CODE DEVELOPMENT FOR COLLECTIVE EFFECTS

K. Li*, H. Bartosik, G. Iadarola, A. Oeftiger, A. Passarelli, A. Romano, G. Rumolo, M. Schenk,
CERN, Switzerland,
S. Hegglin, ETH Zürich, Switzerland

Abstract

The presentation will cover approaches and strategies of modeling and implementing collective effects in modern simulation codes. We will review some of the general approaches to numerically model collective beam dynamics in circular accelerators. We will then look into modern ways of implementing collective effects with a focus on plainness, modularity and flexibility, using the example of the PyHEADTAIL framework, and highlight some of the advantages and drawbacks emerging from this method. To ameliorate one of the main drawbacks, namely a potential loss of performance compared to the classical fully compiled codes, several options for speed improvements will be mentioned and discussed. Finally some examples and applications will be shown together with future plans and perspectives.

INTRODUCTION

Collective effects can lead to beam instabilities and brightness limitations and, thus, have a considerable detrimental impact on the performance of high brightness machines. Numerical modeling and simulations are a fundamental tool in understanding the physics of collective effects in circular particle accelerators. Moreover, they are a valuable means to evaluate and propose mitigation techniques to improve these limitations.

With the push towards higher brightness and higher energy these limitations play an increasingly important role. They involve several effects, among them impedance driven instabilities, electron cloud effects, the impact of long-range and head-on beam-beam collisions and single and multi-bunch effects.

In the past, simulation tools were often geared to modeling certain types or subsets of these effects. Meanwhile, the understanding of the individual effects has improved by a large amount and the combination of the different effects is now becoming increasingly important. To systematically study these combined effects on the beam stability, it is mandatory to bring together all the specific features of collective effects simulation codes.

In this paper we will investigate modern approaches to code development for collective effects. We will briefly illustrate the numerical modeling of collective effects in circular accelerators and then mention some general concepts and strategies for modern code style. We will then embark into a more specific discussion on the utilization of modern programming languages where we will use the example of the PyHEADTAIL framework. We will try to give an objective view on the advantages this type of approach can provide and show how to cope with potential limitations. Finally, we will present some specific applications illustrating the particular usefulness of this type of approach.

* kevin.shing.bruce.li@cern.ch

BASIC MODEL OF THE ACCELERATOR-BEAM SYSTEM

The numerical model that we will adopt to illustrate some of the concepts is the macroparticle model. Macroparticle models provide a direct and intuitive mapping of physics onto computer systems. Nearly any physical effect linked to particle beam dynamics can be easily implemented which makes these models extremely flexible and powerful. Macroparticles are essentially a numerical representation of a cluster of spatially neighbouring physical particles. As such, they follow the same dynamics following the same equations of motion that hold for physical particles.

A macroparticle system's dynamics is fully described by the evolution of its six phase space variables, the generalized coordinates and canonically conjugate momenta. Hence, a physical particle system, or a particle beam, can be easily represented via a macroparticle system on a computer system as an allocated chunk of memory where for each macroparticle all values of the six phase space variables are stored.

The accelerator is represented as a concatenation of elements each individually performing a distinct particle tracking. On a computer system, this can be represented by dedicated functions or methods that act on a macroparticle system in a defined and specific manner. Typically, a ring is split into a set of segments. A particle beam is transported from one segment to another by means of linear transfer matrices based on the machine optics in the transverse planes. In the longitudinal plane, tracking is performed assuming linear synchrotron motion, but also multi-harmonic RF systems can be easily taken into account as long a symplectic integration scheme is employed to assure numerical stability. Nonlinearities are treated via effective machine parameters such as chromaticity or detuning with amplitude by adjusting the phase advance of each individual macroparticle correspondingly after tracking along one segment. At each segment node, collective interactions can take place such as the application of different forms of wake field kicks, beam – electron cloud interaction, space charge kicks etc.

Typically, these effects are correlated with the longitudinal position of particles within the beam. To make the computations numerically efficient, a beam is longitudinally binned into a set of slices via a 1D particle-in-cell (PIC) algorithm. A single slice is then thought to be representative for all the macroparticles contained within. Collective

ISBN 978-3-95450-178-6

effects are correspondingly applied to the macroparticles on a slice-by-slice basis.

