

# JSC Guest Student Programme Proceedings 2021

Ivo Kabadshow (Ed.)

IAS Series

Band / Volume 52

ISBN 978-3-95806-684-7





Forschungszentrum Jülich GmbH  
Institute for Advanced Simulation (IAS)  
Jülich Supercomputing Centre (JSC)

# **JSC Guest Student Programme Proceedings 2021**

Ivo Kabadshow (Ed.)

Schriften des Forschungszentrums Jülich  
IAS Series

Band / Volume 52

---

ISSN 1868-8489

ISBN 978-3-95806-684-7

Bibliografische Information der Deutschen Nationalbibliothek.  
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der  
Deutschen Nationalbibliografie; detaillierte Bibliografische Daten  
sind im Internet über <http://dnb.d-nb.de> abrufbar.

Herausgeber  
und Vertrieb: Forschungszentrum Jülich GmbH  
Zentralbibliothek, Verlag  
52425 Jülich  
Tel.: +49 2461 61-5368  
Fax: +49 2461 61-6103  
zb-publikation@fz-juelich.de  
[www.fz-juelich.de/zb](http://www.fz-juelich.de/zb)

Umschlaggestaltung: Grafische Medien, Forschungszentrum Jülich GmbH

Titelbild: Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH

Druck: Grafische Medien, Forschungszentrum Jülich GmbH

Copyright: Forschungszentrum Jülich 2023

Schriften des Forschungszentrums Jülich  
IAS Series, Band / Volume 52

ISSN 1868-8489  
ISBN 978-3-95806-684-7

Vollständig frei verfügbar über das Publikationsportal des Forschungszentrums Jülich (JuSER)  
unter [www.fz-juelich.de/zb/openaccess](http://www.fz-juelich.de/zb/openaccess).



This is an Open Access publication distributed under the terms of the [Creative Commons Attribution License 4.0](https://creativecommons.org/licenses/by/4.0/),  
which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

**GPU**  
 cuda  
 run pack  
 library particles  
 nanotubes  
 projects  
 forward example  
 control  
 outer result  
 api  
 jsc  
 dimensions  
 algorithm  
 atoms  
 speedup  
 created  
 scattering  
 nvidia tree  
 kernels  
 graphene  
 carbon  
 matrix  
 play  
 centre  
 codes  
 original  
 section  
 window  
 structure  
 call  
 operations  
 computation

**Scatter**  
 games  
 graphs  
 single  
 kernel  
 stream  
 systems  
 algorithms  
 computer  
 block  
 based  
 games  
 check running  
 size  
 power  
 introduction  
 properties  
 means  
 steps  
 application  
 level  
 oppamp  
 hand  
 parallel  
 density  
 cells  
 bcs  
 materials  
 initial  
 ray  
 phase  
 quantum  
 mpi  
 streams  
 left  
 following  
 parameters  
 science  
 minimax  
 mode  
 accessed  
 method  
 files  
 zero  
 efficient  
 radiative

**code**  
 task  
 graph  
 tensor  
 program  
 implementation  
 move  
 functions  
 lattice  
 void  
 memory  
 application  
 wall  
 jlich  
 solve  
 inner  
 difference  
 hence  
 test  
 report  
 mps  
 timestep  
 wave  
 child  
 step  
 bolzmann  
 models  
 do  
 primary  
 differences  
 system  
 listing  
 communication  
 october  
 term  
 python  
 world  
 calls  
 walberla  
 node  
 kkrnano  
 nsight  
 create  
 bond  
 total  
 prepare  
 version  
 research  
 seen  
 range  
 main

**jurassic**  
 time  
 model  
 function  
 data  
 performance  
 queue  
 value  
 project  
 structures  
 nsf  
 tools  
 future  
 gpus  
 solving  
 type  
 toe  
 space  
 methods  
 execution  
 added  
 forward  
 invariance  
 implemented  
 similar  
 clone  
 ai  
 player  
 max  
 cu  
 computational  
 cloud  
 finally  
 values  
 solid  
 physics  
 site  
 implemented  
 tfqmr gpu  
 array  
 linear  
 simulation  
 file  
 source domain  
 information  
 tac  
 benchmark  
 org  
 collision  
 board  
 nodes  
 theory  
 nodes  
 common  
 git hub

# EDITORIAL

The Jülich Supercomputing Centre (JSC) provides HPC infrastructure of the highest class and promotes the education of young scientists willing to enter the HPC domain. Our ten-week guest student program offers interested students the opportunity to work within one of the world's most powerful HPC environments. Within this programme, students with a major in natural sciences, engineering, computer science or mathematics get the opportunity to familiarize themselves with different aspects of scientific computing. Together with local scientists, the participants work on a large range of different topics in research and development. A special emphasis in the training and project selection is placed on the use of modern supercomputers. Depending on previous knowledge and on the participant's interest, the assignment can be chosen out of different areas. These fields include mathematics, physics, chemistry, neuroscience, software development tools, artificial intelligence and distributed computing.













The JSC Guest Student Programme has already been successfully running for 22 years. Since the first programme in 2000, a total of 233 students have seized the opportunity to join research teams from JSC on the Forschungszentrum Jülich campus each summer. Working on challenging scientific projects, they gained experience with modern hardware and software as well as HPC-related methods and algorithms. For many students, the programme has been the foundation for a career in HPC and the basis for fruitful continuing cooperations.

The JSC Guest Student Programme 2021 took place from August 2<sup>nd</sup> to October 8<sup>th</sup>. Once again it was run with support from CECAM (Centre Européen de Calcul Atomique et Moléculaire). It targeted students who have already completed their first degree but have not yet finished their master's course.

Since an on-premise hosting due to COVID was not possible, 6 students were accepted to work remotely for a project in Jülich. This publication summarized the findings of these research projects.

*Ivo Kabadshow*  
Jülich, December 2022

# CONTENTS

 <i>Rahul Mananvalan</i>	
 High Performance Tensor Contraction Algorithms in Tensor Networks. . . .	1
 <i>Cem Oran</i>	
 Accelerating KKRnano . . . . .	15
 <i>Anastasia Papadaki</i>	
 NSL(Nanosystem Simulation Library) . . . . .	24
 <i>Stjepan Požgaj</i>	
 JURASSIC-scatter-GPU . . . . .	34
 <i>Mert Saner</i>	
 Game Tree Implementations Using Python . . . . .	48
 <i>Milena Veneva</i>	
 GPU-Based Optimizations for WALBERLA . . . . .	59



# HIGH PERFORMANCE TENSOR CONTRACTION ALGORITHMS IN TENSOR NETWORKS.

## An Investigation.

Rahul Manavalan  
Computational Science and  
Engineering,  
Technical University of  
Munich,  
Germany  
rahul.manavalan@tum.de

**Abstract** *Tensor Networks have been used to study strongly interacting quantum materials. Understanding the physics of quantum materials is becoming increasingly important to engineer such materials. However, there are unanswered questions regarding strongly interacting quantum systems such as high temperature superconductivity. As a result, there is an impending need for faster algorithms that allow us to simulate more materials in greater detail. This Investigation is one that tries to adopt a high performance tensor contraction library to Tensor Network applications, with the aim of investigating the speedup that one obtains with and without the use of this library. Preliminary investigations would suggest that it is of advantage to port TN implementations to said high performance library.*

## 1. INTRODUCTION

An increase in the computational resources and progress in modern physics has always been hand in hand. By making use of mathematical models and computer simulations, hypotheses have been tested and corrected at an accelerated pace. Until early 2005, clock speed of processors increased almost with every new release. This meant that scientists could simply wait for the next generation of processors for their codes to become faster and in turn refine the parameters of their computer simulations. However with the end of Dennard scaling, this is no longer true. Modern scientific software must make use of not only the inherent parallelism that the present day microarchitectures offer, but also be amenable to execution on accelerators such as GPUs and by extension on distributed architectures. In this article, we investigate the viability of high performance tensor contraction algorithms to Tensor Network applications.

### 1.1. TENSOR NETWORKS

Understanding quantum many-body systems is probably the most challenging problem in condensed matter physics.[6] One explanation for this difficulty is the exponential size of the quantum wave function even for moderately sized systems. This follows that the wavefunction  $|\psi\rangle$  cannot be efficiently represented on a computer. As a result, alternative compressed and parametrized representations of the wavefunction is necessary. Tensor Networks (henceforth TN) is one of the many Ansatzes, which extends the range of models that can be simulated with a classical computer in new and unprecedented

directions.[6]

We shall attempt build on some of the building blocks for doing quantum mechanics with TN. Let us start of with representing something as fundamental as the quantum wave function  $\psi$  and then explore how one might represent a Hamiltonian.

### 1.1.1. WAVE FUNCTION

The quantum wave function is an integral part of performing quantum mechanical calculations for any system. Pragmatically,one should regard this as a mathematical entity, although there have been attempts to interpret it.[2]

If we would like to represent a  $n$  body 1/2 system.

$$|\psi\rangle \in \mathbb{C}^N \text{ where } N = 2^n$$

$|\psi\rangle$  is mathematically a  $n$ -way tensor, whose graphical representation is found on the left of Figure 1.1. In order to make this representation tractable on a computer, we could prescribe to the following trick.(Figure 1.1)

1. Isolate index 1 and combine the rest of the indices. This results in a matrix  $M$ .
2. Perform singular value decomposition on the matrix  $M = U\Sigma V'$ .
3. Assign  $A = U\Sigma$
4. Repeat 1 for all remaining indices on  $V$ .

One can compress the representation of  $|\psi\rangle$  by choosing the first  $k$  singular values in the decomposition.

### 1.1.2. HAMILTONIAN

A similar obstacle for QM calculations is the size of the Hamiltonian  $H$ . If we stick to our 1/2 system with  $n$  bodies.

$$H \in \mathbb{C}^{N \times N} \text{ where } N = 2^n.$$

Solving the eigenproblem to such a matrix  $H$  is computationally intensive and in some cases impractical. As a result, we require a more efficient representation of the Hamiltonian. One method that is prevalent in TN algorithms is to encode the Hamiltonian locally using a Finite State Automata approach.[3] Figure 1.2 denotes graphically how the global Hamiltonian could be represented using a train of Local Hamiltonians.

With the aid of these two representations, we are in a position to understand most of what would follow.

## 1.2. TENSOR CONTRACTION

Now we enter the computational part of our discussion. As one would realize,a recurring theme of TNs is Tensor Contraction. Formally, Tensor Contraction (henceforth TC) is a generalization of the trace of a Matrix. However this operation is often recast in the form of a Matrix Multiplication, preferably using some highly optimized BLAS routines

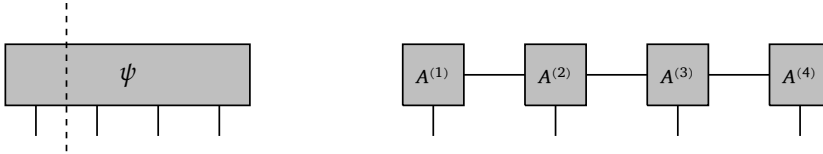


Figure 1.1.:  $|\psi\rangle$  and corresponding MPS state

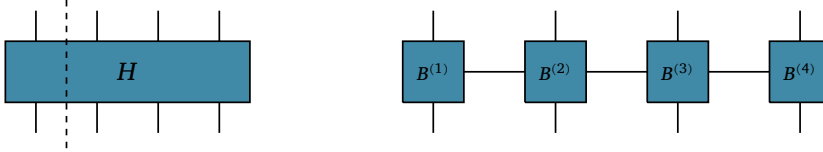


Figure 1.2.:  $H$  and corresponding MPO

such as `dgemm` or `zgemm`. There are multitudes of methods for performing tensor contraction. The range of algorithms include those that make use of BLAS, those that are BLAS independent, those that contract two tensors at a time, those that contract multiple tensors at a time etc. It goes without saying that all of these implementations have optimized versions as well for accelerators and distributed memory architectures.

Some common algorithms for TC are

1. LoG := Loop of GEMMs
2. TTGT := Transpose Tranpose GEMM Transpose
3. GETT := GEMM like Tensor Contraction

We abstain from discussing the details of these methods, however one may find [8] interesting.

## 2. PREVIOUS WORK

Tensor contraction is a complex operation often involving several sub-computations like Transposition, GEMM etc. Owing to its complexity, development of an efficient implementation of such a kernel is laborious. Fortunately, there are several prior implementations to choose from [7]. For the sake of our investigation, we adopt high performance tensor transposition compiler library **HPPT** [10] and high performance tensor contraction library **TCL** [9] owing to its proven benchmarks. TCL makes use of HPPT, to realize increased performance. During this investigation we make use of the C++ and Python interfaces that TCL has at its disposal. It is a given that, for the rest of the discussion all TCs are performed using either direct calls to TCL or the corresponding interfaces.

### 3. PROBLEM DESCRIPTION

We are interested to determine whether the aforementioned compiler library-TCL provides sufficient speedup over the existing implementations, such that a reimplementa-tion of TN algorithms using TCL is warranted.

### 4. APPROACH AND RESULTS

Tensor Networks have a diverse portfolio. Therefore for the sake of this investigation, we focus on MPS algorithms. Ideally, one would reimplement the entire MPS suite of algorithms using TCL for performance measurements. An alternative approach would be to identify recurring kernels and measure their performance instead. Further investigation reveals two groups of algorithms are responsible for large portions of computations.

#### 4.1. DMRG KERNELS

Determining the ground state of a quantum system is a recurring problem in condensed matter physics. The MPS algorithm for determining the ground state is called the DMRG (Density Matrix Renormalization Group) algorithm. DMRG is a variational algorithm. It optimizes the MPS train such that the energy associated with the state becomes a minimum. Figure x illustrates in TN notation, the algorithm for DMRG. Evidently, one observes three sub-kernels in the DMRG algorithm that repeat in the entire train of computation.

- UpdateL
- UpdateR
- Update HEff

##### 4.1.1. UPDATEL

Mathematically this kernel comprises of contraction among three 3-way tensors and one 4-way tensor. It is interesting to note that the order in which the tensors are contracted determines the complexity of the computation. Therefore, we distinguish between 6 different contraction sequences namely:

1. 0123
2. 0132
3. 0213
4. 0231
5. 0312
6. 0321

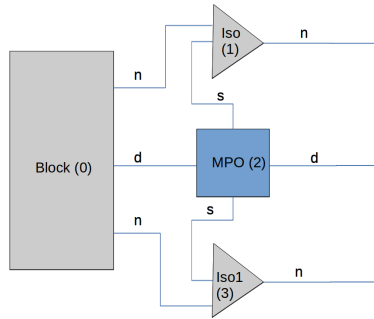


Figure 4.1.: UpdateL  $n$ : = bond dims,  $d$ : = Size of the local H,  $s$ : = DOF

To clarify what these permutations mean. "0123" refers to contraction of Block(0) and Iso(1); followed by contraction with MPO(2); finally contracting this result with Iso1(3). An implementation of all these permutations of this kernel using **numpy** is realized. Later, they are re-implemented using **tcl**.

#### 4.1.2. UPDATEL - RESULTS

We perform test using the following parameters:

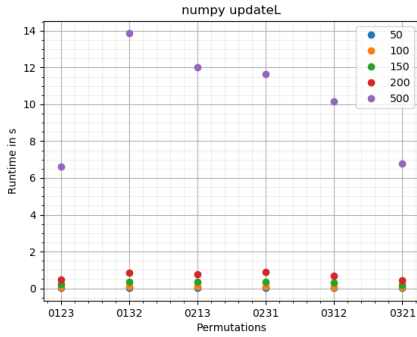
- $s = 2$
- $d = 5$
- $n = [50, 100, 150, 200, 500]$

There are two aspects from Figure 4.2 that are significant. Firstly, in Figure 4.2 a), we validate that different permutations take different times for computation. Secondly, in Figure 4.2 b) we see that the TCL implementation brings down the runtime significantly. More interesting is the fact that the TCL implementation has different coefficients when one regards complexity. It is certainly worth exploring why a discrepancy in the coefficients are seen.

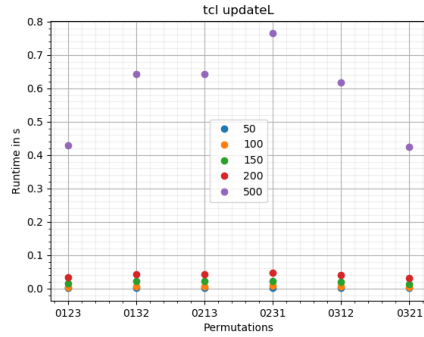
In Figure 4.3, we see the overall gains that are as a consequence of using the high performant implementation. In a) one sees in the same scale how the runtimes vary for different permutations of this kernel. In b) the actual speedup is observable. In summary we were able to observe a speedup of approximately 15x in the best case scenario.

#### 4.1.3. UPDATER

Perhaps the first observation that one may make from Figure 4.4 is that it is a reflection of UpdateL. However one should note that the memory access patterns for the two computations are significantly different. It is due to this, that we devote an entire section for this computation. While the approach to measuring the performance of the two types is the same, we do not expect similar results.

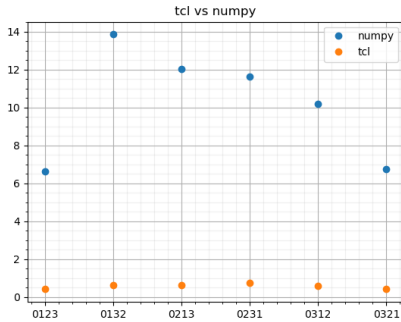


(a) *numpy*

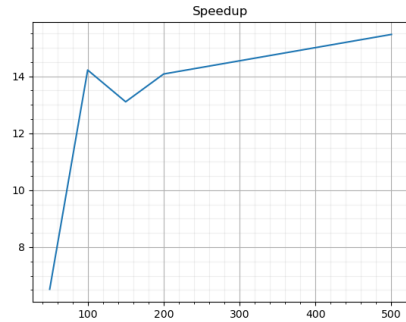


(b) *tcl*

Figure 4.2.: UpdateL with  $s = 2$   $d = 5$



(a) Comparison@ $n = 200$



(b) Speedup with bond dims -  $n$

Figure 4.3.: Speedup

#### 4.1.4. UPDATER - RESULTS

We perform test using the following parameters:

- $s = 2$
- $d = 5$
- $n = [50, 100, 150, 200, 500]$

The results to UpdateR has a similar layout to that of UpdateL. As one would observe the runtime even for the same permutations are different with respect to UpdateL. Aside from this fact, the speedups that one observes is quite similar and is approximately 15x for the best case complexity.

#### 4.1.5. HEFF

HEff is a made up abbreviation for updating the effective Hamiltonian on the lattice sites. As Figure x, would suggest the central part of the DMRG kernel comprises of contracting

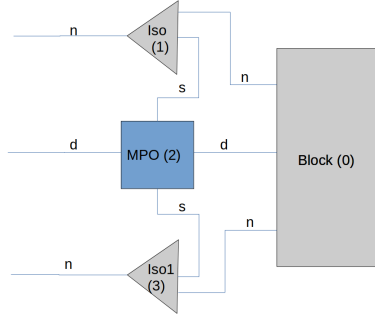


Figure 4.4.: UpdateR  $n$ : = bond dims,  $d$ : = Size of the local H,  $s$ : = DOF

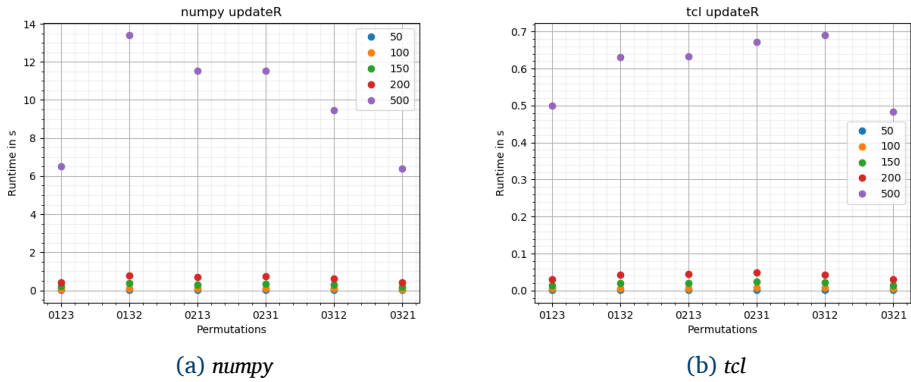


Figure 4.5.: UpdateR with  $s = 2$   $d = 5$

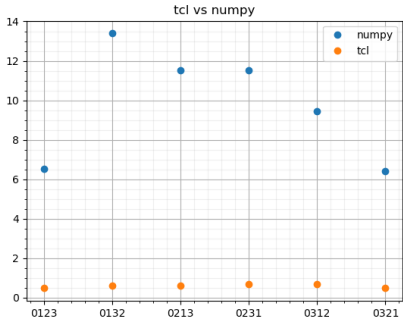
two combined blocks with a MPS,MPO pair. We take a similar approach for measuring the speedup.

#### 4.1.6. HEFF - RESULTS

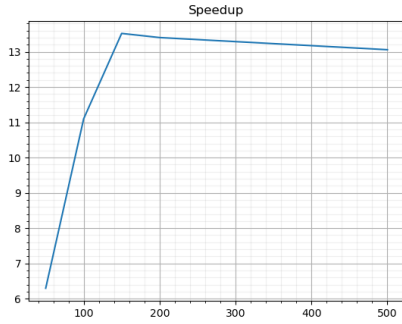
We perform test using the following parameters:

- $s = 2$
- $d = 5$
- $n = [50, 100]$

We had to restrict ourselves to smaller bond dimensions due to memory insufficiency that arises when one tries to contract blocks of higher dimensions. For instance, assume  $n = 150$  then one such contraction pattern would result in an intermediate tensor of dimensions  $150 \times 150 \times 2 \times 2 \times 150 \times 150$ , which makes it a clear exclusion. However for a



(a) Comparison@n = 200



(b) Speedup with bond dims - n

Figure 4.6.: Speedup

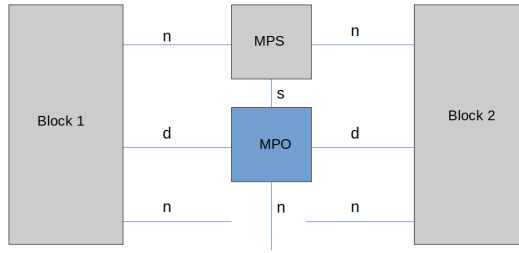
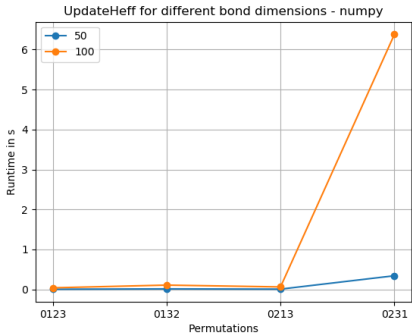
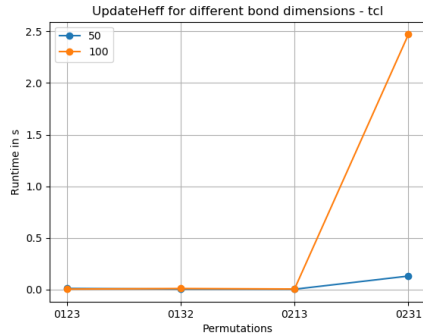


Figure 4.7.: Update H Effective n: = bond dims, d: = Size of the local H, s: = DOF



(a) numpy

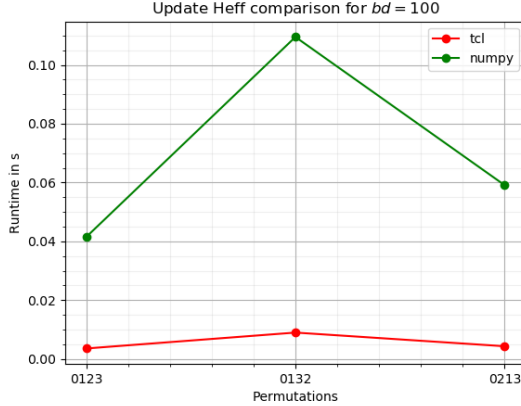


(b) tcl

Figure 4.8.: HEff with  $s = 2$   $d = 5$

lack of time, we are unable to prune out permutations of that nature. Hence the smaller sample size for the computation.





(a) Comparison at  $n = 100$

Figure 4.9.: Comparison

For preserving the homogeneity of the scale of the measurements, one of the permutations has been thrown out. This is justified, as this makes no real difference to the measurement of the speed up that we are interested in. It is of interest to the PIs, that higher bond dimensions are explored to ensure that one achieves scaling with respect to the bond dimension.

## 4.2. ISOMETRIZATION KERNELS

We begin with the definition of an Isometry.

$$\text{Let } U \in \mathbb{C}^{m \times n}$$

$$\text{If } U^*U = I$$

Then  $U$  is called an isometry.

Isometrization is important in the MPS context as it facilitates the conversion of  $A_i$  to an isometry. This inturn has advantages both in terms of reduction of computational effort and improving the stability of certain class of problems that can be solved with MPS. One should take note that a Matrix can be right isometric and left isometric depending on the order of contraction. In this section we explore the computational time that it takes for isometrization on a single site. If one can prove an improvement in the performance for once site, this naturally translates to the rest of the MPS train.

### 4.2.1. LEFT GAUGE TRANSFORM

Left of Figure 4.10 shows the input to the routine. This routine converts the local MPS tensor  $A_i$  to a left isometry as follows:

- 1) The input  $R (R_{i-1})$  is contracted with  $A_i$  i.e.  $K = R_{i-1} * A_i$
- 2) Perform QR decomposition on  $K$ . i.e.  $K = U_i * R_i$  (Right of Figure x)
- 3) Return  $R$

This way one can recursively orthonormalize the entire MPS train.

