Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Octo-Tiger: Binary Star Systems with HPX on Nvidia P100

Gregor Daiß

Course of Study:	Informatik
Examiner:	Prof. Dr. Dirk Pflüger
Supervisor:	David Pfander
Commenced:	November 3, 2017
Completed:	May 3, 2018

Abstract

Stellar mergers between two suns are a significant field of study since they can lead to astrophysical phenomena such as type Ia supernovae. Octo-Tiger simulates merging stars by computing self-gravitating astrophysical fluids. By relying on the high-level library HPX for parallelization and Vc for vectorization, Octo-Tiger combines high performance with ease of development. For accurate simulations, Octo-Tiger requires massive computational resources. To improve hardware utilization, we introduce a stencil-based approach for computing the gravitational field using the fast multipole method. This approach was tailored for machines with wide vector units like Intel's Knights Landing or modern GPUs. Our implementation targets AVX512 enabled processors and is backward compatible with older vector extensions (AVX2, AVX, SSE). We further extended our approach to make use of available NVIDIA GPUs as coprocessors. We developed a tasking system that processes critical compute kernels on the GPU or the processor, depending on their utilization. Using the stencil-based fast multipole method, we gain a consistent speedup on all platforms, over the classical interaction-list-based implementation. On an Intel Xeon Phi 7210, we achieve a speedup of 1.9x. On a heterogeneous node with an Intel Xeon E5-2690 v3, we can obtain a speedup of 1.46x by adding an NVIDIA P100 GPU.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

The exchange of mass between two stars is an important field of research. It can lead to a multitude of different astrophysical phenomena depending on the exact scenario. In case of unstable mass transfer, it can lead to a merger which in turn can cause various events. For example, the merger between two white dwarfs alone can provoke the formation of R Coronae Borealis stars [CGH+07; Web84]. These are rare, short-lived stars that produce a large amount of dust and are thus interesting for studies of dust formation and evolution [Cla96]. Such a merger could also result in a type Ia supernovae. These supernovas lead to many important discoveries, such as the accelerated expansion of the universe [PAD+98; RFC+98]. They are also interesting points of study as the endpoints of stellar evolution and contributors to the chemical evolution of the universe [Mag17]. Mergers of white dwarfs are also considered to be a source of gravitational waves [BB98; CT02; Kat16].

Octo-Tiger is a code for simulating self-gravitating astrophysical fluids, primarily in the form of interacting binary star systems. It can, and has been used, to simulate the exchange of mass between stars. It employs adaptive mesh refinement (AMR) and uses a cell-based n-body solver, the fast multipole method (FMM), for the computation of the gravitational field and relies on high-level abstractions for parallelization [MKC+ew]. Unlike other similar simulations, Octo-Tiger is capable of conserving the angular momentum down the machine precision by using a modified, angular momentum conserving tree code for the FMM [Mar17]. Octo-Tiger and its predecessor, Scorpio, have been used to study the merger of double white dwarf systems resulting in the formation of R Coronae Borealis stars [MCML15]. It has also been used to test and develop the bi-polytropic self-consistent field which is outlined in [KMF+16] and is being used to simulate merging bi-polytropic systems [KCM+17]. Unlike other simulations, Octo-Tiger can not only scale to thousands of CPU cores, but hundreds of thousands. In prior works, Octo-Tiger achieved a parallel efficiency of 97% in a run using 655,520 Knights Landing cores [HLH+18]. This enables us to run accurate and large simulations of binary star systems.

In this work, we focus on the node-level performance of Octo-Tiger. We can break it down into two main objectives.

First, we take a look at the fast multipole method (FMM) that is used to calculate the effect of gravity. We primarily focus on its performance on the Intel Knights Landing platform. To improve the node-level performance, we propose a break from the traditional FMM implementation, by introducing a stencil-based FMM approach. This way, we still have all the advantages of the FMM algorithm, but achieve more cache and SIMD friendly code. In our prior work [PDM+18], we have already used this stencil-based FMM algorithm and studied its performance on the Intel Knights Landing platform. In this work, we formally introduce and compare it with the more standard FMM implementation previously used by Octo-Tiger.

Our second objective is to utilize heterogeneous systems containing a powerful CPU and powerful GPUs. Considering the massive resource requirements Octo-Tiger has for simulating astrophysical phenomena, these systems are especially interesting because they offer a high peak performance. We show how we can use the GPUs of these systems as a coprocessor for processing parts of the stencil-based FMM algorithm. We propose a system to offload work to the GPU, while still using the CPU to its fullest potential, without experiencing synchronization problems.

1.1 Related Works

Researchers have been attempting to simulate the mass exchange between stars as early as 1990 [BBCP90; Kat16] using hydrodynamical equations to collect more data and to better understand the phenomena itself, as well as the events it can lead to. However, the loss of angular momentum was still a problem, and the desire to utilize modern hardware drove the development of more simulation codes. A more recent example of such a simulation code is CASTRO [Kat16; KZC+16], which uses finite difference approximations and a multigrid solver to calculate the gravitational field using the hydrodynamic equations. It uses the Message Passing Interface (MPI) and OpenMP to run thousands of CPU cores in parallel. While there are some optimizations in place to reduce the loss of angular momentum, conservation is still violated [KZC+16].

In our prior work [PDM+18], we already studied the node-level performance of Octo-Tiger Intel Knights Landing platforms using the stencil-based FMM. Here, we also measured both the computational intensity of the most important parts of the FMM implementation, as well as the GFLOPS on an Intel Knights Landing platform.

1.2 Structure of this Work

The remainder of this work is structured as follows.

- **Chapter 2 - Octo-Tiger:** Here, we describe Octo-Tiger, the equations it is based upon and its two major components. Our main focus is the FMM component of Octo-Tiger.
- **Chapter 3 - Parallelism Libraries:** In this chapter, we introduce the two parallelism libraries Octo-Tiger uses. Utilizing these two libraries, HPX and Vc, Octo-Tiger can rely solely on high-level abstraction for parallelization and vectorization.
- **Chapter 4 - Octo-Tiger on Intel Knights Landing:** In this part of the work, we look into ways to improve the node-level performance of Octo-Tiger on the Intel Knights Landing platform. Here, we introduce the stencil-based FMM as a more cache and SIMD efficient approach to calculate the FMM interactions within Octo-Tiger.
- **Chapter 5 - Octo-Tiger's FMM in a Heterogeneous Environment:** Here, we first show how we can execute parts of the FMM algorithm on a GPU. We further introduce a way of using the GPU coprocessor for the FMM algorithm, that both avoids blocking and starvation on the CPU.

- **Chapter 6 - Results:** We discuss two sets of test results. In the first tests, we compare the performance stencil-based FMM algorithm with the more traditional FMM algorithm on multiple platforms. The second set of tests measures the performance gain we get by using both CPU and GPU, instead of just the CPU.
- **Chapter 7 - Conclusion and Outlook:** We follow up with a conclusion and look at future work regarding Octo-Tiger.

2 Octo-Tiger

In this chapter, we introduce Octo-Tiger and explain how it works. We explain the equations Octo-Tiger solves to simulate self-gravitating astrophysical fluids. To do so, we cover two major components of Octo-Tiger: The hydrodynamics solver and the gravity solver. We briefly talk about the the hydrodynamics solver. Afterward, we talk about the gravity solver in more detail. Since this solver takes up a majority of the runtime of Octo-Tiger, we do not only introduce the solver itself but also talk about the optimizations of the solver which are implemented in Octo-Tiger.

2.1 Hydrodynamics Solver

The first major component of Octo-Tiger is the hydrodynamics solver. We describe the fluid using the Navier-Stokes Equations. We only consider the case of compressible, inviscid fluids and can thus use the simplified version of the Navier-Stokes equation, the Euler equation. The equation for ensuring mass conservation reads as follows:

$$\frac{\partial}{\partial t}\rho + \nabla \cdot \rho \boldsymbol{v} = 0 \tag{2.1}$$

where ρ is the mass density and $\boldsymbol{v}$ the rotating frame fluid velocity. We further need the inertial frame fluid velocity u. Given the angular frequency $\boldsymbol{\Omega}$ and the distance from the origin $\boldsymbol{x}$, it can be calculated as $u := \boldsymbol{v} + \boldsymbol{x} \times \boldsymbol{\Omega}$. The equation for impulse conservation is the aforementioned simplified version of the original Navier-Stokes equation:

$$\frac{\partial}{\partial t}\boldsymbol{s} + \nabla \cdot \boldsymbol{v}\boldsymbol{s} + \nabla p = \rho \boldsymbol{g} + \boldsymbol{\Omega} \times \boldsymbol{s} \tag{2.2}$$

where $\boldsymbol{s}$ is the inertial frame linear momentum density, p the gas pressure and $\boldsymbol{g} = -\nabla\phi$ the gravitational acceleration.

In addition to the mass and impulse conservation, we also need to make sure that the energy is conserved. This inertial frame gas energy density E includes the kinetic, internal and degenerate gas energies. To conserve this energy and avoid numerical difficulties, we use the equation [MKC+ew].

$$\frac{\partial}{\partial t}\left(E + \frac{1}{2}\rho\phi\right) + \nabla \cdot (\boldsymbol{v}E + \boldsymbol{u}\rho + \boldsymbol{u}\rho\phi) = \frac{1}{2}\left(\phi\frac{\partial}{\partial t}\phi - \phi\frac{\partial}{\partial t}\rho\right) \tag{2.3}$$

Since we cannot get an analytical solution of these equations, we discretize these equations over time and employ a numerical solver. Octo-tiger uses a finite volume solver to do this. This solver is also modified to not only conserve linear momentum but also the angular momentum. The solver and its modifications are described in detail in [MKC+ew].

To get any meaningful results with Octo-Tiger, we need an inital state of the fluids for the first timestep. The model needs to resemble a binary star system from the beginning so we have to chose this initialization with care. Octo-Tiger uses the self concistent field technique [ET09; Hac86] to generate an inital binary star model. These models are an equilibrium state and can be three different models of binary star systems. The stars are either completely detached, semi-detached or in contact. Octo-Tiger supports all three of these models [MKC+ew].