This type of modeling is illustrated in Fig. 1. Looking at this model, it inherently exhibits a modular structure and it becomes somewhat natural to follow the same modular architecture also for the code design itself.

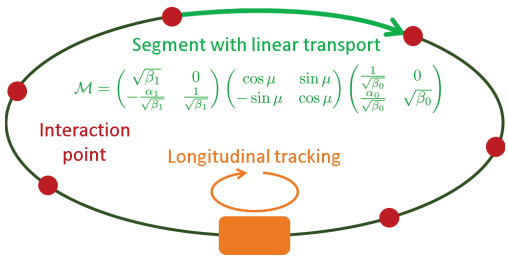

Figure 1: Illustration of the typical numerical model of the accelerator-beam system for collective effects. α, β denote the Twiss parameters, μ the phase advance between the interaction points (red).

MODERN APPROACHES AND PROGRAM ARCHITECTURES

General Concepts

Modern programming languages have support for multiple programming paradigms. This gives flexibility in the choice of programming style in order to describe and control the underlying numerical model in the most suited manner.

There are some minimum requirements, apart from the obvious one that the underlying physics should be correctly modeled, that can be specified for modern computer simulation codes. A key guideline should be to keep them as short and as simple as possible. In addition, they should be designed in a modular way and they should be usable in a dynamic manner. We will explain what we mean by these requirements in more detail in the following.

Writing short and simple code keeps the code base compact and concise and results in programs that are usually much more straightforward to read and to understand, making it easier to maintain and to extend and, at the same time, less susceptible to errors. The philosophy here is not to produce code bloat of which the quality and capabilities is measured by the number of lines of code but instead to write this code in as few lines as possible while capturing all the essentials of the underlying physical processes. Although this process does require some effort it usually pays off in the long run.

A modular architecture was already identified in the previous section to be well-suited to describe and to set up an accelerator-beam system simulation. The idea here is to have an orthogonal set of code blocks that can be developed independently and that can be combined to allow for a nearly infinite spectrum of simulation scenarios to be carried out. Each of these code blocks should follow the guiding philosophy of being compact and concise and ideally self-documenting in order to make them easily accessible, maintainable and usable. The concept of object-oriented programming blends in perfectly with the idea of a modular architecture. One can independently instantiate different objects that represent different elements of a machine. Each object can have its own interface, properties and methods to then manipulate the macroparticle beam.

Finally, requiring the simulation codes to be usable in a dynamic manner means several things. For one, the code should be fast both in development time as well as in execution time. Incremental and interactive development allows for efficient and effective implementation of complex physical processes. Code optimizations and parallelization allow for speed gains during execution.

Technologies nowadays are changing extremely fast and simulation codes should be able to adapt accordingly. For this, the classical static and heavy code basis is not well suited and should instead be replaced by more lightweight and flexible code.

Choice of Programming Language

Most of the requirements stated above can be satisfied by several modern programming languages. For the purpose of illustration and because it was the choice made for PyHEADTAIL, we will use the Python programming language [1] as an example. Python is a high-level, interpreted and dynamic programming language with a very large user community. It supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles. Its design philosophy is to foster short and simple code with an emphasis on code readability.

In terms of modular architecture, the layout and structure forms the backbone of the code design. Python's package and module management provide an easy and natural way of implementing any given structure with only few constraints. Thus, one is left with the pure architectural task of crafting the different parts of the code and their interactions. Python's strong support for object-oriented programming allows to build on this structure and to include additional functionality provided by the Python class mechanism such as instantiation of objects, inheritance, special methods, abstract classes etc.

Code development itself becomes very easy, for one due to the strongly reduced number in lines of code needed along with the very expressive language and syntax of Python. On the other hand, Python is an interpreted language. This makes the programming procedure essentially interactive. Online inspection and interaction with the program become possible making programming much more efficient and reliable. Programs are developed and tested at the same time almost naturally encouraging test-driven development with all its benefits.

A potential problem remains due to the very fact that Python is an interpreted language which is dynamically typed. This comes with considerable overhead during execution of the program and can render the simulation code

ISBN 978-3-95450-178-6

significantly slower compared to fully compiled simulation codes. We will address this issue in the next section.