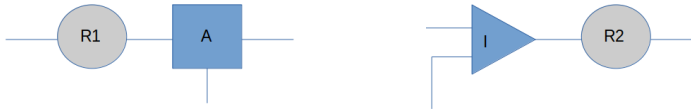


Figure 4.10.: Left Gauge Transform kernel

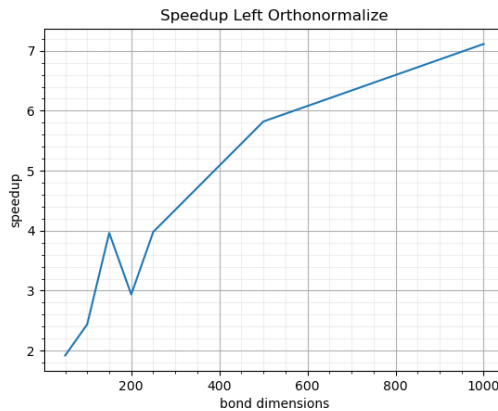


Figure 4.11.: Speedup Left orthonormalize

#### 4.2.2. RESULTS - LEFT GAUGE TRANSFORM

We test the kernel for the following parameters

- $s = 2$
- $d = 5$
- $n = [50, 100, 150, 200, 250, 500]$

As it has been for the previous kernels, we perform measurements in numpy and then the tcl version of the kernel. Figure 4.11 depicts that for higher bond dimensions we observe a speedup of approx 7x.

#### 4.2.3. RIGHT GAUGE TRANSFORM

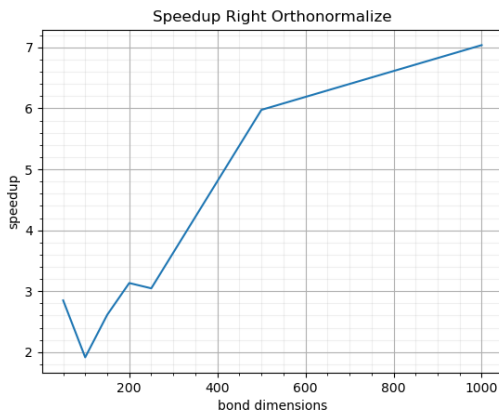
Under right gauge transformation one realized the isometrization using a similar algorithm. The fundamental difference arises in the fact that one starts at the right of the MPS train. In addition one is interested in performing the RQ decomposition as opposed to the QR decomposition in Left Gauge transform.

#### 4.2.4. RESULTS - RIGHT GAUGE TRANSFORM

We test the kernel for the following parameters



Figure 4.12.: Right Gauge Transform kernel



(a) Speedup Right orthonormalize

Figure 4.13.: Speedup Left orthonormalize

- $s = 2$
- $d = 5$
- $n = [50, 100, 150, 200, 250, 500]$

A striking feature about this kernel is that we perform RQ decomposition. This involves transposing the tensor prior to decomposing it. It is of special disadvantage to make use of numpy ecosystems' default transpose function. This slows down the computation significantly. On the otherhand tcl and its highly optimized version of the HPTT code performs better. If we port the implementation to numpy's einsum function, then results similar to that of the Isometrize L is obtained.

## 5. ADDITIONAL WORK

On the account of the positive results, the next step would be to develop MPS suite of algorithms - HPTN that makes use of contraction realized using TCL. In that vein, we started the construction of the initial building blocks necessary for HPTN. The outline for the same can be found in Figure 5.1.

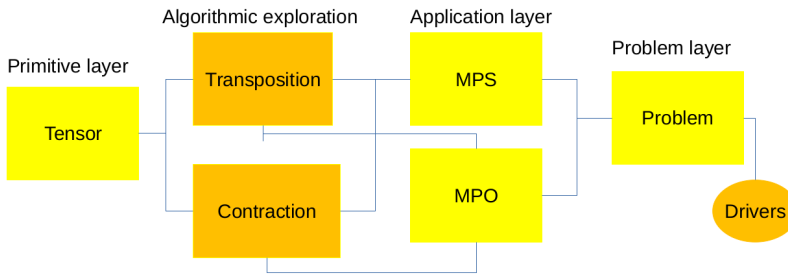


Figure 5.1.: Outline of current HPTN code

## 5.1. COMPUTATION LAYER

Including the Primitive layer and Algorithmic Exploration layer (Fig.5.1), this forms the core of the computation. There are a vast array of tensor contraction and tensor transposition algorithms that are designed for different architectures. By abstracting out the software stack into a computation layer that implements these algorithms, performance of tensor network computations on a wide spectrum of devices could be realized. Of special interest to the PIs, include implementation in GPUs and distributed devices. This is a course that we lay now for future work.

## 5.2. APPLICATION LAYER

While Figure 5.1 alludes to mere MPS and MPO implementations, there are more TN algorithms appropriate for several applications that can be implemented. This includes algorithms for 2D problems such as PEPS[6],MERA[1]. These are also under consideration for future implementations.

## 5.3. PROBLEM LAYER

Finally, it is worth reminding that TN approximations are especially useful for solving problems of strongly correlated quantum systems. In that respect, it would be interest to solve a complete problem and test the relative performance with respect to other TN libraries. Questions on the choice of problem and the appropriate libraries to make the comparisons are still unresolved.

## 6. CONCLUSION

We summarize the following from the results of the experiments.

- DMRG kernels scale well with increasing bond dimensions, suggesting a clear advantage.
- MPS Orthonormalization does not result in as significant an improvement with respect to DMRG routines.

It is safe to infer that a TCL implementation allows us to explore higher bond dimensions on a single node machine. So an obvious recommendation is to reimplement TN algorithms using TCL. One obvious advantage such an endeavour would have, is the significant speedup that it might have over TN libraries that are developed in python and make use of the numpy ecosystem. Some examples of these libraries include PyTeNet[5] and TenPy[4].

However it remains to be proven, if other optimized TN implementations such as ITensors and the like are better. This question can only be resolved with more testing and benchmarking, which we plan to pursue after the end of the guest student programme.

## 7. ACKNOWLEDGMENTS

I would like to thank my advisors Dr. Edoardo Di Napoli and Dr. Matteo Rizzi for dedicating their time in advising and reviewing my work over the course of the summer. In addition, I would also like to thank Dr. Ivo Kabadshow for his lectures and guidance with the reports and presentations.

## REFERENCES

- [1] J. Biamonte and V. Bergholm. **Tensor networks in a nutshell**, 2017. [arXiv:1708.00006](https://arxiv.org/abs/1708.00006).
- [2] C. Blood. **A primer on quantum mechanics and its interpretations**, 2010. [arXiv:1001.3080](https://arxiv.org/abs/1001.3080).
- [3] J. Hauschild and F. Pollmann. **Efficient numerical simulations with tensor networks: Tensor network python (tenpy)**. *SciPost Physics Lecture Notes*, Oct 2018. <http://dx.doi.org/10.21468/SciPostPhysLectNotes.5>, [doi:10.21468/scipostphyslectnotes.5](https://doi.org/10.21468/scipostphyslectnotes.5).
- [4] J. Hauschild and F. Pollmann. **Efficient numerical simulations with Tensor Networks: Tensor Network Python (TeNPy)**. *SciPost Phys. Lect. Notes*, page 5, 2018. Code available from <https://github.com/tenpy/tenpy>. <https://scipost.org/10.21468/SciPostPhysLectNotes.5>, [arXiv:1805.00055](https://arxiv.org/abs/1805.00055), [doi:10.21468/SciPostPhysLectNotes.5](https://doi.org/10.21468/SciPostPhysLectNotes.5).
- [5] C. B. Mendl. **Pytenet: A concise python implementation of quantum tensor network algorithms**. *Journal of Open Source Software*, 3(30):948, 2018. [doi:10.21105/joss.00948](https://doi.org/10.21105/joss.00948).
- [6] R. Orús. **A practical introduction to tensor networks: Matrix product states and projected entangled pair states**. *Annals of Physics*, 349:117–158, Oct 2014. <http://dx.doi.org/10.1016/j.aop.2014.06.013>, [doi:10.1016/j.aop.2014.06.013](https://doi.org/10.1016/j.aop.2014.06.013).
- [7] C. Psarras, L. Karlsson, J. Li, and P. Bientinesi. **The landscape of software for tensor computations**, 2021. [arXiv:2103.13756](https://arxiv.org/abs/2103.13756).
- [8] S.-J. Ran, E. Tirrito, C. Peng, X. Chen, L. Tagliacozzo, G. Su, and M. Lewenstein. **Tensor network contractions**. *Lecture Notes in Physics*, 2020. <http://dx.doi.org/10.1007/978-3-030-34489-4>, [doi:10.1007/978-3-030-34489-4](https://doi.org/10.1007/978-3-030-34489-4).

- [9] P. Springer and P. Bientinesi. **Design of a high-performance gemm-like tensor-tensor multiplication.** ACM Transactions on Mathematical Software (TOMS), 44(3):28:1–28:29, January 2018. 🌐 <https://arxiv.org/pdf/1607.00145.pdf>.
- [10] P. Springer, T. Su, and P. Bientinesi. **Hptt: A high-performance tensor transposition C++ library.** In Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, ARRAY. ACM, June 2017. 🌐 <https://arxiv.org/pdf/1704.04374.pdf>.

# ACCELERATING KKRnANO

## A Green Function Based Electronic Structure Code

**Abstract** *Even though quantum mechanics gives a comprehensive mathematical description of the atomic world, in practice most of the existing precise electronic-structure codes based on density functional theory (DFT) are applicable for systems containing at most several hundred atoms as a result of enormous computational costs. Therefore, developing both accurate and computationally cheap methods is a hot topic in today's materials science. KKRnano is an electronic structure code which provides enormously parallel density-functional calculations for large systems with thousands of atoms. With the power of GPU-accelerated parallelism, KKRnano provides solutions for systems with complex defects or large nanostructures which are beyond the limit that classical DFT methods can reach. This study is dedicated to implement a GPU library, `tFQMRgpu`, into the KKRnano code in order to accelerate the Dyson equation solver. In this report, a brief overview about quantum mechanical calculations, the KKRnano code and the `tFQMRgpu` library is given. Results of preliminary performance tests of KKRnano for CPU and GPU parallelism are presented.*

Cem Oran  
Computational Science and  
Engineering  
Istanbul Technical  
University  
Turkey  
cemoran.01@gmail.com

## 1. INTRODUCTION/MOTIVATION

Discovering new materials with desired properties is one of the most prominent research fields in material science. In this manner, large databases collecting information about properties of various materials are the main resource to guide scientists in their research. The traditional way of obtaining such data is doing experiments in laboratories and for a long time, the data required for research have been obtained mostly from experimental studies. However, improvements during the last 15 – 20 years in both, the application of theoretical framework to algorithms and cheaper high performance computation resources, made calculations of huge amounts of material properties in computer environment feasible [5]. Especially, the development of practical and scalable DFT (Density Functional Theory) algorithms allowed accurately predicting the electronic properties of crystal structures from first principles. These developments deduced a paradigm shift in the materials science towards computation based data-driven approach. Today, large online materials databases which are established on the top of decades long dedicated experimental and computational works represent the backbone of this new paradigm.

Quantum mechanics gives comprehensive mathematical description of the atomic world. Even though the theory is capable of providing explanation of the behavior of electrons, it is not possible to obtain the exact solution of chemical systems bigger than the hydrogen atom. As the chemical system grows, the system of mathematical equations gets forbiddingly more complex. Therefore, it can be said that the approximate solutions

have a very crucial role in computational quantum mechanics.

There are various methods widely used in quantum mechanical calculations, and each of them has pros and cons. On the other hand, those quantum mechanical methods, in general, are far too expensive to be used for the systems containing several hundred atoms. Therefore, developing both accurate and computationally efficient (linear scaling) methods is an active research field today. In this manner, the linear-scaling all-electron DFT implementation KKRnano is an efficient tool in order to deal with such large systems. Moreover, the computational performance of the KKRnano code can be increased even more by empowering it with GPU parallelism. This study is dedicated to improve the computational performance of KKRnano by implementing the GPU library `tfQMRgpu`. In the following sections, a brief introduction is given about some of the popular methods for quantum mechanical calculations. Then, KKRnano with implementation of the GPU library `tfQMRgpu` is presented. Finally, the effect of this implementation on the performance is showed for crystal systems with various sizes.

## 2. AB-INITIO ASPECT AND DFT

The term 'ab-initio' corresponds to 'from the beginning' in Latin. This term means that a computation is directly based on theoretical principles without contribution of experimental findings. Because, in terms of quantum mechanics, exact theoretical solutions are not possible for systems larger than a hydrogen atom, ab-initio calculations mainly deal with obtaining an approximate solution of the time-independent Schrödinger Equation ( $H \cdot \Psi = E \cdot \Psi$ ) for the system of interest. However, pure ab-initio methods in general, are sufficiently accurate but expensive. For example, the Hartree-Fock (HF) method is a fundamental ab-initio method in which Coulombic electron-electron repulsion is taken as an average integrated effect (each electron is affected by the presence of the other electrons indirectly through the mean field created by them). As a method, it lacks the electron correlation in its description of electron-electron interactions, and it scales as the forth power of the number of basis functions ( $N_b^4$ ) [2]. Some more elaborated methods including electron correlation corrections scale considerably harsher than this. Therefore, these methods are only considered as suitable for relatively small sized atomic systems.

On the other hand, DFT is a very popular and strong alternative to pure ab-initio methods. In a typical DFT code, calculations regarding eigenvalue problem scales as the third power of the number of basis functions ( $N_b^3$ ) and it provides a similar level of accuracy compared to some of the correlation corrected ab-initio methods [2]. The theoretical foundation of DFT was established by Hohenberg and Kohn while the first practical application was advanced by Kohn and Sham. The idea behind DFT is that electron density ( $\rho$ ) can be used as primary feature to predict any observable property of a chemical system, contrary to ab-initio methods doing this via a wave function. The Hamiltonian depends on positions and atomic numbers of the nuclei and the total number of electrons,  $N_e$ . Nuclei are assumed as point charges [2].  $N_e$  can be obtained by integrating  $\rho$  over all space as,

$$N_e = \int \rho(\mathbf{r}) d\mathbf{r} \quad (2.1)$$

Using the total electron density is advantageous because the many-body wave function



is difficult to deal with and interpret. The total energy is determined by using  $\rho$ , so if we have  $\rho$ , we can get around the wave function. But in practice, most of the DFT codes are effective for systems with less than 100 atoms. Beyond this limit, most of the time DFT codes are found expensive because their computational cost grows with  $N_{atom}^3$ .

### 3. KKRnano

KKRnano is an electronic structure code which is based on the Korringa-Kohn-Rostoker (KKR) Green's function method. It provides enormously parallel density-functional calculations for large systems which are beyond the limit of classical DFT methods. KKRnano can deal with systems having thousands of atoms, so its main focus is nanostructures, disordered solids, defects, interfaces, etc. and their electronic and magnetic properties [4]. In this method,  $\rho$  can be obtained from the Green's function of the Kohn-Sham equation via the Kramers-Kronig relation. With this way, the problem is reduced to solving large sets of sparse linear equations. The Kohn-Sham equation for the electronic Green's function is similar to a Schrödinger-type equation,

$$\left[-\vec{\nabla}^2 + V(\vec{r}) - E\right] G(E; \vec{r}, \vec{r}') = \delta(\vec{r} - \vec{r}')$$

The main task demanding computational cost is the solving of large sets of sparse linear equations iteratively. The cost of this task is decreased for large system by implementing Kohn's nearsightedness principle of electronic matter [3]. According to this principal, Green's function elements describing long-range interactions can be neglected in order to save large amounts of computing time and storage requirements. With this way, when an atom  $A$  is considered for calculation, only a limited amount of neighboring atoms which are interacting with atom  $A$  are taken into account as shown in Figure 3.1. The number of interacting atoms, so the number of zero Green's function elements, is determined by a truncation radius ( $R_{trunc}$ ) as

$$\text{Linear scaling if } G(E; \vec{r}, \vec{r}') = 0 \text{ for } |\vec{r} - \vec{r}'| > R_{trunc}$$

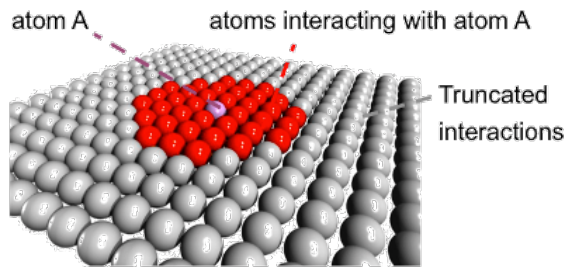


Figure 3.1.: Negligible interaction of far remote atoms implemented by neglecting Green-function elements describing long-range interactions. Picture credit: Alexander Thiess.

Therefore, in KKRnano the computation cost increases only linearly with the number of atoms (it is proportional to  $R_{trunc}^3 N_{atoms}$ ). This feature of KKRnano allows a highly parallelizable method for the quantum-mechanical treatment for thousands of atoms.

## 4. IMPLEMENTATION OF GPU LIBRARY `tfQMRgpu`

`tfQMRgpu` is a CUDA implementation for GPUs of the transpose-free Quasi-Minimal Residual (tfQMR) method developed for the iterative solution of linear systems of equations. Here the linear systems of equations correspond to the Dyson equation. `tfQMRgpu`, a tfQMR method which is ported to GPUs, takes advantage of solving for several right hand sides (RHSs) at a time using vectorization over CUDA threads [1].

The main task of `tfQMRgpu` is solving  $AX = B$  for the operators  $A$ ,  $X$  and  $B$ , as illustrated in Figure 4.1. The operators used here are block sparse matrices stored in a Block-Compressed Sparse Row (BSR) format. In our problem,  $A$ ,  $X$  and  $B$  represent  $H - E$ ,  $G$  and  $\delta$  (here it equals to unity) in a Schrödinger-type equation for the electronic Green's function, respectively. Operator  $B$  consists of the right hand site vectors in its columns. As a result of calculation, operator  $X$  contains the solution vectors as columns. Operator  $A$  is very sparse since only nearest neighbor cells are taken into account as a result of the tight-binding KKR formalism. The block-sparsity of  $X$  is due to truncation of the Green's function for the purpose of obtaining linear scaling of KKRnano [1].

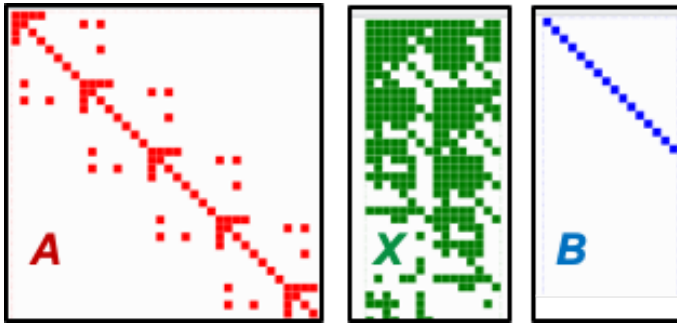


Figure 4.1.: The main task of `tfQMRgpu` is solving  $AX = B$  for the block-sparse operators  $A$ ,  $X$  and  $B$

An example for  $AX = B$  with 2 block columns in  $X$  and  $B$  is shown in Figure 4.2. In this case, if the non-zero entries of the 2 block columns of  $X$  have maximum overlap, the number of shared elements of  $A$  is maximized, so saving of bandwidth is achieved.

In this project we implemented the `tfQMRgpu` library to KKRnano. The original KKRnano code has two different solver modes labelled with integers; Mode 3 represents the iterative solver, while mode 4 corresponds to the direct solver which is based on matrix inversion (scales as  $N^3$ ). In this project we implemented two new solver options: mode 5 and 35. According to this denotation, mode 5 represents the GPU based iterative solver. This means that the algorithm can only be ran with mode 5 on a partition including GPU units, otherwise the program results in failure. In order to bypass the failure and to provide some flexibility to the user, we defined mode 35 which is able to switch between mode 3 and 5 according to resources. If KKRnano has been compiled with `tfQMRgpu`, it turns into mode 5, otherwise it turns into mode 3. By this way, KKRnano code can run according to resources in the machine.

`tfQMRgpu` is a complicated library by itself. It consists of a core written in CUDA C++

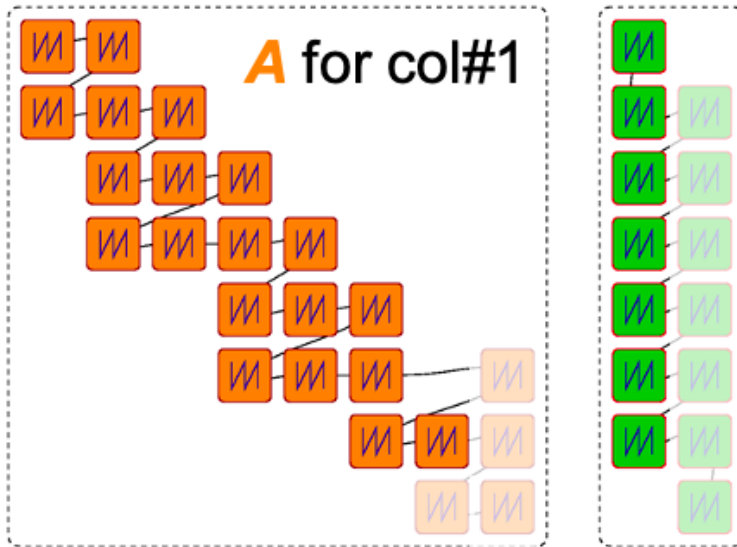


Figure 4.2.: solving  $AX = B$  with 2 block columns in  $X$  and  $B$  ( $B$  is not shown).

which is compiled to a shared object with C-interfaces, C wrappers and a Fortran 90 module around it (see Figure 4.3). Therefore, linking and compiling all those codes with KKRnano was where most of our efforts have gone. During this study, the complicated linking and compiling procedure is reduced to simple commands. In the final version of the codes, in order to run KKRnano with tfQMRgpu library, these steps should be followed:

➤ Clone and build tfQMRgpu:

```

1 # clone the tfQMRgpu repository
2 $ git clone https://github.com/real-space/tfQMRgpu.git
3
4 # create a build dir
5 $ mkdir build
6
7 # configure the project
8 $ cmake .. -DCMAKE_INSTALL_PREFIX=~/.tfQMRgpu
9             -DCMAKE_PREFIX_PATH=~/.tfQMRgpu
10
11 # compile the tfQMRgpu
12 $ make

```

➤ Clone and build KKRnano:

```

1 # clone the JuKKR repository
2 $ git clone https://iffigit.fz-juelich.de/kkr/jukkr.git
3
4 # switch to the branch that has the latest version of KKRnano

```

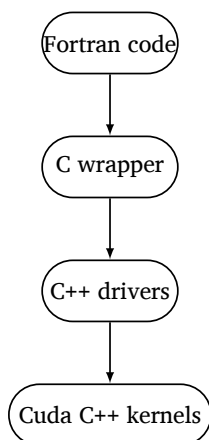


Figure 4.3.: Flowchart of tfQMRgpu

```

5 $ git checkout kkrnano-chebyshev-tfQMRgpu
6
7 # go into the source dir
8 $ cd jukkr/source/KKRnano/source
9
10 # create a build dir
11 $ mkdir build
12
13 # Build the tfQMRgpu Fortran90 module
14 $ make tfQMRgpu tfQMRgpu=yes
15
16 # compile the KKRnano
17 $ make tfQMRgpu=yes
  
```

## 5. PRELIMINARY RESULTS

After the implementation step, we tested the KKRnano code on the GPU partition of the HPC System "JUSUF" installed at Jülich Supercomputing Centre containing NVIDIA V100 GPUs. As a first test, we checked whether the newly implemented tfQMRgpu code can regenerate the same results with the original (CPU-based) KKRnano code. For this purpose, we ran both versions of the code for crystal structures of Cu<sub>4</sub>, GaN, and Fe<sub>8</sub>Co<sub>8</sub> compounds. The sum of band energies and total energy results are shown in Table 5.1 and 5.2, respectively.

As it is seen, tfQMRgpu implemented version can successfully regenerate the results with high accuracy. Then, as a second test, we checked the effect of the tfQMRgpu library implementation on performance in terms of wall clock time for different size of structures. In order to compare the performance, we ran the CPU and GPU versions of the code on bulk Cu crystal structures containing 4, 32, 108 and 256 atoms. All structures have face centered cubic symmetry which means that they have lattice points

on the faces and the corners of their cubic unit cells. As a result that this symmetry pattern repeats along the three-dimensional space, each lattice point in the faces is shared by 2 neighbouring cells (gives 1/2 contribution), while ones in the corners are shared by 8 neighboring cells (gives 1/8 contribution). Figure 5.1 represents the  $\text{Cu}_4$  and  $\text{Cu}_{32}$  crystal structures.

Those structures were created by using a supercell generator code which is written in python. This code is responsible for repeating the lattice parameters in the desired directions, adding enough atoms to keep the symmetry of the structure and creating new potential and shapefun files, the input files required by KKRnano.

Table 5.1.: Sum of band energies comparison between CPU and GPU versions of KKRnano.

Compound	CPU	GPU	Difference	
$\text{Cu}_4$	7.9826650492	7.9826650507	$1.5 \cdot 10^{-9}$	Ry
GaN	5.4259190350	5.4259190350	0	Ry
$\text{Fe}_3\text{Co}_8$	42.5224924129	42.5224924151	$2.2 \cdot 10^{-9}$	Ry

Table 5.2.: Total energy comparison between CPU and GPU versions of KKRnano.

Compound	CPU	GPU	Difference	
$\text{Cu}_4$	-179851.75480643	-179851.75480640	$3.001 \cdot 10^{-8}$	eV
GaN	-53346.71045016	-53346.71045016	0	eV
$\text{Fe}_8\text{Co}_8$	-579058.23717074	-579058.23717072	$2.002 \cdot 10^{-8}$	eV

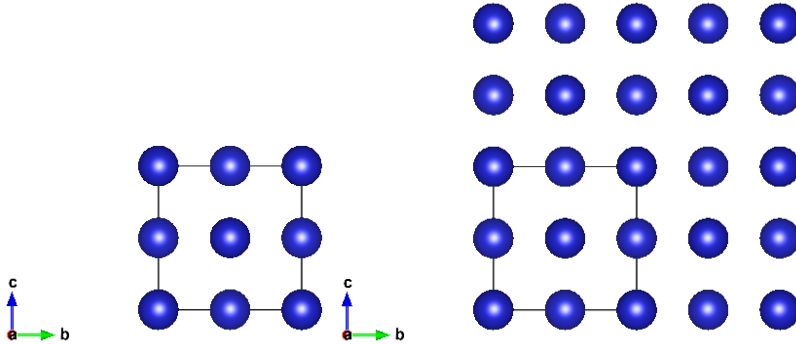


Figure 5.1.: Face-centered cubic symmetry of bulk Cu crystals. Left:  $\text{Cu}_4$ , Right:  $\text{Cu}_{32}$

We used the systems with the same element (Cu atom) but different sizes in order to clearly see the performance versus structure size. Figure 5.2 shows the performance

change with the number of atoms for the CPU and GPU versions of KKRnano for the energy loop calculation in one SCF cycle, i.e. measuring only the GPU-accelerated part.