The solver additionally requires the gravitational acceleration $\boldsymbol{g} := \nabla\phi$ and the associated gravitational potential ϕ. Due to this, we have to calculate the gravity in every timestep in order to solve the hydrodynamics. To do this we need an additional solver for the gravity.

2.2 Gravity Solver

The gravitational potential of the fluids at the position x is described by the equation

$$\phi[\boldsymbol{x}] := -\int_V \frac{G\rho'}{|\boldsymbol{x}-\boldsymbol{x}'|}dV' \tag{2.4}$$

we also need the solution time differentiate for the equation 2.3

$$\frac{\partial}{\partial t}\phi[\boldsymbol{x}] := -\int_V \frac{G\frac{\partial}{\partial t}\rho'}{|\boldsymbol{x}-\boldsymbol{x}'|}dV' \tag{2.5}$$

Octo-Tiger spends most of its runtime within this gravity solver, hence we take a closer look at it. To compute the gravitational potential ϕ given by the equation 2.4, we discretize the volume V and use an n-body based solver.

2.2.1 Discretization

We dicretize the volume V into cells. These cells contain the required variables for their position and are located on a rotating Cartesian adaptive mesh refinement (AMR) grid. This grid is decomposed as an octree. A node in this tree does not only manage one cell; it manages N^3 cells as a subgrid. For the rest of this work, we will use $N := 8$ as a parameter of the size of said subgrid, yielding a size of 512 cells per node. The parameter N can be seen as a trade-off parameter between how finely the grid is adapted to the problem and the computational efficiency, since it is easier to process more data on one node instead of constantly moving to the children of a node. Each of these nodes either has eight children or is not refined at all. In case a node is refined, its children's subgrids have twice the resolution of its own subgrid. Whether or not a node gets refined is decided by the refinement criteria based on the local density ρ as outlined in [MKC+ew]. This criterion is checked every 15 time steps of the hydrodynamics solver. In the following, we call a cell a monopole cell if it is based on a subgrid in a leaf node. Otherwise, we call it a multipole cell.

An exemplary depiction of an octree can be seen in figure 2.1. This tree has been refined either two or one time, depending on the location of the cells we consider.

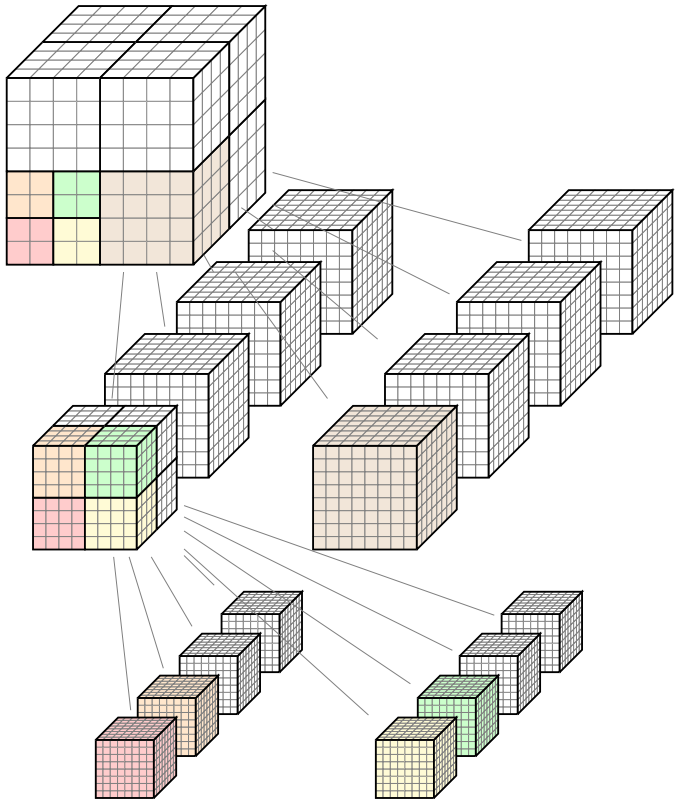

Figure 2.1: Visualization of the adaptive octree from our prior work [PDM+18]

2.2.2 n-body Solver

We have to calculate the gravitational field ϕ for each cell of the octree by considering the influence of all other cells. For n cells this would lead to a runtime complexity of $O(n^2)$ if we use a naive approach. We could already reduce this to a complexity of $O(n \log n)$ by using the Barnes-Hut algorithm [BH86].

However, we are using the Fast Multipole Method (FMM) as first outlined in [GR87] to approximate the gravitational potential and achieve a runtime complexity of $O(n)$. When computing the gravitational field at one point, the FMM algorithm approximates the influences of distant collections of masses with a multipole approximation [Tho80]. By using the modifications in [Deh00] we conserve linear momentum to machine precision. Furthermore, one of Octo-Tiger's most important features is that it conserves angular momentum as well. To achieve this we use modifactions to the FMM algorithm as outlined in [Mar17].

Unlike other FMM implementations, we are not considering particles which are contained in a cell. We are just considering the density of a cell, which is stored in a singular value in each cell ρ. Essentially, the FMM solver gets the current density at each cell as an input, and outputs Taylor coefficients for each cell. Using these coefficients of a cell, the hydrodynamics solver can easily compute the gravitational field ϕ at this cell location. The Taylor coefficients themselves are computed in the FMM by considering cell to cell interactions, or approximating interactions of well separated cells.

Notation

In the following, we use a special notation for variables, since we are dealing with multiple coefficients. The same notation is used in the orignal Octo-Tiger work [MKC+ew]. We demonstrate this notation at the example of the multipole variable that we need later. A multipole M^n of the order n and below, consists of multiple coefficients depending on n. For $n = 0$ it contains a single coefficient, for $n = 1$ three coefficients, for $n = 2$ nine coefficients and for $n = 3$ we deal with 27 coefficients. To refer to a single element within a multipole M^n we use three helper variables m, n, p. These form a multi-index. Each of the three helper variables is an element of the set $\{0, 1, 2\}$. We will not predetermine the value of the helper variables, instead each of them could be any of the three possible elements. This helps use to denote multipoles of order higher than zero. $M^{(1)}_m$ thus stands for three possible coefficients, which are $M^{(1)}_0$, $M^{(1)}_1$, and $M^{(1)}_2$. $M^{(2)}_{mn}$ stand for nine coefficients and $M^{(3)}_{mnp}$ for 27 coefficients. The same notation holds true for other variables we need. For example, the Taylor coefficients of a cell q are denoted by $L^{(0)}_q$, $L^{(1)}_{q,m}$,$L^{(2)}_{q,mn}$ and $L^{(3)}_{q,mnp}$, containing as much coefficients as the multipoles. Other variables we need are the cell q to cell l interactions $A^{(0)}_{ql}$,$A^{(1)}_{ql,m}$,$A^{(2)}_{ql,mn}$,$A^{(3)}_{ql,mnp}$, $B^{(1)}_{q,m}$ and the gradients of the Green's function $D^{(0)}_{ql}$,$D^{(1)}_{ql,m}$,$D^{(2)}_{ql,mn}$,$D^{(3)}_{ql,mnp}$, all of which will be defined in the next section. We will further need variables for the terms we require to conserve angular momentum $\mathcal{L}^{(1)}_{q,m}$.

Basic Algorithm

The FMM implementation consists of three basic steps, which are explained in more detail after a short overview:

Data: Density ρ_i at each monopole cell i
Result: Taylor coefficients $L^{(1)}_{q,m}$,$L^{(2)}_{q,mn}$, $L^{(3)}_{q,mnp}$ and the angular momentum correction $\mathcal{L}^{(1)}_{q,m}$ for each cell q
Step 1: Compute multipole moments $M^{(0)}_{ql}$,$M^{(1)}_{ql,m}$,$M^{(2)}_{ql,mn}$ and $M^{(3)}_{ql,mnp}$ at each cell q in a bottom up octree traversal using the density at the monopole cells;
Step 2: Compute the cell to cell interactions $A^{(0)}_{ql}$,$A^{(1)}_{ql,m}$,$A^{(2)}_{ql,mn}$,$A^{(3)}_{ql,mnp}$ and $B^{(1)}_{q,m}$ of all cell pairs (q, l) which are on the same level in the octree and are located close to each other;
Step 3: In a top down traversal, combine all cell to cell interactions for a cell with the Taylor coefficients of its parent, to get the Taylor coefficient for this cell. Do the same for the angular momentum corrections;

Algorithm 1: The basic Fast Multipole Method (FMM) algorithm.

We now take a detailed look at each of these three steps.

Step 1: Computation of multipole moments

We compute the multipole moments for each cell in a bottom-up tree traversal. At the leaf nodes each cell l just has a zeroth level multipole, or monopole, which gets calculated by the volume of the cell $(\triangle x)^3$ and its local density ρ_l.

$$M_l^{(0)} = \rho_l (\triangle x)^3 \tag{2.6}$$

With the following equations, we calculate the multipole moments of a non-leaf node using the multipole of its eight children

$$M^{(0)} := \sum_{l=1}^{8} M_l^{(0)} \tag{2.7}$$

$$M_m^{(1)} := \sum_{l=1}^{8} \left(M_l^{(0)} x_{l,m} + M_{l,m}^{(1)} \right) \tag{2.8}$$

$$M_{mn}^{(2)} := \sum_{l=1}^{8} \left(M_l^{(0)} x_{l,m} x_{l,n} + M_l^{(1)} x_{l,m} + M_{l,m}^{(2)} \right) \tag{2.9}$$

$$M_{mnp}^{(3)} := \sum_{l=1}^{8} \left(M_l^{(0)} x_{l,m} x_{l,n} x_{l,p} + M_l^{(1)} x_{l,m} x_{l,n} + M_l^{(2)} x_{l,m} + M_{l,m}^{(3)} \right) \tag{2.10}$$

where x_l is the center of the child cells expansion [MKC+ew]. After the bottom-up tree traversal we have the multipole moments of all cells. Using these, we can compute the cell to cell interactions in the next step.

