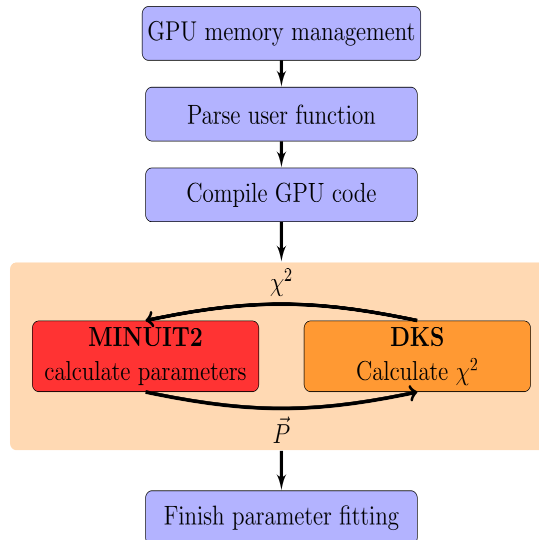

Figure 5.2: Flow diagram of parameter fitting with Musrfit using Minuit2 and DKS

Using DKS, Musrfit allocates memory on the GPU for every data set used in the fitting and transfers the data to the device. Since the data sets do not change during the fitting, this operation can be performed only once.

One of the most important features of Musrfit is the ability for users to define the theory function using an input file. A mechanism needs to be created where this user defined function can be passed to the GPU at run-time and used in the kernel code. To handle this problem, run-time compilation was used. The user defined function is parsed by Musrfit from the input file and passed to DKS where a CUDA or OpenCL device function is created to be used in the GPU kernels. This process is described in more detail in section 5.4. When the new GPU program is created and compiled by DKS, Musrfit begins the process of minimizing the χ^2 value by invoking the CUDA or OpenCL kernels to calculate the χ^2 value and using Minuit2 to fit the parameter set. The sequence diagram

of this process is shown in Figure 5.2.

The most time consuming part of the parameter fitting are the calculation of the χ^2 function for each data set. This calculation can be easily parallelized and therefore is an ideal candidate to offload to the GPU. The CUDA kernel to compute the χ^2 value creates a thread for each data point in a data set. Shared memory is used to store parameter, function, and map (see section 5.4) values since these values are accessed multiple times by each thread. Using the new parameters, functions, and maps for the data set, the theory function is evaluated at each point and the χ^2 value at that point is calculated and stored in a temporary allocated global memory array. After the kernel completes the calculation of χ^2 for each individual data point, all these values are summed up using CUBLAS [47] to get the χ^2 value of the whole data set. This process is repeated for every data set used in the calculation.

5.4 User defined kernels

To allow users to define functions, the GPU code must be created at run-time. For OpenCL this is the standard execution method, while for the CUDA framework the CUDA run-time compilation library [48] was used. MUSRFIT parses the user input file to get the user defined function and creates a string with a C++ mathematical expression. This expression can use standard C++ mathematical operators and functions, and in addition it is able to utilize a set of predefined functions which are commonly used in the μSR field. CUDA implementations of exponential and Gaussian distribution functions are shown below in the code sample 5.1. A full list of available predefined functions is listed in MUSRFIT user guide [45].

The mathematical expression can use the parameter array to access parameter values and the function array to access precomputed function values. The function array is a convenience feature for the user. A subset of the parameter array is data set specific. In order to keep the mathematical expression compact, an indirect addressing of these parameters is needed. This is accomplished with the map array. For more details see [45, 49]. An example of a created user function for use in CUDA kernels is shown in code sample 5.2.

```
__device__
double se(double t, double lambda) {
  return exp( -lambda*t );
}

__device__
double ge(double t, double lambda, double beta) {
  return exp( -pow(lambda*t, beta) );
}

__device__
double sg(double t, double sigma) {
  return exp( -0.5 * pow(sigma*t, 2) );
}

__device__
double stg(double t, double sigma) {
  double sigmatsq = pow(sigma*t,2);
  return (1/3) + (2/3)*(1 - sigmatsq) * exp(-0.5 * sigmatsq);
}
```

Code example 5.1: CUDA examples of predefined functions that can be used to create the user function.

```
__device__
double fTheory(double t, double *p, double *f, int *m)
{
  return p[m[0]] * sg(t,p[m[1]]) * tf(t,p[m[2]],f[m[3]]);
}
```

Code example 5.2: Example of parsed user defined function ready for compilation.

After MUSRFIT has created the string containing the mathematical expression of the user defined function, it is added to the string containing the CUDA program. The CUDA program consists of a user defined function definition, predefined functions, and the kernel for χ^2 calculation. This newly created program is compiled at run-time and used by DKS to evaluate the χ^2 of a given data set.

5.5 Musrfit speedups with GPUs

The parameter fitting tests were run on two systems. The first system was equipped with two Intel(R) Xeon(R) CPU E5-2609 v2 processors and one Nvidia Tesla K40c GPU. The second system was equipped with two Intel(R) Xeon(R) CPU E5620 processors and AMD Radeon R9 390x. MUSRFIT parallelizes CPU code using OpenMP so the performance of the fitting using this implementation,

with 8 threads, was chosen as the baseline. To test the CUDA and OpenCL performance, the same example was run on the GPU using both of these frameworks. Another benchmark was run with OpenCL using the CPU as the target device on the first machine.

For the tests, a typical muon polarization function was chosen to determine the magnetic shift of a para-/diamagnetic material [46]. It is given by:

$$A^j(\vec{p}, t) = A_0^j \exp\left[-\frac{1}{2}(\sigma t)^2\right] \cos(\gamma_\mu B t + \phi^j), \tag{5.5}$$

where $j = 1$ to 16, where 16 is the number of positron detectors in this example. A_0^j is the asymmetry of each positron detector, σ is the depolarization rate of the muon spin ensemble, γ_μ is the gyromagnetic ratio of the muon, B is the magnetic induction at the muon stopping site, t is the time, and ϕ^j is the phase of the initial muon spin in respect to the positron detector.

Table 5.1: Parameter fitting with χ^2 function running on the GPU. The given time is for the execution of the `minimize` command of Minuit2 [43].

		χ^2			**MLH**		
Data size	**Device**	**Iter.**	**Time(s)**	**Speedup**	**Iter**	**Time(s)**	**Speedup**
16x142200	Intel	9319	4029.4		9028	5294.5	
	OpenMP Intel		**513.672**	**x8**		**673.718**	**x8**
	OpenCL Intel		**439.864**	**x9**		**685.638**	**x8**
	CUDA Nvidia		**18.262**	**x221**		**18.0678**	**x293**
	OpenCL Nvidia		**22.7205**	**x177**		**23.2791**	**x227**
	OpenCL AMD		**45.8842**	**x88**		**51.7018**	**x102**
16x213300	Intel	8052	5237.12		8318	7294.39	
	OpenMP Intel		**658.485**	**x8**		**927.567**	**x8**
	OpenCL Intel		**729.055**	**x7**		**927.046**	**x8**
	CUDA Nvidia		**20.0328**	**x261**		**21.6166**	**x337**
	OpenCL Nvidia		**24.6488**	**x212**		**26.9296**	**x271**
	OpenCL AMD		**42.992**	**x122**		**54.3796**	**x134**
16x426601	Intel	6313	8128.89		8439	14238.2	
	OpenMP Intel		**1028.31**	**x8**		**1877.73**	**x8**
	OpenCL Intel		**885.274**	**x9**		**1242.16**	**x11**
	CUDA Nvidia		**25.6257**	**x317**		**35.9603**	**x396**
	OpenCL Nvidia		**29.5987**	**x275**		**41.92**	**x340**
	OpenCL AMD		**43.6503**	**x186**		**68.5996**	**x208**

The results of these tests are shown in the Table 5.1. The results show that for the chosen test function, the total execution time of the parameter fitting

can be improved by around $\times30...\times40$ on the GPU, compared to multi CPU performance, depending on the size of the problem. Such a performance allows one to analyze the experimental data in real time.

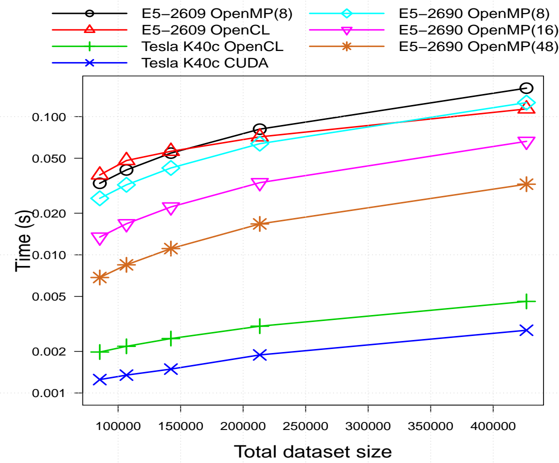

Figure 5.3: Parameter fitting with χ^2 function running on the GPU. The time is shown for the execution of one iteration of the `minimize` command of Minuit2 [43].