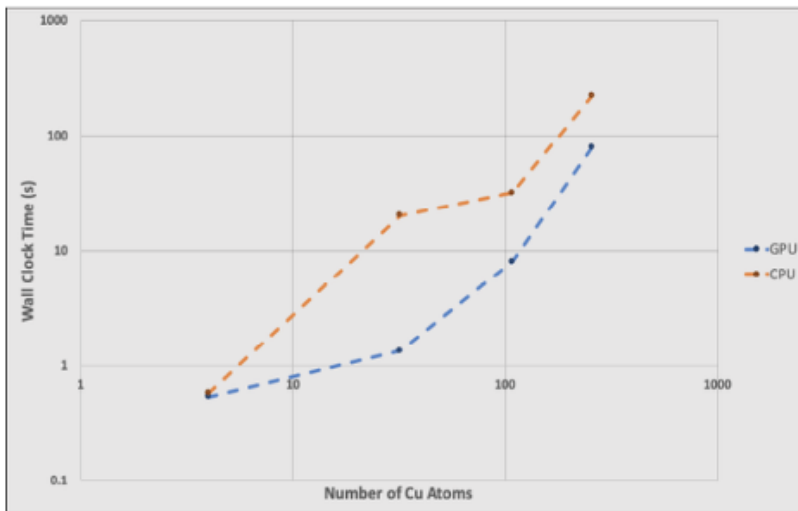


Figure 5.2.: Comparison between CPU and GPU versions of KKRnano for solving Dyson equation.

As it is seen, the implementation of `tfQMRgpu` has a positive effect on the performance. We do not see a linear trend in the performance with increasing number of atoms, since we are dealing with relatively small size systems. Since the number of atoms is small, the number of neglected long range interactions is also small and, so the effect of truncated Green's function elements is not seen here. No Green's function elements are dropped in these setups. In addition, it seems that the performance gain is changing with the size of the crystal structures. This may be resulted that there are only a few structures for which we test the performance. With adding larger systems, it is expected to see a clear linear trend with different cost prefactors.

## 6. REMARKS AND FUTURE WORK

In this study we have successfully implemented the `tfQMRgpu` library into the linear-scaling all-electron density functional theory application KKRnano and validated that the results are compatible with the original CPU version of the main code for relatively simple systems. In addition, we showed that `tfQMRgpu` provides a performance gain even for these kind of simple systems. On the other hand, because of the limited time, more complex systems, such as compounds having magnetic properties and spin-orbit coupling, which requiring more attention to deal with properly have not been tested. As a future work the GPU accelerated KKRnano can be applied to such complex systems for additional verification of the correctness of the results.

In addition, our performance test results cover systems having up to 256 atoms. Performance tests can be extended to larger systems with thousands of atoms in order to see whether the computational cost grows linearly or not for the GPU accelerated

case. In summary, it can be said that the tfQMRgpu version of KKRnano has successfully passed preliminary correctness and performance tests. With this work, a first and strong step for future studies is constituted.

## 7. ACKNOWLEDGMENTS

First of all, I would like to present my special thanks to my advisors Paul Baumeister and Philipp Rießmann. Their guidance and contributions helped me a lot through this project. I also want to thank Ivo Kabadshow for all his precious support. Finally, I would like to thank everyone who contributed to the realization of the Guest Student Programme. This program was a great opportunity to improve myself in many ways.

## REFERENCES

- [1] P. F. Baumeister. **tfQMRgpu**. <https://github.com/real-space/tfQMRgpu>.
- [2] C. J. Cramer. Essentials of computational chemistry: theories and models. Wiley, 2 edition, 2004.
- [3] W. Kohn. **Density functional and density matrix method scaling linearly with the number of atoms**. Phys. Rev. Lett., 76:3168–3171, Apr 1996. <https://link.aps.org/doi/10.1103/PhysRevLett.76.3168>, [doi:10.1103/PhysRevLett.76.3168](https://doi.org/10.1103/PhysRevLett.76.3168).
- [4] A. Thiess, R. Zeller, M. Bolten, P. H. Dederichs, and S. Blügel. **Massively parallel density functional calculations for thousands of atoms: Kkrnano**. Phys. Rev. B, 85:235103, Jun 2012. <https://link.aps.org/doi/10.1103/PhysRevB.85.235103>, [doi:10.1103/PhysRevB.85.235103](https://doi.org/10.1103/PhysRevB.85.235103).
- [5] D. Young. Computational chemistry: a practical guide for applying techniques to real world problems. Wiley New York, 1 edition, 2001.

# NSL(NANOSYSTEM SIMULATION LIBRARY)

Anastasia Papadaki  
Computational Engineering  
Friedrich-Alexander  
University  
Erlangen-Nürnberg  
Germany  
anni1302@gmail.com

**Abstract** *The Nanosystem Simulation Library (NSL) is a library that implements statistical simulations for nanoscale systems.*

*The two software programs CNS and isle are combined in one library. More features are added to provide a self-contained modeling library for nanostructures like graphene and carbon nanotubes. We used Kokkos C++ in our project to approach the project and to suggest the benchmark problem [5].*

## 1. INTRODUCTION/MOTIVATION

Nowadays, a simulation technique that can link on the nanoscale is essential for accurate design and modeling of nano-enabled systems. For accurately and efficiently modeling a material, a variety of simulation methodologies and tools are available. Also there are several approaches for linking and coupling multiple hierarchical scales. The present consortium's major purpose is to create a modeling environment for nanomaterials and system design. The tools will be built primarily by enhancing existing open-source and commercial simulation tools with sophisticated libraries. The simulation environment will also act as a basis for harmonizing and speeding up the development of new simulation modules by providing interface libraries to sophisticated pre- and postprocessing tools as well as computational modules which can be easily integrated and reused in new applications. Through a proof-of-concept design of innovative simulation tools, the efficiency of the newly created simulation environment will be proven, especially for reducing the development process and time to find novel nano-enabled products. Beyond this research, we plan to investigate how Kokkos optimizes code for the architectures utilized in these trials, as well as the layered parallelisation option used in NSL.

## 2. THE HUBBARD MODEL

The Hubbard model is an approximate model used in solid state physics to describe the transition between conductive and insulating systems. The Hubbard model, named after John Hubbard, is a simple model of particles in a lattice, with only two terms in the Hamiltonian a kinetic term allowing the tunneling of particles between sites of the lattice and a potential term consisting of on-site interaction. Particles can be either fermions, as in Hubbard's original work, or bosons, in which case model is called the "Bose-Hubbard model". The Hubbard model is a useful approximation for a periodic potential at sufficiently low temperatures. All particles can be assumed to be in the lowest Bloch state, and interactions to distance between particles can be ignored. If the interactions between particles at different lattice sites are included, the model is often referred to as "Extended Hubbard". The physics of the Hubbard model is determined by competition between the strength of the hopping integral, which characterizes the system's kinetic energy, and the strength of the interaction term. As we mentioned earlier, the Hubbard model was proposed in the 1960s to describe electrons in 3d



transition metals. In these elements, the radial wave function of the 3d-electrons has a very small spatial extent. Therefore, when the 3d shell is occupied by several electrons, these are forced to be close to one another on the average so that the electrostatic energy is large. In (statistical) physics, a model is specified by its Hamiltonian. More specific first, we would like to account that there is a regular array of nuclear position that in good approximation is fixed. This implies that we start with a lattice of ions (sites) on that the fermions move. Of course, a single real atom is already a really complicated structure, with many various energy levels (orbitals). The HH approximates the atoms in a solid with one level (orbital). Given this approximation (of one orbital) and applying the Pauli principle yields 4 possible electron states per site: empty, a single spin up fermion, a single spin down fermion, or double occupation by a pair of spin up and spin down fermions. In a solid where electrons can move around, the electrons interact via a screened Coulomb interaction. The biggest interaction will be for two electrons on the same site. Moreover, interactions are modeled by a term which is zero if the site is empty or has only a single fermion. Then, it takes the value  $U$  if the site is doubly occupied (necessarily, by the Pauli principle, by fermions of opposite spin) [2]. The expression  $Un_i n_j$  captures this property. In the simplest HH, there is no interaction  $V_{n_i n_j}$  between fermions on different sites  $i$  and  $j$ , although such terms are included in the "extended" HH. With all that the Hubbard Hamiltonian Model can be written with two terms, a hopping  $H_0$  and an interaction term  $H_I$ :  $H = H_0 + H_I$ . So, it takes the form:

$$H = - \sum_{ij} \sum_{\sigma} t_{ij} c_{i\sigma}^{\dagger} c_{j\sigma} + U \sum_i n_{i\uparrow} n_{i\downarrow} \quad (2.1)$$

where  $c_{i\sigma}^{\dagger} c_{i\sigma}$  is the spin-density operator for spin  $\sigma$  on the  $i$ -*th* site and the total density operator is  $n_{i\uparrow} n_{i\downarrow}$ . Also, typically  $t$  is taken to be positive, and  $U$  may be either positive or negative in general, but is assumed to be positive when considering electronic systems as we are here. We refer to the situation where there is one fermion per site as "half-filling" since the lattice contains half as many fermions as the maximum number (two per site). Studies of the HH often focus on the half-filled case because, as we shall see, it exhibits a lot of interesting phenomena (Mott insulating behavior, anti-ferromagnetic order, etc.)

## 3. GRAPHENE-CARBON NANOTUBES

### 3.1. GRAPHENE

The discovery of graphene and new research comes from the Institute of Electronic Structure and Laser. The discovery of graphene (Nobel Prize in Physics in 2010), the first two-dimensional material (about the thickness of a carbon atom), revolutionised science and technology. due to its exceptional physical properties. Two-dimensional materials offer significant advantages over silicon-based materials, so they are expected to be used in future devices very soon [4]. Especially when two two-dimensional materials are placed on top of each other, the new material that is created presents new exceptional properties that do not exist in the individual two-dimensional materials that the product produces. For example, when two single-layer graphene sheets are placed one on top of the other at the angle of the molecule, the new material created exhibits superconducting properties. This angle is called the "magic-angle" [3].

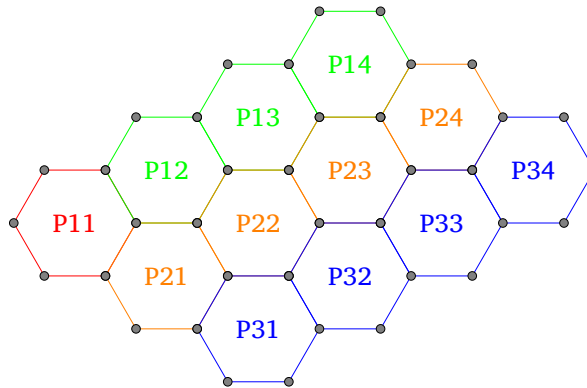


Figure 3.1.: Graphene

### 3.1.1. PROPERTIES

Graphene is the most slender material known to man at one particle thick, additionally fantastically solid - around 200 times more grounded than steel. On beat of that, graphene is a great conductor of warm and power and has curiously light assimilation capacities. It is genuinely a fabric that seem alter the world, with boundless potential for integration in nearly any industry.

### 3.1.2. POTENTIAL APPLICATIONS

Graphene is a greatly differing material , and can be combined with other components (counting gasses and metals) to create distinctive materials with different predominant properties. Analysts all over the world proceed to continually explore and obvious graphene to investigate its different properties and conceivable applications, which include: batteries transistors computer chips energy generation supercapacitors DNA sequencing water filters antennas touchscreens (for LCD or OLED displays) solar cells.

### 3.1.3. PRODUCING GRAPHENE

Graphene is in fact exceptionally “strong”, but creating tall quality materials is still a challenge. Handfuls of companies around the world are creating distinctive sorts and grades of graphene materials - extending from tall quality single-layer graphene synthesized employing a CVD-based handle to graphene drops delivered from graphite in huge volumes. High-end graphene sheets are for the most part utilized in R&D exercises or in extraordinary applications such as sensors, but graphene chips, delivered in huge volumes and at lower costs, are embraced in numerous applications such as sports gear, buyer hardware, car and more [10].

## 3.2. CARBON NANOTUBES

Carbon nanotubes (CNTs) are tubes made of carbon with distances across regularly measured in nanometers. Single-wall carbon nanotubes are one of the allotropes of carbon, halfway between fullerene cages and level graphene. Although not made this way, single-wall carbon nanotubes can be idealized as set patterns from a two-dimensional hexagonal grid of carbon molecules rolled up along one of the Bravais grid vectors of the hexagonal grid to create a empty barrel. In this development, intermittent

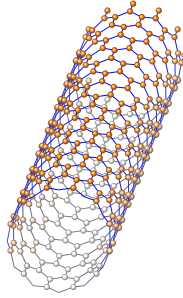


Figure 3.2.: Carbon Nanotube

boundary conditions are forced over the length of this roll-up vector to abdicate a helical cross section of consistently reinforced carbon particles on the barrel surface. Carbon nanotubes too frequently allude to multi-wall carbon nanotubes (MWCNTs) comprising of settled single-wall carbon nanotubes pitifully bound together by van der Waals intuitive in a tree ring-like structure. In the event that not indistinguishable, these tubes are exceptionally comparable to Oberlin, Endo, and Koyama's long straight and parallel carbon layers circularly orchestrated around an empty tube. Multi-wall carbon nanotubes are moreover in some cases utilized to allude to twofold- and triple-wall carbon nanotubes. Such tubes were found in 1952 by Radushkevich and Lukyanovich. The length of a carbon nanotube created by common generation strategies is frequently not detailed, but is regularly much bigger than its distance across. In this way, for numerous purposes, conclusion impacts are ignored and the length of carbon nanotubes is expected infinite. Carbon nanotubes can show surprising electrical conductivity, whereas others are semiconductors. They moreover have uncommon pliable quality and warm conductivity due to their nanostructure and quality of the bonds between carbon molecules. In expansion, they can be chemically adjusted. These properties are anticipated to be important in numerous zones of innovation, such as gadgets, optics, composite materials (supplanting or complementing carbon filaments), nanotechnology, and other applications of materials science. In addition, most are chiral, meaning the tube and its reflect picture cannot be superimposed. This development too permits single-wall carbon nanotubes to be labeled by a match of integers. An extraordinary bunch of achiral single-wall carbon nanotubes are metallic, but all the rest are either little or direct band crevice semiconductors. These electrical properties, be that as it may, don't depend on whether the hexagonal cross section is rolled from its back to front or from its front to back and thus are the same for the tube and its reflect image. Most are chiral, meaning the tube and its reflect picture cannot be superimposed. This development too permits single-wall carbon nanotubes to be labeled by a match of integers. An extraordinary bunch of single-wall carbon nanotubes are metallic, but all the rest are either little or direct band semiconductors [9].

#### 4. NANOSYSTEM SIMULATION LIBRARY

First of all, NSL can be used for different kind od models. In the current work, we want to code a NSL for the Hubbard model. As we mentioned in the introduction, we use two

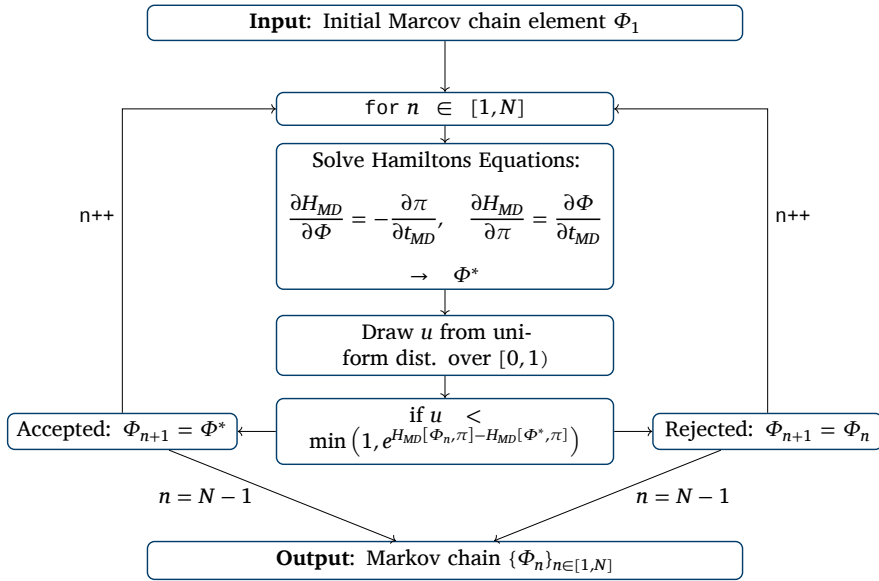


Figure 4.1.: Blueprint of a Hybrid (Hamilton) Monte Carlo (HMC) algorithm

materials, graphene and carbon nanotubes and our aim is to simulate their properties through NSL. To achieve that we have to apply Markov Chain Monte Carlo. The general idea of this implementation is the following:

- Create a configuration. However, this configuration is nothing more than a snapshot of a quantum mechanical state of the system.
- Compute observable, for instance magnetization, on each configuration.
- Average those results to get estimate (+ error) of this observable.

This kind of algorithms required plenty of high-dimensional linear algebra operations. The GPU provides high parallelism execution model that is well suited for this. Consequently, we try to ofload as many operations as possible.

## 5. KOKKOS LIBRARY

Kokkos is a C++ programming paradigm for creating high-performance portable programs that can run on many different architectures. It accomplishes this by providing interfaces for both parallel code execution and data management. OpenMP, Pthreads, and CUDA are now supported as backend programming paradigms [7].

### 5.1. KOKKOS TOOLS

Kokkos Tools provide you access to Kokkos' built-in instrumentation for profiling and debugging. They make it much simpler to comprehend what's going on in a big Kokkos application, which aids in the detection of mistakes and performance concerns. For a quick search of syntax, here is the place to go. Developers familiar with other shared

memory models such as OpenMP, CUDA, or OpenCL may be able to check up how specific features are employed in Kokkos [6].

## 5.2. KOKKOS-KERNELS

Graph kernels or dense and sparse BLAS capabilities are used in many, if not all, high-performance computing applications. Kokkos-Kernels is a Kokkos View-based interface that provides efficient kernels and interfaces to vendor libraries. There's no need to find out what your data structures' dimensions, strides, and memory spaces are since Kokkos Views already know.

## 5.3. KOKKOS GPU

As we mentioned above, Kokkos may be a templated C++ library that gives deliberations to permit a single usage of an application bit (e.g. a collision fashion) to run productively on diverse sorts of equipment, such as GPUs, Intel Xeon Phi, or many-core CPUs. Kokkos maps the C++ bit onto diverse backend dialects such as CUDA, OpenMP, or Pthreads. The Kokkos library also gives information deliberations to alter (at compile time) the memory format of information structures like 2d and 3d clusters to optimize execution on distinctive equipment. For more data on Kokkos, see Github. Kokkos is portion of Trilinos. The Kokkos library was composed fundamentally by Carter Edwards, Christian Trott, and Dan Sunderland (all Sandia). For half neighbor records and OpenMP, the KOKKOS bundle employments information duplication (i.e. thread-private clusters) by default to dodge thread-level type in clashes within the constrain clusters (and other information structures as essential). Information duplication is regularly speediest for little numbers of strings (i.e. 8 or less) but does increase memory impression and isn't versatile to huge numbers of strings. An elective to information duplication is to utilize thread-level nuclear operations which don't require information duplication. The utilize of nuclear operations can be upheld by compiling LAMMPS with the "-DLMP\_KOKKOS\_USE\_ATOMICS" pre-processor hail. Most but not all Kokkos-enabled pair\_styles back information duplication. On the other hand, full neighbor records maintain a strategic distance from the require for duplication or nuclear operations but require more compute operations per molecule. When utilizing the Kokkos Serial back conclusion or the OpenMP back conclusion with a single string, no duplication or nuclear operations are utilized. The KOKKOS package currently provides support for 3 modes of execution (per MPI task). These are Serial (MPI-only for CPUs and Intel Phi), OpenMP (threading for many-core CPUs and Intel Phi), and CUDA (for NVIDIA GPUs). You choose the mode at build time to produce an executable compatible with specific hardware. When using a GPU, you will achieve the best performance if your input script does not use fix or compute styles which are not yet Kokkos-enabled. This allows data to stay on the GPU for multiple timesteps, without being copied back to the host CPU. Invoking a non-Kokkos fix or compute, or performing I/O for thermo or dump output will cause data to be copied back to the CPU incurring a performance penalty [6].

## 6. BENCHMARK

In computing, a benchmark is the act of running a computer program, a set of programs, or other operations, in arrange to survey the relative execution of time, regularly by running a number of standard tests and trials against it. The term benchmark is addi-

tionally commonly utilized for the purposes of extravagantly planned benchmarking programs themselves. Benchmarking is more often than not related with evaluating execution characteristics of computer equipment, for illustration, the coasting point operation execution of a CPU, but there are circumstances when the strategy is additionally pertinent to computer program. As computer engineering progressed, it got to be more troublesome to compare the execution of different computer frameworks essentially by looking at their details. In this manner, tests were created that permitted comparison of distinctive structures. For illustration, Pentium 4 processors by and large worked at the next clock recurrence than Athlon XP or PowerPC processors, which did not essentially decipher to more computational control; a processor with a slower clock recurrence might perform as well as or indeed superior than a processor working at the next recurrence. See BogoMips and the megahertz myth. Benchmarks are outlined to imitate a specific sort of workload on a component or framework. Manufactured benchmarks do this by uncommonly made programs that force the workload on the component. Application benchmarks run real-world programs on the framework. Benchmarks are especially vital in CPU plan, giving processor designers the capacity to degree and make tradeoffs in microarchitectural choices. For illustration, in case a benchmark extricates the key algorithms of an application, it'll contain the performance-sensitive perspectives of that application. Running this much littler piece on a cycle-accurate test system can donate clues on how to move forward execution. Computer producers are known to arrange their frameworks to grant unreasonably tall execution on benchmark tests that are not duplicated in genuine utilization. For occasion, amid the 1980s a few compilers might identify a particular numerical operation utilized in a well-known floating-point benchmark and supplant the operation with a speedier numerically identical operation. Be that as it may, such a change was seldom valuable exterior the benchmark until the mid-1990s, when RISC and VLIW models emphasized the significance of compiler innovation because it related to execution. Benchmarks are presently frequently utilized by compiler companies to move forward not as it were their possess benchmark scores, but genuine application performance. CPUs that have numerous execution units — such as a superscalar CPU, a VLIW CPU, or a reconfigurable computing CPU — regularly have slower clock rates than a successive CPU with one or two execution units when built from transistors that are fair as quick. By the by, CPUs with numerous execution units regularly total real-world and benchmark assignments in less time than the as far as anyone knows quicker high-clock-rate CPU. Given the expansive number of benchmarks accessible, a producer can as a rule discover at slightest one benchmark that appears its framework will outflank another framework; the other frameworks can be appeared to exceed expectations with a diverse benchmark. Manufacturers commonly report as it were those benchmarks (or viewpoints of benchmarks) that appear their items within the best light. They too have been known to mis-represent the noteworthiness of benchmarks, once more to show their items within the best conceivable light. Taken together, these hones are called bench-marketing. Ideally benchmarks should only substitute for real applications if the application is unavailable, or too difficult or costly to port to a specific processor or computer system. If performance is critical, the only benchmark that matters is the target environment's application suite [8].

```

// =====
// Slicing
// =====
Tensor<Type, Kokkos::LayoutStrided> slice(const std::size_t & dim, const std::size_t & start, const std::size_t & end){

    const std::size_t rank = data_.rank();
    if (dim >= rank){
        throw std::runtime_error("NSL::Tensor.slice(dim,start,end) can only have dim=0,1,...,"+std::to_string(rank)+" but got dim="+std::to_string(dim)+"\n");
    }

    // Kokkos only implements up to rank 7. This can be used as the subview must take at least rank+1 arguments
    // additional Kokkos::ALL arguments are ignored by the subview function.
    // The returned Kokkos::DynamicRankView<Type,LayoutStrided> is then converted into NSL::Tensor<Type, LayoutStrided> at return.
    switch(dim) {
        case 0: return Kokkos::subview(data_, std::make_pair(start, end), Kokkos::ALL, Kokkos::ALL, Kokkos::ALL, Kokkos::ALL, Kokkos::ALL, Kokkos::ALL);
        case 1: return Kokkos::subview(data_, Kokkos::ALL, std::make_pair(start, end), Kokkos::ALL, Kokkos::ALL, Kokkos::ALL, Kokkos::ALL, Kokkos::ALL);
        case 2: return Kokkos::subview(data_, Kokkos::ALL, Kokkos::ALL, std::make_pair(start, end), Kokkos::ALL, Kokkos::ALL, Kokkos::ALL, Kokkos::ALL);
        case 3: return Kokkos::subview(data_, Kokkos::ALL, Kokkos::ALL, Kokkos::ALL, std::make_pair(start, end), Kokkos::ALL, Kokkos::ALL, Kokkos::ALL);
        case 4: return Kokkos::subview(data_, Kokkos::ALL, Kokkos::ALL, Kokkos::ALL, Kokkos::ALL, std::make_pair(start, end), Kokkos::ALL, Kokkos::ALL);
        case 5: return Kokkos::subview(data_, Kokkos::ALL, Kokkos::ALL, Kokkos::ALL, Kokkos::ALL, Kokkos::ALL, std::make_pair(start, end), Kokkos::ALL);
        case 6: return Kokkos::subview(data_, Kokkos::ALL, Kokkos::ALL, Kokkos::ALL, Kokkos::ALL, Kokkos::ALL, Kokkos::ALL, std::make_pair(start, end));
        default: throw std::runtime_error("NSL::Tensor.slice(dim,start,end) can only have dim=0,1,...,"+std::to_string(rank)+" but got dim="+std::to_string(dim)+"\n");
    }
}

```

Figure 7.1.: Example of our code

## 7. RESULTS

First of, a Tensor class with a slice method was created. Then a condition was created that specifies that if the number of dimensions is greater than the number of ranks, then the program will display an error. After that we have the main program where as the following picture shows that Kokkos only implements up to 7 ranks. Also, this can be used as the subview must take at least rank + 1 arguments. However, Kokkos::ALL arguments are ignored by the subview function. Finally, the returned Kokkos::DynamicRankView <Type, LayoutStrided> is then converted into NSL::Tensor <Type, LayoutStrided> at return.