Step 2: Computation of the Cell to Cell Interactions

Using the multipole moments we calculated in the bottom-up tree traversal, we can calculate the cell to cell interactions for all cells q and their respective, close-by, neighbor cells l. To do this, we must first define what qualifies as a close-by neighbor. For this we use the opening criteria:

$$\frac{1}{|\mathbf{Z}_q^f - \mathbf{Z}_l^f|} \leq \Theta < \frac{1}{|\mathbf{Z}_q^c - \mathbf{Z}_l^c|}. \tag{2.11}$$

where Z_q^f and Z_l^f are the coordinate centers of two cells, and Z_q^C and Z_l^C are the coordinate centers of their parent cells. For all neighboring cells which are (1) on the same level in the octree and (2) fulfill this criteria, we calculate the cell to cell interactions $A_{ql}^{(0)}$,$A_{ql,m}^{(1)}$,$A_{ql,mn}^{(2)}$,$A_{ql,mnp}^{(3)}$ and $B_{q,m}^{(1)}$.

The number of neighbors that are considered due to the opening criteria is always the same. The computational complexity per cell is thus constant and does not depend on the total number of cells n. Since the number of valid neighbors is always constant, we can simply store the indices of interacting cells in an interaction list $ilist$. This list stores index tuples in the form of (q, l). Each of this tuple means that cell q is directly interacting with cell l. For convenience we define ilist$(q) = \{l|(q, l) \in \text{ilist}\}$, to get a list over all interaction partners of a cell q. The neighbors of a cell can also be interpreted in the form of a discretized sphere. A visualization of that stencil can be seen in figure 2.2.

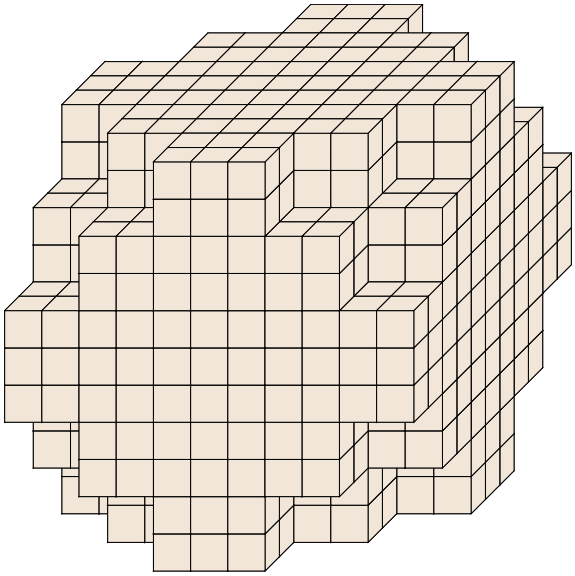

Figure 2.2: Visualization of discretized spherical stencil from our prior work [PDM+18]. This stencil is implied by the opening criteria.

Given the list with interacting pairs of cells (q, l), we now need to compute the cell to cell interactions for each of these pairs. Do do this, we first need the gradients of the Green's function for gravitational potential. This gets calculated using the distance between the cells.

$$D^{(0)}(\boldsymbol{R}_{l,q}) = -\frac{1}{R}, \tag{2.12}$$

$$D^{(1)}_i(\boldsymbol{R}_{l,q}) = \frac{R_i}{R^3}, \tag{2.13}$$

$$D^{(2)}_{ij}(\boldsymbol{R}_{l,q}) = -\frac{3R_iR_j-\delta_{ij}R^2}{R^5}, \tag{2.14}$$

$$D^{(3)}_{ijk}(\boldsymbol{R}_{l,q}) = \frac{15R_iR_jR_k-3(\delta_{ij}R_k+\delta_{jk}R_i+\delta_{ki}R_j)R^2}{R^7}, \tag{2.15}$$

where $\boldsymbol{R}_{l,q} := \boldsymbol{R}_l - \boldsymbol{R}_q$ is the distance vector between the two interacting cells, $R := ||\boldsymbol{R}_{l,q}||_2$. If a cell q is a monopole cell, we simply use the middle of the cell as the position $\boldsymbol{R_q}$, otherwise we use the center of mass, which can be calculated using its child cells. The same holds true for the position $\boldsymbol{R_l}$.

With both the multipoles and the coefficients Green's function we can calculate the cell to cell interactions $A^{(0)}_{ql}$,$A^{(1)}_{ql,m}$,$A^{(2)}_{ql,mn}$ and $A^{(3)}_{ql,mnp}$.

$$A^{(0)}_{ql} := M^{(0)}_l D^{(0)}(\boldsymbol{R}_{l,q}) + M^{(1)}_{l,m} D^{(1)}_m(\boldsymbol{R}_{l,q}) + M^{(2)}_{l,mn} D^{(2)}_{mn}(\boldsymbol{R}_{l,q}) + M^{(3)}_{l,mnp} D^{(3)}_{mnp}(\boldsymbol{R}_{l,q}), \tag{2.16}$$

$$A^{(1)}_{ql,m} := M^{(0)}_l D^{(1)}_m(\boldsymbol{R}_{l,q}) + M^{(1)}_{l,n} D^{(2)}_{mn}(\boldsymbol{R}_{l,q}) + M^{(2)}_{l,np} D^{(3)}_{mnp}(\boldsymbol{R}_{l,q}), \tag{2.17}$$

$$A^{(2)}_{ql,mn} := M^{(0)}_l D^{(2)}_{mn}(\boldsymbol{R}_{l,q}) + M^{(1)}_{l,p} D^{(3)}_{mnp}(\boldsymbol{R}_{l,q}), \tag{2.18}$$

$$A^{(3)}_{ql,mnp} := M^{(0)}_l D^{(3)}_{mnp}(\boldsymbol{R}_{l,q}). \tag{2.19}$$

If we calculate these coefficients this way, the linear momentum is already conserved. However, to also conserve angular momentum we have to add a corrective term to $L^{(1)}_{q,m}$, as outlined in [Mar17]. The corrective term which which we need for the cell to cell interaction between q and l is $B^{(1)}_{q,m}$. $r \in \{0, 1, 2\}$ is like a m, n, and p, an additional helper variable.

$$B^{(1)}_{ql.m} := \frac{1}{2}\left(M^{(2)}_{l,n,pr} - \frac{M^{(0)}_{l}}{M^{(0)}_{q}} M^{(2)}_{q,n,pr}\right) D'^{(4)}_{mnpr}(R_{l,q}) \tag{2.20}$$

As shown in [MKC+ew], we calculate the coefficients $D'^{(4)}_{mnpr}(R)$ like this

$$D'^{(4)}_{mnpr}(\boldsymbol{R}) := \frac{15\left(\delta_{ij}R_kR_l + \delta_{ik}R_jR_l + \delta_{il}R_jR_k\right)}{R^7} - \frac{3\left(\delta_{ij}\delta_{kl} + \delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}\right)}{R^5}. \tag{2.21}$$

Step 3: Computation of the Taylor Coefficients of each Cell

We introduced the cell to cell interactions for cell pairs on the same octree level that are located close to each other acording to the opening criteria 2.11. However, to calculate the the Taylor coefficients for a cell, we still need its interactions with the cells outside of its opening criteria. These interactions are not calculated directly. Instead, we use the parent cells, since they interact with other cells that are outside of the opening criteria of the respective child cells. We can add these contributions to the Taylor coefficients by using an up-down tree traversal. Variables with C stand for the value of the parent of the current cell. These values are already up to date, since the Taylor coefficients of the parent cell are always processed first in this tree traversal. The root node simply uses empty values instead the parent values.

Given all cell to cell interactions $A^{(0)}_{ql}$,$A^{(1)}_{ql,m}$,$A^{(2)}_{ql,mn}$,$A^{(3)}_{ql,mnp}$ of a cell q with all same-level interaction partners $l \in \text{ilist}(q)$, as well as the Taylor coefficients of the parent cell $L^{C,(0)}_{q}$,$L^{C,(1)}_{q,m}$,$L^{C,(2)}_{q,mn}$, $L^{C,(3)}_{q,mnp}$ we can now calculate the Taylor coefficients of the cell itself:

$$L^{(0)}_{q} := \left(\sum_{l}^{\text{ilist}(q)} A^{(0)}_{ql}\right) + L^{C,(0)}_{q} - L^{C,(1)}_{q,m}x_m + L^{(C,(2)}_{q,mm}x_nx_m - L^{C,(2)}_{q,mnp}x_xx_mx_p \tag{2.22}$$

$$L^{(1)}_{q,m} := \left(\sum_{l}^{\text{ilist}(q)} A^{(1)}_{q,m}\right) + L^{C,(1)}_{q,m} + L^{(C,(2)}_{q,mm}x_n - L^{C,(2)}_{q,mnp}x_xx_p \tag{2.23}$$

$$L^{(2)}_{q,mn} := \left(\sum_{l}^{\text{ilist}(q)} A^{(2)}_{q,mn}\right) + L^{(C,(2)}_{q,mm} - L^{C,(2)}_{q,mnp}x_p \tag{2.24}$$

$$L^{(3)}_{q,mnp} := \left(\sum_{l}^{\text{ilist}(q)} A^{(3)}_{q,mnp}\right) + L^{C,(2)}_{q,mnp}x_p \tag{2.25}$$

Again a correction term is required to ensure angular momentum conservation. For this, we add up the terms $B^{(1)}_{q,m}$ from all cells and combine it with the corrective term of its parent $\mathcal{L}^{C,(1)}_{q.m}$.

$$\mathcal{L}^{(1)}_{q,m} := \left(\sum_{l}^{\text{ilist}(q)} B^{(1)}_{q,m}\right) + \mathcal{L}^{C,(1)}_{q.m} \tag{2.26}$$

After we have completed the up-down traversal of the octree, we have to finish all steps of the FMM algorithm and get the Taylor coefficients $L_q^{C,(0)}$,$L_{q,m}^{C,(1)}$,$L_{q,mn}^{C,(2)}$, $L_{q,mnp}^{C,(3)}$ for each monopole cell, as well as its correcitve terms $\mathcal{L}_{q.m}^{C,(1)}$. These terms can be used to calculate $\boldsymbol{g}$, ϕ and $\frac{\partial}{\partial t}\phi$ within the hydrodynamics solver [MKC+ew].