Example of PyHEADTAIL

PyHEADTAIL [2] is a project that was started at CERN at the beginning of 2014. It is based on the well established HEADTAIL code [3]. However, being more a framework than a pure simulation code it builds on the very principles stated above. As such, it is predominantly written in Python and aims for a compact and concise code base making extensive use of Python's powerful expressiveness. It is built in a strictly modular way. The basic building blocks are a particles package, several packages containing machine elements and some utilities packages.

The particles package manages macroparticle systems which representing the physical particle beams. Essentially, this is a collection of data structures to keep track of the macroparticle system's dynamics and the evolution of its six phase space variables along with some specialised methods to help handling these data structures. It also contains all the methods for the generation of different macroparticle distributions and is mainly used to instantiate a bunch or beam object.

Machine elements are derived from an abstract base class and as such must all contain a *track* method. This method is always called on a bunch or beam object and manipulates the underlying data structures in accordance to the machine element that the object represents. There are classes that describe single particle tracking such as the ones contained in the longitudinal tracking module describing synchrotron motion. The multipoles module describes non-linear kicks in the thin lens approximation. There are other classes that simulate collective effects such as from wake fields or space charge. The architecture is set up in a way to minimize dependencies and to support the straightforward implementation of additional packages, modules or classes in order to streamline the code design and development. It is continuously being improved.

In providing the structure and classes that allow a modular composition of basic building blocks for customized simulations, the question may arise how to create a suitable inputfile syntax that can support and exploit this feature. We found that it is not necessary to create a new syntax and input files as used in the classical computer simulation codes such as MAD-X [4] or MAFIA [5]. Instead, it would be much easier to build on a syntax that is already available and widely known – the Python syntax itself. The input file becomes essentially a Python script and along with this comes the full power of the Python programming language injected up to the level of inputfile writing.

With this approach for example Python control flows become available in the input file. Examples are conditional execution of different parts of the input file (e.g. having a trigger on file i/o) or dynamic change of object attributes (i.e. resembling the implementation of trim functions). Any missing functionality can be easily programmed at the level of the input file. This can serve as a test and can later be reworked up to form a new module, for example. We will show some use cases in the last section.

Figure 2 shows the typical workflow in setting up and running a PyHEADTAIL simulation.

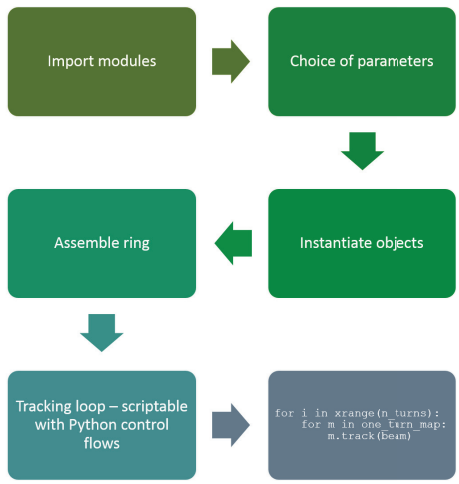

Figure 2: Typical workflow when setting up a PyHEADTAIL simulation.

PERFORMANCE CONSIDERATIONS

We mentioned in the previous section that the advantages provided by an interpreted language typically come with the price of performance loss in execution speed. Fortunately, there are ways out of this limitation at least for the case of Python.

First, Python contains a large set of third party libraries specialized for scientific computing, most importantly to be mentioned here, the NumPy and SciPy packages [6]. The core functionality of NumPy is its *ndarray* (for n-dimensional array) data structure. These arrays are strided views on contiguous memory buffers and they are homogeneously typed [1]. Any arithmetic operation on an ndarray is then automatically propagated down to a lower level such that the inner loops are performed on the level of the compiled language. This is called vectorization [2]. Algorithms that are not expressible as a vectorized operation will typically run slowly because they must be implemented in "pure Python". For these, Python can be interfaced with compiled languages such as Fortan or C routines, for example, to speed up computationally demanding parts of the code and thus to overcome performance bottlenecks. There are basically two different strategies for extending Python in this manner.

Python can be extended to use it as a "glue" language in which the core programming language is another, usually lower level compiled language, and Python is just used at the high level to stich different components of the core program together in a script. Different tools can be used for this

[1] A new package called Blaze attempts to generalize the NumPy functionalities to distributed and heterogeneous data structures and out-of-core computations [7]

[2] This should not be confused with the automatic vectorization provided by modern CPUs via SIMD instruction sets.