As we expected, by our assumptions the total execution time is constant, because throughput (how much time our system needs to do the process (slicing) per MB) is linear rising. Finally, the error-bar themselves are smaller than the average ns as we expected.

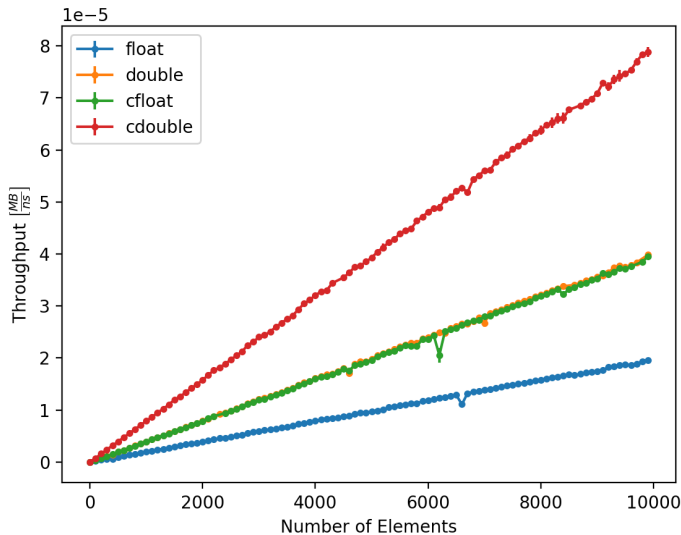


Figure 7.2.: Benchmark Results

## 8. CURRENT WORK

First of all, we tried to comprehensive how Kokkos C++ works exactly. So, we started by implemented some constructors and destructors.

After that, we created a **class Tensor** and we inserted the **Kokkos::DynRankView**.

**DynRankView** is a potentially reference counted multi dimensional array with compile time layouts and memory space. Its semantics are similar to that of `std::shared_ptr`. The DynRankView differs from the View in that its rank is not provided with the DataType template parameter; it is determined dynamically based on the number of extent arguments passed to the constructor. The rank has an upper bound of 7 dimensions.

From that point forward and adding some more constructors and destructors, we ended up creating the method of **slicing**. In the first place, we created a class Tensor using **LayoutType**. LayoutType determines the mapping of indices into the underlying 1D memory storage. Custom Layouts can be implemented, but Kokkos comes with some built-in ones: LayoutStride where strides can be arbitrary for each dimension. At this point we have to say that in Kokkos, a subview is a slice of a View. A slice of a multidimensional array behaves as an array, and is a view of a structured subset of the original array. "Behaves as an array" means that the slice has the same syntax as an array does; one can access its entries using array indexing notation. "View" means that the slice and the original array point to the same data, i.e, the slice sees changes to the original array and vice versa. "Structured subset" means a cross product of indices along each dimension, as for example a plane or face of a cube. If the original array has dimensions  $(N_0, N_1, \dots, N_{k-1})$ , then a slice views all entries whose indices are  $(a_0, a_1, \dots, a_{k-1})$ , where  $a_j$  is an ordered subset of  $N_0, N_1, \dots, N_{j-1}$ .



Array slices are handy for encapsulation. A slice looks and acts like an array, so you can pass it into functions that expect an array. For example, you can write a function for processing boundaries (as slices) of a structured grid without needing to tell that function properties of the entire grid. To take a subview of a View, you can use the `Kokkos::subview` function. This function is overloaded for all different kinds of Views and index ranges. Moreover, as we mentioned above Kokkos only implements up to rank 7. This can be used as the subview must take at least rank + 1 arguments. Additional `Kokkos::ALL` arguments are ignored by the subview function [1]. So, we ended up that the returned `Kokkos::DynamicRankView<Type, LayoutStrided>` is then converted into `NSL::Tensor<Type, LayoutStrided>` at return.

## REFERENCES

- [1] D. Arndt. **Kokkos\_DynRankView.hpp**, 2018. Last accessed 25 February 2021. 🌐 [https://github.com/kokkos/kokkos/blob/master/containers/src/Kokkos\\_DynRankView.hpp](https://github.com/kokkos/kokkos/blob/master/containers/src/Kokkos_DynRankView.hpp).
- [2] V. Celebonovic. **The two dimensional hubbard model: a theoretical tool for molecular electronics**. *Journal of Physics: Conference Series*, 253:1–10, 2010.
- [3] E. Gibney. **How magic angle graphene is stirring up physics**. page 1, 2019.
- [4] H. Hill. **Magic-angle bilayer graphene enters a new phase**. *Physics Today*, 18:1, 2019.
- [5] M. Rodekamp. **Nsl**, 2021. Last accessed 27 March 2021. 🌐 <https://github.com/Marcel-Rodekamp/NSL>.
- [6] C. Trott. **Kokkos: The C++ performance portability programming model**, 2018. Last accessed 4 July 2018. 🌐 <https://github.com/kokkos/kokkos/wiki/OverlappingHostAndDeviceWork>.
- [7] C. Trott. **Overlapping host and device work**, 2021. Last accessed 27 March 2021. 🌐 <https://github.com/kokkos/kokkos/wiki/OverlappingHostAndDeviceWork>.
- [8] Wikipedia. **Benchmark (computing)**, 2021. Last accessed 17 August 2021. 🌐 [https://en.wikipedia.org/wiki/Benchmark\\_\(computing\)](https://en.wikipedia.org/wiki/Benchmark_(computing)).
- [9] Wikipedia. **Carbon nanotube**, 2021. Last accessed 20 September 2021. 🌐 [https://en.wikipedia.org/wiki/Carbon\\_nanotube](https://en.wikipedia.org/wiki/Carbon_nanotube).
- [10] Wikipedia. **Graphene**, 2021. Last accessed 27 September 2021. 🌐 <https://en.wikipedia.org/wiki/Graphene>.

# JURASSIC-SCATTER-GPU

## Accelerating Multiple Scattering for Radiation Transport

Stjepan Požgaj  
Faculty of Science,  
Department of Mathematics  
University of Zagreb  
Croatia  
stjepan.pozgaj1@gmail.com

**Abstract** *JURASSIC is a fast radiative transfer model for the analysis of atmospheric remote sensing measurements in the mid-infrared spectral region. An important research field is to incorporate particles (ice and water clouds, volcanic aerosols, dust particles, etc.) into the model. For this, scattering of the infrared radiation on the liquid or solid particles was accounted for in JURASSIC-scatter. In this paper we present JURASSIC-scatter-GPU, an upgrade of JURASSIC-scatter which now also benefits from GPU tuning and acceleration of JURASSIC and because of that achieves better performance.*

## 1. INTRODUCTION

Infrared measurements from polar orbiting satellite instruments are an important pillar of Earth observation systems. In order to derive the state of the atmosphere (temperature, pressure, trace gas concentrations, aerosol and cloud properties) from the measured spectra, we need to model the radiative transfer through the atmosphere along a ray path given by the position and orientation of the satellite instrument. The atmosphere variables can be varied until the calculated spectra match with the measured spectra. This process is called retrieval and relies on a fast execution of the radiative transport computation.

The Juelich Rapid Spectral Simulation Code (JURASSIC) [6, 7, 8] is a fast radiative transfer model for the analysis of atmospheric remote sensing measurements in the mid-infrared spectral region. It performs the radiative transport forward calculation and provides the retrieval algorithm around it. It was originally developed by Hoffman [7]. Successful analyses with this JURASSIC have been performed on MIPAS, CRISTA-NF and AIRS data [5]. JURASSIC was originally written in C and has been ported to GPUs using the CUDA programming language by Baumeister et al. [2, 1].

An important research field is to incorporate particles (ice and water clouds, volcanic aerosol, dust particles, etc.) into the model. For this, scattering of the infrared radiation on the liquid or solid particles needs to be accounted for so JURASSIC was cloned and modified to become JURASSIC-scatter by Grießbach [5, 4].

JURASSIC without scattering is available as vectorized CPU and GPU version and as reference implementation, however, JURASSIC-scatter so far did not benefit from tuning and acceleration. As it can be seen in Figure 1.1, our goal in this paper is to connect JURASSIC-scatter and JURASSIC-GPU into a new JURASSIC-scatter-GPU project, which will perform the same task as JURASSIC-scatter, but will use JURASSIC-GPU in some of its parts to get better performance.

The remainder of this paper is structured as follows. The JURASSIC and the JURASSIC-scatter forward models are explained in more detail in Section 2. Files organization of the new project and source code differences between given projects are described in Section 3. In Section 4 we show how JURASSIC-scatter is modified in order to incorporate the GPU-enabled JURASSIC functions and in Section 5 we compare performance results of

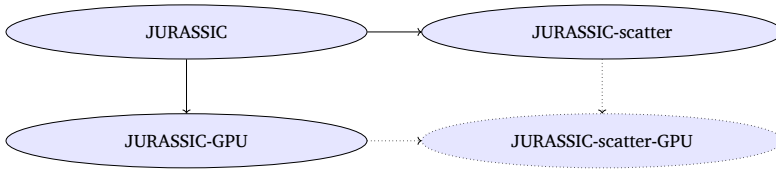


Figure 1.1.: Projects based on JURASSIC. The design question was whether a new repository needed to be created for a GPU-accelerated version of JURASSIC-scatter.

our program and the reference implementation. Finally, Section 6 summarizes the paper and gives an outlook.

## 2. FORWARD MODELS

As we mentioned above, JURASSIC is a coupled forward and retrieval model which allows analyses of different remote sensing measurements. In this paper we will not go into the details of the retrieval, but the JURASSIC and JURASSIC-scatter forward models are analysed to identify their differences and more easily determine how to combine these two projects.

### 2.1. JURASSIC FORWARD MODEL

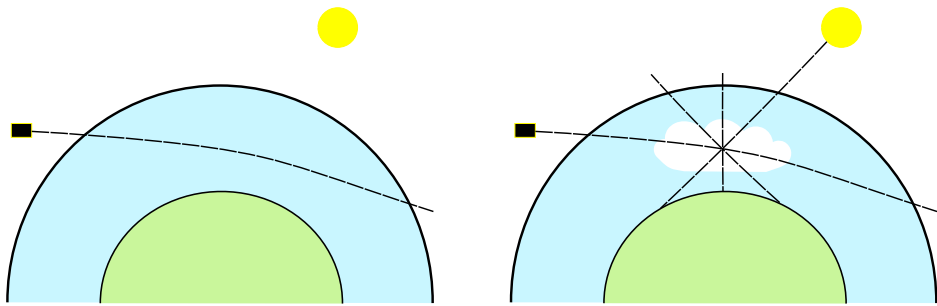
The JURASSIC forward model is shown in solid rectangles of Figure 2.2. In the input block, the pre-calculated emissivity tables, the atmospheric data containing pressure, temperature, volume mixing ratios of atmospheric gases, and the control file are given.

At the beginning of a radiative transfer calculation the path of a single ray through the atmosphere, which is referred to as pencil beam, has to be calculated. In the case of a limb geometry the pencil beam is not just a straight line tangentially through the atmosphere but a curve refracted towards the Earth’s surface. The degree of curvature depends on the change of the refractive index of the atmosphere with altitude and refractive index depends on atmospheric conditions. The step length chosen for determining segments in raytracing has to fulfil two constraints: on the one hand fewer steps are advantageous when considering the computation time and on the other hand the step length must be short enough so that the atmosphere properties along one step can be assumed as constant. You can see an example of a limb path through an atmosphere in Figure 2.1a.

After raytracing, the evaluation of the radiative transfer is done by calculating spectrally averaged radiances, emissivities and Planck’s functions applying pre-calculated emissivity tables according to the instrument’s characteristics [5]. In Figure 2.2 we refer to those three calculations as ”computing radiative transport”.

### 2.2. JURASSIC-SCATTER FORWARD MODEL

The implementation of scattering into JURASSIC is schematically shown in Figure 2.2, but in this case the dotted rectangles and the diamond from it also have to be included. Compared to the JURASSIC model without scattering JURASSIC-scatter contains some new input parameters: scattering order, number of scattering modules, and log-normal parameters of the particle size distribution (particle number concentration, median radius and standard deviation). The scattering order  $sca_{mult}$  defines how many scattering levels the forward model has and it will be discussed in the Section 4. The other three



(a) Limb path through an atmosphere.

(b) Limb path through a cloudy atmosphere.

Figure 2.1.: The clear air case can be treated using JURASSIC or JURASSIC-GPU. In the cloudy case incoming radiance from all directions is scattered towards the detector by cloud particles so it is treated by JURASSIC-scatter.

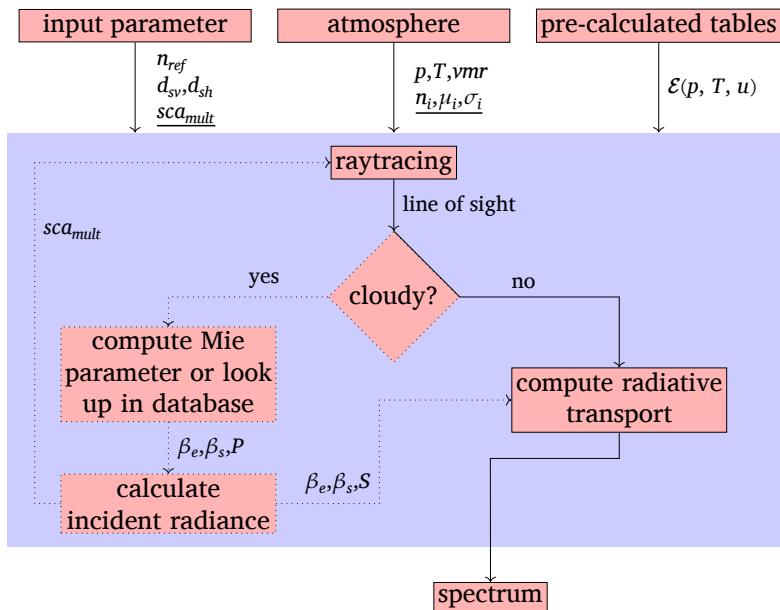


Figure 2.2.: Forward models. Dotted arrows and underlined variables indicate the difference of JURASSIC-scatter over JURASSIC.

new input parameters are not part of the JURASSIC forward model without scattering because those are aerosol and cloud properties.

In the JURASSIC-scatter forward model, the raytracing of the line of sight is again calculated first. But in this case segments located in the cloud are determined and for each of these segments the extinction and scattering coefficients as well as the phase function are determined and new ray paths ending at that segment are set up to determine the incoming radiance from all directions [5]. The incoming ray paths also pass through the cloud and undergo scattering processes. Based on this approach single or multiple scattering, depending on scattering order, can be computed. An example of a limb path through a cloud atmosphere is shown in Figure 2.1b.

We see that JURASSIC-scatter is equivalent to JURASSIC without scattering in cases where the segment is not part of the cloud or when it is at the recursion base case, i.e. at the last level of scattering and this is precisely the main motivation for merging the two models.

### 3. CONNECTING PROJECTS

In this section we will discuss how we managed to connect the JURASSIC-scatter and JURASSIC-GPU codes. Each of these codes is in its own GitHub repository and our task was to merge them. Wanted that, we had to figure out how to send data from one project to another because we want to have uninterrupted communication between them. Also, we want to determine if we have to create a new project or it is best to somehow combine these two projects to not further complicate code structures and maintainability.

#### 3.1. FILES ORGANIZATION

In Figure 3.1 you can see the files organization. There are actually more files than it is shown, but the most important ones are there. We decided not to create a new GitHub repository but to use the `git clone --branch` commands (see appendix) to put JURASSIC-scatter and JURASSIC-GPU in the same place. More precisely, to use JURASSIC-scatter-GPU one first has to clone the `dev` branch of JURASSIC-scatter and then clone the `scatter` branch of JURASSIC-GPU into it.

Compiling is also a challenging step when merging two projects. To compile JURASSIC-scatter-GPU one has to run a script which can also be seen in the appendix. This script first compiles JURASSIC-GPU and wraps it into a static C library which is then linked to JURASSIC-scatter.

JURASSIC-scatter and JURASSIC-GPU both have their own header files which contain macros, constants and declarations of important structures. One way to merge them would be to leave these header files separated and convert one type to another when needed, but since we would like to have a unified code in the future we decided on a common header file that both projects must include.

Since the projects have a number of files and functions with the same name, when merging, we decided to put declarations of all its functions that JURASSIC-scatter directly uses in the header file `inter-folder/interface.h` on the `scatter` branch of the JURASSIC-GPU project. For some functions this interface trick was not enough so we solved the name collision problem by adding the prefix `"jur_"` to the names of such functions of the JURASSIC-GPU project.

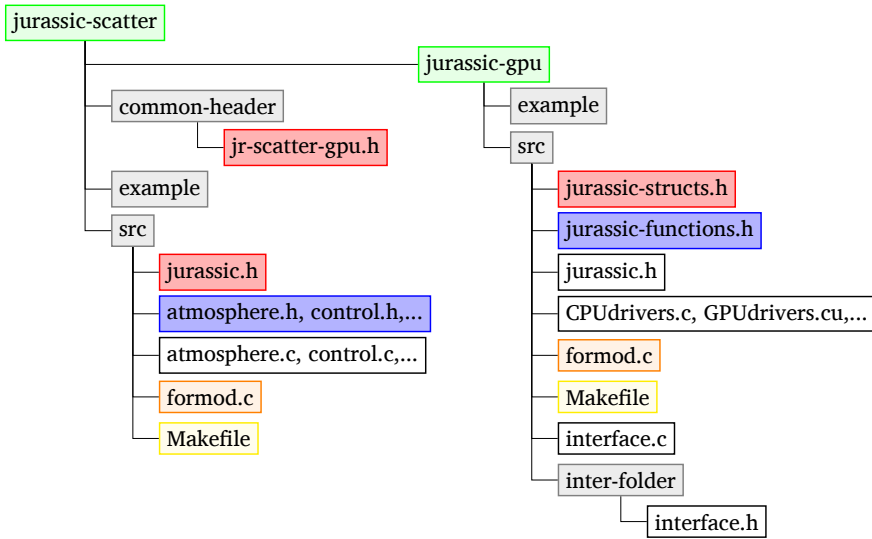


Figure 3.1.: Merged repositories. To connect both codes the git repository of the GPU-accelerated project "jurassic-gpu" is cloned in a subfolder inside "jurassic-scatter".

### 3.2. SOURCE CODE DIFFERENCES

Since the JURASSIC-scatter and JURASSIC-GPU projects are not developed at the same time nor by the same person an important part of merging them was to determine all differences between them. At the moment, we have not resolved all the differences in the best possible way, but we have managed to connect the codes and determine how to resolve these differences more elegantly in the future. The first difference we noticed were the different names of the constants for dimensions, e.g. in JURASSIC-scatter constant which describes the number of rays is called NRMAX, while the same constant in JURASSIC-GPU is called NR. To solve this problem without changing the codes much we decided to have both versions in the common header file, and we used macro definitions to make sure these two constants have the same value.

The main difference between these two projects were the different data structures they use, so that is why we will explain their data structures one by one, the differences between them and the way we solved those differences:

- `ct1_t` – forward model control parameters: Minor differences, so it wasn't hard to have only one `ct1_t` structure in the common header file.
- `los_t` – line-of-sight data: In JURASSIC-scatter struct of arrays (SoA) is used to represent the segments of the line of sight, while in JURASSIC-GPU that is done by array of structs (AoS). The best solution would be only to have arrays of structs, but at the moment we use both of them and convert one type to another when it is necessary.
- `obs_t` – observation geometry and radiance data: Structures from the two projects are almost equal, but the difference is that the indices of the 2d-array `rad` do not

have the same order. In one case it is declared as `rad[NR][ND]` and in another as `rad[ND][NR]`, where NR is number of rays and ND number of detectors or radiance channels. In JURASSIC-scatter-GPU we want to have only one `obs_t` structure and we opted for `rad[NR][ND]` case.

- `tbl_t` – emissivity look-up tables: Here we had similar problem as with `obs_t`, but here we decided to have two different types of emissivity tables in the common header file, because otherwise we would have to adapt too many lines of code. One more difference about JURASSIC-scatter and JURASSIC-GPU emissivity tables is that their data is not read from the disk in the same way. One is optimized to read in binary, and another in ASCII format, so an important future step should be to have unified emissivity tables.
- `atm_t` – atmospheric data: This is the only structure which was completely the same in both projects.
- `aero_t` – aerosol and cloud properties: It is not part of JURASSIC-GPU and that is why it was simple to introduce it to JURASSIC-GPU.
- `ret_t` – retrieval control parameters: Again, only part of JURASSIC-scatter.

## 4. JURASSIC-SCATTER-GPU REALIZATION

The most important part of porting JURASSIC-scatter to GPUs is to identify what this project has in common with JURASSIC without scattering and what are the main differences between them. We have already said that the number of scattering levels  $sca_{mult}$  is one of the input parameters of JURASSIC-scatter forward model which is not present in the JURASSIC forward model. This parameter can also be seen as the depth of recursion. If  $sca_{mult} = 0$ , then JURASSIC-scatter calculates only primary rays which are treated without scattering, so this case is equivalent to JURASSIC without scattering. If  $sca_{mult} = 1$ , then there are primary rays treated with scattering and secondary rays treated without scattering. This case is the most common in practice. In Figure 4.1 a scattering scenario for  $sca_{mult} = 2$  is presented. Note that each ray at the lowest level of recursion will exactly perform the operations provided by the forward model in JURASSIC-GPU. This allows to limit the efforts of porting the scattering module to a restructuring of the high-level routines. To do this we have divided the work done by the JURASSIC-scatter-GPU into 3 phases:

1. CPU-Prepare
2. GPU-Execute
3. CPU-Collect

In the following subsections we will explain these 3 phases in more detail.

## 4.1. CPU-PREPARE

The main idea is that raytracing and recursively calculating the radiance of rays with scattering stays on the CPU and only in the case without scattering it is executed on the GPU, leveraging JURASSIC-GPU CUDA kernels. That is why the CPU-Prepare phase, until coming to the lowest level of recursion does the same thing as JURASSIC-scatter and at the lowest level saves these rays in memory and sends them to the JURASSIC-GPU project. The lowest level rays could be saved in memory with various data structures. Our first idea was to use an array, but we realized that for  $sca_{mult} > 1$  it might be difficult to determine the size of array and ray indices if we want to allocate memory for it statically. That is why we decided to use a queue structure which we will call work-queue because the information about the rays from the lowest level of recursion will be written into it. If we use a call-tree to represent this recursive function calls, as shown in Figure 4.1, by representing each ray by one node we can say that information about leaf nodes is written into the work-queue. First In, First Out (FIFO) is the most important property of the queue structure. It means that leaf nodes of the tree have to be visited in the same order in the CPU-Prepare and CPU-Collect phases, and for this reason we cannot parallelize these phases without further modification.

## 4.2. GPU-EXECUTE

In the GPU-Execute phase the radiation for each ray from the work-queue is calculated using functions from the JURASSIC-GPU project. Rays from the work-queue are divided into smaller packages which are given to JURASSIC-GPU kernel functions in parallel using OpenMP parallelism. More precisely, at the beginning we calculate how many packages can be processed at the same time on one GPU and allocate enough GPU memory for these packages. Then the same number of OpenMP threads is used to calculate the radiances for the rays in the given packages in parallel. With this mechanism GPU memory balancing is ensured.

## 4.3. CPU-COLLECT

The CPU-Collect phase is very similar to the CPU-Prepare phase. The difference is that in this case instead of pushing information about leaf nodes of the call-tree to the work-queue, its radiances calculated in the GPU-execute phase have to be read and passed to the parent node. For levels that are not the lowest level, the calculation is again exactly the same as in the original JURASSIC-scatter.

In the CPU-Prepare subsection we said that because of the FIFO property we had to do additional adjustments to parallelize the CPU-Prepare and CPU-Collect phases. To do that we introduced multiple work-queues. Each primary ray has its own work-queue, so inside a subtree of a primary ray node the calculation has to be performed in serial, but we can OpenMP parallelize over primary rays. Figure 4.2 visualizes this idea.



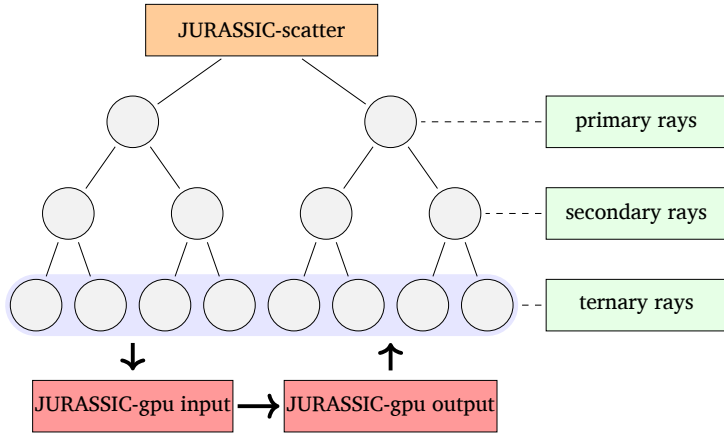


Figure 4.1.: JURASSIC-scatter-GPU. Leaf nodes of the tree submit work packages to a work-queue which then is processed on the GPU.