2.2.3 Introducing the FMM Interaction Kernel

Octo-Tiger does not store the cell to cell interactions $A_{ql}^{(0)}$,$A_{ql,m}^{(1)}$,$A_{ql,mn}^{(2)}$, $A_{ql,mnp}^{(3)}$, $B_{ql.m}^{f,(1)}$ explictly. Instead, we can immediately sum them up directly after their calculation:

$$A^{(0)}_{\text{combined},q} := \left(\sum_{l}^{\text{ilist}(q)} A^{(0)}_{ql}\right) \tag{2.27}$$

$$A^{(1)}_{\text{combined},q,m} := \left(\sum_{l}^{\text{ilist}(q)} A^{(1)}_{q,m}\right) \tag{2.28}$$

$$A^{(2)}_{\text{combined},q,mn} := \left(\sum_{l}^{\text{ilist}(q)} A^{(2)}_{q,mn}\right) \tag{2.29}$$

$$A^{(3)}_{\text{combined},q,mnp} := \left(\sum_{l}^{\text{ilist}(q)} A^{(3)}_{q,mnp}\right) \tag{2.30}$$

$$B^{(1)}_{\text{combined},q,m} := \left(\sum_{l}^{\text{ilist}(q)} B^{(1)}_{q,m}\right) \tag{2.31}$$

Thus, we separate the summation of the interaction from the eventual computation of the Taylor coefficients. This computation can now be written as:

$$L_q^{(0)} := A^{(0)}_{\text{combined},q} + L_q^{C,(0)} - L_{q,m}^{C,(1)} x_m + L_{q,mm}^{(C,(2)} x_n x_m - L_{q,mnp}^{C,(2)} x_x x_m x_p \tag{2.32}$$

$$L_{q,m}^{(1)} := A^{(1)}_{\text{combined},q,m} + L_{q,m}^{C,(1)} + L_{q,mm}^{(C,(2)} x_n - L_{q,mnp}^{C,(2)} x_x x_p \tag{2.33}$$

$$L_{q,mn}^{(2)} := A^{(2)}_{\text{combined},q,mn} + L_{q,mm}^{(C,(2)} - L_{q,mnp}^{C,(2)} x_p \tag{2.34}$$

$$L_{q,mnp}^{(3)} := A^{(3)}_{\text{combined},q,mnp} + L_{q,mnp}^{C,(2)} x_p \tag{2.35}$$

$$\mathcal{L}_{q,m}^{f,(1)} := B^{(1)}_{\text{combined},q,m} + \mathcal{L}_{q.m}^{C,(1)} \tag{2.36}$$

$$\tag{2.37}$$

Now that we separated the summation of the cell to cell interactions into the equations we can describe the algorithm that computes them. Since cells in Octo-Tiger are always managed in groups of 512 cells per octree node, this algorithm does not only describe the summation for one cell, but for a whole subgrid of 512 cells. Thus we must have an interaction list for each node of the octree. These lists contains the interactions of all 512 cells with all cells either from the same subgrid, or from one of the neighboring subgrids. Again, the opening criteria 2.11 decides whether two cells directly interact and are thus part of the list.

Using these node-specific interaction lists we can formulate the algorithm 2. We call this algorithm the FMM interaction kernel. Since it is responsible for most of the computations within Octo-Tiger, it is an apt target for optimizations.

Data: All multipoles and center of masses of the cells within the subgrid and its neighbor subgrids, as well ilist of this node
Result: All combined cell to cell interactions for each cell within the subgrid
while *we have not processed all elements of the interaction list* $ilist$ **do**
  Step 1: Load next cell pair (q, l) from interaction list $ilist$;
  Step 2: Load multipole moments and the center of masses for these two cells;
  Step 3: Compute Green's gradients using the center of masses with equation 2.12;
  Step 4: Compute cell to cell interaction results $A^{(0)}_{ql}$,$A^{(1)}_{ql,m}$,$A^{(2)}_{ql,mn}$,$A^{(3)}_{ql,mnp}$between l and q using the equations 2.16-2.19;
  Step 5: Add cell to cell interactions to combined result for this cell;
  Step 6: Compute cell to cell interaction corrective term $B^{(1)}_{q,m}$ using the equation 2.20;
  Step 7: Add to corrective term $B^{(1)}_{q,m}$ to the combined corrective term.
end

Algorithm 2: This algorithm shows the general FMM interaction kernel. It computes all cell to cell interactions required by the 512 cells in one subgrid

2.3 Optimized Gravity Solver

We have to compute the solution for the gravitational potential for every time step in the hydrodynamics solver. The computation of the gravity thus takes up the majority of Octo-Tiger's runtime. Especially the computation of the cell to cell interactions in step two is very time-consuming. In this step, we have to compute the interactions of every cell with its close-by neighbors. In praxis, this means we have to compute about a thousand cell to cell interactions for each singular cell. This is done for all cells by computing the FMM interaction kernel in algorithm 2 for each node in the octree.

Octo-Tiger employs multiple optimizations to lessen the runtime of this step. (1) We reduce the number of coefficients per cell by exploiting symmetry. (2) We adapt the equations of the cell to cell interaction to the type of cells considered. This leads to four specialised FMM kernels. (3) In some steps we can avoid computing the angular corrections. Using this, we can create three additional kernels which skip this step.

In this section, we will look at each of these optimizations and conclude with an overview of these newly introduced FMM kernels.

2.3.1 Using less Coefficients

A first optimization for the FMM interaction kernel is the reduction of required coefficients. As we have mentioned earlier, each cell q holds multiple multipole coefficients as well as multiple Taylor coefficients depending on the order. For the zeroth order $M^{(0)}_q$ there is one coefficient, three coefficients for the first order $M^{(1)}_m$, nine for the second $M^{(2)}_{mn}$ and lastly 27 coefficients for the third

order $M^{(3)}_{mnp}$. Overall this means we have 40 coefficients for the multipole moments of one cell alone. The same holds true for the Taylor coefficients L and the cell to cell interactions A. Exploiting symmetry, we can reduce the number of required coefficients for each of them from 40 to 20, more specifically from the zeroth to the third order we need one, three, six, and ten elements. This reduces the number of computations and the memory requirements.

2.3.2 Use specialized FMM interaction kernels depending on the Cell Type

The equations 2.16 to 2.19 can be simplified depending on the nature of the interacting cells. Since nodes on the same level in the octree can be differently refined, there are four possible interaction types: Two multipole cells interact ($m2m$ interaction), two non-refined monopole cells interact ($p2p$ interaction) and the mixed interactions with monopole-multipole ($p2m$) pairs and multipole-monopole ($m2p$) pairs.

There are two possible ways to optimize the equations depending on these types: We can reduce the number of multipole moments to consider and thus the number of computations since only $M^{(0)}$ is non-zero for monopole cells. For direct interactions in leaf nodes (p2p or p2m) Octo-Tiger only calculates Taylor coefficients of the zeroth and first order as outlined in [Deh02].

Using this we can introduce three new kernels. In order to maintain a simple notation, we keep it similar to the notation of the general FMM interaction kernel in algorithm 2.

To define these new, simpler kernels, we need to split the interaction list into four interaction lists depending on the type of the interacting cells:

- $ilist_{p2p}$ containing the cell pairs (q, l), where both cells are monopole cells
- $ilist_{p2m}$ containing the cell pairs (q, l), where q is a monopole cell, but l is a multipole cell
- $ilist_{m2p}$ where q is a multipole cell and l is a monopole cell
- $ilist_{m2m}$ where all cells are multipole cells

Monopole-Monopole (p2p) Specialization

For interactions between monopole cells, we can reduce the number of multipole moments to consider, we only need to compute the Taylor coefficients of the zeroth and first order. With these simplifications, we get the following equations to calculate the Taylor coefficients for a monopole cell q which is interacting with other monopoles:

$$A^{(0)}_{\text{p2p},ql} := M^{(0)}_l D^{(0)}(\boldsymbol{R}_{l,q}), \tag{2.38}$$

$$A^{(1)}_{\text{p2p},ql,m} := M^{(0)}_l D^{(1)}_m(\boldsymbol{R}_{l,q}) \tag{2.39}$$

We need to compute only $D^{(0)}$ and $D^{(1)}_m$ instead of all gradients of the Green's function. Furthermore, we can also drop the equation 2.20 in this case, since the term would be zero for interactions between monopoles.

We can thus formulate a p2p FMM interaction kernel 3 that requires less computations than the original FMM kernel in algorithm 2.

Data: All multipoles moments $M^{(0)}$ of the cells within the subgrid and its neighbor subgrids, as well as the $ilist_{\text{p2p}}$ of this node
Result: All monopole monopole interactions are added to the combined interaction variable
while *we have not processed all elements of the interaction list* $ilist_{p2p}$ **do**
  Step 1: Load next cell pair (q, l) from interaction list $ilist_{\text{p2p}}$;
  Step 2: Load multipole moments $M^{(0)}$ for these two cells;
  Step 3: Compute Green's gradients $D^{(0)}$ and $D^{(1)}_m$;
  Step 4: Compute cell to cell interaction results $A^{(0)}_{ql}$,$A^{(1)}_{ql,m}$ between l and q using the equations 2.38-2.39;
  Step 5: Add cell to cell interactions to combined result for this cell;
end

Algorithm 3: FMM interaction kernel that computes all monopole-monopole interactions for a subgrid.