The OpenCL implementation of parameter fitting in DKS allows the use of other accelerator devices to speed up the calculations. This makes the application more portable and more accessible to users. The results of OpenCL tests are shown in Figure 5.3. This figure also shows OpenMP results when using up to 48 CPU cores to run the fitting on the second test system equipped with two Intel(R) Xeon(R) CPU E5-2690 v3 processors each consisting of 24 virtual cores with hyper-threading enabled.

Chapter 6

PET Image reconstruction and analysis

6.1 PET Image reconstruction basics

The SAFIR (Small Animal Fast Insert for MRI) project is developing a fast PET insert (positron emission tomography scanner that can be integrated in MRI system) for a pre-clinical MRI (Magnetic Resonance Imaging) system for dynamic in vivo PET-MRI studies with excellent temporal resolution. This requires tomographic image reconstruction followed by image data analysis adapted to the conducted study. While, under idealized assumptions, the image can be obtained analytically by a filtered inverse Fourier transform. Modeling the system details and irregularities results in better images. However, this second approach involves the manipulation of huge matrices. Therefore typically iterative image reconstruction algorithms are applied, which still constitute a significant computational burden.

Image analysis, such as feature finding, is computationally time intensive as well. Moreover, in dynamic studies sequences of dozens of images need to be reconstructed and analyzed. In particular the aim is to reconstruct one image about every 5 seconds. Thus 60 images need to be reconstructed for a typical acquisition of 5 minutes, each comprising 5 seconds worth of data. Computation times of hours to days would be required for data acquired in a few minutes. Speeding up

the computation is therefore of prime importance. A first big improvement would be the reconstruction of the image series on a time-scale of one hour. Ultimately, a quasi-online visualization of the process dynamics would be very beneficial to control and optimize the experiments requiring to reconstruct one image within 5 seconds.

There are several articles in the literature that describe the efforts of accelerating PET image reconstruction codes using GPUs [50, 51, 52]. The algorithm described in this work uses list-mode data for image reconstruction and follows a similar approach as proposed in [50] and [52]. The results obtained in previous works show that PET image reconstruction is a good algorithm for GPU acceleration and would greatly benefit the reconstruction algorithms used in SAFIR project.

6.1.1 Image reconstruction

In PET image reconstruction, the goal is to find the source activity distribution in the object to be studied. The activity distribution is found from the projection measurements of the set of coincident detector pairs, which form the whole PET scanner. In PET imaging, a positron emitting radiotracer is used. The positron annihilates producing two back-to-back photons with 511 keV energy. These are measured, typically with a set of cylindrically arranged detectors surrounding the source. A schematic sketch of one ring of such an arrangement is shown in figure 6.1.

If within a short time window, two detectors each register a photon, it indicates that an annihilation event has occurred on the line joining the two detectors (line-of-response, LOR). The list of all these coincidence events (listmode data) can directly be used to reconstruct the image. A general description of PET image reconstruction can be found in [53].

Consider a discrete tracer distribution $\vec{f} = (f_1, \ldots, f_J)^T$ on a 3D Cartesian grid, where the index labels the volume elements called voxels and J is the number of voxels. The coincidence measurement yields an estimation of the mean counts $\vec{\bar{y}} = (\bar{y}_1, \ldots, \bar{y}_I)^T$ per detector pair i, where I is the number of possible pairs.

Using a linear model for $\bar{y}_i$, the relationship between tracer distribution and

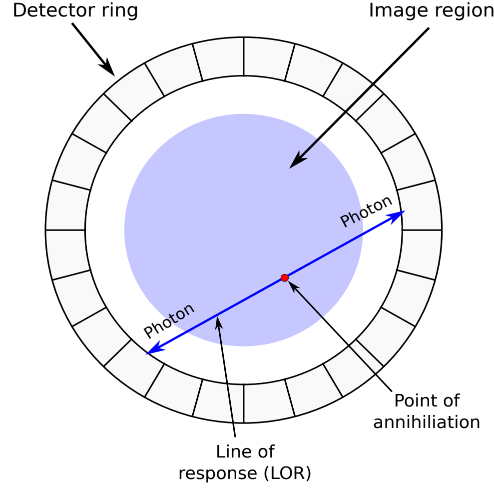

Figure 6.1: PET imaging basic principles.

mean measured counts can be written as

$$\bar{y}_i = \sum_{j=1}^{J} a_{ij} f_j + n_i, \tag{6.1}$$

where the matrix **a** is called the system matrix and $\vec{n}$ is a noise term. PET reconstruction aims at solving the inverse problem, i.e. finding $\vec{f}$ given $\vec{\bar{y}}$. Inverting the matrix **a** is computationally unaffordable due to its size. Therefore, iterative approaches are employed.

The aim of the iterative image reconstruction algorithm is to find $\vec{f}$ such that equation 6.1 produces $\vec{\bar{y}}$ which matches the measured coincidence events $\vec{m}$. Most iterative algorithms aim to minimize the distance measure between $\vec{\bar{y}}$ and the measured data $\vec{m}$ [53].

The stochastic nature of the radioactive decay, together with the detection of the events, can be modeled with a Poisson process. The likelihood function is

$$p(\vec{\bar{y}}|\vec{f}) = \prod_{i=1}^{I} p(m_i|\bar{y}_i) = \prod_{i=1}^{I} e^{-\bar{y}_i} \frac{\bar{y}_i^{m_i}}{m_i!}, \tag{6.2}$$

where m_i is the actually measured number of counts in the i-th line-of-response (LOR).

The Bayes factors for the inversion of the conditional probability are neglected.

Finding the minimum of the likelihood function with respect to $\vec{f}$ yields an iterative formula for the distribution

$$\begin{aligned} f_j^{k+1} &= \frac{f_j^k}{\sum_{i=1}^{I} a_{ij}} \sum_{i=1}^{I} a_{ij} \frac{m_i}{\bar{y}_i^k}, \\ \bar{y}_i^k &= \sum_{j=1}^{J} a_{ij} f_j^k + n_i. \end{aligned} \tag{6.3}$$

This algorithm can be rewritten for listmode processing

$$\begin{aligned} f_j^{k+1} = \frac{f_j^k}{\sum_{i=1}^{I} a_{ij}} \sum_{l=1}^{L} \sum_{i=1}^{I} \delta_{i,c(l)} \frac{a_{ij}}{\bar{y}_i^k} = \frac{f_j^k}{\sum_{i=1}^{I} a_{ij}} \sum_{l=1}^{L} \frac{a_{c(l),j}}{\bar{y}_{c(l)}^k}, \\ c(l) = \text{index of detector paircorresponding to } l\text{-th listmode event} \end{aligned} \tag{6.4}$$

where L is the number of listmode events and δ_{ij} is the Kronecker delta. The matrix element $a_{c(l),j}$ describes the probability of detecting an annihilation from the j-th voxel in the $c(l)$-th LOR, in which the l-th coincidence event was detected.

Equation (6.3) implicitly reappears in the denominator of the last term of (6.4) and is called forward projection because it constitutes a map from the spatial activity distribution to the number of counts in the LORs. The sum over l in (6.4) is called backward projection because it constitutes a map from the set of LOR count values to the spatial activity distribution.

The matrix elements a_{ij} can be estimated using an adapted raytracing algorithm. To reduce the computation effort, the LOR's predominant direction of propagation is determined to be along the x- or y-axis and the planes perpendicular to that axis, through the voxel centers, are considered. For each plane, the intersection point $\vec{p} = (p_x, p_y, p_z)^T$ of the LOR i with that plane is determined and the voxel j with center coordinate $(v_{jx}, v_{jy}, v_{jz})^T$, containing this intersection point is identified.

Without loss of generality, let the predominant direction be along the x-axis. In each plane, the matrix element is calculated for the voxel j and its three neighbors j', j'', j''' in the positive y- and z-directions. The matrix element a_{ij} is approximated to be related to the distance of the voxel to the intersection point

in the following way

$$a_{ij} \approx m_d - \sqrt{(p_y - v_{jy})^2 + (p_z - v_{jz})^2}, \tag{6.5}$$

where m_d is the matrix distance factor. The matrix distance factor acts as a weight for the influence of the distance of the voxel to the intersection point to the system matrix element.

6.1.2 Image analysis

An important task for the envisaged research is to identify a small spot, with a volume of about 5 to 10 mm^3, in a rodent brain with enhanced activity. This spot needs to be identified in non-uniform background and with the enhanced activity of the order of 20% compared to its normal state. The stochastic nature of the data and the relatively low number of counts in the few second time intervals results in large variance of the reconstructed activity concentration. It is therefore important to distinguish true features from fluctuations with quantifiable significance.