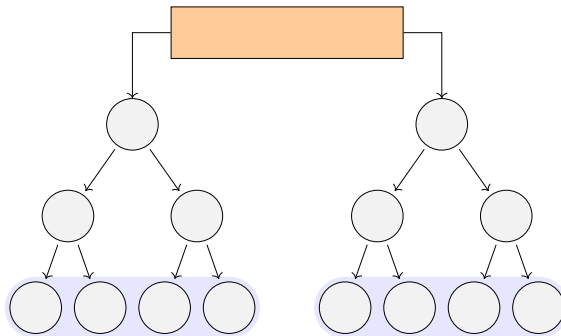


Figure 4.2.: Multiple queues. Separate queues are necessary to exploit thread parallelism during the prepare and collect phase.

## 5. PERFORMANCE RESULTS

All performance results reported here were obtained on the Jülich Wizzard for European Leadership Science (JUWELS) supercomputing system at the Jülich Supercomputing Centre which consists of two partitions: Cluster and Booster [10]. The JUWELS Booster nodes comprise a Dual AMD EPYC Rome 7402 CPU with  $2 \times 24$  cores and four NVIDIA A100 GPUs. The JUWELS Cluster nodes comprise Dual Intel Xeon Platinum 8168 CPU with  $2 \times 24$  cores. JURASSIC-scatter-GPU was benchmarked on a JUWELS Booster and JURASSIC-scatter on a JUWELS Cluster nodes. Since a JUWELS Booster node contains 4 GPUs, we decided to have 4 MPI ranks per node in both cases to use their full capacity. Because of that, for JURASSIC-scatter-GPU each MPI rank has its own GPU and for both JURASSIC-scatter-GPU and JURASSIC-scatter there are 12 OpenMP threads per MPI rank. The CUDA runtime and compiler version are 11.3 while GCC version 9.3.0 was employed to compile the CPU code. Also, to run the programs successfully the GSL/2.6 module has to be loaded.

### ABOUT TEST CASES:

- data observed by the CRISTA-NF airborne instrument which measures the thermal emissions of the atmosphere in the mid-infrared region from 4 to 15  $\mu\text{m}$  in an altitude range from flight altitude (up to 20 km) down to approximately 5 km [9]
- ~2000 test cases
- test cases depend on:
  1. profile of 13 trace gases, temperature and pressure
  2. cloud vertical thickness
  3. cloud layer bottom altitude
  4. particle size distribution (log-normal)
- 32 spectral radiance channels
- 84 primary rays per test case
- $sca_{mult} = 1$ : only primary and secondary rays

Figure 5.1 shows a comparison of execution times of our JURASSIC-scatter-GPU and the reference JURASSIC-scatter program for ten different cloud scenarios. In each scenario the number of primary rays equals 84, but the number of secondary rays depends on the thickness of the cloud because in a thicker cloud more secondary rays are generated per primary ray so both programs need a longer execution time. It can be seen that the achieved speedup is around  $9\times$ .

In Figure 5.2 JURASSIC-scatter-GPU and JURASSIC-scatter are again compared, but here for the same cloud scenario, but different numbers of primary rays. In each of these test cases there are around 2300 secondary rays per primary ray. That is why the execution time of JURASSIC-scatter-GPU increases linearly with the number of primary rays. For the test case with 80 rays the achieved speedup is again around  $9\times$ , and for smaller numbers of rays this number is even better. For example, the speedup for the test case with 10 rays is around  $17\times$ .

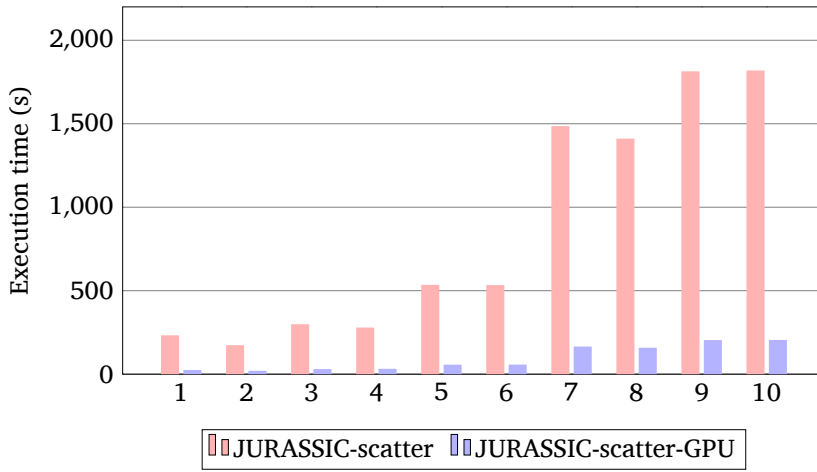


Figure 5.1.: Execution times for different scenarios. JURASSIC-scatter-GPU was benchmarked on NVIDIA A100 GPUs and JURASSIC-scatter on Intel Xeon Platinum 8168 CPUs.

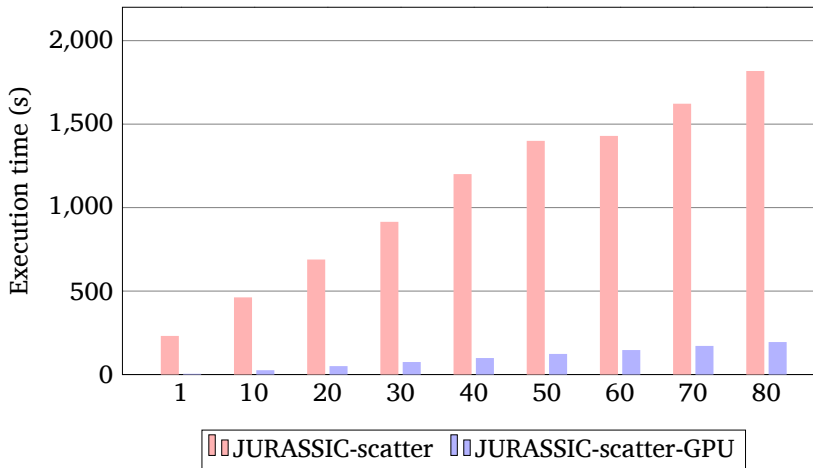


Figure 5.2.: Execution times for different number of primary rays. The achieved speedup is at least 9x.

## 6. SUMMARY & OUTLOOK

We managed to connect the JURASSIC-scatter and JURASSIC-GPU projects by restructuring both codes and introducing a common header file. We identified differences between the JURASSIC and JURASSIC-scatter forward model and used the fact that each ray at the lowest level of recursion in JURASSIC-scatter model performs exactly the same operations as in the JURASSIC model. Because of this we introduced 3 phases in JURASSIC-scatter-GPU: CPU-Prepare, GPU-Execute and CPU-Collect. Performance results show that our JURASSIC-scatter-GPU has a better execution time than JURASSIC-scatter and the speedup factor depends on the test cases, but is at least  $9\times$ .

An additional reason for optimism is given by the fact that in the current implementation raytracing is performed on the CPU, and that after porting of that part to the GPU we could expect even better results. Furthermore, after our work on the JURASSIC-scatter-GPU project, we are much closer to a unified JURASSIC code that would combine all the functions of JURASSIC, JURASSIC-scatter and JURASSIC-GPU codes, which is an even more important result than the speedup factors that we have achieved.

## 7. ACKNOWLEDGMENTS

This report was done during the Guest Student Programme 2021 at Jülich Supercomputing Centre.



I sincerely thank my supervisors Dr. Sabine Griebach, Dr. Paul F. Baumeister and Dr. Lars Hoffman for their valuable advices and their guidance during this research program.

Also, I truly appreciate Dr. Ivo Kabadshow for his great organisation, for always being there for all the guest students, and especially for his help with the presentation.

Furthermore, I am grateful to Prof. Dr. Sc. Zlatko Drmač for helping me to be invited to the program by writing a referral letter.

Last but not least, I want to thank my family and my girlfriend Ana for enormous support during the program.

## REFERENCES

- [1] P. F. Baumeister and L. Hoffmann. **GitHub Source Repository of JURASSIC-GPU**. 2021.  <https://github.com/slcs-jsc/jurassic-gpu>.
- [2] P. F. Baumeister, B. Rombach, T. Hater, S. Griessbach, L. Hoffmann, M. Bühler, and D. Pleiter. **Strategies for forward modelling of infrared radiative transfer on GPUs**. *Parallel Computing is Everywhere*, 32:369–380, 2017.  [doi:https://doi.org/10.3233/978-1-61499-843-3-369](https://doi.org/10.3233/978-1-61499-843-3-369).
- [3] S. Griessbach and L. Hoffmann. **JURASSIC-scatter v1.3 documentation**. 2019.  <https://github.com/slcs-jsc/jurassic-scatter/blob/v1.3/docu/jurassic.pdf>.
- [4] S. Griessbach and L. Hoffmann. **GitHub Source Repository of JURASSIC-scatter**. 2021.  <https://github.com/slcs-jsc/jurassic-scatter>.
- [5] S. Griessbach, L. Hoffmann, M. Höpfner, M. Riese, and R. Spang. **Scattering in infrared radiative transfer: A comparison between the**

- spectrally averaging model JURASSIC and the line-by-line model KOPRA.** *Journal of Quantitative Spectroscopy and Radiative Transfer*, 127:102–118, 2013. <https://www.sciencedirect.com/science/article/pii/S0022407313001969>, [doi:https://doi.org/10.1016/j.jqsrt.2013.05.004](https://doi.org/10.1016/j.jqsrt.2013.05.004).
- [6] L. Hoffmann and M. Alexander. **Retrieval of stratospheric temperatures from Atmospheric Infrared Sounder radiance measurements for gravity wave studies.** *J. Geophys.*, 114, 2009.
- [7] L. Hoffmann, M. Kaufmann, R. Spang, R. Müller, J. Remedios, D. Moore, C. Volk, T. von Clarmann, and M. Riese. **Envisat MIPAS measurements of CFC-11: retrieval, validation, and climatology.** *Atmos. Chem. Phys.*, 8:3671–3688, 2008.
- [8] L. Hoffmann. **GitHub Source Repository of JURASSIC.** 2021. <https://github.com/slcs-jsc/jurassic>.
- [9] C. Kalicinsky, S. Griessbach, and R. Spang. **Radiative transfer simulations and observations of infrared spectra in the presence of polar stratospheric clouds: Detection and discrimination of cloud types.** 07 2020. [doi:10.5194/amt-2020-144](https://doi.org/10.5194/amt-2020-144).
- [10] S. Support. **JUWELS: Modular Tier-0/1 Supercomputer at Jülich Supercomputing Centre.** *Journal of large-scale research facilities JLSRF*, 5, 02 2019. [doi:10.17815/jlsrf-5-171](https://doi.org/10.17815/jlsrf-5-171).

## A. APPENDIX

### A.1. HOW TO BUILD AND RUN THE PROGRAM

In this appendix, we explain how to connect the JURASSIC-scatter and JURASSIC-GPU projects with git clone & branch and how to run the program. Also, we introduce new input parameters that are not part of the JURASSIC-scatter v1.3 documentation [3].

#### GETTING STARTED

As we explained in Section 3, for creating JURASSIC-scatter-GPU no new repository is created, but to run it you have to clone the `dev` branch of the JURASSIC-scatter project and then clone the `scatter` branch of the JURASSIC-GPU project into it.

JURASSIC-GPU has to be compiled first, wrapped into C static library which is then linked to JURASSIC-scatter. Test cases and scripts we used during development are given in the `dev` branch of the JURASSIC-scatter project. There are two sets of tests: small and large. We will here show how to run the program on the small testset, but in the second case the procedure is the same. You have to make sure that all modules we need are loaded and then get into the `small_testset` folder. This folder contains 10 test cases, their common control file `cloud-785-798.ct1`, and a few scripts: `submit.sh`, `jurun-ice-785.sh` and `diff.py`, which are used to compile, run and benchmark the JURASSIC-scatter-GPU program and check the correctness of the resulting radiances. To do all that you only need to run the `submit.sh` script. Listing 1 presents how to do it.

**LISTING 1: CLONE, LOAD MODULES AND SUBMIT**

```

git clone --branch dev https://github.com/stjepan/jurassic-scatter.git
cd jurassic-scatter
git clone --branch scatter https://github.com/stjepan/jurassic-gpu.git
cd testing
source module_load.sh
cd small_testset
./submit.sh

```

**NEW INPUT PARAMETERS**

Compared to the JURASSIC-scatter v1.3 documentation [3], in JURASSIC-scatter-GPU we have two new control parameters, which are in our case written in the `cloud-785-798.ct1` control file. You can see more information about the new parameters in Table A.1.

Depending on the values of these two new parameters different modules are activated:

- if `MAX_QUEUE=0`: the work-queue is not used, so this module is very similar to the original JURASSIC-scatter v1.3
- if `MAX_QUEUE<0`: the memory for `|MAX_QUEUE|` rays is statically allocated for the work-queue, but in this scenario the Execute phase is done on CPUs, as part of the JURASSIC-scatter implementation
- if `MAX_QUEUE>0` and `USEGPU=0`: the Execute phase is again performed on CPUs, but as part of the JURASSIC-GPU implementation
- if `MAX_QUEUE>0` and `USEGPU=1`: the Execute phase is performed on GPUs, if JURASSIC-GPU is not compiled with CUDA, the program will abort with an error
- if `MAX_QUEUE>0` and `USEGPU=-1`: the Execute phase is tried to be performed on GPUs, if JURASSIC-GPU is not compiled with CUDA then the CPU-Execute phase from JURASSIC-GPU implementation will be used

These different modules are also shown in Figure A.1.

Table A.1.: New control flags for JURASSIC-scatter-GPU

flag name	purpose	default	options
Accelerating parameters			
MAX_QUEUE	upper bound of number of rays in the work-queue	0	0: do not use work-queue >0: call JURASSIC-GPU functions <0: do not call JURASSIC-GPU functions, in that case size of the work-queue is <code> MAX_QUEUE </code>
USEGPU	Use GPU-accelerated formod implementation	0	0: never 1: always -1: if possible

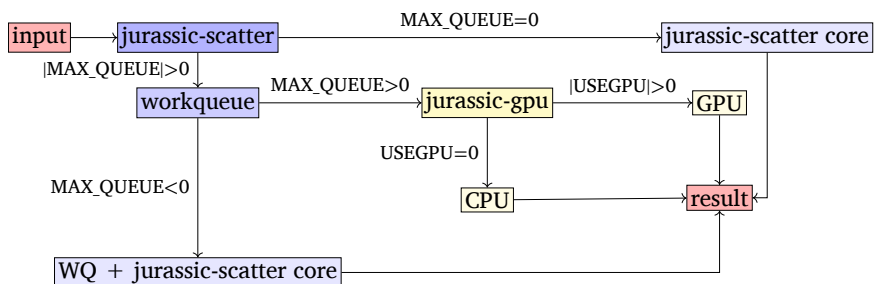


Figure A.1.: JURASSIC-scatter-GPU dataflow. Using a work-queue structure, JURASSIC-scatter can call JURASSIC-GPU as solver.

# GAME TREE IMPLEMENTATIONS USING PYTHON

## Analysis of $m, n, k$ games

**Abstract** *In this project, we try to solve  $m, n, k$  games. Our investigation of  $m, n, k$  games was carried out with a Python implementation of the minimax algorithm, which was extended with alpha-beta pruning and heuristic algorithms that check for shortcuts in the move finding process. We will try to get closer to board sizes where the value of  $m, n, k$  games is still unknown. In particular, we are interested in where the transition from a drawn game to a win for the first player occurs. Furthermore, we verified game values for  $m < 5$ ,  $n < 5$ ,  $k < 5$  game and achieved considerable speedup compared to the naive minimax implementation.*

Mert Saner  
Scientific Computing  
Technical University of  
Berlin  
Germany  
saner@campus.tu-berlin.de

## 1. INTRODUCTION

Games have been one of the entertainment activities of humankind for thousands of years. Historians believe that the origin of the first games goes back to 6000 BC. Throughout the years games have evolved and different types of games have appeared. However, until the last century games have been played by humans only and people have required another person to play the game. Starting in the 1960s, the development of computer systems showed that machines could be opponents of the humans in the games. Computer opponents in the games were one of the first examples of artificial intelligence (AI) in the history of humankind. The birth of AI was declared around the same years in the Dartmouth workshop of 1956. The proposal for the conference included the assertion that "every aspect of learning or any other feature of intelligence can be so precisely described that a machine can be made to simulate it"[7].

### 1.1. GAMES & ARTIFICIAL INTELLIGENCE

One of the important reasons that make games a research area for AI is real-world applications can be modelled as games. Every game logic constitutes game theory in it. Although game theory covers a broader range of application areas than games, logic in the games could be applied to areas that use game theory in it. This includes a wide range of areas including but not limited to operations research, economics, finance, regulation, military, politics, and energy[4, 1]. In the early days, the starting point of the AI plays a board game was in the sense that human intelligence is copied by the computer. Initially, computers tried to emulate the human approach. However, later on, it is found out that human-like strategies are not necessarily the best computational strategies[5].

In the 1960s when computers have in their very early development scheme, we did not see perfect computer players. Starting from late the 1990s, AI in computers are even more developed thanks to improvements in information technologies and algorithms. IBM Deep Blue computer was one of the first examples of this phenomenon. Nowadays,



good chess software in mobile phones can beat Garry Kasparov. Nevertheless, not every game structure is like chess and hence not every game has a superhuman AI player. The reason for this is some games are so hard that we are not able to explore all relevant parts of the state space. One of those games is  $m, n, k$  games for large  $m \times n$  board. With this project, we aim to move one step closer to unknown game values and see the possibilities we have.

It is important to classify the games before analyzing them from a scientific standpoint. Games can be classified according to many different characteristics[9]. We will classify the games according to whether:

- Game is concurrent or round-based
- Players involved are cooperative or against each other
- State space is perfect information or imperfect information
- Observations and actions are deterministic or stochastic
- Zero-sum game or non-zero-sum game

In this paper, we will focus on a two-player deterministic zero-sum game with perfect information that always results in win, loss or draw. Despite their simple rules, there are still unanswered questions about the game values of some of the  $m, n, k$  games. The objective is to find a draw/win transition for large game boards with big  $k$  values. We will discuss the methodology and implementation steps used throughout the guest student program 2021.

## 2. M,N,K GAME

### 2.1. DEFINITION AND FUNDAMENTALS

In the  $m, n, k$ -game board game, two players take turns in placing their marks on an  $m \cdot n$  board. The winner is the first to get  $k$  of their mark in a row, in a horizontal, vertical or diagonal manner. Tic-tac-toe is the 3, 3, 3 game and freestyle Gomoku is the 15, 15, 5 game. An  $m, n, k$ -game is also called a  $k$ -in-a-row game on an  $m \times n$  board. The purpose of analyzing  $m, n, k$  games is to find their game-theoretic values. Game value is the result of the game with perfect play. Finding the game value of the game is known as solving the game. In this project, we try to solve the  $m, n, k$  games using the Python programming language. Despite the simple rules, there are still unanswered questions about the values of  $m, n, k$  games.

### 2.2. TREE STRUCTURE

To solve  $m, n, k$  games we used game trees as a tool for our research. In graph theory, a tree is an undirected graph in which any two vertices are connected by exactly one path, or equivalently a connected acyclic undirected graph. [Figure 2.1a](#) shows how the tree structure is represented. The root node is starting node of the tree. [Figure 2.1b](#) show tree structure with the parent and child nodes are implemented. Every node that is a step above its child node is called a parent node.

Every single step in the game is demonstrated by nodes in the game tree structure. After each step, there are one or more possibilities that the player can play and these

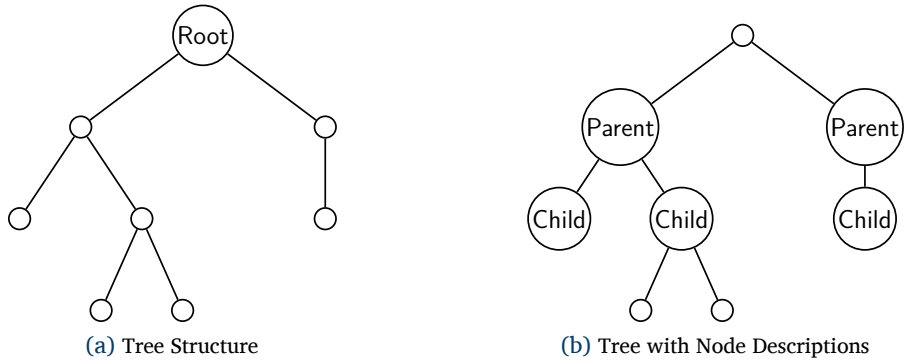


Figure 2.1.: Trees

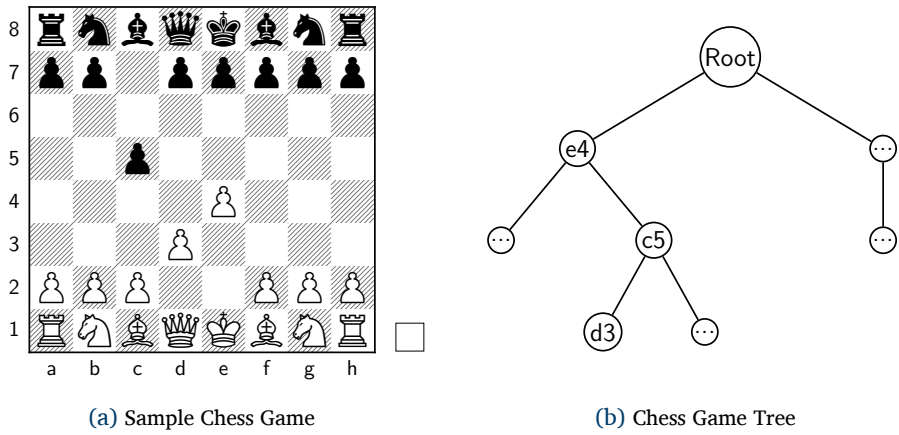


Figure 2.2.: Chess Game

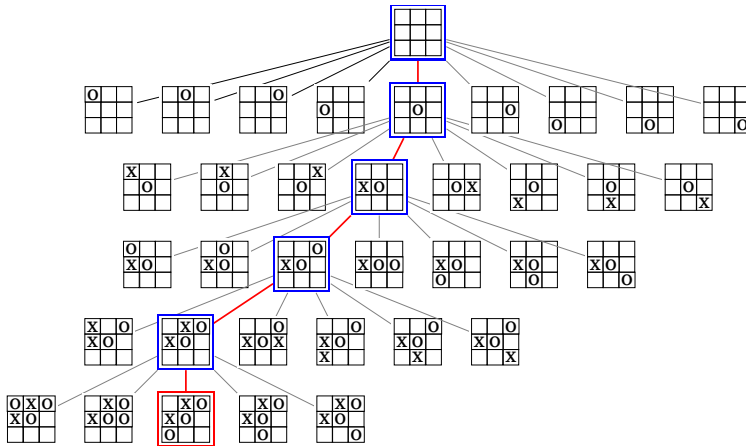


Figure 2.3.: Tic-Tac-Toe Game Tree Structure

are represented by connected child nodes to the parent node. Figure 2.2b shows how the third step of the white player could be represented as a game tree. Similarly, we can also analyze the  $m, n, k$  games using a tree structure. Figure 2.3 shows the tree structure of Tic-Tac-Toe game. The first player is O who can play nine different places in total and the opponent X could play the remaining eight places. When one of the players reaches three consecutive marks they win the game. One of the winning cases for the O player is red coloured and can be seen in the figure. Since the game value of Tic-Tac-Toe is zero if both players play optimally the result is a draw. Hence, to enter a winning case like in Figure 2.3, one player must make a wrong move in some part of the game [3].

### 3. IMPLEMENTATION

#### 3.1. APPROACH

The problem with the big game tree implementations is the complexity. In the Tic-tac-toe game, there are 9 possible first moves, 8 for the second moves, and in total approximately  $9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 9! = 362880$  different combinations. Although this number shows worst-case scenarios and games can end up earlier, there is huge complexity in  $m, n, k$  games.

#### 3.2. MINIMAX

Minimax is a decision rule that is widely used in computer science and artificial intelligence. The maximizing player always tries to maximize game value while the minimizing player tries to minimize the game value. Moreover, zero game value define draw between the players. Figure 3.1 and Figure 3.2 show how minimax functions work. Since we are analyzing the Tic-Tac-Toe game there are there different possible outcomes which are win, draw and loss. Leaf of the value of 1 in the graph defines win, zero defines draw, -1 define loss in the game. According to the outcome of each step, players can take play various moves and it can lead to different leaf values. To find game value, we need to look at the case in which both players play optimally. According to where

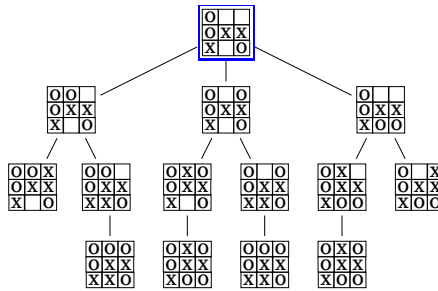


Figure 3.1.: Tic-Tac-Toe Tree Structure

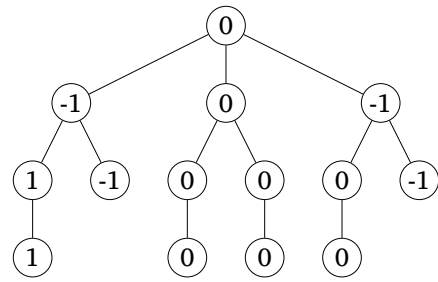


Figure 3.2.: Leaf Values and Game Value

the player moved you can see the values vary between -1, 0 and 1. In Figure 3.2, O is the maximizing and X is the minimizing player. The root value of the tree is zero, which means that the given situation is a draw if both players play optimally. The value of the root node defines the game value and it is the value we are looking for. The logic of finding the game value is valid not only for 3,3,3 games but also for bigger games like 5,5,5.

### 3.3. GAME TREE COMPLEXITY

Tree Complexity is defined as the number of leaves in the tree structure. General rule of thumb for the  $m, n, k$  is;

$$(m \cdot n)^k$$

For instance; the 4, 4, 4 game requires 16! and Gomoku requires at most 225! different game combination. Hence, the implementations require high computing power and parallelization.