Monopole-Multipole (p2m) Specialization

For monopole-multipole interactions we have to consider all multipole moments, since the interaction partners l of the monopole q are multipoles. We still benefit from the fact that q is a monopole by having to compute fewer Taylor coefficients. Thus, the mixed interactions between a monopole q and its neighboring multipoles l can be written as:

$$A^{(0)}_{\text{p2m},ql} := M^{(0)}_l D^{(0)}(\boldsymbol{R}_{l,q}) + M^{(1)}_{l,m} D^{(1)}_m(\boldsymbol{R}_{l,q}) + M^{(2)}_{l,mn} D^{(2)}_{mn}(\boldsymbol{R}_{l,q}) + M^{(3)}_{l,mnp} D^{(3)}_{mnp}(\boldsymbol{R}_{l,q}) \quad (2.40)$$

$$A^{(1)}_{\text{p2m},ql,m} := M^{(0)}_l D^{(1)}_m(\boldsymbol{R}_{l,q}) + M^{(1)}_{l,n} D^{(2)}_{mn}(\boldsymbol{R}_{l,q}) + M^{(2)}_{l,np} D^{(3)}_{mnp}(\boldsymbol{R}_{l,q}) \quad (2.41)$$

$$(2.42)$$

Unlike the with the p2p specialization, we still have to compute the angular corrections for this case. However, since q is a monopole cell, and $M^{(2)}_{l,pr}$ is 0, and we can thus simplify the equation to:

$$B^{(1)}_{\text{p2m},ql.m} := \frac{1}{2} M^{(2)}_{l,n,pr} D'^{(4)}_{mnpr}(R_{l,q}) \quad (2.43)$$

Unfortunately, we still have to compute all of the D coefficients. Using these new equations, we can rewrite the general FMM kernel from algorithm 2 to save some computations. This yields the p2m FMM interaction kernel, as shown in algorithm 4.

Data: All multipoles and center of masses of the cells within the subgrid and its neighbor subgrids, as well $ilist_{\text{p2m}}$ of this node
Result: All combined cell to cell interactions for each cell within the subgrid
while *we have not processed all elements of the interaction list* $ilist_{p2m}$ **do**
  Step 1: Load next cell pair (q, l) from interaction list $ilist_{\text{p2m}}$;
  Step 2: Load multipole moments and the center of masses for these two cells;
  Step 3: Compute Green's gradients using the center of masses with equation 2.12;
  Step 4: Compute cell to cell interaction results $A^{(0)}_{\text{p2m},ql}$,$A^{(1)}_{\text{p2m},ql,m}$ between l and q using the equations 2.40-2.41;
  Step 5: Add cell to cell interactions to combined result for this cell;
  Step 6: Compute cell to cell interaction corrective term $B^{(1)}_{\text{p2m},q,m}$ using the equation 2.43;
  Step 7: Add to corrective term $B^{(1)}_{\text{p2m},q,m}$ to the combined corrective term.
end

Algorithm 4: FMM interaction kernel that computes all monopole-multipole (p2m) interactions for a subgrid.

Multipole-Monopole (m2p) Specialization

In the case of multipole-monopole interactions, we have to compute all Taylor coefficients, but can ignore all but the zeroth multipole moment (the monopole).

$$A^{(0)}_{\text{m2p},q} := M^{(0)}_l D^{(0)}(\boldsymbol{R}_{l,q}), \tag{2.44}$$

$$A^{(1)}_{\text{m2p},q,m} := M^{(0)}_l D^{(1)}_m(\boldsymbol{R}_{l,q}), \tag{2.45}$$

$$A^{(2)}_{\text{m2p},q,mn} := M^{(0)}_l D^{(2)}_{mn}(\boldsymbol{R}_{l,q}), \tag{2.46}$$

$$A^{(3)}_{\text{m2p},q,mnp} := M^{(0)}_l D^{(3)}_{mnp}(\boldsymbol{R}_{l,q}) \tag{2.47}$$

Again, we can also simplify the equation for the corrective term $B^{(1)}_{\text{m2p},ql.m}$, since one of the cells is a monopole.

$$B^{(1)}_{\text{m2p},ql.m} := \tfrac{1}{2}\left(-\frac{M^{(0)}_l}{M^{(0)}_q} M^{(2)}_{q,n,pr}\right) D'^{(4)}_{mnpr}(R_{l,q}) \tag{2.48}$$

Using these new equations, we can write the multipole-monopole FMM interaction kernel, as seen in algorithm 5.

Data: All multipoles and center of masses of the cells within the subgrid and its neighbor subgrids, as well $ilist_{\text{mp2}}$ of this node
Result: All combined cell to cell interactions for each cell within the subgrid
while *we have not processed all elements of the interaction list* $ilist_{mp2}$ **do**
  Step 1: Load next cell pair (q, l) from interaction list $ilist_{\text{mp2}}$;
  Step 2: Load multipole moments and the center of masses for these two cells;
  Step 3: Compute Green's gradients using the center of masses with equation 2.12;
  Step 4: Compute cell to cell interaction results $A^{(0)}_{\text{m2p},ql}$,$A^{(1)}_{\text{m2p},ql,m}$,$A^{(2)}_{\text{m2p},ql,mn}$,$A^{(3)}_{\text{m2p},ql,mnp}$between l and q using the equations 2.44-2.47;
  Step 5: Add cell to cell interactions to combined result for this cell;
  Step 6: Compute cell to cell interaction corrective term $B^{(1)}_{q,m}$ using the equation 2.48;
  Step 7: Add to corrective term $B^{(1)}_{\text{m2p},q,m}$ to the combined corrective term.
end

Algorithm 5: FMM interaction kernel that computes all multipole-monopole interactions for a subgrid.

Multipole-Multipole (m2m) Specialization

Unfortunately, we cannot reduce the amount of computations that are required to calculate the cell to cell interactions between two multipole cells. In this case, we have to compute all Taylor coefficients of the cell q, using all multipole moments of both cells. Thus the m2m FMM interaction kernel is the same as the general FMM kernel, seen in Algorithmus 2.

2.3.3 Adding Kernels without Angular Correction

In some steps of the gravity solver, Octo-Tiger requires the Taylor coefficients for each cell, but not the angular corrections. In these steps, we can skip the computation of B. The FMM kernel presents the inner-most loop of computation, thus we should not skip the computation by introducing conditional code within this critical part. Instead, we create new kernels, which exclude the steps to calculate B and to add it to the overall corrections. This yields another FMM kernel for each of the already introduced kernels, except for the p2p FMM kernel since it does not compute angular corrections. We call these new kernels that skip the computation of the angular corrections non-rho kernels.

For the p2m FMM interaction kernel, we add the p2m FMM non-rho kernel. Accordingly, we add the m2p and the m2m non-rho FMM kernels. Each of these kernels essentially works in the same way as the respective kernel that computes the angular corrections, where just the steps six and seven of the algorithm are omitted.

Kernel	Algorithm
p2p FMM interaction kernel	Algorithm 3
p2m FMM interaction kernel	Algorithm 4
m2p FMM interaction kernel	Algorithm 5
m2m FMM interaction kernel	Algorithm 2
p2m non-rho FMM interaction kernel	Algorithm 4, skipping step 6 and 7
m2p non-rho FMM interaction kernel	Algorithm 5, skipping step 6 and 7
m2m non-rho FMM interaction kernel	Algorithm 2, skipping step 6 and 7

Table 2.1: Overview over all interaction list FMM kernels and their respective, scalar algorithm.

2.3.4 Overview over the FMM Interaction Kernels

Using all of specializations we now have not one, but seven FMM interaction kernels. These include the four specializations depending on the cell types, as well as three specializations which skip the computation of the corrective term for the angular moment.

The seven kernels represent the most compute-intensive parts of Octo-Tiger. An overview over them is given in table 2.1.

3 Parallelism Libraries

Octo-Tiger utilizes two libraries to achieve distributed, local and data-level parallelism. It uses HPX to for the former two types of parallelism, and Vc for vectorization to achieve data-level parallelism.

3.1 HPX - A High Level Runtime Framework for Local and Distributed Task Parallelism

To obtain meaningful results using Octo-Tiger we have to run large scenarios. Therefore Octo-Tiger not only needs to be able to use all resources on a single compute node, it needs to run distributed across many nodes to be able to finish the computation in a reasonable time.

While machines are readily available, developing applications that scale to a lot of nodes while preserving parallel efficiency is a problem that has occupied both science and industry. In [KHA+14], the problems which are limiting us are called the SLOW factors: **S**tarvation, **L**atencies, **O**verheads and **W**aiting. Starvation will occur when there is just not enough parallel work to keep all resources busy. An example for this would be global barriers since there are usually threads that finish sooner and have to wait. Latencies arise when we have to wait for communication or remote resources. Managing a high number of different compute nodes also introduces overheads for the management of the parallel actions. Lastly, if we oversubscribe a shared resource, we have to wait for contention resolution.

In order to solve these issues, Octo-Tiger has been developed using High Performance ParalleX, or HPX [HDB+17; HKD+16; KHA+14]. HPX is an open source C++ runtime system which implements the ParalleX computation model [GSS+07; KBS09] for parallel and distributed applications in order to tackle the aforementioned problems. It uses an asynchronous many-task runtime system (AMT) allowing developers to convey complex data flow execution trees with up to billions of tasks, which are being scheduled when their dependencies are met.

HPX has already been successfully used with good scaling results on heterogeneous supercomputers [HKSF13], it performs well against more traditional Frameworks as MPI or OpenMP [HKD+16; HKI13] and Octo-Tiger has been shown to achieve a parallel efficiency of 97% using 655,520 Knight Landing cores [HLH+18]. Other applications that use HPX include the LibGeoDecomp N-Body Code or Storm Surge Forecasting [HDB+17]. Furthermore, HPX includes GPU support, albeit it is still experimental [HDB+17; WKH15].

HPX is designed around multiple overarching principles which we will take advantage of for executing the FMM kernels of Octo-Tiger to the GPU, while still being able to work on other tasks on the CPU. Therefore we will take a short look at these principles. A more thorough description is to be found at [KHA+14].

One of these principles concerns latencies. HPX focuses on hiding latencies instead of avoiding them. It is impossible to completely hide latencies, as they arise for a multitude of tasks, for example, the communication with a GPU or over a network. We should use the idle time these latencies introduce to do other useful computations. We effectively want to hide the latency by switching the thread to another context where it can continue to work. To achieve this HPX supports very light-weight threads with short context switching time.