As explained above, the goal is to find the relative activity increase in a region over some normal (background) activity. Hence the excess E and its standard deviation ΔE are defined

$$\begin{aligned} E &= \frac{S - B}{B}, \\ \Delta E &= \frac{S}{B}\sqrt{\left(\frac{1}{S} + \frac{1}{B}\right)} \end{aligned} \tag{6.6}$$

where B is the background activity and S is the activity in the region of interest including the background. For simplicity, two concentric spheres are used, the smaller (inner) one representing the signal region and the larger, with the volume of the smaller sphere cut out, representing the background region. The significance of the excess can be expressed in terms of its standard deviations and a threshold can be used to separate true from random signals.

The image is processed by displacing the center of the sphere into the center of each voxel. Applying a threshold to the transformed image allows to locate

features of a certain significance.

6.2 GPU kernels for PET image reconstruction and analysis

For image reconstruction, the most time consuming parts of the algorithm are the forward and backward projections. Both of the projections loop through the projection lines and either accumulate image data along the line (forward projection) or distribute projection values into the image data along the same lane (backward projection). Every line in the list can be processed independently so this problem can be parallelized to take advantage of the computational resources available on GPU. However, there are several challenges that must be considered for the GPU algorithm to achieve the desired performance:

- Some lines in the list do not require processing and different predominant line directions require alternative processing which results in a large thread divergence;
- Each line requires a different set of voxels from the image resulting in random memory access;
- Multiple lines need to update the same voxel in the image and thus requires atomic operations.

For image analysis, the most time consuming part is the calculation of the average value and the standard deviation of the voxels inside a sphere. Two spheres are placed at the source location. First the average value and standard deviation of voxels that are inside the smaller of the two spheres (source value) are calculated. The second part of calculations finds the same values for voxels that are outside of the smaller sphere, but inside the larger sphere (background value).

Since the spheres are placed at the center of each voxel, this can be parallelized on the GPU by every thread calculating the average value for a different sphere.

To add GPU support to host application CUDA kernels for forward projection, backward projection, source calculation and background calculations were

implemented in DKS. The host application was updated to use DKS instead of CPU implementation when a GPU is available. The host application uses DKS interface to invoke memory management, data transfer and kernel calls while all the temporary memory, kernel details and kernel launch parameters are handled by DKS.

6.2.1 Forward and backward projections

The forward projection in DKS is implemented as two kernel calls. The first kernel call calculates the predominant direction of each line, and assigns a label to it:

- 0 - line does not need to be processed
- 1 - predominant direction in the x plane
- 2 - predominant direction in the y plane

Once the predominant direction of the line is known, the Thrust sort by key function is used, sorting the lines according to its direction. In this way we minimize the thread divergence in the kernel call that performs the forward projection.

After the lines are sorted, the forward projection processes the image in slices along the predominant direction as shown in Figure 6.2. Whether the line requires any values from a slice is determined by the position and angle of the line. This process is the same for both forward and backwards projections. For each slice the position where the line crosses the slice is determined, if this position is inside the image region the values are loaded from global memory (forward projection) or global memory is updated with the correction value (backward projection).

The assigned label allows us to process all the lines with a single kernel call and avoid repeated calculations on how the line should be processed. After the lines are sorted, they are grouped by the predominant direction. The divergence of threads within the warp will be very minimal.

Since the lines require very few values from a slice and not all the lines use every slice, each thread checks if any voxels from the slice are needed and only then perform a load from global memory. This will result in an un-coalesced global memory access since lines in the same warp are accessing random voxels in the

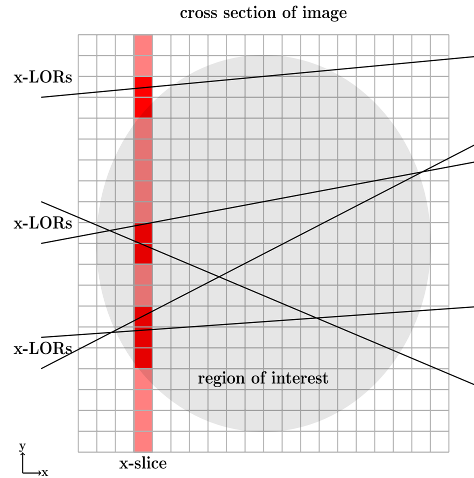

Figure 6.2: Cross section of the image showing LORs, with predominant direction in the x plane, and a slice of the image along this direction being processed.

image. Since only a few of the slices are used for each line, and only a few values from each slice are used, this results in a better performance than loading all of the voxels in the slice in shared memory.

Backward projection takes the correction value calculated for each line and distributes it back to the voxels along this line. During backward projection, the same sorted list of lines is used to tackle thread divergence. The main bottleneck for the backward projection is the need for multiple threads to update the same voxels because different lines can cross the same positions in the image. To avoid race conditions when multiple threads need to write to the same global memory address CUDAs atomic operations are used.

DKS calls are inserted in the host application to offload tasks if a GPU device is present. Using the DKS, the host application allocates memory on the device and transfers data from the host, holding voxel positions, voxel values, detector pair list and detector positions. In addition memory is allocated on the GPU to hold the correction values for each detector pair calculated by forward projection and corrected voxel values calculated by background projection. After memory allocation and data transfer the host application loops trough the set reconstruction iterations and uses DKS to call forward and backward projection kernels on the GPU. Every iteration requires a read of corrected voxel values from the

GPU, and since the final processing of the image is done by the host application, before the next iteration new voxel values are written back to the GPU. Each iteration also requires a write of list of detector pairs used for reconstruction. Since after every iteration half of the detector pairs are discarded this list needs to be updated and resorted before every forward projection. The example code of host application and DKS integration for image reconstruction is shown in the code sample 6.1.

```
1  for (int iter = 0; iter<num_of_iteration; iter++)
2  {
3    //transfer image data to GPU every time step
4    dksbase.writeData<float>(*image_gpu, *recon_image_host, image_size);
5
6    //calc forward projections on the GPU
7    dksbase.callForwardProjection(*line_correction, *image_gpu,
         *list_detectors, *detector_position, *image_position, event_number);
8
9    //calc backward projections on the GPU
10   dksbase.callBackwardProjection(*line_correction, *image_correction,
         *list_detectors, *detector_position, *image_position, event_number,
         image_size);
11
12   //read recon_image_3d_corrector form GPU
13   dksbase.readData<float>(*image_correction, *recon_image_host, image_size);
14
15   //final processing of the reconstructed image
16   //output operations
17 }
```

Code example 6.1: Example code of DKS interface integrated in the host application for image reconstruction.

6.2.2 Source and background calculation

Source and background calculations are separated in two kernels. To calculate the source values at each voxel position, every thread places a sphere with the center at this voxel. Then knowing the diameter of the sphere, the position of the voxel, and the size of each voxel, a box is calculated that contains this sphere. This process is illustrated in Figure 6.3.

After the box is formed the thread loops through the voxels in this box and calculates the average value and standard deviation using only voxels that lie inside the sphere. The box is necessary to minimize the number of voxels that each thread has to process. This approach requires a lot of global memory access.

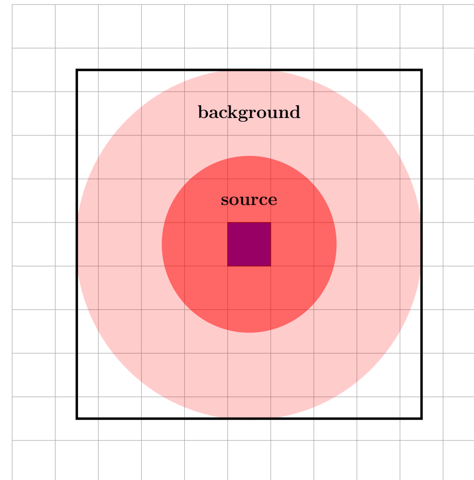

Figure 6.3: 2D representation of sphere placement at the voxel position for source and background calculation.

Shared memory usage could be explored to improve the performance of the kernel, since voxels used by threads in the same thread block are overlapping.

To calculate source and background values on the GPU, the host application first needs to allocate memory on the device and transfer data for voxel positions and voxel values in the image. Then memory is allocated to hold temporary values for average values and standard deviation at each sphere. When all the necessary data is transferred to the device, kernels to calculate source and background values are invoked trough DKS. In the final stage of the algorithm the host application reads the average and standard deviation values for each sphere from the GPU and performs final signal to noise ratio calculations. As in the case of forward and backward transformation the host application is responsible when memory allocation and data transfer is scheduled, in order to ensure that the host application can access the data from GPU when needed, but DKS handles all the device code details and kernel launch parameters.