### 3.4. NAIVE MINIMAX ALGORITHM

First of all, we started to implement the Tic-Tac-Toe game in Python. The reason for this is, although Tic-Tac-Toe is a fairly simple game and its state space smaller than more advanced games, it is a 3, 3, 3 game and the purpose was to create a solid source code base that we could extend the initial application to the bigger boards. In the first step, we created a Python program in which two AI players compete against each other. The program worked and we verified the game value 0 in three seconds. The next step was

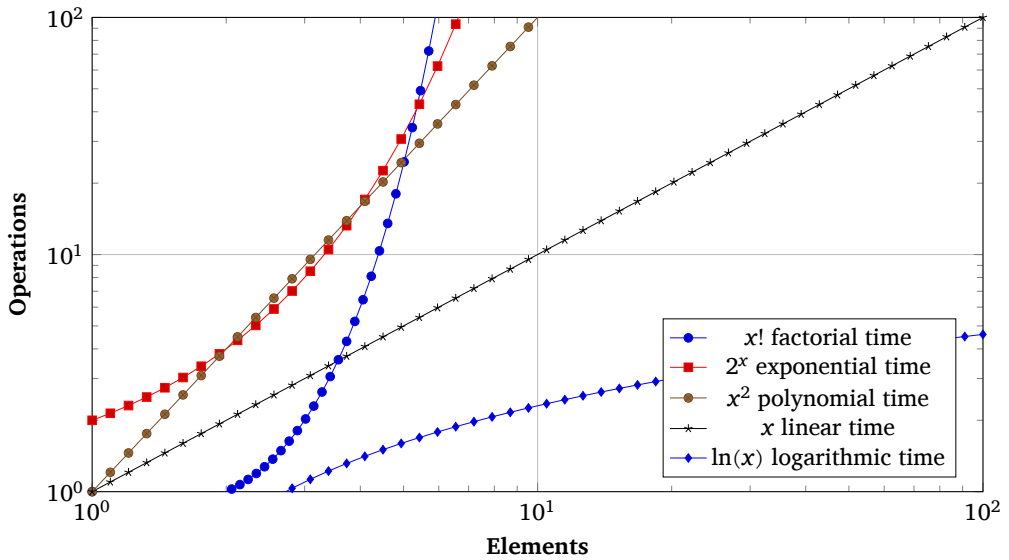


Figure 3.3.: Factorial Time Complexity Increase

---

**Algorithm 1** Minimax Algorithm[6]

---

```

procedure MINIMAX(node, depth, maxPlayer) is
  if depth = 0 or leaf node then
    return the value of node
  if maxPlayer then
    value :=  $-\infty$ 
    for each child of node do
      value :=  $\max(\text{value}, \text{MINIMAX}(\text{child}, \text{depth} - 1, \text{false}))$ 
    return value
  else (min player)
    value :=  $\infty$ 
    for each child of node do
      value :=  $\min(\text{value}, \text{MINIMAX}(\text{child}, \text{depth} - 1, \text{true}))$ 
    return value

```

---

---

**Algorithm 2** Alpha-Beta Pruning Algorithm[8]

---

```
procedure ALPHABETA( $\square$ node, depth,  $\alpha, \beta$ , maxPlayer) is
  if depth = 0 or leaf node then
    return the value of node
  if maxPlayer then
    value :=  $-\infty$ 
    for each child of node do
      value := max(value, ALPHABETA(child, depth - 1,  $\alpha, \beta$ , false))
      if value  $\geq \beta$  then
        break (*  $\beta$  cutoff *)
       $\alpha$  := max( $\alpha$ , value)
    return value
  else
    value :=  $\infty$ 
    for each child of node do
      value := min(value, ALPHABETA(child, depth - 1,  $\alpha, \beta$ , true))
      if value  $\leq \alpha$  then
        break (*  $\alpha$  cutoff *)
       $\beta$  := min( $\beta$ , value)
    return value
```

---

to make the board size larger. We tried 4, 4, 3 games and ran our program. However, the program struggled to make the first move and even after 10 minutes and it did not make the first move. The reason for this is that we used the naive minimax function to find the values. We quickly realized that is not useful for any application bigger than 3, 3, 3.

In all of our implementations, we define X is as a starting player. To test whether our new 4, 4, 3 program working, we set the  $4 \times 4$  table with four moves initialized earlier. So the program ran on the table which is initialized with four moves played. Even with the initial move setup, it took 126 seconds to run 4, 4, 3 game. The program showed that X is the winner. Since X is the winner as a result, we know that the game value of that game is 1 and if it is draw then the game value is 0.

After our first implementation, we realized that we need to improve the minimax function and need to add heuristic algorithms and so that the program will enter minimax only if there is no other option in that particular case. Hence, we integrated the Alpha-Beta Pruning algorithm into our initial implementation to improve the initial minimax function.

### 3.5. ALPHA-BETA PRUNING ALGORITHM

Alpha-Beta pruning is a search algorithm whose purpose is to decrease the number of iterations made by minimax. It is a tree search algorithm. Since the m, n, k game is an adversarial game, it was well suited for our implementation. The advantage of using alpha-beta pruning instead of the initial version was the reduced complexity of the state space. The idea behind the alpha-beta pruning was to prune off the parts of the tree which are not needed. For instance, if there is a move that is known to be better, then it

would cut the unnecessary branches of the tree.

On the other hand, minimax was looking at every single configuration available in the game and so that it was so slow. After alpha-Beta pruning was implemented, we saw drastic improvements in our program. Moreover, alpha-beta pruning is more efficient if good moves are explored early stage of the game. The effects of the alpha-beta pruning algorithm are shown in the results section.

## **3.6. HEURISTIC ALGORITHMS**

### **3.6.1. MOVE GENERATION FUNCTIONS**

Move generation functions can be divided into two types, which are the matein function and saving move function. We also created three matein functions for the program. These are mate in one, mate in two and mate in three functions. As the name suggests the functions check in every iteration of the loop program whether is mate in one, two or three. If there is such a case then the program does not enter to minimax function instead it enters the heuristic algorithm. Thanks to this implementation we have seen dramatic speed improvements over the initial implementation for special cases.

### **3.6.2. BOARDMAKER FUNCTION**

After evaluating bigger tables with a  $k = 4$ , we realized that most of the computing time and power was spent in the first steps of the board. Hence, we developed the board maker function, which initializes the board according to the heuristic structure. To make this heuristic structure we analyzed many different games. After analyzing different gameplays and researching in literature we decided that placing the mark to the centre is the most advantageous for the first player[2]. Hence, we created a heuristic function to place the first of moves of the game to the centre of the board.

One of the second observations we made was it is better to play near the cluster of other marks to reach the goal. It also does not make sense to play near corners and far from the main cluster of the marks since that makes finishing harder for the player and give the positional advantage to the opponent. We created a board maker function in a way that it initiates the board by playing one random move in the centre of the board and then the opponent plays one move in the centre for  $3 \times 4$  board. These given first two moves make the program evaluate upcoming moves more efficient than earlier implementation thanks to decreasing complexity. In the bigger board like  $4 \times 4$ , it plays three moves to the centre square. That reduces minimax function steps and the program enters matein functions more quickly especially for  $k = 3$ .

## **3.7. TOOLS**

We implemented all algorithms in the Python programming language. The easy syntax of Python enables us to focus on programming itself instead of the syntax of the language. I used Pycharm as an integrated development environment (IDE). Pycharm is fully compatible with the Git version control system. Hence, without requiring a command line, I was able to post the latest version of the program to GitHub. Hence, my supervisor was able to see every single change I did in the program. Sometimes, especially while testing different board sizes, the program was requiring more than 10 minutes of running time. Since that cause my local machine to heat up and that makes machine unable to run any other Python program, I took advantage of one of the nodes in supercomputers JURECA, JUWELS in Jülich Supercomputing Centre (JSC) and a Ubuntu server in Amazon

Web Services (AWS). They help me to run the same program in different board sizes and get results simultaneously.

## 4. RESULTS

After implementing alpha-beta pruning and heuristic functions we reached more efficient program. We verified solutions for  $m < 5$ ,  $n < 5$  and  $k < 5$ , and found their game values by using different game tree implementations and the aforementioned algorithms.

### 4.1. SPEED

The biggest impact of all improved algorithms was speed improvements. After incorporating the techniques dramatic speed improvements was obvious in the run time of the program. Tic-Tac-Toe full artificial intelligence program runs  $\sim 0.3$  seconds compared  $\sim 3$  seconds initial on implementation. We have reached more than 10 times faster working code thanks to these improvements. After implementing heuristic algorithms we have reached efficient working Python code that can solve 4, 4, 4 game and find its game value. In our machine MacBook Air M1, we can solve 4, 4, 4 game for around six minutes. When we run the program every game steps and at the end, its game value of zero can be seen in the console output. Zero game value of 4, 4, 4 demonstrates that it is a draw game.

### 4.2. LARGE STATE SPACE

Efficient and faster working code also allowed us to implement our program on larger  $m \times n$  boards. We tried to confirm results for 9, 4, 4 and tried to solve even larger games like 8, 9, 5; 8, 10, 5; 8, 11, 5 and explore the draw/win transition for  $k = 5, 6, 7$ . The complexity of the game tree directly correlated with the size of the board. Hence, larger boards were hard to tackle at the beginning for us. However, in the later steps, we were able to solve large board games including  $5 \times 5$  and  $6 \times 6$  game in a matter of seconds for the  $k = 3$ . The real challenge appeared when we increased the  $k$  value. With  $k = 4$  the biggest board we were able to solve in a reasonable time was  $4 \times 4$ .

### 4.3. PARALLELIZATION

One of the other important results we reached was to tackle unknown game values, immense computing power was required. Although we used supercomputers in our project, their main purpose was to work as an additional server to run the program. Hence, we only used one of their nodes to run the program. To compare the speed of the machines we utilize, an example could be useful. In the very early stage of our project, we run the same 3, 4, 3 AI versus AI implementation on three different machines namely, MacBook Air M1, JURECA Supercomputer and Ubuntu Server in AWS. In JURECA, it is completed in 853.008 seconds. In MacBook, it took 921.854 seconds. So there was only  $\sim 1$  minute difference between one node of the supercomputer and MacBook. On the other hand, the Amazon AWS instance added additional two minutes and finished it in 1042.004 seconds. This was the early implementation of our program and now the same program run in MacBook in 1.237 seconds, and in JURECA 1.334 seconds, in AWS 1.443 seconds. The important point we would like to make is there was not a big difference between running times of machines despite their very different hardware specifications. Hence, to explore the game tree faster, much more computing power is needed. JURECA



has that much power but we did not utilize this power. Therefore, the result was to take advantage of the immense power of the supercomputer, parallelization of the nodes is needed. This will unleash the full power of the supercomputer and it will make it the perfect tool to improve the program. My supervisor and I believe to find unknown game values for  $m, n, k$  games, parallelization of the supercomputer are required.

## 5. CONCLUSION & SUMMARY

We verified game values for  $m < 5, n < 5, k < 5$  games and achieved considerable speedup compared to the naive minimax implementation. It is observed that it is possible to increase game tree search speed by a considerable margin by using efficient algorithms. More advanced algorithms and computing power are required for the larger  $m \times n$  board since the games become more sophisticated in every more advanced step. Although we improved base minimax function a lot and reached impressive results. To find game values of unknown games like 15, 5, 5 there is a long way to go. One of the reasons for this is complexity increases in the factorial time (upper bound). Therefore, although with proper implementation of heuristic algorithms complexity can be decreased enormously. Considering native minimax evaluating every single move, it is complexity was so high in our initial implementation. The problem with factorial time complexity is that after increasing complexity this much, most of the algorithms become obsolete. This was particularly valid for very efficient working algorithms which work in comparatively small state space.

We believe adding more heuristics will bring this implementation to an even better stage. Nevertheless, adding heuristics to the program is not straightforward. It requires lots of time and critical analysis. We believe this research project created a solid ground base for the scientists who would like to move one step further by using our findings. In the following section, we list some of our projections that can be useful for the researchers who would like to build on our research.

## 6. POSSIBLE FUTURE IMPLEMENTATIONS

More improvements could be added to the program. These improvements will make the program faster. One of the improvements that could be done is making use of symmetries. There are two different symmetry axes in non-quadratic (rectangular) boards, and they have in total four symmetries. In the quadratic board, there are four different symmetry axes and eight in total. If there are functions implemented that take advantage of these symmetries then the program can understand that most of the starting phases are equivalent to each other. Therefore, state-space could decrease to one-fourth of the initial version and decrease the complexity by a factor of 4. Secondly, there can be a live database implemented to the program. The advantage of this running database is that it can prevent multiple evaluations. It will help to cut more unnecessary leaves which will result in a more efficient working program. Thirdly, we thought that a function called double threat check can be added. The function could be improved mate in two functions. It would check if there is a double threat and then the program plays it and does not enter the minimax algorithm[10].

## 7. ACKNOWLEDGMENTS

I would like to thank my family and friends, who supported me throughout this process. I am very thankful to the program director Dr. Ivo Kabadshow who gave me the opportunity to join this program. His constant support made this program a very enjoyable experience. At the same time, I would like to express my deepest thanks to my supervisor Julian Clausnitzer who guide me in each step of our project. He helped me to understand advanced topics and always answered my questions and concerns throughout the program.

## REFERENCES

- [1] B. A. Bhuiyan. **An overview of game theory and some applications.** 59:121–122, 2016. <https://www.banglajol.info/index.php/PP/article/view/36683/24721>.
- [2] V. Charatsidis. **Solving four-in-a-row on  $9 \times 4$  and  $10 \times 4$  boards, Bachelor Thesis.**, 2017.
- [3] J. Clausnitzer. **A stochastic approach to game solving, Master Thesis.** 2020.
- [4] S. I. Gass. **What is game theory and what are some of its applications?** 2003. <https://www.scientificamerican.com/article/what-is-game-theory-and-w/>.
- [5] Y. B. Jonathan Schaeffer, Neil Burch. **Checkers is solved.** *Scienceexpress*, 10.1126:1, 2007. <http://dit.unitn.it/~montreso/asd/docs/checkers.pdf>.
- [6] **Minimax.** <https://en.wikipedia.org/wiki/Minimax>.
- [7] J. McCarthy, M. Minsky, N. Rochester, and C. Shannon. **A proposal for the dartmouth summer research project on artificial intelligence.** page 1, 1955. <http://raysolomonoff.com/dartmouth/boxa/dart564props.pdf>.
- [8] S. J. Russell and P. Norvig. **Artificial intelligence modern approach.** pages 144–158, 2003. <https://www.cin.ufpe.br/~tf12/artificial-intelligence-modern-approach.9780131038059.25368.pdf>.
- [9] U. Schmid and D. Wolter. **Introduction to artificial intelligence.** 2019. [https://cogsys.uni-bamberg.de/teaching/ss21/eki/slides/lecture\\_4\\_games.pdf](https://cogsys.uni-bamberg.de/teaching/ss21/eki/slides/lecture_4_games.pdf).
- [10] J. W. Uiterwijk. **Solving strong and weak 4-in-a-row.** *IEEE*, pages 3–5, 2019. [https://iee-cog.org/2019/papers/paper\\_115.pdf](https://iee-cog.org/2019/papers/paper_115.pdf).

# GPU-BASED OPTIMIZATIONS FOR WALBERLA

## Porting WALBERLA to CUDA task graphs.

**Abstract** *WALBERLA's workflow for 2 different timestep strategies was moved to CUDA task graphs. Different approaches for constructing the communication graphs were tested. A more effective fan-out of kernel execution was achieved for the pack and unpack kernels. CUDA task graphs allowed the code to be restructured and simplified by removing the manually created forking streams for each kernel, as well as the additionally introduced CUDA API calls by the stream manager. All this led to performance improvement. NVTX was added for more comprehensive Nsight Systems reports. The bottlenecks for merging the 4 graphs into one were investigated and listed. Some remedies were suggested.*

Milena Veneva  
student trainee at R-CCS  
RIKEN  
Bulgaria  
milena.p.veneva@gmail.com

## 1. INTRODUCTION

WALBERLA (acronym for widely applicable Lattice Boltzmann from Erlangen) [3, 27, 26] is a massively parallel framework for general-purpose multiphysics simulations whose first prototype was released in 2007. It consists of three main modules: computational fluid dynamics simulations with the lattice Boltzmann method (LBM) (this is the module of interest in this report); rigid particle dynamics simulations of particulate systems with the discrete element method or hard-contact models; and phase-field simulations.

WALBERLA consists of more than 2800 source files, as well as it maintains auto-generation of target-specific LBM kernel codes. Besides that, its code also has many levels of abstraction, and uses different parallel programming techniques such as CUDA, MPI, and OpenMP.

### 1.1. PROBLEM DESCRIPTION AND RESEARCH AIM

The open questions that we wanted to answer in this project were:

- is it possible to move WALBERLA's workflow (which runs on concurrent CUDA streams) to CUDA task graph(s)?
- If affirmative, does this bring performance benefits? How much?
- Could CUDA task graphs help for better parallelization of the application?
- Any other optimizations and recommendations are welcomed.

The case study application is `lbm_pull_srt_d3q27`, that is, a 3-dimensional lattice Boltzmann method with 27 probability distribution functions (PDFs) per cell, pulling updating scheme (that is, in the time loop the streaming comes before the collision), and single-relaxation time (SRT) approximation model for the collision operator.

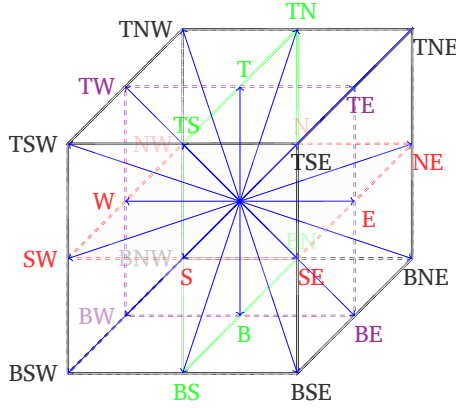


Figure 1.1.: Lattice cell for the D3Q27 LBM model.

The layout of the report is as follows: in the next section 2, the outline of the lattice Boltzmann method is given. Afterwards, in Section 3 the CUDA task graphs are introduced and discussed. Section 4 contains code snippets for porting a single kernel to CUDA task graph. In Section 5 the experimental setup is presented, and the obtained results from the computational experiments are analyzed. Section 6 shows results from using JUBE. Last, in Section 7 are listed future work recommendations, and in Section 8 the summary is given.

## 2. LATTICE BOLTZMANN METHOD

### 2.1. INTRODUCTION TO LBM

Lattice Boltzmann method (LBM) [16, 7, 2, 5, 1] is a mathematical technique for simulating fluid flow based on mesoscopic description (i. e. relying on particles' clusters probability distribution functions). The computational domain is discretized by using a stencil of lattice cells  $DdQq$ , that is,  $d$ -dimensional domain in which every cell contains  $q$  probability distribution functions (PDFs)  $f_i(\vec{x}, t), i = \overline{1, q}$ . This means that the system has  $q$  degrees of freedom. Each PDF determines the probability of the fluid particles' cluster  $i$  to move from position  $\vec{x}$  to  $\vec{x} + \vec{v}_i dt$  after a period of time  $dt$ . Frequently used LBM lattice cells models are e. g. D2Q9, D3Q15, D3Q19, and D3Q27 (for a schematic visualization of a lattice cell for the D3Q27 model see Figure 1.1). In particular, D3Q27 associates 26 PDFs with the particles' clusters that move to the neighboring lattice cells, and one PDF to the resting particles.

The method consists of two main steps – collision, and streaming (propagation). During the former, all particles which are in one and the same cell interact with each other (collide), but do not execute any movement to different cells. At the beginning of the collision step each cell linkage has a pre-collision incoming PDF value  $f_i^{in}$  which after being updated, is converted to a post-collision outgoing PDF value  $f_i^{out}$  at the end of the collision. The simulation does not move in time during this step. On the next step, i. e. the streaming, the particles move to the neighbor cells according to their motion direction (lattice link) and this leads to a simulation timestep. All the post-collision

outgoing PDFs from the collision step are transferred to the neighboring lattice cells according to the direction of their velocities, and become incoming pre-collision PDFs in their new cells. The values of the PDFs do not change during this step.

There are two main strategies for updating the PDFs – *pull* (that is, the streaming comes before the collision in the time loop), and *push* (the collision takes place before the streaming in the time loop).

Many different boundary conditions could be applied to LBM. A list of all the BCs that WALBERLA maintains work with can be seen in [25]. Below, we are going to use UBB (velocity bounce-back), and no-slip (bounce-back solid walls). The first adds velocity to the molecules which bounce to the boundary, while the second means that the molecules switch direction and remain at the same place (see Figure 2.2).

## 2.2. LBM FEATURES

The features of LBM that make it a preferable method in comparison with other computational fluid dynamics methods are as follow:

- the method is explicit in time;
- the two stages LBM consists of (namely, collision, and streaming) are local processes. This together with the explicitity make LBM an embarrassingly parallelizable. In particular, by applying domain decomposition the computational domain is partitioned into several sub-domains that are processed by different processors. The data communication is done through halo cells exchange once per timestep. Communication is of importance for the boundary and near boundary cells, while the inner part of the computational domain does not rely on the exchanged data.
- extensibility: by changing the local equilibrium formula for the PDFs (it must satisfy conservation of mass and momentum), and the relaxation scheme for approximation of the collision operator (e. g. single-relaxation time (SRT), two-relaxation time (TRT), and multi-relaxation-time (MRT)), many different complex physical phenomena can be simulated;
- stability: allows for larger timesteps than with other computational methods (the stability of LBM is conditional, because the scheme is explicit in time, but it is much better than the stability of FDM for instance);
- computational efficiency;
- simplicity of the arithmetic calculations;
- ease of BCs implementation.

## 2.3. LBM THEORY

The main LBM equations [5, 1] are given below.

- Boltzmann transport equation describes the rate of change of the molecular density (i. e. molecules per volume per velocity) in a cell taking into account that the molecules interact with each other:

$$\frac{\partial f}{\partial t} + \frac{\partial f}{\partial \vec{x}} \vec{v} = \Omega - F, \quad (2.1)$$

where  $F$  is the force term,  $\Omega$  is the collision operator.

- The collision operator can be modelled by different models. For instance, a single-relaxation time (SRT) Bhatnagar-Gross-Krook collision operator is defined as follows:

$$\Omega = -\frac{f - f^{(eq)}}{\tau} = -\omega(f - f^{(eq)}), \quad (2.2)$$

where  $f^{(eq)}$  is the Maxwell-Boltzmann distribution,  $\tau$  is the relaxation time which controls the rate of approaching equilibrium;

- The Maxwell-Boltzmann distribution function has the following form:

$$f^{(eq)} = \rho \left( \frac{m}{2\pi KT} \right)^{\frac{D}{2}} e^{-\frac{m(\vec{v}-\vec{u})^2}{2KT}}, \quad (2.3)$$

where:  $\rho$  – macroscopic density,  $u$  – macroscopic velocity,  $T$  – macroscopic temperature,  $K$  – Boltzmann constant,  $m$  – mass of the gas molecule, and  $D$  – # of spatial dimensions.

- Taking Equations (2.1) and (2.2); discretizing the velocities and the time we obtain the lattice Boltzmann equation:

$$\underbrace{\bar{f}_i^{(n+1)}}_{\text{streaming}} = \underbrace{\bar{f}_i^{(n)} - \frac{1}{\tau} [\bar{f}_i^{(n)} - f_i^{(eq)(n)}]}_{\text{collision}} - \left[ 1 - \frac{1}{2\tau} \right] F_i^{(n)} \delta t, \quad (2.4)$$

where  $n$  is the timestep, the bar over  $f$  denotes that this is a modified function. Equation (2.4) is usually divided into two parts – collision, and streaming.

## 2.4. APPLICATIONS OF LBM

LBM is used for modelling and simulation of many different problems and application [5, 1]. For instance,

- oil-water displacement in porous media,
- fingering phenomenon,
- blood flow,
- flow in lungs to treat asthma and cancer,
- flow in filtration processes,
- traffic management,
- atmospheric turbulence,
- washing machines,
- fixed/sweeping air-conditioning in a room, and
- falling water column.

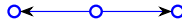


Figure 2.1.: Lattice cell for the D1Q3 LBM model.

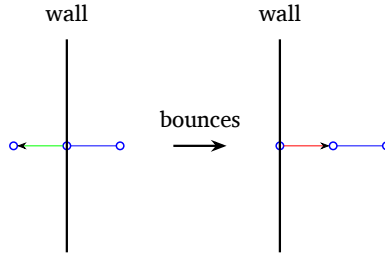


Figure 2.2.: Bounce-back BCs.

## 2.5. SIMPLE EXAMPLE

A toy problem task was taken from [16] (example 5.5.2).

**The problem:** a slab is initially at temperature equal to zero,  $T = 0.0$ . For time  $t \geq 0$ , the left surface of the slab is subjected to a high temperature and equal to unity,  $T = 1.0$ . The slab length is 100 units. Calculate the temperature distribution in the slab for  $t = 200$ . Use thermal diffusion coefficient  $\alpha = 0.25$ .

Two different simulations were implemented for solving the heat diffusion equation in C++. The first one is based on a simple one-dimensional D1Q3 LBM model (a schematic visualization of a lattice cell in the case of the D1Q3 model can be seen on Figure 2.1). The second one uses a second-order finite difference method (FDM). Figure 2.4 compares the predicted results of the LBM and FDM. As one can see, the results fully coincide.

Note that for the stability condition the timestep for the FDM is 0.5, while it is 1.0 for the LBM. Therefore the FDM has to do 400 timesteps. Hence, LBM is much faster and efficient than the FDM. The algorithmic steps for solving the problem numerically with the help of LBM are shown on Figure 2.3. We are applying *push* updating scheme, and no-slip BCs (see Figure 2.2).