Another principle is the avoidance of global barriers. Each task should be able to continue as soon as its preconditions are met, using a constraint-based synchronization. This type of computing was already described in [Den74]. Instead of relying on barriers to synchronize the control flow between several threads, in HPX an operation can proceed as soon as its preconditions are met. Using Futurization, it is possible to express dataflow execution trees in HPX with up to billions of different tasks. These tasks are only scheduled when their dependencies are satisfied, and will then be executed by a worker thread. Through the usage of thread local variables, it is also possible to have memory associated with a worker thread, instead of the tasks. This memory can be useful for reusing variables in between HPX tasks.

Other design principles of HPX concern the distributed case where we have to deal with a large number of separated nodes. While these are important to Octo-Tiger at large, for this work we will focus on the node level performance itself and thus only mention these principles shortly. The API of HPX is the same for local and distributed operations, thus enabling developers to maintain a high-level view of the application. HPX takes care of the load balancing by the use of an active global address space (AGAS) to enable object migration. It prefers moving work to the data instead of the other way around since operations can usually be encoded in fewer bytes than the data they are operating upon. For the communication between nodes, message driving computation is preferred over message passing.

3.2 Vc - A High Level Library for Data-Level Parallelism

To use the available hardware to its fullest potential, we have to utilize data-level parallelism. Most CPUs support SIMD instructions (single instruction, multiple data) to apply an operation to multiple data values at once. However, using SIMD instructions and registers requires developers to either trust the auto-vectorization of the compiler or to use the special instructions. Both ways have disadvantages.

As shown in [KL12], compilers cannot simply auto-vectorize purely scalar code. While all major C++ compilers can auto-vectorize loops, the success of these automatic vectorizations depends on specific loop layouts, employed data structures and the dependencies between data accesses. The speed-ups obtained are therefore dependent on the compiler and usually much smaller than what we would get by explicit vectorization [PCM+16]. Another disadvantage is that even small changes to the scalar code might break auto-vectorization and cause slowdowns.

To consistently and adequately utilize SIMD capabilities we have to add explicit vectorization to our applications. A usual way of doing this is using the AVX Intrinsics instructions. However, the explicit use of these instructions comes with its own set of problems: SIMD intrinsic operations are more verbose than their scalar counterparts and use prefix notation. This verbosity makes it harder to port scalar code to intrinsic instructions because we have to replace every instruction. Even

worse, the code is not portable to other SIMD architectures. With the introduction of AVX512, we would have to manually convert all the instructions to the new register lengths and instruction names.

The Vc library [KL12; Kre+15] aims to provide a link between both solutions, by easing explicit vectorization of C++ code. The data parallelism is added with the type system, with Vc types explicitly stating that operations will operate on multiple values.

Furthermore, Vc can be compiled for multiple SIMD instructions sets: AVX, AVX2, AVX-512, SSE4.2 or just for plain scalar code. To port the code we just have to compile for the target architecture without changing the code itself. Operations in Vc are written similarly to scalar operations. This not only makes it a bit easier to write said Vc code, but it also allows us to write template functions which can be instanced as a vectorized version or a scalar version at compile time. We can use this feature to call the same function template on the GPU as on the CPU. We use the vectorized instance of the function on the CPU to use its SIMD capabilities, and can use the scalar instance on the GPU.

Alternative libraries to Vc are Boost.SIMD [EFGL14], or the vector classes of Agner Fog [Fog13]. A more comprehensive overview over multiple SIMD libraries, including Vc, and a comparison between them can be found in [PCM+16].

4 Octo-Tiger on Intel Knights Landing

The Intel Knights Landing (KNL) processor is a manycore processor that provides a similar double-precision as a modern GPU [Sod15]. It achieves this performance by supporting AVX512 instructions and containing up to 72 cores. Unlike a GPU it still has the advantage of being a processor, which simplifies programming. For example, we can still run programs which are compiled for other CPUs, and the caches are larger than the ones on a typical GPU. However, the cores of the Knights Landing Processor run a lower clock rate than the ones of a standard processor. Also, while the caches are larger than the one on a GPU, they are still smaller than the caches of other processors. These two attributes make the Knights Landing processor more punishing to scalar bottlenecks, and cache inefficient code. While these attributes are undesirable from a practical standpoint, they are also emphasizing parts of an application that need to be fixed. While these fixes usually yield the most significant benefit on a KNL processor, they are also speeding up applications on other processors [BCD+16].

In prior work, Octo-Tiger has already been ported to the KNL platform and has successfully been run on 655,520 Knights Landing cores in parallel. In this work, we focus on the node-level performance of Octo-Tiger. We try to improve it by optimizing the most compute intensive parts of Octo-Tiger, the computation of the cell to cell interactions in the Fast Multipole Method. Our goal is to achieve a more cache and SIMD friendly FMM algorithm.

To do this, we first take a look how the seven FMM interaction kernels of Octo-Tiger are vectorized. An overview of these kernels and their scalar algorithm can be found in table 2.1. Afterward, we present an alternative implementation that no longer relies on an interaction list. Instead, it uses a stencil-based approach.

4.1 Vectorized FMM Interaction Kernels Using an Interaction List

During an initial profiler run for a small scenario using a Haswell processor, we find that Octo-Tiger spends 67.5% of the runtime calculating the FMM cell to cell interactions with the FMM interaction kernels.

Essentially all the FMM kernels of table 2.1 follow the same pattern. (1) We find a pair of interacting cells using the interaction list. (2) We load the values for the 20 multipole coefficients and the center of masses for those two cells (3) We calculate the cell to cell interactions using an implementation of the equations 2.16 - 2.19 or one of the specializations. (4) We add the result to the combined interactions for the two cells.

However, vectorizing the algorithm is challenging. We want to process multiple pairs of interacting cells at once. Using AVX512, we can handle eight pairs at the same time if we load the values into vector variables of width eight. During step (1) we load not only one pair from the interaction list,

but eight pairs. In step (2) we now have to gather the coefficients into vector variables. Instead of having a variable for each of the 20 multipole coefficients of a cell, we now have vector variables with eight multipole coefficients, one for each cell. Using Vc, we can write the code for the calculation of the Taylor coefficients in step (3) as we would write it for a scalar code. However, the result of these computations is another vector variable. To add the results back to the Taylor coefficients of the cells in step (4), we need to scatter the vector variable back onto multiple scalar Taylor coefficients.

The indirect loading of the interaction partner with the interaction list and the subsequent gathering of the coefficients in the vector registers is a considerable performance hit. While gathering the scalar coefficients into the vector register, we have to potentially touch many different cachelines. These cachelines are not getting fully used, meaning we load a lot of data we do not actually need. In the worst case, we load eight different cachelines, each 64 Bytes wide, yet do not use more than the one value we wanted to load per cacheline. In this case, we would load 512 Bytes but only use 64. These issues offset the performance gain that we could theoretically get by using AVX512 instead of AVX2.

4.2 Vectorized FMM Interaction Kernels using a Stencil-based Approach

To improve the node level performance we would like to have a more linear memory access pattern. We want to avoid these explicit gather steps and fully use the cached data.

We can achieve this by moving to a different data structure. In the initial version of Octo-Tiger, all of the coefficients and results are stored in an Array of Structs (AoS) data structure. That is why we need to gather all of the coefficients with scalar operations first since for each cell all 20 multipole coefficients are consecutive in memory. We need to have the first coefficient, of all eight cells consecutive in memory in order to load them with a single instruction. The same needs to hold true for the second coefficient and so on. Effectively, instead of an AoS data structure, we need a Struct of Array (SoA) data structure. Within the SoA data structure, all of the coefficients are ordered by their component index first, and the cell index is less important. Thus, we could load eight coefficients without explicitly having to gather them. This only works if the cells are in the right order. If we want to load the coefficients for eight pairs of the interaction list, it is unlikely that the coefficients for these exact eight interaction pairs are consecutive. So we still would have to gather the right cells.

We can solve this by replacing the interaction list with a stencil-based approach. We first introduced this approach for Octo-Tiger in our prior work [PDM+18]. The opening criteria 2.11 implies a stencil around each cell, which includes all cells it directly interacts with. This stencil always has the same size. Instead of loading eight interaction pairs from the interaction list to make use of the vectorization, we now can process this stencil for eight cells. If these eight cells are in order, the interaction partners indicated by the stencil will be, too.

By combining the SoA data structure with this stencil-based approach of identifying interaction partners, we can completely remove the gather and scatter steps. We also eliminated the indirect addressing the interaction list forced us to use. We can also remove the memory intensive interaction list itself. For each node in the octree, we process this stencil for 512 cells. This allows us to scale

this approach to handle more than eight interaction pairs simultaneously. Given enough parallel capabilities, we can process the stencil for all 512 elements, and the data we load to this will be in order.

4.3 Implementation of the Stencil FMM Interaction Kernels

Our goal is to create new high-performance stencil FMM interaction kernels to replace the interaction list FMM kernels. These new stencil kernels are using the SoA data structure, instead of the original AoS data structure that the rest of Octo-Tiger uses. Therefore, before we calculate the FMM cell to cell interactions using the kernels, we have to convert the data from the AoS to the SoA data structure. We do this once per cell before we start calculating the FMM cell to cell interactions. The interaction list FMM kernels are doing something similar when they are gathering the coefficients into the vector registers, however, they do this once per cell to cell interaction instead.

On each octree node, the SoA data structure holds all required coefficients in a three-dimensional grid. For example, these coefficients include the multipole moments $M_q^{(0)}$, $M_{q,m}^{(1)}$,$M_{q,mn}^{(2)}$ and $M_{q,mnp}^{(3)}$ for each cell q. This does not only include the coefficients of all 512 cells within the subgrid of the octree node, but also the ones of the neighboring octree nodes on the same refinement level, since we are interacting with their cells as well. In the SoA conversion step on each node, we have to fill a grid the size of $24 \cdot 24 \cdot 24 = 13824$ cells. The middle part of this grid consists of the $8 \cdot 8 \cdot 8$ subgrid of the current octree node. There are two additional cases to consider while filling the SoA data structure of a node. First, a neighbor could be of the wrong type. For example, monopole cells only contain the multipole moment of the zeroth order $M_q^{(0)}$, and none of any higher order. Second, a neighbor on the same refinement level could simply not exist. In both cases, we fill the SoA data structure with zeros at these positions. This would yield the right result because the cell to cell interactions would also result in zeros. However, to avoid these useless computations we employ an early exit strategy to skip the cell to cell interactions in these cases.