ISBN 978-3-95450-178-6

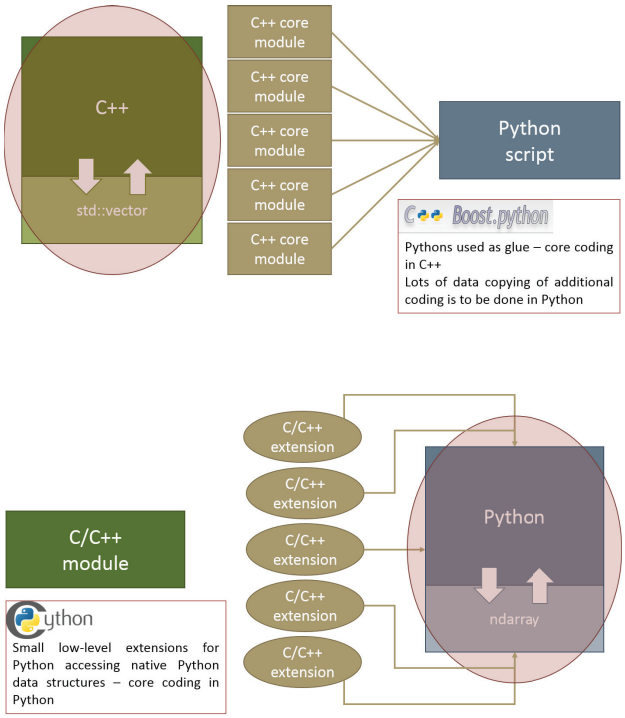

Figure 3: Comparison of the two strategies using Python as a "glue" language (top) and using Python as the core programming language (bottom) with the lower level language being C++ in this example. The core programming language is encircled in red in each case along with the basic data structure used.

with SWIG [8] or Boost.Python [9] being among the most popular. While most of the advantages mentioned earlier about having the input file actually written as a Python script still hold, the core programming is performed with a lower level language, e.g. C++, and the core data structures are the constructs of this language (i.e. for C++ the std::vector). For most scientists this makes the fundamental principle of writing short and simple code a lot more challenging. The other option is to use Python as the core programming language and to use lower level compiled languages only for the parts that actually pose a performance bottleneck. There are several tools available for Python that can be used for this purpose, among them statically compiling ones such as F2PY [10], ctypes which is part of the Python standard library, Cython [11] or alternatives that interoperate with NumPy, including scipy.weave, numexpr [12] or Numba [13]. Figure 3 shows and illustrative comparison of the two approaches.

In fact, PyHEADTAIL emerged from another project that was started already in 2012 and actually followed the first approach. This was cobra-HEADTAIL [14] which had its core written in C++, exporting modules to Python via Boost.Python. It was more cumbersome to add code written in pure Python and to exploit the advantages provided by NumPy since the basic data structures were C++ std::vectors. For this reason and after some research on the ndarray data structure it was finally decided to move to the second approach and to migrate to PyHEADTAIL.

Finally, parallelisation techniques exist for Python to further boost the performance both by multi-threading via OpenMP [15] (where the global interpreter lock, or GIL, needs to be released) and by multi-processing via mpi4py [16]. In addition, parallelsiation on the GPU is easily added with libraries such as PyCUDA [17] or can be added as CUDA [18] extensions via Cython or even via the CUDA just-in-time (JIT) compilation offered by Numba [13].

APPLICATIONS, PRESENT STATUS AND PERSPECTIVES

One of the big advantages that comes with running simulations in a scripting language is the dynamic control over the simulation process. For example, it is possible to access instance variables representing simulation parameters and to modify them at run time. This allows to seamlessly implement trim functions into the simulation. In principle, real machine cycles can thus be realistically simulated. One example where dynamically changing parameters have been employed is the creation of longitudinally hollow bunches in the CERN Proton Synchrotron Booster (PSB) used for space charge mitigation [19]. Here, the phase loop offset is modulated around the synchronous phase to excite a dipolar parametric resonance, effectively depleting the bunch core in longitudinal phase space. Figure 4 shows the function for phase loop offset modulation. The same function was implemented in the simulation to reproduce the dynamics during the formation of those observed in the experiments (Fig. 5).