## 2.6. ALGORITHMIC STEPS OF WALBERLA'S LBM

As it was previously mentioned, WALBERLA is a massively parallel application, so its implementation of LBM uses copy-compute overlap. The computational domain is split into blocks of cells, and each block is processed by a different process. Additionally, for the sake of the copy-compute overlap, the domain is divided into inner and outer parts. In each timestep communication and inner domain computation are happening concurrently. After these steps are done, the outer part of the domain is computed. As shown on Figure 2.5, the communication step consists of data preparation (by filling up sending buffers), data sending and receiving, and data extraction (from the receiving buffers). During the computational parts the BCs are applied, and the LBM takes place. The order of collision and streaming depends on the configuration that was chosen.

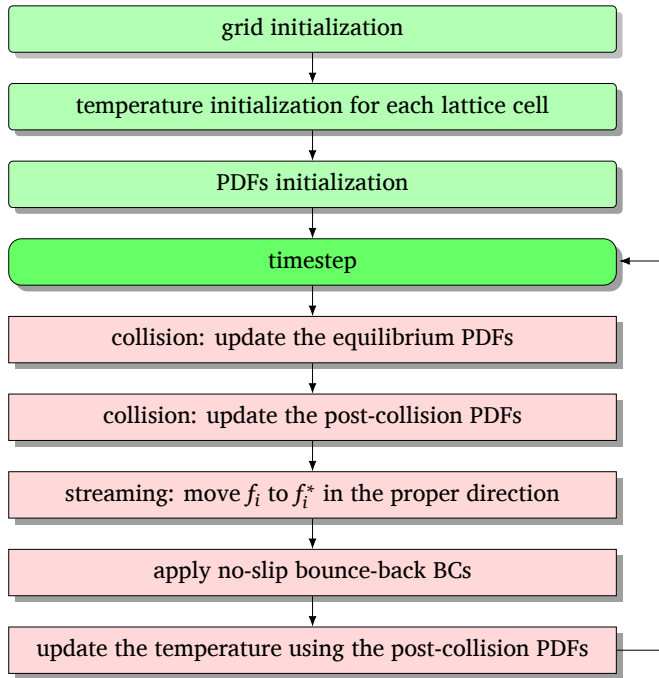


Figure 2.3.: Algorithmic steps of LBM.

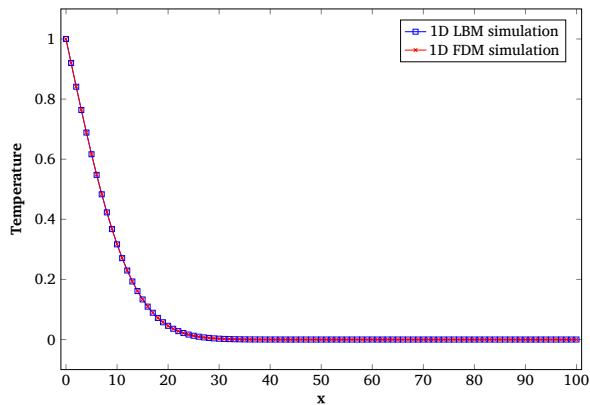


Figure 2.4.: A comparison of the results predicted by the LBM and FDM for a slab subjected to a constant temperature at the boundary.



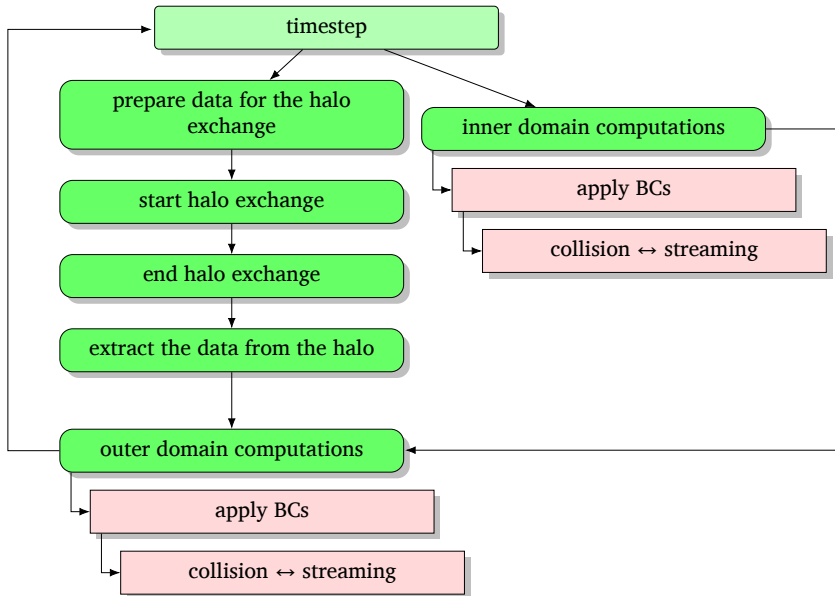


Figure 2.5.: Algorithmic steps of WALBERLA’s LBM.

### 3. CUDA TASK GRAPHS

CUDA task graphs [23, 19, 11, 6, 8, 9] allow a mechanism for designing the program’s workflow consisting of multiple GPU operations as a graph of tasks, and launching them as a single GPU operation. Any CUDA stream workflow can be moved to CUDA task graph(s). This approach reduces the overhead for separately launched GPU operations, because we pay it once, but then we can reuse the graph many times, and hence amortize the launch cost. CUDA task graph is especially applicable in the case of short kernels whose execution time is dominated by the time for their launch as it might reduce the gaps between the kernel launches. There are two ways to create a CUDA graph – explicit/manual (by using the CUDA Graph API), and implicit (by capturing streams’ workflow). The manual approach gives more control than the implicit one. After the creation is done, the graph is instantiated, and launched. Since the instantiating is the overhead when working with task graphs, the aim is to minimize the number of instantiations. If in the course of work, we need to change something in the task graph, we can follow one of two paths. If the graph’s topology needs to be updated, the already created graph needs a re-instantiation. Otherwise, if only the kernel nodes’ parameters should be changed, we can update them without re-instantiating. Our application’s workflow does not change with time, therefore we do not need to re-instantiate, but only update the graph’s parameters.

Figure 3.1 shows a subset of all the possible nodes we can add to a task graph, namely we can call a kernel or a CPU function within the graph. Malloc and memset operations can be done inside the task graph. Finally, a node can be yet another graph – child

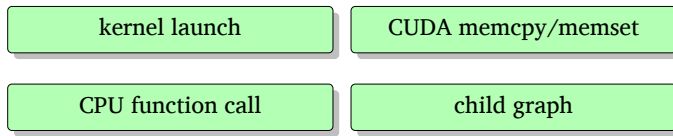


Figure 3.1.: Some of the graph operations.

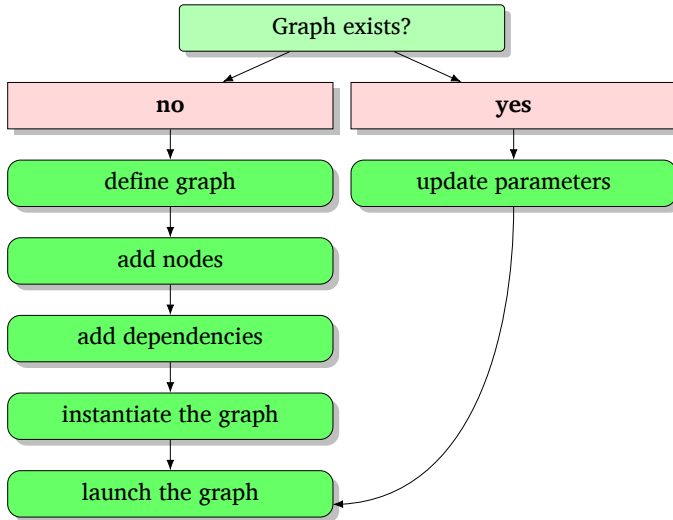


Figure 3.2.: Explicit construction of graph, using CUDA Graph API calls.

graph.

The workflow in the case of explicit creation of the task graph is described on Figure 3.2. As it can be seen, if the graph does not exist, we need to define it, add nodes and describe the dependencies between the nodes. Then, the graph is instantiated (that is, converted into an executable graph), and launched (executed). If the executable graph is existing, we can update the parameters (if needed), and launch it.

Figure 3.3 depicts how an example stream workflow running on three CUDA streams can be moved to a task graph. On both the pictures the upper case letters denote kernel calls. For example, to the left, we can see that kernel D depends on kernel B, but as they are run on two different streams, a waiting point is needed to ensure that kernel D is not going to start before kernel B has completed. Such waiting points are of no need in task graphs since an edge between two nodes defines a dependency, and hence, kernel D will wait for kernel B to finish its work.

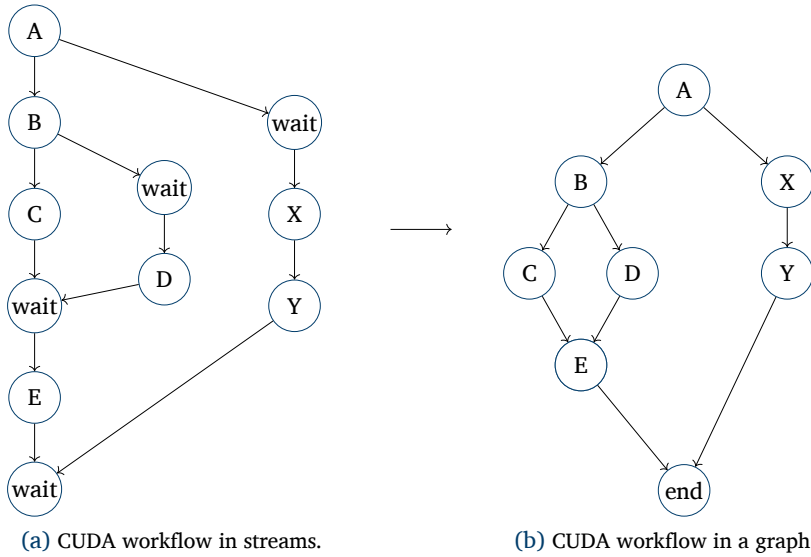


Figure 3.3.: Porting stream workflow to CUDA task graph.

## 4. PORTING WALBERLA TO CUDA TASK GRAPHS

### 4.1. CODE SNIPPETS

What follows is the procedure for moving a single kernel to a task graph. Similar steps (sometimes with more complicated logic, e. g. because BCs may or may not be applied and therefore dependencies must or should not be added to the compute kernels' nodes) were followed for all other kernels ( $62 * 2 = 124$  overall). I am showing here the simplest case.

The initial code is given in Listing 2. As one can see, this is a single kernel call with particular launch configuration and kernel arguments.

#### LISTING 2: UNIFORMGRIDGPU\_PACKINFOODD/EVEN.CU

```
1 internal_pack_BSW::pack_BSW <<< _grid, _block, 0, stream >>> (_data_buffer, _data_pdfs,\
2   _size_pdfs_0, _size_pdfs_1, _size_pdfs_2,\
3   _stride_pdfs_0, _stride_pdfs_1, _stride_pdfs_2, _stride_pdfs_3);
```

As my aim was to prove a concept, and therefore did not want to restructure the WALBERLA's code, a header file **graph\_params.h** (part of which is shown in Listing 3) was introduced for keeping the definitions of all the new variables I needed. All of them were defined as global.

#### LISTING 3: GRAPH\_PARAMS.H

```
1 inline cudaGraph_t graph_pack;
2 inline bool graph_created_pack = false;
3 inline cudaGraphExec_t graph_exec_pack;
4 inline cudaGraphNode_t emptyNode_pack;
5 inline std::vector< cudaGraphNode_t > nodeDependencies_pack(1);
6 inline cudaGraphNode_t kernelNode_pack[26];
```

Listing 4 shows what augmentations should be made to the **.cu** file in order to add the kernel to a CUDA task graph. One needs to define the `cudaKernelNodeParams`

structure, and fill it in with a pointer to the kernel, pointers to the kernel’s arguments, and the launch configuration (grid and block sizes, and amount of dynamic shared memory) that will be used. After that, if the executable graph has not been created yet, a kernel node with proper dependencies is added to the task graph. Otherwise, the kernel node’s parameters are only updated.

#### LISTING 4: AUGMENTED .CU FILE

```

1  cudaKernelNodeParams kernelNodeParams = {0};
2
3  void *kernel_args[9] = {(void *)&_data_buffer, (void *)&_data_pdfs, (void *)&_size_pdfs_0,\
4      (void *)&_size_pdfs_1, (void *)&_size_pdfs_2, (void *)&_stride_pdfs_0,\
5      (void *)&_stride_pdfs_1, (void *)&_stride_pdfs_2, (void *)&_stride_pdfs_3};
6
7  kernelNodeParams.func = (void *)internal_pack_BSW::pack_BSW;
8  kernelNodeParams.gridDim = _grid;
9  kernelNodeParams.blockDim = _block;
10 kernelNodeParams.sharedMemBytes = 0;
11 kernelNodeParams.kernelParams = (void **)kernel_args;
12 kernelNodeParams.extra = NULL;
13
14 if(!graph_created_pack){
15     WALBERLA_CUDA_CHECK(cudaGraphAddKernelNode(&kernelNode_pack[0],\
16         graph_pack, nodeDependencies_pack.data(), nodeDependencies_pack.size(),\
17         &kernelNodeParams));
18 }
19 else{
20     WALBERLA_CUDA_CHECK(cudaGraphExecKernelNodeSetParams(graph_exec_pack,\
21         kernelNode_pack[0], &kernelNodeParams));
22 }

```

Finally, in the main C++ function we need to create a stream on which the task graph is going to be run, and to create the task graph itself. Then, after the function (colored in pink) which eventually calls the kernel of interest completes, the task graph is instantiated (if not instantiated yet), and launched. To ensure usage of correct data, the stream needs to be synchronized after the launch. The green part of Listing 5 is used only in a debug mode (because it adds overhead), that is, we can use the CUDA Graph API to print the number of nodes we have created, as well as the task graph itself, and thus to check if we have applied all the dependencies correctly. Finally, the used resources need to be freed. Note that if the stream is not destroyed, it leads to a segmentation fault.

#### LISTING 5: AUGMENTED MAIN FUNCTION

```

1  cudaStream_t graph_stream;
2  WALBERLA_CUDA_CHECK(cudaStreamCreate(&graph_stream));
3  WALBERLA_CUDA_CHECK(cudaGraphCreate(&graph_pack, 0));
4  WALBERLA_CUDA_CHECK(cudaGraphAddEmptyNode(&emptyNode_pack, graph_pack, NULL, 0));
5  nodeDependencies_pack[0] = emptyNode_pack;
6  comm.startCommunication(defaultStream);
7  if(!graph_created_pack){
8      WALBERLA_CUDA_CHECK(cudaGraphAddEmptyNode(&emptyNode_pack, graph_pack,\
9          nodeDependencies_from_pack.data(), nodeDependencies_from_pack.size()));
10     WALBERLA_CUDA_CHECK(cudaGraphInstantiate(&graph_exec_pack, graph_pack, NULL, NULL, 0));
11     graph_created_pack = true;
12 }
13 WALBERLA_CUDA_CHECK(cudaGraphLaunch(graph_exec_pack, graph_stream));
14 WALBERLA_CUDA_CHECK(cudaStreamSynchronize(graph_stream));
15 cudaGraphNode_t *nodes = NULL;
16 size_t numNodes = 0;
17 WALBERLA_CUDA_CHECK(cudaGraphGetNodes(graph_pack, nodes, &numNodes));
18 printf("\nNum of nodes created manually = %zu\n", numNodes);
19 WALBERLA_CUDA_CHECK(cudaGraphDebugDotPrint(graph_pack, "dot_file_pack.dot", {0}));
20 ...

```

```
21 WALBERLA_CUDA_CHECK(cudaStreamDestroy(graph_stream));
22 WALBERLA_CUDA_CHECK(cudaGraphDestroy(graph_pack));
23 WALBERLA_CUDA_CHECK(cudaGraphExecDestroy(graph_exec_pack));
```

## 4.2. WORKFLOW FOR THE KERNELONLY TIMESTEP STRATEGY

As a first step the kernelOnly timestep strategy was moved to a task graph with a single LBM kernel node. There was no hope that such an attempt would lead to a performance boost, it was made for the sake of proof of concept only.

## 4.3. WORKFLOW FOR THE SIMPLEOVERLAPTIMESTEP STRATEGY

As a next step the simpleOverlapTimeStep strategy was ported to four CUDA task graphs as depicted on Figure 4.1, as follows:

1. communication step: packing (26 pack kernel nodes);
2. [inner BCs and] inner domain computation ([2 BCs kernel nodes] + 1 LBM kernel node);
3. communication step: unpacking (26 unpack kernel nodes);
4. [outer BCs and] outer domain computation ([2 BCs kernel nodes] + 6 LBM kernel nodes (one for each face of the cube)).

N.B. The square brackets above denote that these nodes may or may not be part of the graph depending on the problem's configuration. The procedure above was done twice, because WALBERLA differentiates between odd and even timesteps by introducing different kernel calls.

Note that on Figure 4.1 it becomes clear that we have external dependencies on MPI in our workflow. This fact enforces calling `cudaStreamSynchronize` after the pack graph is launched. So as to be sure that the correct data (that is, to prevent race conditions) is going to be manipulated in the CPU function which eventually calls the outer kernels, yet another synchronization point is needed after the unpack graph. Finally, at the end of each timestep, the stream is synchronized so as the next iteration to use the updated data. Figure 4.2 shows the GPU operations needed in the case of these four graphs being executed on two concurrent streams (the MPI part is hidden). As the inner computation does not depend on the rest of the graphs, it can or not be executed on a separate stream.

## 4.4. CHANGED WALBERLA'S FILES

In order to achieve the porting from stream workflow to CUDA task graphs, we needed to augment all the pink files shown on Figure 4.3a. These were originally auto-generated target-specific LBM kernel code files. For our purposes we stopped re-generating them, and started building on them. This is the reason why the CMake files needed some changes.

The pink-colored files on Figure 4.3b show which source files needed changes. These are the files that were building the stream-based execution of WALBERLA. Originally, a stream manager was constructing concurrent streams for each kernel call, and so was making CUDA API calls so as to ensure synchronization. Since CUDA task graph creates the needed streams on its own, all this became redundant.

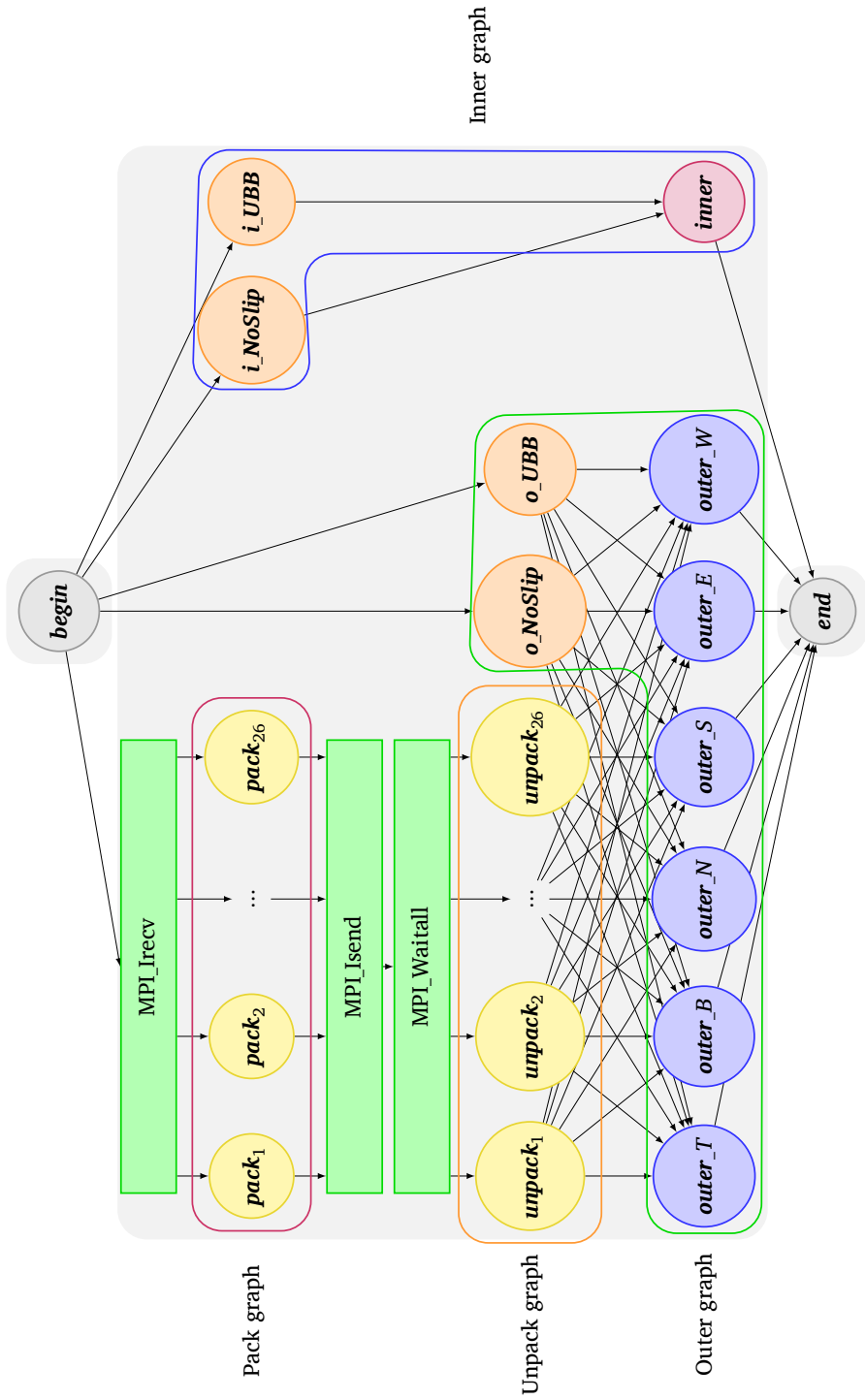


Figure 4.1.: Dependencies and task graphs splitting.

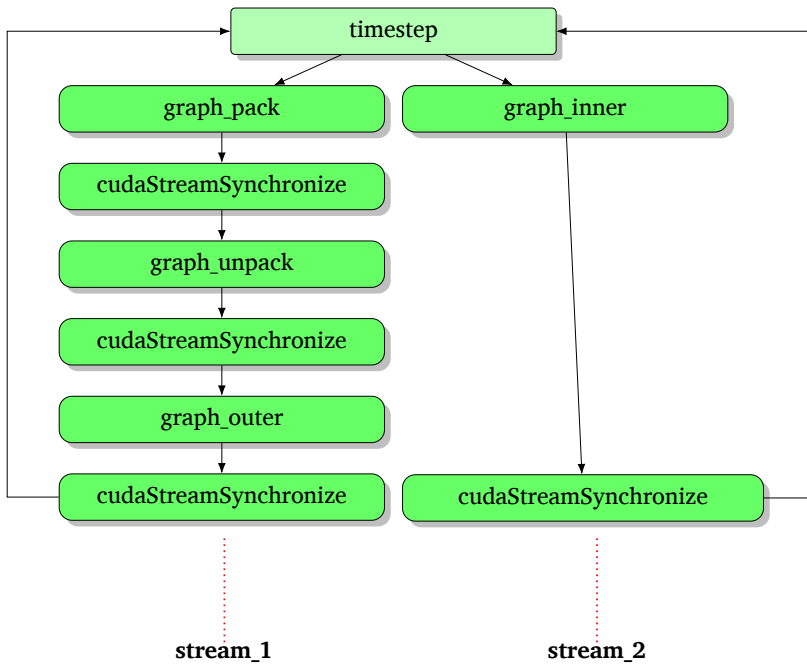
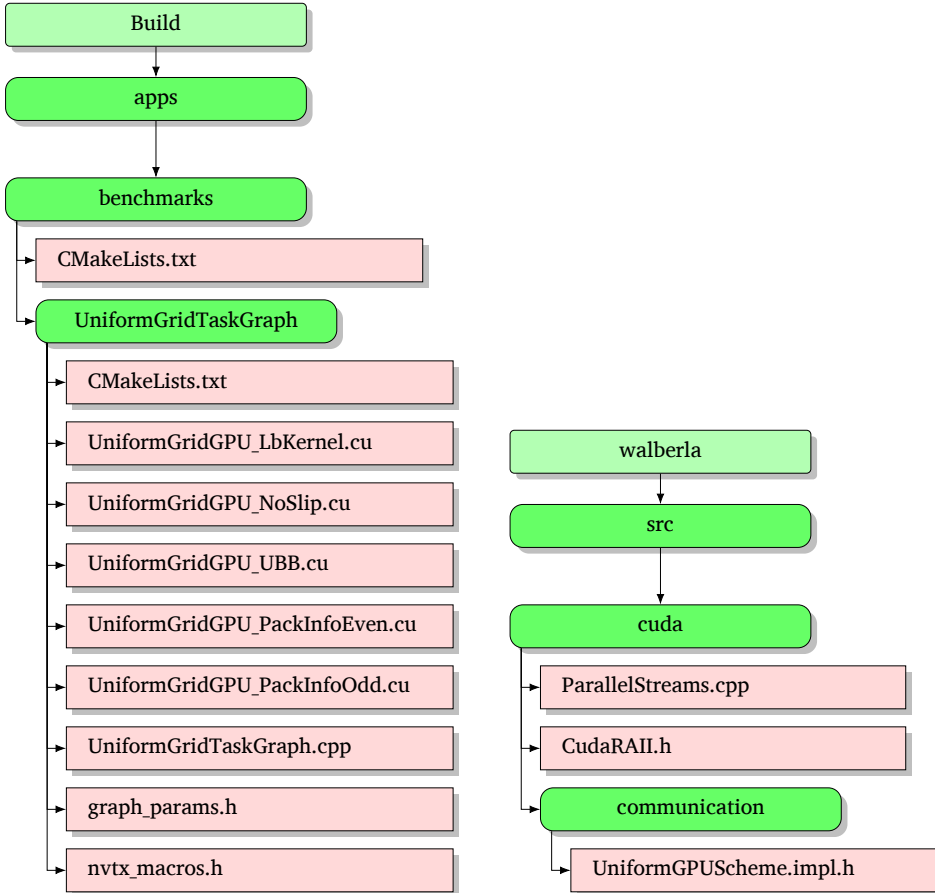


Figure 4.2.: Timelooop of 4 graphs on 2 streams.