Once the SoA data structures are ready on an octree node, calculating the cell to cell interactions is very similar to the interaction list FMM kernels method. We are still implementing the same equations, however the difference is how we identify interacting cells and how we load the data.

For each cell q within the 8^3 subgrid of a node, we interact with all other cells on the 24^3 sized SoA coefficient grid that are in range of the stencil. The stencil itself is defined by the opening criteria equation 2.11. A picture of this stencil for Octo-Tiger's default parameters can be seen in figure 2.2. For example, a scalar version of the stencil FMM kernel for multipole-multipole interaction can be seen in algorithm 6. The stencil-based equivalent to the interaction list kernel is seen in algorithm 2.

We can now easily vectorize this stencil FMM kernel for AVX512. Again, we want to process eight cell to cell interactions, instead of just a one. To accomplish this, we iterate not over one cell in the first loop, but over eight cells. This means each variable that previously only held one coefficient must now hold eight coefficients. We accomplish this by using Vc variables. If the eight cells we loaded are aligned in memory, then their respective eight interacting cells l in the inner loop will be as well. Loading a coefficient for the first of the eight interacting cells will give us the same coefficient of the other cells within the cacheline since they are consecutive in memory, because of

the SoA data structure. This means we only have to load a single cacheline if we want to load a coefficient of all eight cells. Instead of the eight cachelines, we require using the interaction list FMM kernel.

Data: All multipoles and center of masses of the cells within the subgrid and its neighbor subgrids in the form of SoA data structures
Result: All combined cell to cell interactions for each cell within the subgrid in the form of a SoA data structure
for *cell* $q \leftarrow 0$ **to** 512 **by** 1 **do**
  Step 0: Load SoA multipole moments and the center of masses for cell q;
  forall *cells l within the stencil of q* **do**
    Step 1: Load SoA multipole moments and the center of masses for cell l;
    Step 2: Compute Green's gradients using the center of masses with equation 2.12;
    Step 3: Compute cell to cell interaction results $A^{(0)}_{ql}$,$A^{(1)}_{ql,m}$,$A^{(2)}_{ql,mn}$,$A^{(3)}_{ql,mnp}$between l and q using the equations 2.16-2.19;
    Step 4: Add cell to cell interactions to combined result for this cell;
    Step 5: Compute cell to cell interaction corrective term $B^{(1)}_{q,m}$ using the equation 2.20;
    Step 6: Add to corrective term $B^{(1)}_{q,m}$ to the combined corrective term.
  end
end

Algorithm 6: This algorithm shows a scalar version of the stencil multipole-multipole (m2m) FMM interaction kernel.

4.4 Further optimizations

To further improve the runtime of the FMM implementation on all platforms we included various other optimizations:

(1) We combined the multipole-multipole kernel with the multipole-monopole kernel. By structuring the input data the right way and doing a couple of masking operations within the kernel itself, we ensure the right result. This improved the cache usage since both kernels essentially require almost the same data. Staying true to our notation of the kernels, we are calling this kernel the m2mp kernel, since it computes interactions for both multipole and monopole partners. Since there is a version with the angular corrections and one without, it brings our overall number of stencil FMM kernels down to five.

(2) We removed the explicit calculation of all the values for the gradients of the Green's function 2.12. We sorted the calculations of the m2m kernel in a way that we only need one of the coefficients at a time and thus have no need to store them all. This reduced the usage of CPU registers and sped up the execution of this kernel.

(3) We moved all SoA data structures into thread local memory. Instead of allocating this memory per subgrid, we now allocate it once per HPX worker thread. This means we have one SoA subgrid with all the multipole moments and the center of masses per worker thread. This subgrid will

be filled by the worker thread with the SoA conversions with the data of the node it is currently working on. This optimization brought us back to nearly identical memory requirements as the initial code, while still avoiding costly memory allocations.

We also put other, more conventional, optimizations into place, like loop unrolling and memory blocking. However, the three optimizations above are more specific to Octo-Tiger and thus important to know for other people working with the code.

4.5 Overview over the Stencil FMM Interaction Kernels

We initially created a stencil-based FMM interaction kernel, for each of the seven interaction list FMM interaction kernels. However, due to the merger between the m2m and the m2p kernel we end up with five stencil FMM interaction kernels:

Stencil Kernel
p2p stencil FMM interaction kernel
p2m stencil FMM interaction kernel
m2mp stencil FMM interaction kernel
p2m non-rho stencil FMM interaction kernel
m2mp non-rho stencil FMM interaction kernel

5 Octo-Tiger's FMM in a Heterogeneous Environment

Supercomputers that include one or more GPUs per compute node in addition to the standard processor, have become more commonplace recently. This can be seen in the top 500 list of supercomputers. An example for such a supercomputer would be Piz Daint, which includes an NVIDIA Tesla P100 GPU and a Xeon E5-2690v3 CPU on each compute node. In such heterogeneous environments, Octo-Tiger's performance can be further improved by utilizing the GPU. More specifically we want to use the GPU as a coprocessor for computing some of the stencil FMM interaction kernels. We are targeting NVIDIA GPUs and are therefore using Cuda to use their devices.

In this chapter, we first explain a basic version of how to launch the FMM kernels on a NVIDIA GPU. Afterward, we discuss the necessary changes to make Octo-Tiger use both the CPU and the GPU at the same time.

5.1 Basic Version

To make an FMM kernel executable on a Cuda device, we only need to slightly modify the original scalar stencil FMM kernel. We already saw an exemplary scalar version, which is the multipole-multipole stencil FMM kernel in algorithm 6. Here we iterated over all 512 cells of the octree node, calculating all cell to cell interactions within the stencil for each. Instead of iterating over the 512 cells, we now launch 512 threads on the Cuda device for one kernel launch. Each of these threads takes care of all cell to cell interactions of its respective cell. However, 512 threads are not enough to fully use a GPU. We need to be able to execute multiple FMM kernels concurrently on the GPU. Cuda GPUs are capable of executing a number of kernels concurrently. How many depends on the GPU itself, but it usually varies between 16 and 128 for modern NVIDIA graphic cards. To do this, we need to use Cuda streams. We can launch each FMM kernel execution in a Cuda stream and it will be executed on the Cuda device. If other FMM kernels are launched in different streams, they will be executed in parallel on the GPU if the device has the capability to do so. Otherwise, the kernel will be executed as soon as the resources on the GPU are available. We can create a Cuda stream for each octree node, in which we launch the FMM kernel for this node.

However, to launch an FMM kernel on the GPU involves more steps than doing so on the CPU. To launch a stencil FMM kernel on the CPU, we only have to do two steps. (1) Convert the AoS data of the octree node and its neighbors into a SoA grid. (2) Launch stencil FMM kernel on the CPU.

On the GPU even the most basic version requires more steps. To launch an FMM kernel on a GPU we have to: (1) Again, convert the AoS input data into the SoA grid. (2) Allocate all required buffers for the kernel input and output data on the Cuda device. (3) Move the SoA input data into

the respective Cuda device buffers. (4) Launch the FMM kernel on the Cuda device and wait for it to finish. (5) Move the resulting data back from the device buffers to the CPU host-side buffers. (6) Deallocate the the device buffers.

If we are doing this on each octree node, we fully outsourced the computation of the stencil FMM interaction kernels to the GPU. Since we launch each FMM kernel in a different stream, we will fill up the GPU with multiple concurrent kernel executions, each consisting of 512 threads itself. However, this way of launching the FMM kernels on the GPU comes with two major disadvantages. (1) We continuously block the CPU by waiting on either the copying or the kernel execution to finish. For this timeframe, the HPX worker thread just waits for the GPU to finish its operation and cannot do anything else. (2) Even small Octo-Tiger scenarios require more than 16 GB of memory, making it impossible to store the complete scenario in the GPU memory. Instead, we have to allocate and deallocate the device buffers we need for each kernel launch.

These two disadvantages result in a considerable overhead for launching FMM kernels on the GPU, and prevent the CPU from doing useful work at the same time. To truly use the GPU as a coprocessor to execute some of the FMM kernels, while leaving the CPU free to do other work, we have to solve the problems causing these two disadvantages.

5.2 Hetereogenous Version

We will show how we can solve the problems mentioned in the previous section. These problems include: (1) Constant allocation and deallocation of Cuda device buffers. (2) Host CPU has to wait for memory copies to and from the device to finish. (3) Host CPU cannot do any other work while waiting for the kernel execution on the GPU to finish.

The first problem we need to solve is the allocation and deallocation of the Cuda device buffers. We can do this by moving the host-side handles of the device buffers into thread local memory. Instead of allocating the device buffers once per kernel execution, we now only allocate them once per HPX worker thread. This means we can persistently reuse the device buffers for each worker thread, and consequently only have to allocate them once during the startup of Octo-Tiger. We call the thread local struct containing the handles for the device buffers the `kernel device enviroment` from now on. Similarly, instead of having one Cuda stream per octree node, it is sufficient to have one thread local stream per HPX worker thread.

The second problem to solve is how to copy data to and from the Cuda device without blocking the CPU. Cuda supports asynchronous memory operations, doing precisely this. Using these, we can directly queue a memory copy in a Cuda stream, without blocking the host CPU. Since operations in a Cuda stream are guaranteed to be in order, we do not need to worry when this memory copy is going to finish. If we queue the launch of the kernel after the memory copy in the same stream, the copy operation will be finished before the kernel execution starts on the device. However, there is one requirement we have to fulfill to use these asynchronous copies. The host side memory of this copy must be non-pageable memory. This memory is also often called pinned memory. Since the allocation of such memory is more expensive than the allocation of normal memory, we employ the same trick as we do regarding the handles of Cuda device buffers. We allocate the pinned memory only once per HPX worker thread, again using thread local memory. We call the struct containing all SoA thread local memory of the host side the `kernel staging area` from now on. Since we

need to convert the AoS input data into an SoA grid anyway, we simply use the pinned SoA grids in the `kernel staging area` as the target for these conversions. Therefore we avoid costly allocations, while still supporting asynchronous memory copy to and from the Cuda device.