6.3 Results

The tests for image reconstruction and analysis where performed on two different systems. The first system uses Intel(R) Xeon(R) CPU E5-2609 v2 processor and

Nvidia Tesla K40c GPU, while the second system uses Intel(R) Xeon(R) CPU E5-2690 v3 CPU. The CPU implementation of reconstruction and analysis is not parallelized, so the single CPU core performance was chosen as the baseline performance.

To test the GPU performance, simulated data were used. The data were generated using GEANT4 [54, 55] for an idealized scanner made from 91 rings of 180 detectors. The detector crystals are 2.0 mm x 2.0 mm and are 12.0 mm long in the radial direction. The pitch between adjacent detectors in a ring, as well as between the rings, is 2.2 mm. Simulations of a Derenzo [56] type phantom were performed: six groups of spheres with different diameters (1.0 mm, 1.2 mm, 1.6 mm, 2.4 mm, 3.2 mm, and 4.0 mm) were embedded into a rat phantom. The rat phantom was implemented as a high density polyethylene cylinder, with a length of 150 mm and a diameter of 50 mm. The simulation was performed for one second with 500 MBq distributed evenly over the spheres volume. This corresponds to 1.42 MBq/mm^3 and zero activity in the rat phantom. Reconstruction and image analysis were performed using the algorithms as described above.

The reconstructed image size was 90x90x50 voxels with a voxel size of 0.7 mm x 0.7 mm x 0.7 mm, and the reconstruction was performed using $13,901,607$ coincidence events. The reconstruction uses all of the available coincidence events and performs forward and backward projections for 15 iterations.

Table 6.1: Performance of image reconstruction and analysis example.

Device	Recon	Speedup	Analysis	Speedup
E5-2609 v2	800s		8.8s	
E5-2690 v3	599s		5.9s	
Nvidia Tesla K40c	14s	×**57**	2.7s	×**3**

When performing image analysis, the example performs two separate types of analysis. The first analysis places the spheres at previously defined source positions and the second analysis places the spheres at every voxel that lies inside the image region.

The results of the reconstruction for 1s of data and the analysis are shown in Table 6.1. As can be seen from the results, the implemented GPU version cuts the execution time from almost half an hour to around 30 seconds. The time

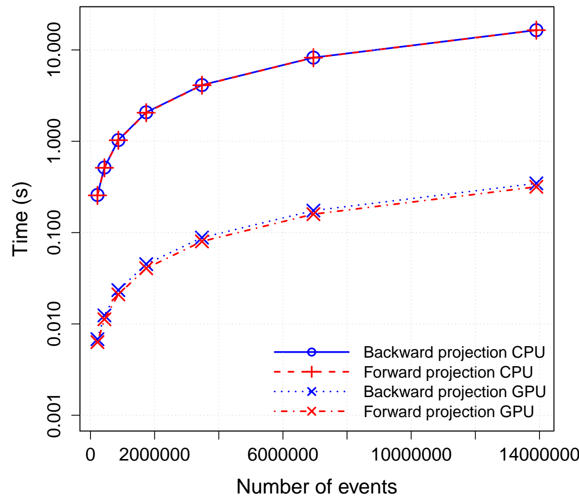

Figure 6.4: Execution time for forward and backward projections. Run on Intel(R) Xeon(R) CPU E5-2690 v3 and Nvidia Tesla K40c

represented in the table shows the total execution time of the reconstruction algorithm, including the input and the output operations. For image analysis input and output operations are excluded from the benchmarking, because they take more that 50% of total analysis time. For the image analysis, the diameter of inner and outer spheres are chosen to be 2mm and 4mm. The performance of individual kernels, for image reconstruction, offloaded to GPU are shown in Figure 6.4. The results in Figure 6.4 illustrate the execution time for forward and backward projections using different numbers of lines for image reconstruction on a CPU and a GPU.

To test the GPU performance and scaling of kernels used for image analysis, tests were repeated with different sphere sizes. Figure 6.5 shows the execution time of calculating source and background values at each voxel position with different sphere diameters. The diameter of the outer sphere is always twice the diameter of the inner sphere.

The computation time for the image reconstruction depends on the total number of detected events and on the number of iterations, but is independent of the true image and the physical scanner used to obtain the data. The computation

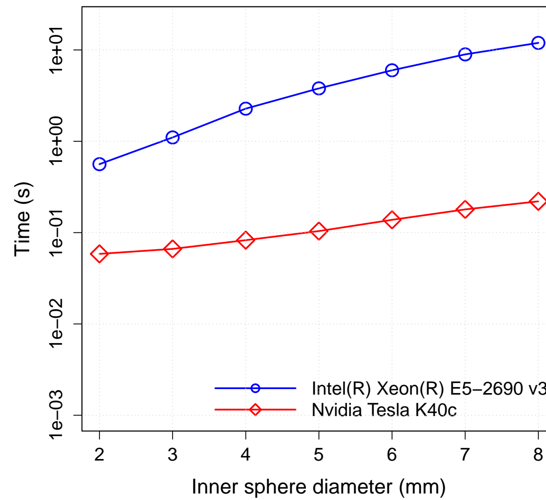

Figure 6.5: Calculation of source and background values with different sphere diameters.

time of the image analysis will scale with the number of voxels and the size of spheres, as can be seen in Figure 6.5. However the computed time is independent of the content of the image and of how the image was actually obtained. Therefore, the results of the examples are representative for all other possible PET systems, applying similar reconstruction and feature finding. In the given application using list-mode processing, the reconstruction time scales linear with the amount of data. Thus reconstructing an image of 5 s using the GPU requires a total of 72.7 s, and the reconstruction of a series of 5 minutes of data into 60 images would require 1 h and 12.7 min. This is a significant improvement compared to the ~66 hours required in the CPU case and already close to the first objective to reconstruct within the order of one hour.

Chapter 7

Multi-bunch tracking code - mbtrack

The final chapter of the thesis moves away from DKS and integrating hardware accelerators in large scientific codes, but rather shows the efforts of porting a smaller application to CUDA. This was done in order to allow moving the simulations from multi-core CPU clusters to a heterogeneous CPU and GPU systems. Since the targeted GPUs for mbtrack were Nvidia Tesla K40c, CUDA was chosen as the language to create the device code.

The application that was ported to the GPUs was mbtrack, which is a single and multi bunch tracking code created at SOLEIL (French National Synchrotron Facility). The reason that the previous approach with DKS was not used for mbtrack, was that in this case we are not looking to offload parts of the code to the accelerator, but rather rewrite the application to create mbtrack-cuda version, so that all the heavy calculations are done on the GPU side, while the CPU is only used for setting up the simulation and input/output operations. Another reason for not using the DKS in mbtrack-cuda version was for the simplicity of distributing the mbtrack application. Since mbtrack is a rather small application and does not use any external libraries a stand alone CUDA version would be easier to distribute and use to current mbtrack users. And the last reason for not using DKS in porting mbtrack to the GPUs was that the targeted accelerators were Nvidias GPUs, so only a CUDA implementation of the application was

needed.

The mbtrack application is used at PSI for simulations of bunch instabilities in the SLS-2 (Swiss Light Source upgrade) project. The rest of the chapter will describe in more detail the algorithms implemented in mbtrack, the simulations that will be run using this application and why a CUDA version was needed as well as the process of porting mbtrack to CUDA. Finally, results achieved with the mbtrack-cuda version will be shown.

7.1 MBTRACK

7.1.1 Collective effects in synchrotrons

A synchrotron is ring-shaped machine to accelerate charged particles or store the particles without changing the energy (storage rings). The particles travel in a cyclic vacuum tube (beam pipe) along which magnets and radio-frequency (RF) cavities are placed. The magnets are used to bend the trajectory of the particles and to focus the beam, while RF cavities are used to accelerate the particles and to shape the particle bunch [57].

While circulating in an accelerator beam pipe, particles create and leave behind wake-fields. The created fields influence the trailing particles, which may lead to energy losses and instabilities in the beam [58].

Mbtrack is a multi-bunch tracking code for the study of beam instabilities in synchrotrons. It is developed to simulate single bunch effects such as head-tail, microwave or transverse mode coupling instabilities and bunch lengthening due to short range geometric and resistive wall wakes. Mbtrack can also simulate multi-bunch effects due to high order mode impedance or long range resistive wall. The simulations of effects from harmonic cavities is also possible with mbtrack [59].

7.1.2 Limitations of mbtrack MPI version

At PSI, mbtrack is used for the detailed study of both single-bunch and coupled-bunch instabilities and transient beam-loading in the SLS-2 upgrade proposal. In the case of the SLS-2 base-line proposal with 500MHz main RF cavity and

a passive third harmonic cavity, it is necessary to simulate the interaction of up to 484 bunches with their wake-fields. Simulations of this scale would require a cluster with 485 CPU cores. This scale is too big for the available cluster at PSI and would require access to a supercomputer, for example at the Swiss Computing Center. Fortunately, mbtrack can be adapted to benefit from GPU-acceleration. This enables us to run our simulations on a relatively cheap GPU accelerated desktop computer.