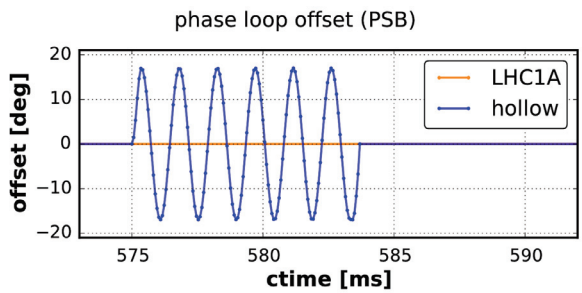

Figure 4: Trim function for the phase loop offset (blue curve).

The advantage provided by the modular design of PyHEADTAIL was exploited when interfacing PyHEADTAIL with PyPIC and PyECLOUD [20] to include the treatment of electron-cloud effects. PyPIC includes a set of Poisson solvers based on the particle-in-cell (PIC) algorithm including, for example, FFT methods using integrated Green's functions or finite-difference time-domain (FDTD) solvers which are able to handle complex boundaries using Shortley-Weller stencils. PyECLOUD is a build-up simulation code to simulate the formation of electron-clouds though multipacting. It therefore contains, among others, a sophisticated multipacting model and very detailed implementation of

ISBN 978-3-95450-178-6

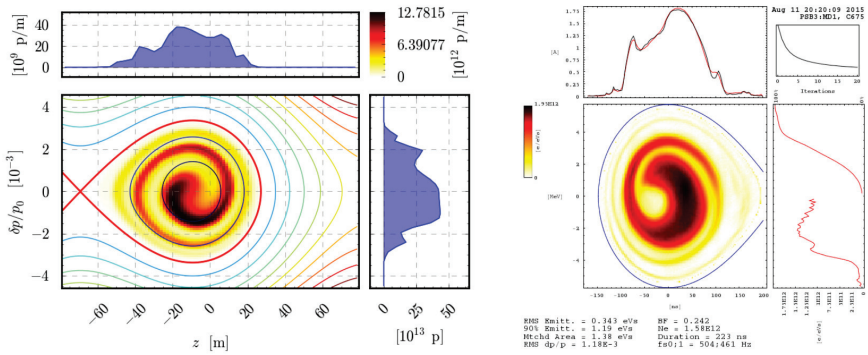

Figure 5: Comparison of the longitudinal phase spaces as obtained in simulation and measured in the experiments. Note that the horizontal axis on the left plot indicated position whereas on the right plot it indicates time.

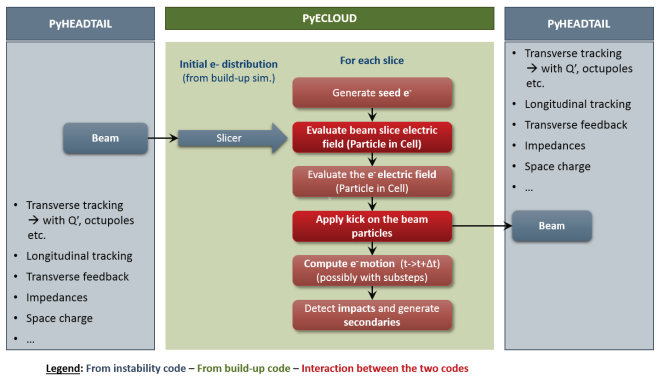

Figure 6: A graphical illustration of how PyHEADTAIL interacts with PyPIC and PyECLOUD.

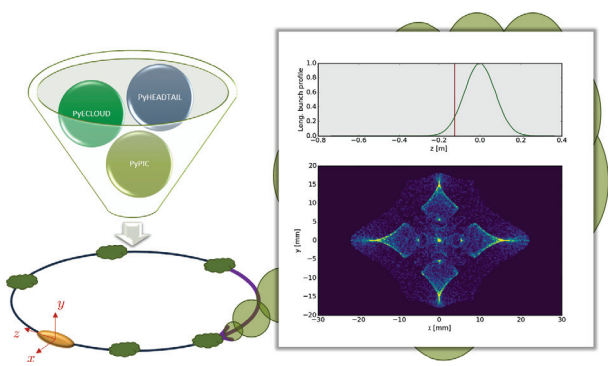

Figure 7: Snapshot of a simulation including electron clouds in quadrupoles. The plot shows the electron density distribution in an LHC vacuum chamber.

the electron dynamics within magnetic fields. PyPIC and PyECLOUD have been widely used in the past to study the formation of electron clouds in different machines with their specific beams and has also been benchmarked against experiments [21]. To include electron cloud effects to study them as collective effects it is therefore natural to reuse all the existing features of the two projects. Thanks to the orthogonal design of PyHEADTAIL no additional knowledge of any of the other modules was necessary for the successful integration of PyPIC and PyECLOUD. This made the inclusion of an entire separate code seamless and drastically reduced the probability of introducing any bugs in any of the working routines. Figure 6 illustrates how the different parts of the codes interact with each other.