(a) Changed WALBERLA's auto-generated code files.

(b) Changed WALBERLA's src files.

Figure 4.3.: Changed WALBERLA's files.



## 5. RESULTS

### 5.1. EXPERIMENTAL SETUP

Computation were held on the basis of the [supercomputer JUWELS Booster](https://www.top500.org) (ranked 8th in the TOP500 list (<https://www.top500.org>) as of June 2021) [4] on a single GPU NVIDIA Tesla A100 (see Figure 5.1).

Four different scenarios with different number of timesteps were tested (for more details see Figure 5.2).

### 5.2. COMPUTATIONAL RESULTS

Figure 5.3 shows the results (in mega lattice updates per second (MLUPS)) from the runs of the different versions of the application, as follows: the original code; the code ported to four CUDA task graphs with the pack and unpack graphs being wide, depth 1; the third and fourth columns correspond to the cases when the communication task graphs start/end with an empty node; the fifth column explores both starting and ending empty nodes, while the last two columns investigate the same kind of pack/unpack graphs as in column 5, but running the inner graph on a separate stream or separate low-priority CUDA stream.

As one can see, CUDA task graphs give better performance results than the original code. Additionally, the starting empty node gives some more performance benefits (because it helps the scheduler to handle the graphs better). Since the results in columns 3 and 5 have negligible differences (due to the cluster noise), we chose to continue with the version of the graphs from column 5. Two variant of parallel streams were tested. None of them gave better performance since the inner graph is computationally greedy, and the hardware utilization cannot be improved further than that.

### 5.3. FAN-OUT OF KERNELS

Screenshots from Nsight Systems [18, 20] reports are shown on Figure 5.4. Comparing Figure 5.4a which depicts part of the timeline of the original code with Figure 5.4b which shows the fan-out of the ported to the CUDA task graphs code, one can see that with CUDA task graphs the fan-out of the communication kernels is increased (25 vs. 8). That is, we have more kernels running in parallel. This means that the application's parallelization was indeed improved by the CUDA task graphs.

### 5.4. CUDA API CALLS

As it was discussed earlier, the stream manager which is part of the WALBERLA's initial code introduces a lot of CUDA API calls as it can be observed on Figure 5.5. There, the screenshot of the Nsight Systems report shows in green all the CUDA API calls for a single timestep. The following list gives some of the CUDA API calls that became redundant, and therefore were removed:

- `cudaEventCreate`
- `cudaEventRecord`
- `cudaStreamCreate`
- `cudaStreamCreateWithPriority`

NVIDIA GPU	Tesla A100
Cores	6912
SMs	108
Memory	40 GB HBM2
FP32 [TFLOPS]	19.5
FP64 [TFLOPS]	9.7
Capability	8.0
FREQ [MHz]	1410
Mem bandwidth [GB/s]	1555

Figure 5.1.: NVIDIA Tesla A100 specifications.

No	cellsPerBlock	gpuBlockSize	timesteps
1	( 64, 64, 64)	(32, 1, 1)	1600
2	(128, 128, 128)	(32, 1, 1)	200
3	(256, 256, 256)	(32, 1, 1)	25
4	(320, 320, 320)	(32, 1, 1)	12

Figure 5.2.: Tested scenarios.





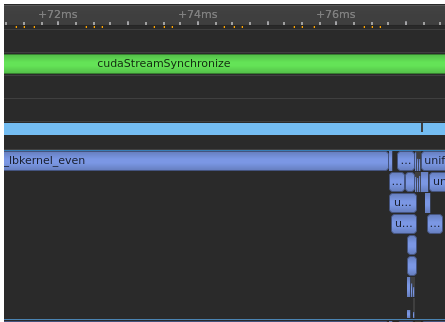
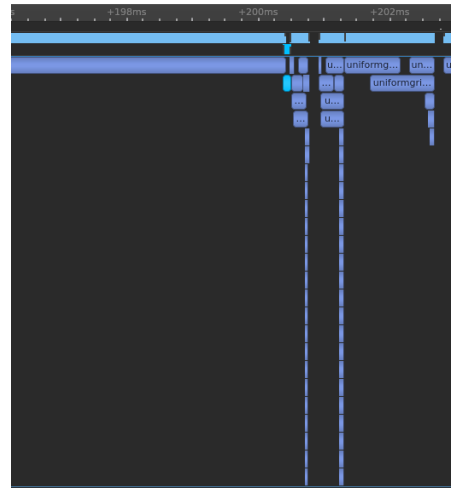
No	no graphs	4 graphs 	4 graphs 	4 graphs 	4 graphs 	4 graphs, 2 streams	4 graphs, 2 priority streams
	MLUPS						
1	255.45	561.78	572.79	556.76	569.71	567.26	568.61
2	1314.27	1554.90	1622.94	1553.71	1628.66	1629.04	1621.82
3	2173.65	2287.76	2309.56	2275.3	2305.53	2303.36	2306.93
4	2315.17	2442.24	2453.90	2441.54	2453.30	2454.94	2454.89

Figure 5.3.: Results from different runs.



(a) Communication kernels fan-out – before.



(b) Communication kernels fan-out – after.

Figure 5.4.: Fan-out of the communication kernels – before and after task graphs.

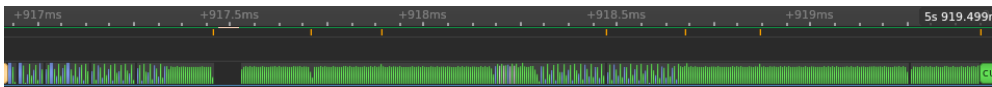


Figure 5.5.: Removed events – before.

- `cudaStreamWaitEvent`
- `cudaStreamSynchronize`
- `cudaEventDestroy`

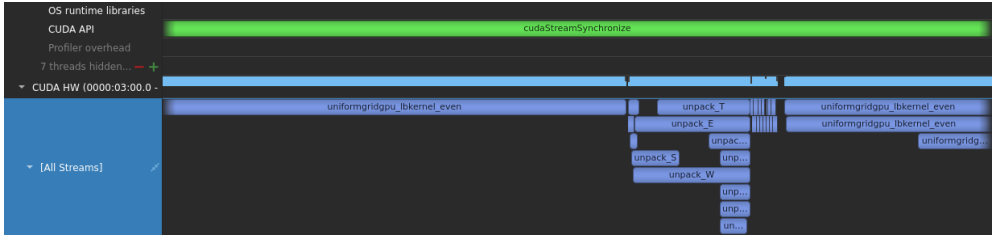
This increased the performance boost.

## 5.5. NVTX FOR ANNOTATED REPORTS

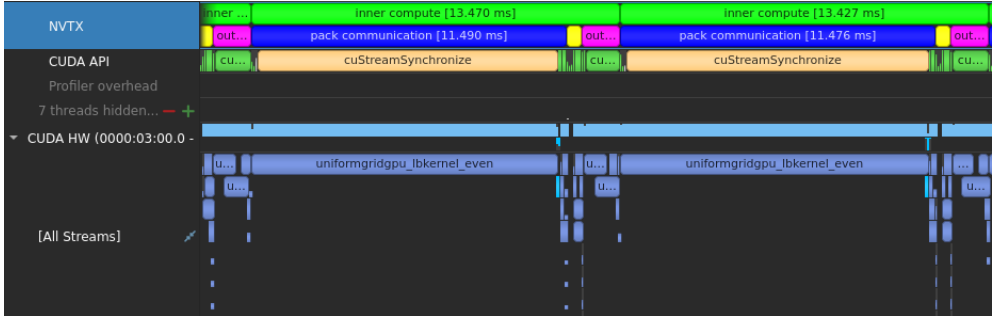
On Figure 5.6a one can see a section of the timeline of the Nsight Systems profiler, applied to the original WALBERLA’s code, including some kernel launches. More precisely, both to the left and to the right there are `uniformgridgpu_lbkernel_even` calls, but for a reader who is not aware with the implementation details of WALBERLA, it will not be clear what these calls denote, namely the left one corresponds to the inner kernel, and to the right we see three of the outer computational kernels. Therefore, NVIDIA Extension Tools (NVTX) library [22] was used so as to make the Nsight Systems profiles more readable. The results can be seen on Figure 5.6b where it is now obvious what corresponds to what.

## 5.6. IN THE CASE OF A SINGLE GPU

In the case when we run WALBERLA on a single GPU card, the communication effectively disappears (converting to send-to-self instead). Therefore, the pack and unpack task



(a) Nsight Systems report – before.



(b) Nsight Systems report – after.

Figure 5.6.: Nsight Systems report – before and after NVTX annotation.

graphs can be merged into a single communication task graph (see Figure 5.7). Additionally, if we have only a single block of cells, the for loops which are iterated over the blocks disappear. Hence, the computational task graphs that are otherwise launched on every loop iteration can be added to the communication task graph, and thus merged into a single task graph (see Figure 5.8).

It is impossible to use less than 4 graphs on more than one GPU, as CUDA-aware MPI [14] is not stream and CUDA task graph aware. NVSHMEM [21] might be a remedy, because it is CUDA stream-aware.

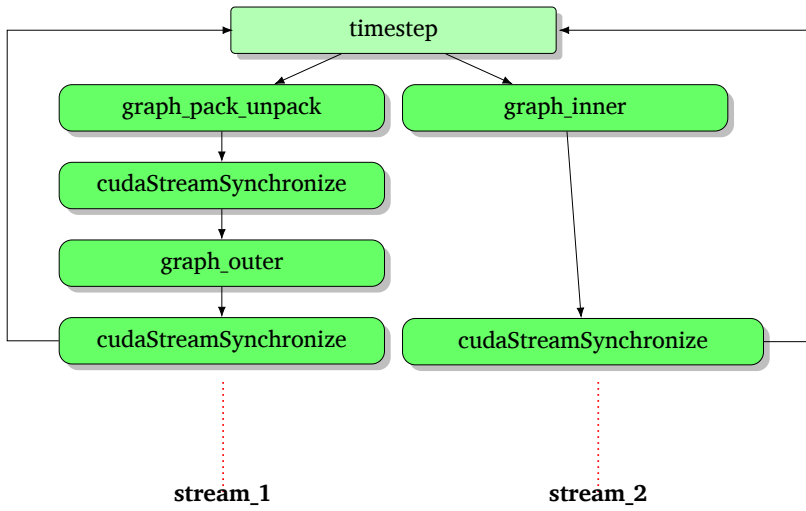


Figure 5.7.: Timeloop of 3 graphs on 2 streams.

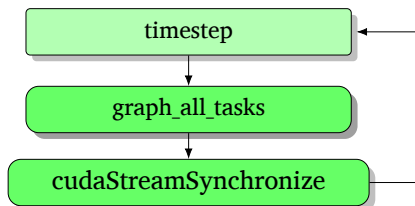


Figure 5.8.: Timeloop of 1 graph.

## 6. JUBE

JUBE (Juelich Benchmark Environment) [13, 12, 10, 15] is a script-based framework which facilitates the creation of benchmarks sets, their running on different computer systems, and evaluation of the results. Beside automated benchmarking JUBE can be used also for building, testing, profiling of software, parameters sweeps and production automatization.

A JUBE benchmark script for the application of interest was created. It incorporates the following steps: compilation, execution, results analysis, and summary. Listing 6 shows the result of running the benchmark, while Listing 7 gives the generated result table after the application results have been analyzed.

**LISTING 6: RUNNING JUBE.**

```
1 $ jube run UniformGridGPU.xml
2 #####
3 # benchmark: UniformGridGPU
4 # id: 59
5 #
6 #
7 #####
8
9 Running workpackages (#=done, 0=wait, E=error):
10 ##### ( 3/ 3)
11
12 | stepname | all | open | wait | error | done |
13 |-----|-----|-----|-----|-----|-----|
14 | compile | 1 | 0 | 0 | 0 | 1 |
15 | build_copy | 1 | 0 | 0 | 0 | 1 |
16 | execute | 1 | 0 | 0 | 0 | 1 |
17
18 >>> Benchmark information and further useful commands:
19 >>> id: 59
20 >>> handle: UniformGridGPU_jube_benchmark
21 >>> dir: UniformGridGPU_jube_benchmark/000059
22 >>> analyse: jube analyse UniformGridGPU_jube_benchmark --id 59
23 >>> result: jube result UniformGridGPU_jube_benchmark --id 59
24 >>> info: jube info UniformGridGPU_jube_benchmark --id 59
25 >>> log: jube log UniformGridGPU_jube_benchmark --id 59
26 #####
```

**LISTING 7: JUBE RESULTS.**

```
1 $ jube result UniformGridGPU_jube_benchmark --id 59
2 #####
3 result_table:
4 | number_of_GPUs | mlups_per_GPU_avg |
5 |-----|-----|
6 | 1 | 567.72 |
7 #####
```

## 7. FUTURE WORK RECOMMENDATIONS

The following recommendations are made for the future work:

1. In case CUDA-aware MPI implementation is not available or when offloading to multiple GPUs, memcpy calls H2D and D2H are needed during the communication step. It is recommended that they are added to the task graph using memcpy nodes (see Listing 8). It might be beneficial to set single dependency between the pack/unpack and memcpy nodes, that is, the memcpy that is needed after kernel  $\text{pack}_1$  is executed depends only on it, not on all the 26 pack kernels. Likewise for the unpack kernels and the memcpy that precedes them (see Figure 7.1).

2. Instead of launching the computational task graphs on each for loop iteration, it is worth trying cloning the 3-node graphs instead.
3. Important note is that synchronization points (MPI and CUDA) are not task-graph-friendly. It is recommended that they are avoided.
4. NVIDIA collective communication library (NCCL) [17] might be a good replacement of MPI for the communication.

#### LISTING 8: MEMCPY NODE

```

1  uint_t sizeBytes = sizeof(double) * numCells * elementsPerCell;
2  uint_t size = numCells * elementsPerCell;
3
4  cudaGraphNode_t memcpyNode;
5  cudaMemcpy3DParms memcpyParams = {0};
6
7  memcpyParams.srcArray = NULL;
8  memcpyParams.srcPos = make_cudaPos(0, 0, 0);
9  memcpyParams.srcPtr = make_cudaPitchedPtr((void *)cpuDataPtr, sizeBytes, size, 1);
10
11 memcpyParams.dstArray = NULL;
12 memcpyParams.dstPos = make_cudaPos(0, 0, 0);
13 memcpyParams.dstPtr = make_cudaPitchedPtr(gpuDataPtr, sizeBytes, size, 1);
14
15 memcpyParams.extent = make_cudaExtent(sizeBytes, 1, 1);
16 memcpyParams.kind = cudaMemcpyHostToDevice;
17
18 WALBERLA_CUDA_CHECK(cudaGraphAddMemcpyNode(&memcpyNode, graph, NULL, 0, &memcpyParams));

```

In the Listing 8 the `cudaMemcpy3DParms` structure is filled in with the CUDA pitched memory pointers (with the respective sizes and offsets) for both the source and the destination arrays, extent (that is, number of elements to be transferred), and type of memory transfer to be conducted. In this example code no dependencies are passed in. Finally, the `memcpy` operation can also be done by using the `cudaGraphAddMemcpyNode1D` in the case of contiguous data.

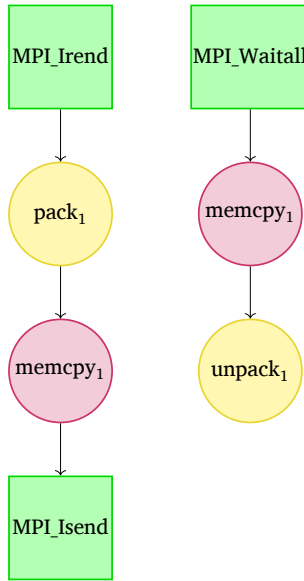


Figure 7.1.: Memcpy nodes and its dependencies.

## 8. CONCLUSION

### 8.1. DIFFICULTIES

The main difficulties while working on the project were related to the huge and complex code of WALBERLA. Additionally, many programming and data dependencies should be taken into account in the working process. The code has many levels of abstraction, e. g. finding a particular functions' call chain might be a time-consuming and difficult process. It is important to realize that neither can a developer fix everything at once, nor is it always possible to foresee what might go wrong when introducing new features. Tackling such an amount of code was very challenging, e. g. one single change in the code led to a bug we could not have recovered from for a week. On the other hand, it is important to know that synchronizing CUDA streams and CUDA-aware MPI is an art, and should be done with a great care so as to prevent from race conditions. Finally, the structure of the code did not allow for stream capture approach to be used for the graphs' creation, because of the many synchronization points that are used.

### 8.2. SUMMARY

WALBERLA's workflow for 2 different timestep strategies was moved to CUDA task graphs. Different approaches for constructing the communication graphs were tested. A more effective fan-out of kernel execution was achieved for the pack and unpack kernels. CUDA task graphs allowed the code to be restructured and simplified by removing the manually created forking streams for each kernel, as well as the additionally introduced CUDA API calls by the stream manager. All this led to performance improvement. NVTX was added for more comprehensive Nsight Systems reports. The bottlenecks for merging the 4 graphs into one were investigated and listed. Some remedies were suggested.




















All figures were generated with the help of the Tikz [24] package.

## 9. ACKNOWLEDGMENTS

The author would like to express their deep gratitude to (ordered alphabetically): Andreas Herten (JSC), Ivo Kabadshow (JSC), Jayesh Badwaik (JSC), Jiri Kraus (NVIDIA), Markus Holzer (Cerfacs), Markus Hrywniak (NVIDIA), Thorsten Hater (JSC) for their invaluable help.

## REFERENCES

- [1] A. Alzaabi, C. Hoydic, and S. Joon. **Pennsylvania State University. Mathematical Modeling in Energy and Mineral Engineering EGEE 520 lecture: Mathematical Modeling. Lattice-Boltzmann Method.** Lecture, 2019.
- [2] P. ASINARI. **Chapter 5: Lattice Boltzmann Method.** In Multi-Scale Analysis of Heat and Mass Transfer in Mini/Micro-Structures. Doctoral thesis, 2005.
- [3] M. Bauer, S. Eibl, C. Godenschwager, N. Kohl, M. Kuron, C. Rettinger, F. Schornbaum, C. Schwarzmeier, D. Thönnies, H. Köstler, and U. Rude. **Walberla: A block-structured high-performance framework for multiphysics simulations.** Computers & Mathematics with Applications, 2020.  [doi:10.1016/j.camwa.2020.01.007](https://doi.org/10.1016/j.camwa.2020.01.007).
- [4] J. S. Centre. **JUWELS: Modular Tier-0/1 Supercomputer at the Jülich Supercomputing.** Journal of large-scale research facilities, 5(A171), 2019.  [doi:10.17815/jlsrf-5-171](https://doi.org/10.17815/jlsrf-5-171).
- [5] K. Doyle, L. Li, C. Wang, and C. Elsworth. **Pennsylvania State University. Mathematical Modeling in Energy and Mineral Engineering EME 521 lecture: Lattice Boltzmann Method.** Lecture, 2014.
- [6] R. V. der Wijngaart. **GTC Digital Talk: Effortless CUDA Graphs.** Talk, April 2021.
- [7] A. B. Elton. Doctoral thesis, 1990.  [doi:10.2172/6480937](https://doi.org/10.2172/6480937).
- [8] A. Gray. **NVIDIA Developer Blog. Getting Started with CUDA Graphs,** 2019. (Accessed: 8 October 2021).  <https://developer.nvidia.com/blog/cuda-graphs/>.
- [9] G. Gutmann. **CUDA Graph Usage: CUDA Feature Testing,** 2020. (Accessed: 8 October 2021).  <https://codingbyexample.com/2020/09/25/cuda-graph-usage/>.
- [10] M.-A. Hermanns. **North Rhine-Westphalia competence network for high performance computing (HPC\_NRW) talk: JUBE by Example.** Talk, August 2021.
- [11] S. Jones, S. Gurfinkel, A. Gray, J. Larkin, and E. Weinberg. **TC2020 Talk: Connect with the Experts. CUDA Graphs (CWE21914).** Talk, April 2020.

- [12] **JUBE's documentation.** (Accessed: 8 October 2021).  <https://apps.fz-juelich.de/jsc/jube/jube2/docu/>.
- [13] **JUBE's website.** (Accessed: 8 October 2021).  [https://www.fz-juelich.de/ias/jsc/EN/Expertise/Support/Software/JUBE/\\_node.html](https://www.fz-juelich.de/ias/jsc/EN/Expertise/Support/Software/JUBE/_node.html).
- [14] J. Kraus. **NVIDIA Developer Blog. An Introduction to CUDA-Aware MPI,** 2013. (Accessed: 8 October 2021).  <https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/>.
- [15] S. Lührs. **PC Knowledge Meeting'20 (HPCKP'20) talk: Automated Benchmarking with JUBE.** Talk, June 2020.
- [16] A. A. Mohamad. Lattice Boltzmann Method. Fundamentals and Engineering Applications with Computer Codes. Springer-Verlag London, 2nd edition, 2019.  [doi:10.1007/978-1-4471-7423-3](https://doi.org/10.1007/978-1-4471-7423-3).
- [17] **NVIDIA NCCL Library.** (Accessed: 8 October 2021).  <https://developer.nvidia.com/nccl>.
- [18] **NVIDIA Nsight Systems.** (Accessed: 8 October 2021).  <https://developer.nvidia.com/nsight-systems>.
- [19] NVIDIA. **CUDA C++ Programming Guide. Design Guide,** 2021.
- [20] NVIDIA. **Nsight Systems USER GUIDE. User Manual,** 2021.
- [21] **NVIDIA NVSHMEM Library.** (Accessed: 8 October 2021).  <https://developer.nvidia.com/nvshmem>.
- [22] **NVIDIA NVTX Library.** (Accessed: 8 October 2021).  [https://docs.nvidia.com/gameworks/content/gameworkslibrary/nvtx/nvidia\\_tools\\_extension\\_library\\_nvtx.htm](https://docs.nvidia.com/gameworks/content/gameworkslibrary/nvtx/nvidia_tools_extension_library_nvtx.htm).
- [23] P. Ramarao. **NVIDIA Developer Blog. CUDA 10 Features Revealed: Turing, CUDA Graphs, and More,** 2018. (Accessed: 8 October 2021).  <https://developer.nvidia.com/blog/cuda-10-features-revealed/>.
- [24] T. Tantau. The TikZ and PGF Packages. Manual for version 3.1.9. Documentation, 2021.
- [25] **Tutorial – LBM 6: Boundary Conditions.** (Accessed: 8 October 2021).  [https://www.walberla.net/doxygen/tutorial\\_lbm06.html](https://www.walberla.net/doxygen/tutorial_lbm06.html).
- [26] **walBerla's official repo.** (Accessed: 8 October 2021).  <https://i10git.cs.fau.de/walberla/walberla>.
- [27] **walBerla's website.** (Accessed: 8 October 2021).  <https://walberla.net>.

Band / Volume 40

**Extreme Data Workshop 2018**

Forschungszentrum Jülich, 18-19 September 2018  
Proceedings

M. Schultz, D. Pleiter, P. Bauer (Eds.) (2019), 64 pp

ISBN: 978-3-95806-392-1

URN: urn:nbn:de:0001-2019032102

Band / Volume 41

**A lattice QCD study of nucleon structure with physical quark masses**

N. Hasan (2020), xiii, 157 pp

ISBN: 978-3-95806-456-0

URN: urn:nbn:de:0001-2020012307

Band / Volume 42

**Mikroskopische Fundamentaldiagramme der Fußgängerdynamik –  
Empirische Untersuchung von Experimenten eindimensionaler Bewegung  
sowie quantitative Beschreibung von Stau-Charakteristika**

V. Ziemer (2020), XVIII, 155 pp

ISBN: 978-3-95806-470-6

URN: urn:nbn:de:0001-2020051000

Band / Volume 43

**Algorithms for massively parallel generic *hp*-adaptive finite element methods**

M. Fehling (2020), vii, 78 pp

ISBN: 978-3-95806-486-7

URN: urn:nbn:de:0001-2020071402

Band / Volume 44

**The method of fundamental solutions for computing interior transmission  
eigenvalues**

L. Pieronek (2020), 115 pp

ISBN: 978-3-95806-504-8

Band / Volume 45

**Supercomputer simulations of transmon quantum computers**

D. Willsch (2020), IX, 237 pp

ISBN: 978-3-95806-505-5

Band / Volume 46

**The Influence of Individual Characteristics on Crowd Dynamics**

P. Geörg (2021), xiv, 212 pp

ISBN: 978-3-95806-561-1

Band / Volume 47

**Structural plasticity as a connectivity generation and optimization algorithm in neural networks**

S. Diaz Pier (2021), 167 pp  
ISBN: 978-3-95806-577-2

Band / Volume 48

**Porting applications to a Modular Supercomputer**

Experiences from the DEEP-EST project

A. Kreuzer, E. Suarez, N. Eicker, Th. Lippert (Eds.) (2021), 209 pp  
ISBN: 978-3-95806-590-1

Band / Volume 49

**Operational Navigation of Agents and Self-organization Phenomena in Velocity-based Models for Pedestrian Dynamics**

Q. Xu (2022), xii, 112 pp  
ISBN: 978-3-95806-620-5

Band / Volume 50

**Utilizing Inertial Sensors as an Extension of a Camera Tracking System for Gathering Movement Data in Dense Crowds**

J. Schumann (2022), xii, 155 pp  
ISBN: 978-3-95806-624-3

Band / Volume 51

**Final report of the DeepRain project  
Abschlußbericht des DeepRain Projektes**

(2022), ca. 70 pp  
ISBN: 978-3-95806-675-5

Band / Volume 52

**JSC Guest Student Programme Proceedings 2021**

I. Kabadshow (Ed.) (2023), ii, 82 pp  
ISBN: 978-3-95806-684-7

Weitere **Schriften des Verlags im Forschungszentrum Jülich** unter  
<http://wwwzb1.fz-juelich.de/verlagextern1/index.asp>



IAS Series  
Band / Volume 52  
ISBN 978-3-95806-684-7