The original mbtrack version (mbtrack-mpi) uses MPI to parallelize the multi-bunch simulations. Each bunch in the simulation is assigned to a different core. This requires N+1 cores to complete the simulation, where N is the number of bunches. This scale of the simulations is too big for the available computing resources at PSI and therefore a different solution is required in order to allow the necessary simulations to be completed.

7.1.3 Parallelization in mbtrack MPI version

Mbtrack is parallelized with one 'master' task responsible for the setup of the simulation, data processing and distribution of the data to 'worker' processes. The worker processes are each responsible for single bunch formed of a large number of particles.

The manager process of mbtrack-mpi is responsible of generating the initial conditions for each bunch and sending this information to the cores that are handling this bunch. After the simulation starts the manager process receives the statistics information after every turn from all the bunches and calculates the average statistics over all the bunches.

The worker process receives the initial bunch conditions from the manager and generates the initial particle distribution. After the tracking starts the selffield and optic transformations are performed for each particle and resistive wall effects are calculated. The detailed descriptions of the particle transformations can be found in [58]. After each turn the bunch statistics are sent to the master process and bunch moments are sent between the worker processes.

The biggest disadvantage of the tracking code is that each of the bunches is handled serially by one core and there is no parallelization over the particles in

the bunch. This means that it is necessary to add more cores to the simulation when the total number of bunches is increasing.

7.2 GPU acceleration of MBTRACK

The mbtrack-cuda version of the multibunch tracking software uses CUDA to offload all the heavy computation to the GPU, while the host is used only for setting up the simulation, updating the statistics and logging the results of the simulation. The mbtrack-cuda version is able to simulate all the same physics effects as mbtrack-mpi version, but instead of requiring N + 1 CPU cores to run the simulation it requires CPU + GPU system. The advantage of this version is that if the CUDA version is able to complete the simulation in reasonable time the CPU+GPU system is more cost effective to set up than a small CPU cluster, which would allow to set up a dedicated system for SLS-2 simulations.

7.2.1 mbtrack CUDA

For the ease of porting the application to CUDA, mbtrack-cuda is built on top of the original mbtrack application and reuses most of the original code. The code to setup of the simulation, store data structures, storing and updating statistics, as well as writing output to the files is kept identical to the original version.

In contrast to the mbtrack-mpi version, the mbtrack-cuda version parallelizes the simulation over the macro particles in the bunch rather then over the number of bunches. Since the number of particles is usually large (over 50,000), this parallelization technique ensures, that, even for simulations where the number of bunches is relatively small, all the GPU resources are still utilized.

7.2.2 GPU memory management

After the simulation is initialized and initial particle distributions are generated, all the data needs to be transferred from the host memory to the GPU memory.

For the simplicity of porting the application to CUDA all the data structures used in the CUDA code are kept the same as the ones used in mbtrack-mpi

version. After the simulation is initialized on the host side all the data structures are copied to the GPU using *cudaMemcpyToSymbol* function. This function copies the variables in the data structure from the host memory to the GPU memory. The arrays that are used in the data structures need to be copied separately using the *cudaMalloc* and *cudaMemcpy* to allocate the needed space and copy the data. The copying of the data structures and the arrays needed in the data structures are shown in the code example 7.1.

```
1 //copy ring to device constant memory
2 cudaMemcpyToSymbol( dring, ring, sizeof(ring_t) );
3
4 //allocate memory for dactive_HC in GPU global memory
5 cudaMalloc((void**) &hactive_HC, sizeof(active_HC_t) * ring->active_HC_size);
6 //copy the data to device
7 cudaMemcpy(hactive_HC, ring->active_HC, ring->active_HC_size *
      sizeof(active_HC_t), cudaMemcpyHostToDevice);
8 //assign the copied array to device variable, that can be used inside CUDA
      kernel
9 cudaMemcpyToSymbol(dactive_HC, &hactive_HC, sizeof(hactive_HC));
```

Code example 7.1: Allocate the arrays for the data structures.

The memory is also allocated to hold the particle data for every bunch and the bunches are transferred to the GPU memory one by one. Since particle data is transferred back from GPU at every turn for statistics calculations and output purposes *cudaHostRegister* is used to page-lock the CPU memory for particles. The page-locking of the CPU memory allows to increase the transfer speed between the CPU and GPU, and also allows to use asynchronous data transfer allowing to overlap data movement and calculations on the GPU.

The memory setup of the GPU ends with allocation of all the temporary arrays used by kernels to hold the intermediate results.

7.2.3 The basic single particle transformations

The basic optics transformation is performed once per turn. Mbtrack stores the position of every particle in phase space in a 6 dimensional vector $(x, x', y, y', \tau, \delta)$. The x and y are the horizontal and vertical positions, while $x' = \frac{dx}{ds}$ and $y' = \frac{dy}{ds}$ are transverse momenta at longitudinal position s. Parameter $\delta = \frac{\Delta E}{E_0}$ describes the energy deviation relative to the reference particle's energy E_0 and the longitudinal

coordinate τ is the arrival time with respect to the reference particle [58].

The relative energy spread δ_i at turn $i+1$ is computed by:

$$\delta_{i+1} = \delta_i + \epsilon_i - \frac{U_{rad}}{E_0} \tag{7.1}$$

where ϵ_i is the relative energy gain in the RF cavities and U_{rad} is the synchrotron radiation losses.

The longitudinal coordinate is updated every turn using:

$$\tau_{i+1} = \tau_i + \delta_i T_0 \alpha_c \tag{7.2}$$

where α_c is the momentum compaction factor. The relative energy gain ϵ_i from RF cavities is determined by:

$$\epsilon_i = \frac{e}{E_0}[V_{RF}\sin(\omega_{RF}\tau_i + \phi_s) + \sum_j \hat{V}_j \sin(n_j\omega_{RF}\tau_i + n_j\phi_j)] \tag{7.3}$$

where V_{RF} is the peak voltage, ω_{RF} is the angular RF frequency, ϕ is the phase of the synchronous particle and e is the electron charge. The first term in the equation describes the relative energy gain from main RF cavity. The second term in the equation describes the effect of additional cavities, where $\hat{V}_j$ is the peak voltage of the jth harmonic cavity with angular frequency $n_j\omega_j$ and phase $n_j\phi_j$.

In transverse planes the particles in the beam perform betatron oscillations. These oscillations are described using Twiss parameters $\alpha_{x,y}, \beta_{x,y}, \gamma_{x,y}$, which describe the beam shape, size and orientation, and a phase advance per turn Ψ_{xy} [58]. For particles with non-zero energy deviations the phase advance can be calculated by:

$$\Psi_{xy} = \Psi_{x0y0}(1 + \xi_{xy}\delta) \tag{7.4}$$

where ξ_{xy} is the chromaticity of the lattice. Given the presence of the horizontal dispersion D the transformation in transverse planes are expressed by the transfer

matrices:

$$\begin{pmatrix} x \\ x' \\ \delta \end{pmatrix} = \begin{pmatrix} \cos\Psi_x + \alpha\sin\Psi_x & \beta_x \sin\Psi_x & D \\ -\gamma_x \sin\Psi_x & \cos\Psi_x - \alpha\sin\Psi_x & D' \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ x' \\ \delta \end{pmatrix}$$

$$\begin{pmatrix} y \\ y' \end{pmatrix} = \begin{pmatrix} \cos\Psi_y + \alpha\sin\Psi_y & \beta_y \sin\Psi_y \\ -\gamma_y \sin\Psi_y & \cos\Psi_y - \alpha\sin\Psi_y \end{pmatrix} \begin{pmatrix} y \\ y' \end{pmatrix} \tag{7.5}$$

Changes in the beam energy are also caused by quantum excitation and radiation damping shown in the equations below.

$$\begin{aligned} \tilde{\delta}_{i+1} =& \delta_{i+1}(1 - D_E) + \sigma_E\sqrt{2D_e}\delta_{rand} \\ \tilde{x}_{i+1} =& x_{i+1} + \sigma_x\sqrt{D_x}x_{rand} \\ \tilde{x}'_{i+1} =& x'_{i+1}\frac{1+\delta_{i+1}}{1+\delta_{i+1}+\epsilon_{i+1}} + \sigma_{x'}\sqrt{D_x}x'_{rand} \end{aligned} \tag{7.6}$$

where coefficients D and σ correspond to radiation damping times and bunch energy spread, while δ_{rand}, x_{rand} and x'_{rand} are random numbers from normal distribution with unit standard deviation [58].

The application launches one kernel that performs these transformations for every bunch. Since calculations for each particle are independent, one thread per particle is created inside the kernel. The random numbers needed for the calculations of the radiation damping and quantum excitation are generated using the Nvidias cuRand library. Shared memory is used to hold the data for additional harmonic cavities, since they are shared by all the threads, and shared memory usage allows to improve the load time from global memory.