Equipped with this additional feature, PyHEADTAIL was used to study electron cloud instabilities in different vacuum chambers with different magnetic field configurations for the Super Proton Synchrotron (SPS) at CERN. It is also being used to study the beam stability in presence of electron clouds in the quadrupoles of the LHC and the HL-LHC, where the high beam energies generate very fast cyclotron motion of the electrons which makes the correct modeling of the electron motion numerically very challenging. Figure 7 shows a snapshot of such a simulation.

Finally, PyHEADTAIL has been partly ported to the GPU using PyCUDA [22]. Following the usual philosophy, this was performed in the least invasive manner possible. To keep the interface unchanged at the user level, an abstraction layer was added in the form of a context manager. This context manager handles all dispatches of the different function calls via the context to the matching platform. That way, the performance of the GPU can be exploited when possible without having to pollute any of the existing programs or scripts.

Efforts are ongoing to include parallelisation at different levels using multi-processing via mpi4py in order to boost

ISBN 978-3-95450-178-6

performance for multi-bunch and electron cloud instability simulations.

REFERENCES

[1] "Python", https://python.org/, Jun. 2016.

[2] E. Métral et al.: "Beam Instabilities in Hadron Synchrotrons", in *IEEE Transactions on Nuclear Science*, vol. 63, no. 2, Apr. 2016, pp. 1001-1050.

[3] G. Rumolo and F. Zimmermann: "Practical user guide for HEADTAIL", *SL-Note-2002-036-AP*, Nov. 2002.

[4] "MAD – Methodical Accelerator Design", mad.web.cern.ch, Jun. 2016.

[5] "Die Finite-Integrations-Theorie und das Programmpaket MAFIA",temf.tu-darmstadt.de,Jun.2016.

[6] "SciPy", scipy.org, Jun. 2016.

[7] T. Oliphant, "Passing the torch of NumPy and moving on to Blaze", technicaldiscovery.blogspot.ch, Jun. 2016

[8] "Simplified Wrapper and Interface Generator", swig.org, Jun. 2016.

[9] D. Abrahams and S. Seefeld, "Boost.Python", boost.org, Jun. 2016.

[10] P. Peterson, "F2PY: Fortran to Python interface generator", sysbio.ioc.ee/projects/f2py2e, Jun. 2016.

[11] "Cython C-Extension for Python", cython.org, Jun. 2016.

[12] "Fast numerical expression evaluator for NumPy", pypi.python.org/pypi/numexpr, Jun. 2016.

[13] "Numba", numba.pydata.org, Jun. 2016.

[14] K. Li, "cobra-HeadTail", 95th ICE section Meeting, Nov. 2013.

[15] "The OpenMP API specification for parallel programming", openmp.org, Jun. 2016.

[16] "MPI for Python", bitbucket.org/mpi4py/mpi4py, Jun. 2016.

[17] A. Klöckner, "PyCUDA", mathema.tician.de/software/pycuda, Jun. 2016.

[18] "NVIDIA CUDA 8", developer.nvidia.com/cuda-zone, Jun. 2016.

[19] A. Oeftiger, "Space Charge Mitigation With Longitudinally Hollow Bunches", *in proc.* HB2016, paper MOPR026.

[20] "(Py)thon (Co)llective (M)acro-(P)article Simulation (L)ibrary with (E)xtendible (T)racking (E)lements", github.com/PyCOMPLETE, Jun. 2016.

[21] G. Iadarola, "Electron cloud studies for CERN particle accelerators and simulation code development", CERN-THESIS-2014-047, 2014.

[22] S. Hegglin, "Simulating Collective Effects on GPUs", MSc thesis, ETH Z¨urich, Switzerland, 2016.

ISBN 978-3-95450-178-6