7.2.4 Geometric ring impedance

The simulations in mbtrack can include arbitrary number of resonators as well as purely resistive and inductive components, all contributing to the total geometric wake [58]. The macro-particles in each bunch are grouped into cells (bins) depending on their longitudinal position. The wake functions are calculated corresponding to ensemble of resonators. Each turn the excitation of this wake by

each bin is calculated and the resulting kick is given to each bin. All particles in the same bin receive the same kick. In this way presence of the self-field to which each bunch is subjected is taken into account [59].

The longitudinal wake function of a resonator, when $\tau_j > 0$, is given by:

$$W^{\parallel}(\tau_j) = \frac{\omega_r R_s}{Q} \exp - \frac{\omega_r}{2Q} \tau_j (\cos \omega_r \sqrt{1-(4Q^2)^{-1}} \tau_j - \frac{\sin \omega_r \sqrt{1-(4Q^2)^{-1}} \tau_j}{2Q\sqrt{1-(4Q^2)^{-1}}}) \tag{7.7}$$

where ω_r is the resonant angular frequency, Q is the quality factor and R_s is the longitudinal shunt impedance and τ_j is the position at the bin j. Similarly transverse wake of such a resonator ($\tau_j > 0$) can be expressed as:

$$W^{\perp}(\tau_j) = \frac{\omega_r R_s}{Q\sqrt{1-(4Q^2)^{-1}}} \exp - \frac{\omega_r}{2Q} \tau_j \sin \omega_r \sqrt{1-(4Q^2)^{-1}} \tag{7.8}$$

The components describing purely resistive and inductive impedances are R and $iL\omega$ and form the corresponding wake function:

$$W2^{\parallel}(\tau_j) = R\delta(\tau_j) - L\frac{d\delta(\tau_j)}{d\tau_j} \tag{7.9}$$

where $\delta(\tau)$ is the delta-function.

The wake voltage induced by this bunch is then calculated by:

$$V(\tau_j) = \sum_{k=0}^{j-1} q_k W^{\parallel}(\tau_j - \tau_k) + \frac{q_j W2^{\parallel}}{2} \tag{7.10}$$

where $q_{j,k}$ is the total charge in the bins j and k.

In the longitudinal and transverse planes the transformation is expressed as change in particle energy, where horizontal and vertical planes are treated identically:

$$\begin{aligned} \Delta\delta_j &= \frac{q_j V(\tau_j)}{E_0} \\ \Delta x'_j &= \frac{q_j}{E_0} \sum_{k=0}^{j-1} q_k D_p(\tau_k) W^{\perp}(\tau_j - \tau_k) \end{aligned} \tag{7.11}$$

where $D_p(\tau_k)$ is the dipole-moment of the bunch at position τ_k.

The resistive-wall effects for longitudinal and transverse planes are also calculated and added to the wake voltage $V(\tau_i)$:

$$\begin{aligned} W^{\parallel}_{RW}(\tau_j) &= -\frac{1}{4\pi a}\sqrt{\frac{Z_0}{\sigma c \pi \tau_j^{\frac{3}{2}}}} \\ W^{\perp}_{RW}(\tau_j) &= -\frac{1}{\pi a^3}\sqrt{\frac{Z_0 c}{\sigma \pi \tau_j^{\frac{1}{2}}}} \end{aligned} \tag{7.12}$$

where Z_0 is the impedance of free space, c is the speed of light and a the effective beam pipe radius.

The transformation begins by assigning each particle to a bin (mesh cell), since the calculations of the bin to which the particle belongs is independent this process can again be parallelized over the number of particles in the bunch. A kernel is launched where each thread handles one particle and finds and saves the bin number to which this particle belongs to. The bin number is saved in a temporary memory reserved in the beginning of the calculations and is reused between bunches. The number of bins and the bin size is taken from the configuration file and transferred to the GPU at the start of the simulation as described in the 7.2.2 section.

After each particle has been assigned to a bin a kernel is launched that counts the particles in each bin and the dipole-moments D_p for horizontal and vertical planes. This kernel is also launched with one thread per particle, but since there are multiple particles per bin atomic operations are needed to sum up the particle counts in the bins.

After the particle counts per bin are known a kernel is launched that finds the min and max bin that holds any particles. Since searching for the min and max bins requires communication between threads and it is not a good problem to parallelize, it is done serially on the GPU. One block with multiple threads are launched on the GPU, the multiple threads parallelize the loading of the data from the slow GPU memory to the faster shared memory and then one thread searches for the first and last bins that contains any particles. Since we do not need to loop over all the cells this serialization of the kernel will not cause a

bottleneck for the simulation.

After the particles are assigned to bins and the bins that hold any particles are found, a kernel is launched that constructs the wake potentials at each cell for each of the planes. First a kernel is called that performs the calculations on the LON plane and then separate kernels for HOR and VER planes. To construct the wake potentials one thread is launched for each bin which holds the particles and the wake potential effect on this bin is calculated by the thread and saved in a temporary storage on the GPU memory.

Once the wake potentials are constructed a kernel is called that applies the effects of the wake potentials to each particle in the bunch. This kernel parallelizes the simulation over the particles in the bunch and launches one thread for each particle. Shared memory is used to store the data that is frequently reused by the kernel to minimize the loads from global memory. And the same kernel handles all three planes if they are enable in the simulation. The kernel that applies the effects of the wake potentials to all the planes enabled in the simulation is demonstrated in 7.2

Additionally passing charges also deposit an oscillating field in the resonant structures which decays over time. The following charges experience a kick from the resulting voltage depending on their arrival time [58]. The wake potential V_{HC} acting on a charge at the longitudinal position τ, excited by the last k bunches with the charge distribution ρ_k, depends on the wake field W_{HC} of the harmonic cavity [60]:

$$V_{HC}(\tau) = \sum_{k}^{bunches} \int_{-\infty}^{\infty} d\tau' \rho_k(\tau') W_{HC}(k\Delta\tau_b + \tau - \tau') \tag{7.13}$$

where $\Delta\tau_b$ is the distance between two bunches. The wake field W_{HC} is modeled by:

$$\begin{aligned} W_{HC}(\tau) =& 2\alpha R_s \quad \exp^{-\alpha\tau}(\cos\overline{\omega}\tau - \frac{\alpha}{\overline{\omega}}\sin\overline{\omega\tau}) \\ \alpha = \frac{\omega_r}{2Q},& \qquad\qquad\qquad \overline{\omega} = \sqrt{\omega_r^2 - \alpha^2} \end{aligned} \tag{7.14}$$

A kernel that calculates the wake potential V_{hc} is launched for each bunch and the calculated wake potential is stored in temporary GPU memory lr_wake.

```
__global__ void kernelWakePotentialEffect(particle_t *particles, int
    *mapcell, double *GL1, double *GlambdaV, double *GlambdaH, double
    bunchIb, int Np, int Ncell) {

  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int tid = threadIdx.x;

  //load wake potentials in shared memory
  extern __shared__ double smem[];
  double *sGL1 = (double*)smem;
  double *sGlambdaV = (double*)&smem[Ncell];
  double *sGlambdaH = (double*)&smem[2*Ncell];
  while (tid < Ncell) {
    sGL1[tid] = GL1[tid];
    sGlambdaV[tid] = GlambdaV[tid];
    sGlambdaH[tid] = GlambdaH[tid];
    slr_wake[tid] = lr_wake[tid];
    tid += blockDim.x;
  }
  __syncthreads();

  if (idx < Np) {
    int map = mapcell[idx]; //get the bin of the particle
    //apply wake potential effects
    double factG = -dring.T0 * bunchIb / (Np * dring.E0 * FGIGA);
    particles[idx].slope.xtau += factG * sGL1[map] + slr_wake[map];
    //if VER and HOR planes enabled apply wake potentian effects
    if (dSelfFieldModel.PlaneV > 0)
      particles[idx].slope.z += -factG * sGlambdaV[map];
    if (dSelfFieldModel.PlaneH > 0)
      particles[idx].slope.x += -factG * sGlambdaH[map];
  }
}
```

Code example 7.2: Kernel to apply wake potential effect to particles in a bunch.

These effects are applied to particles together with geometric ring impedance effects as it is shown in the code example 7.2.

At the end of the transformation when the wake potential effects are applied the temporary storage that holds the wake potential data is transferred to the host side and written to disk. After the turn the temporary storage is cleared (all values set to 0) since it is reused by the following bunch.

7.2.5 Statistics calculations

After each turn statistics of each bunch are calculated. Statistics are used to log the information of about the bunches during the simulation and in the calculations of long-range resistive wall effects. Most time consuming part of the statistics calculations is the calculation of average position and momentum for

each dimension, as well as standard deviation of position and momentum for each dimension. Since all the particle data is kept on the GPU these calculations are also performed on the GPU side and results transferred to the CPU. To calculate the average value Thrust libraries *reduce* function is used to calculate the sums of position and momentum for each dimension. After *reduce* is performed data are sent to the CPU side where the average values are computed. To compute the standard deviation Thrusts *transform_reduce* function is used. Transformation is performed for every data point calculating its deviation from the average and reduction is performed to calculate the sums for each dimension. The sums are transferred to the CPU side where square root of the sums is taken to get the standard deviation.

Since each particle in the bunch in mbtrack is represented as a structure of 6 variables a custom operators are defined to perform reduction and transformations correctly on the arrays of particles. The statistics are calculated for each bunch as well as averaged for all the bunches in the simulations. Only the calculations for the individual bunches are performed on the GPU while the rest is done by the host side.

7.3 Results

7.3.1 Verification of the mbtrack-cuda

To validate the results from the GPU version of mbtrack-cuda the results are compared with the original mbtrack-mpi version. Since no new features are introduced in the CUDA version the simulations results should agree from both versions of the code.

The difficulty of comparing the results arises because of the use of random numbers in the simulation. Since for the calculation of synchrotron radiation (SR) effects random numbers are needed and different random number generators are used in both versions, it is not possible to reproduce the results of mbtrack-mpi version with mbtrack-cuda version. For this reason the validation of the code was done without the synchrotron radiation effects.

The achieved results show agreement between the two versions, meaning that

the results of the original application can be reproduced using the GPU version. This allows us to move the simulation from a CPU cluster to a heterogeneous CPU/GPU system.

7.3.2 Performance of the mbtrack-cuda

The performance tests of the mbtrack were performed on a system equipped with 2x Intel E5-2609 CPUs and a Nvidia Tesla K40c. The aim of the tests was to demonstrate the ability to enable the full ring simulations using the mbtrack-cuda version. These simulations are not possible with the mbtrack-mpi version without a computing cluster. The computing time is shown in the tables 7.1 and 7.2.

In the first test simulations each bunch consists of 100,000 particles and the simulation is run for 10,000 turns. This simulation includes basic optics transformations and long range resistive wall effects. The simulation is performed for longitudinal and horizontal planes while the vertical plane is ignored. The simulations are performed with synchrotron radiation effects included and excluded to show the effect of generating random numbers on the simulation time for the CPU version. The reported results show the full execution time of the simulation including the input and output operations. The results for this simulation are presented in table 7.1

Table 7.1: Comparison of mbtrack-mpi and mbtrack-cuda on 8 core machine with 1 Nvidia Tesla K40c GPU for the first test simulation.

Number of bunches	No SR effects		With SR effects	
	mbtrack-mpi	mbtrack-cuda	mbtrack-mpi	mbtrack-cuda
1	116s	44s	942s	44s
2	117s	65s	940s	65s
3	122s	86s	970s	86s
10	457s	231s	2059s	360s
138	-	2992s	-	2982s
416	-	9452s	-	9461s

The second simulations simulates the MAX IV 3GeV ring. The simulation includes the effects of two harmonic cavities and one broad band resonator, but does not simulate the long range resistive wall effects. The simulation also includes

100,000 particles per bunch and is run for 10,000 turns, but in this case transformations are performed only in longitudinal plane. The results for the second test simulation are shown in the table 7.2.

Table 7.2: Comparison of mbtrack-mpi and mbtrack-cuda on 8 core machine with 1 Nvidia Tesla K40c GPU for the second test simulation.

Number of bunches	No SR effects		With SR effects	
	mbtrack-mpi	mbtrack-cuda	mbtrack-mpi	mbtrack-cuda
1	82s	33s	357s	33s
2	81s	59s	355s	60s
3	87s	86s	362s	86s
10	273s	249s	856s	356s
58	-	1668s	-	1689s
176	-	7514s	-	7511s

The results show, that for a small number of bunches the GPU version offers a significant speedup over the CPU version. This is because for a small number of bunches the CPU versions does not utilize all the available parallelism of the system. The results also show that mbtrack-cuda version is able to handle full ring simulations in reasonable time, enabling to run the simulations when a CPU cluster is not available.

Chapter 8

Conclusions

The author has developed Dynamic Kernel Scheduler (DKS) during this work to ease the integration of hardware accelerators in existing scientific applications. DKS allows to separate all the device specific code, written using CUDA, OpenCL or OpenMP, from the host application. This separation eases the development, management and optimization of the device code, while also requiring minimal changes in the original application to integrate the use of new devices. DKS is developed to be easily extendable in the future to support new development frameworks and new devices.

During this work DKS was used to integrate the GPUs and Intel MICs in different scientific applications used at PSI and ETH for particle accelerator simulations and experimental data analysis.

Author used DKS to allow OPAL (Object Oriented Particle Accelerator Library) to offload FFT Poisson Solver and Monte-Carlo simulations for particle matter interaction to GPU or Intel MIC. CUDA and OpenMP was used in DKS to target these devices. Speedup of up to $\times 11$ was demonstrated for FFT Poisson solver on Nvidia Tesla K40 compared to MPI implementation using 8 CPU cores. For particle matter interaction simulations on the GPU speedups of up to $\times 50$ was achieved compared to multicore CPU performance.

The musrfit application for μSR experiments was enhanced to allow the use of GPUs to speed up parameter fitting. CUDA and OpenCL were used to create the GPU code and DKS was used to ease the integration of the created algorithm in musrfit. The demonstrated speedups using the GPU compared to multi-core

CPU implementation where around ×30...×40, which would allow for almost real time analysis of experimental data.

The author added the option to offload the time consuming parts of the algorithm to GPU for PET image reconstruction application developed at ETH. DKS was used together with CUDA to allow application to target Nvidia GPUs. The speedups achieved with the GPU (×57) brings the application closer to the main goal - real time PET image reconstruction.

Mbtrack application used at PSI to analyze collective bunch instabilities in the SLS-2 upgrade was optimized to take advantage of the computational power of GPUs. The application uses CUDA to target Nvidia GPUs and the performance improvements using GPUs allows for the full SLS-2 ring simulation to be run on a single GPU instead of requiring a large CPU cluster.

The results achieved during this work show, that using hardware accelerators can provide a significant boost in application execution time, when compared to more traditional CPU systems. The use of DKS can also ease the integration of these devices in existing applications with a large code base.

References

[1] Top500 supercomputers. 2016. URL: http://www.top500.org/lists/2016/11/. 1

[2] NVIDIA CUDA Zone. 2015. URL: https://developer.nvidia.com/cuda-zone. 1

[3] Khronos Group, OpenCL. 2015. URL: https://www.khronos.org/opencl/. 2

[4] Intel Xeon Phi coprocessor. 2015. URL: https://software.intel.com/en-us/mic-developer. 2

[5] OpenACC. 2015. URL: http://www.openacc.org/. 2, 3

[6] OpenMP. 2015. URL: http://openmp.org/wp/openmp-specifications/. 2

[7] OpenMP. OpenMP Application Program Interface. *The OpenMP Forum, Tech. Rep*, (July):320, 2013. doi:10.1080/08905769008604595. 3

[8] Gpu-accelerated libraries. 2016. URL: https://developer.nvidia.com/gpu-accelerated-libraries. 3, 24

[9] Intel Math Kernel Library (Intel MKL). 2015. URL: https://software.intel.com/en-us/articles/intel-mkl-on-the-intel-xeon-phi-coprocessors. 3, 25

[10] Thrust. 2016. URL: http://docs.nvidia.com/cuda/thrust. 3, 20

[11] Arrayfire. 2016. URL: http://www.arrayfire.com/docs/index.htm. 3, 19, 20

[12] Boostcompute. 2016. URL: https://boostorg.github.io/compute/. 3, 22

[13] Mathias Bourgoin, Emmanuel Chailloux, and Jean Luc Lamotte. Efficient abstractions for GPGPU programming. *International Journal of Parallel Programming*, 42(4):583–600, 2014. 3

[14] J. Svensson, K. Claessen, and M. Sheeran. GPGPU kernel implementation and refinement using Obsidian. *Procedia Computer Science*, 1(1):2065–2074, 2010. 3

[15] M. Viñas, B.B. Fraguela, Z. Bozkus, and D. Andrade. Improving OpenCL Programmability with the Heterogeneous Programming Library. *Procedia Computer Science*, 51:110–119, 2015. 3

[16] Intel xeon phi x100 family coprocessor - the architecture. 2012. URL: https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner. 15, 16

[17] James Malcolm, Pavan Yalamanchili, Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, and John Melonakos. ArrayFire: a GPU acceleration platform. *Proc. SPIE 8403, Modeling and Simulation for Defense Systems and Applications VII*, 8403, 2012. doi:10.1117/12.921122. 19, 20

[18] Nathan Bell and Jared Hoberock. Thrust: A Productivity-Oriented Library for CUDA. *GPU Computing Gems Jade Edition*, pages 359–371, 2012. doi:10.1016/B978-0-12-385963-1.00026-5. 21

[19] Vexcl. 2016. URL: http://vexcl.readthedocs.io. 23

[20] Denis Demidov. VexCL Documentation. 2016. 24

[21] Viennacl. 2016. URL: http://viennacl.sourceforge.net/. 24

[22] Karl Rupp, Florian Rudolf, and J Weinbub. ViennaCL-a high level linear algebra library for GPUs and multi-core CPUs. *Intl. Workshop on GPUs and Scientific Applications*, pages 51–56, 2010. URL: http://www.iue.tuwien.ac.at/pdf/ib_2010/Rupp_GPUScA.pdf. 24

[23] Acl - amd compute libraries. 2016. URL: http://developer.amd.com/tools-and-sdks/opencl-zone/acl-amd-compute-libraries/. 25

[24] Andreas Adelmann, Uldis Locans, and Andreas Suter. The Dynamic Kernel SchedulerPart 1. *Computer Physics Communications*, 207:83–90, 2016. URL: http://linkinghub.elsevier.com/retrieve/pii/S0010465516301370, doi:10.1016/j.cpc.2016.05.013. 25, 64

[25] Ken Naono, Keita Teranishi, John Cavazos, and Reiji Suda. Software automatic tuning: From concepts to state-of-the-art results. *Software Automatic Tuning: From Concepts to State-of-the-Art Results*, pages 1–377, 2010. doi:10.1007/978-1-4419-6935-4. 32

[26] Ping Guo and Liqiang Wang. Auto-tuning CUDA parameters for sparse matrix-vector multiplication on GPUs. *Proceedings - 2010 International Conference on Computational and Information Sciences, ICCIS 2010*, pages 1154–1157, 2010. doi:10.1109/ICCIS.2010.285. 32

[27] Chih-Sheng Lin, Shih-Meng Teng, and Pao-Ann Hsiung. Auto-tuning for GPGPU applications using performance and energy model. *Journal of Systems Architecture*, 62:40–53, 2016. URL: http://linkinghub.elsevier.com/retrieve/pii/S1383762115001514, doi:10.1016/j.sysarc.2015.11.012. 32

[28] Martin Tillmann, Thomas Karcher, Carsten Dachsbacher, and Walter F. Tichy. Application-independent autotuning for GPUs. *Advances in Parallel Computing*, 25(1):626–635, 2014. doi:10.3233/978-1-61499-381-0-626. 32

[29] Michael Vollmer. Meta-Programming and Auto-Tuning in the Search for High Performance GPU Code. pages 1–11, 2015. 32

[30] Natalia Kalinnik, Matthias Korch, and Thomas Rauber. Online auto-tuning for the time-step-based parallel solution of ODEs on shared-memory systems. *Journal of Parallel and Distributed Computing*, 74(8):2722–2744, 2014. URL: http://dx.doi.org/10.1016/j.jpdc.2014.03.006, doi:10.1016/j.jpdc.2014.03.006. 33

[31] R.W Hockney. Methods in computational physics. pages 136–211, 1970. 36

[32] R.W. Hockney and J.W. Eastwood. *Computer Simulation using Particles*. Adam Hilger, 1988. 36

[33] J. W. Eastwood and D. R. K. Brownrigg. *J. Comp. Phys, 32, 24-38*, 1979. 36

[34] Benjamin Ulmer, Uldis Locans, and Andreas Adelmann. Performance analysis of mkl fast fourier transform and fft-poisson solver on intel xeon phi. *ETH Zurich, Semester project*, 2015. 40

[35] Y.J. Bi, A. Adelmann, R. Dölling, M. Humbel, W. Joho, M. Seidel, and T.J. Zhang. Towards quantitative simulations of high power proton cyclotrons. *Physical Review Special Topics - Accelerators and Beams*, 14(5), 2011. 40

[36] J.J. Yang, A. Adelmann, M. Humbel, M. Seidel, and T.J. Zhang. Beam dynamics in high intensity cyclotrons including neighboring bunch effects: Model, implementation, and application. *Physical Review Special Topics - Accelerators and Beams*, 13(6), 2010. 40

[37] A Allisy, AM Kelleler, RS Caswell, et al. Stopping powers and ranges for protons and alpha particles. *ICRU Report*, 49, 1993. 41

[38] K.A. Olive et al. Particle data group. *Chin. Phys. C,* **38**, *090001*, 2014. 41

[39] William R. Leo. *Techniques for nuclear and particle physics experiments*. Springer-Verlag, Berlin Heidelberg New York, 2nd edition, 1994. 41

[40] J. D. Jackson. *Classical Electrodynamics*. John Wiley &. Sons, New York, 3rd edition, 1998. 43

[41] Helene Stachel, Andreas Adelmann, and Prof Klaus Kirch. Double Degrader for Proton Therapy. *ETH Zurich, Masters thesis*, 2013. 46

[42] Matthias Toggweiler. An adaptive time integration method for more efficient simulation of particle accelerators Supervised by. 2011. 48

[43] Lorenzo Moneta, M Winkler, A Zsenei, P Mato-Vila, M Hatlo, and F James. Developments of mathematical software libraries for the LHC experiments. *IEEE Transactions on Nuclear Science*, 52:2818–2822, 2005. 61, 68, 69

[44] CN/ASD Group. Minuit users guide. *Program Library D506, CERN*, 1993. 61

[45] A Suter and BM Wojek. Musrfit: a free platform-independent framework for μsr data analysis. *Physics Procedia*, 30:69–73, 2012. 61, 66

[46] A Youanc and P Dalmas de Réotier. *Muon spin rotation, relaxation and resonance*. Oxford University Press, Oxford, 2011. 63, 68

[47] CUDA Toolkit 4.2. CUBLAS Library. *PG-05326-041_v01*, (March), 2012. 66

[48] CUDA Toolkit 7.5. Nvrtc - cuda runtime compilation. (September), 2015. 66

[49] Musrfit user manual. 2015. URL: http://lmu.web.psi.ch/musrfit/user/MUSR/WebHome.html. 66

[50] Jing-Yu Cui, Guillem Pratx, Sven Prevrhal, and Craig S Levin. *Medical physics*, 38(12):6775–86, 2011. 71

[51] J. L. Herraiz, S. Espana, R. Cabido, A. S. Montemayor, M. Desco, J. J. Vaquero, and J. M. Udias. *IEEE Transactions on Nuclear Science*, 58(5 PART 1):2257–2263, 2011. 71

[52] Guillem Pratx, Jing Yu Cui, Sven Prevrhal, and Craig S. Levin. *3-D tomographic image reconstruction from randomly ordered lines with CUDA*. NVIDIA Corporation and Wen-mei W. Hwu, 2011. 71

[53] Andrew J. Reader and Habib Zaidi. Advances in PET Image Reconstruction. *PET Clinics*, 2(2):173 – 190, 2007. PET Instrumentation and Quantification. URL: http://www.sciencedirect.com/science/article/pii/S1556859807000193, doi:http://dx.doi.org/10.1016/j.cpet.2007.08.001. 71, 72

[54] S. Agostinelli, J. Allison, et al. Geant4 – a simulation toolkit. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 506(3):250–303, 2003. URL: http://www.sciencedirect.com/science/article/pii/S0168900203013688, doi:http://dx.doi.org/10.1016/S0168-9002(03)01368-8. 80

[55] J. Allison, K. Amako, et al. Geant4 developments and applications. *IEEE Transactions on Nuclear Science*, 53(1):270–278, 2006. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1610988, doi:10.1109/TNS.2006.869826. 80

[56] T. F. Budinger, S. E. Derenzo, et al. Emission computer assisted tomography with single-photon and positron annihilation photon emitters. *J Comput Assist Tomogr*, 1(1):131–145, Jan 1977. 80

[57] Stefan Hegglin. Simulating Collective Effects on GPUs. *ETH Zurich, Masters thesis*, 2016. 84

[58] Galina Skripka, Ryutaro Nagaoka, Marit Klein, Francis Cullinan, and Pedro F. Tavares. Simultaneous computation of intrabunch and interbunch collective beam motions in storage rings. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 806:221–230, 2016. URL: http://linkinghub.elsevier.com/retrieve/pii/S016890021501236X, doi:10.1016/j.nima.2015.10.029. 84, 85, 88, 89, 92

[59] Jack Borthwick, Francis Cullinan, Ryutaro Nagaoka, and Galina Skripka. mbtrack : Multi-bunch tracking code. *mbtrack manual*, 2015. 84, 90

[60] M Klein, R Nagaoka, and Synchrotron Soleil. Multibunch Tracking Code Development To Account for Passive Landau Cavities. *International Particle Accelerator Conference 2013*, (5):5–7, 2013. 92