The third problem is the most complicated one to solve. So far, the CPU still has to wait for the completion of the FMM kernel execution before it can continue since it needs its results. Now HPX comes into play. HPX Compute allows us the get a future from a Cuda stream. This future will enter the ready state once the stream has finished all operations that were queued in this stream before the creation of the future. HPX internally realizes this by utilizing Cuda callback functions. If we call the `get` method of this HPX future and the future is not ready yet, it causes the current HPX worker thread to jump away. It will then work on another task, given by the HPX scheduler. This task could, for example, be to calculate the cell to cell interactions of another node in the octree. Once the HPX future becomes ready, the HPX scheduler gets notified with the Cuda callback function and will schedule a worker thread (not necessarily the same one) to continue where the other worker thread left off. This time the results are ready, and the execution can proceed normally. This way the HPX worker thread does not have to explicitly wait for the execution on the GPU to finish, and can work on other tasks instead.

However, the amount non-FMM related tasks is particularly limited as long as we are calculating the gravity. Therefore, it is still likely that we starve the CPU of work if we queue every FMM kernel an HPX worker thread has to process, on the GPU. We need a load balancing solution, which only launches an FMM kernel on the GPU if the GPU is not yet out of resources and can support another concurrent kernel launch. Using Cuda, we can query whether a Cuda stream is currently busy or not. We can use this to implement a load balancer. An HPX worker thread will only launch a kernel on the GPU if its associated Cuda stream is not busy. Otherwise, we launch the kernel on the CPU. While this not accurately represents how busy the GPU currently is since we can have multiple worker threads, each with an associated stream, it is a close enough, yet simple approximation. Using all of the above concepts, we can use the Cuda stream of each HPX worker thread as a small coprocessor. The HPX worker thread can offload some of the stencil FMM kernel launches to the GPU without explicitly having to wait for any response from the GPU before continuing with other tasks. Meanwhile, the GPU can execute the stencil FMM kernel and copy the results back to the CPU host-side memory.

The last thing to consider is the number of Cuda streams we need to fully utilize a GPU. With our current approach, we would have eight Cuda streams if we launch Octo-Tiger with eight HPX worker threads. This would not be enough streams. To allow multiple streams per worker thread, we introduce a thread local scheduler for each HPX worker thread. This scheduler can have multiple launch slots. Each slot has an associated Cuda stream, a `kernel staging area` and a `kernel device enviroment`. If a worker thread requests a launch slot from its scheduler to execute an FMM kernel on the GPU, the scheduler will check whether there is a launch slot with a free Cuda stream. If there is, it will return its ID. This allows the worker thread access to the stream, the `kernel staging area`, and the `kernel device enviroment` of this slot. If there is no slot available, the scheduler will simply return -1, indicating that worker thread should launch the FMM kernel on the CPU. Instead of just one, an HPX worker thread can now have multiple streams to act as coprocessors for executing the stencil FMM kernels.

We can now write the algorithm to launch an FMM kernel, as shown in algorithm 7.

Step 1: i ← query scheduler for launch slot ;
if *i == -1* **then**
  Step 2: No Cuda slot available. Convert AoS into Soa data and launch FMM kernel on CPU ;
else
  Step 3: Get pinned `kernel staging area` for slot *i* from scheduler;
  Step 4: Convert AoS data of the octree node an its neighbors into SoA data, stored in the pinned staging area;
  Step 5: Get `kernel device environment` for slot *i* from scheduler. This environment contains the handles for the Cuda device buffers;
  Step 6: Get Cuda stream *s* for slot *i* from scheduler;
  Step 7: Queue asynchronous copy of pinned SoA input data to Cuda device buffers in stream *s*;
  Step 8: Queue FMM kernel execution in stream *s*;
  Step 9: Queue asynchronous asynchrous copy of results back to host in strean *s*;
  Step 10: Create HPX future *f* which will be ready once the stream *s* has finished its three queued operations;
  Step 11: Call `f.get()`. This allows the current HPX worker thread to jump away to work on other tasks, until the future *f* becomes ready. Once the future becomes ready a worker thread will return to this point;
end

Algorithm 7: Algorithm to launch a stencil FMM interaction kernel on a heterogeneous CPU/GPU system.

5.3 Scheduler Parameters

The number of launch slots each scheduler has is an important knob to turn. On the one hand, if we have too many of these slots, we will exceed the number of parallel kernel executions the GPU supports, while simultaneously starving the CPU of work. On the other hand, if we use too few launch slots, we waste potential resources on the GPU.

We introduce the command line parameter `Cuda_streams_per_locality` to control how many overall Cuda streams should be used by Octo-Tiger on one compute node. These streams will be evenly distributed over the schedulers associated with the individual HPX worker threads. Each scheduler can end up with zero or more launch slots. If it has zero slots, it will simply always use the CPU to compute the FMM interaction kernels.

We also introduce a second command line parameter `Cuda_streams_per_gpu`. It specifies how many streams should be used per GPU. If the other parameter exceeds that number, the remaining streams will be placed onto another GPU if one is available on the current hardware platform. For example, if we launch Octo-Tiger with eight worker threads, 16 Cuda streams per compute node and 16 Cuda streams per GPU, we would solely use one GPU and each of the eight schedulers would have two launch slots. If we launch Octo-Tiger with 24 Cuda streams per compute node and 16 Cuda stream per GPU, we would place 16 streams on the first GPU and eight streams on the second one. With 48 streams per compute node, and still 16 streams per GPU, we would use three GPUs, each with 16 streams. This parameter effectively adds multi GPU support.

6 Results

In this chapter we take a look at the runtime data Octo-Tiger achieves with different platforms, configurations and scenarios. First we look at the scenarios, and the toolchain we are going to use. Next, we compare the stencil based FMM interaction kernels we introduced in chapter 4 with the FMM kernels using the interaction list. Finally, we will show how well Octo-Tiger performs using CPU and GPU at the same time.

6.1 Scenarios

We use multiple scenarios to collect runtime data with Octo-Tiger. All of these scenarios simulate a moving star, however, they use different sizes for the moving star (`Xscale`). Additionally we increased the maximum depth of the octree (`Max_level`) and the amount of steps after which we stop the simulation (`Stopstep`).

1. Scenario:

```
-Disableoutput -Problem=moving_star -Max_level=6 -Odt=0.3 -Stoptime=0.2 -Xscale=20.0
    -Omega=0.1 -Stopstep=20
```

2. Scenario:

```
-Disableoutput -Problem=moving_star -Max_level=7 -Odt=0.3 -Stoptime=0.2 -Xscale=32
    -Omega=0.1 -Stopstep=20
```

3. Scenario:

```
-Disableoutput -Problem=moving_star -Max_level=7 -Odt=0.3 -Stoptime=0.2 -Xscale=20
    -Omega=0.1 -Stopstep=20
```

4. Scenario:

```
-Disableoutput -Problem=moving_star -Max_level=8 -Odt=0.3 -Stoptime=0.2 -Xscale=32
    -Stopstep=6
```

5. Scenario:

```
-Disableoutput -Problem=moving_star -Max_level=9 -Odt=0.3 -Stoptime=0.2 -Xscale=32
    -Omega=0.1 -Stopstep=6
```

We use scenario 1 for the comparison between the stencil FMM kernels and the interaction list FMM kernels. Scenario 2, 3 and 4 require longer running times than scenario 1. We use these scenarios to compare the runtime of Octo-Tiger when we only use the CPU, to the runtime we get when using the CPU and all available GPUs. Scenario 5 is too large to fit into the memory of a single platform. We use it to test the distributed scaling with multiple compute nodes on Piz Daint.

6.2 Toolchain

We always used the same toolchain for compiling Octo-Tiger. We compile each of the dependencies from source. The build scripts to accomplish this can be found on Github [1]. After we have compiled Clang, we use it to compile the rest in the same order as the appear in the following table.

Software	Version/Commit
Clang	7.0.0, Commit 23b713ddc4
Boost	1.65
Jemalloc	5.0.1
Hwloc	1.11.7
Vc	Commit 157026d3
HPX	1.1.0
Octotiger	Commit 0f4af8a

6.3 Platforms

We use multiple platforms to collect the runtime data of the following tests. We have a set of platforms for performance comparison between the stencil FMM kernels and the interaction list FMM kernels, each containing only a CPU. We have another set of platforms for the tests regarding the Cuda implementation.

6.3.1 CPU-only Platforms

These are the platforms we use for the performance comparison:

- **Intel Core i7-7500U** with 2.70 GHz and 2 Cores
- **Intel Core i7-4770K** with 3.50GHz and 4 Cores
- **Intel Xeon CPU E5-2660 v3** with 2.60GHz and 20 Cores
- **Intel Xeon Gold 5120** with 2.2GHz and 28 Cores
- **Intel Xeon Gold 6148** with 2.4GHz and 40 Cores
- **Intel Xeon Phi 7210** with 1.3GHz and 68 Cores

The most interesting of these platforms is the Knights Landing Xeon Phi 7210, since it originally motivated the creation of the stencil FFM kernels. We included two more platforms that are capable of AVX512, the Intel Xeon Gold 6148 and the Intel Xeon Gold 5120. While these two CPUs look similar, the have one key difference. The Xeon Gold 6148 has two FMA units per core, and the Xeon Gold 5120 only one, consequently the peak performance of the Gold 6148 is much higher.

[1] https://github.com/G-071/OctotigerClangScripts.git

We also added three platforms with CPUs that only support AVX2. The first is a consumer grade Kaby Lake CPU, followed by an consumer grade Haswell CPU. To test a Haswell Platform with more cores we added the Intel Xeon CPU E5-2660.

6.3.2 CPU/GPU Platforms

We further use three platforms to test the Cuda implementation. Each of these three platforms contains a powerful CPU, as well as one or more NVIDIA GPUs. For each of the follwing tests with these platforms, we always use the CPUs to their fullest potential by launching as many HPX worker threads as we have CPU cores.

- **argon-gtx** Contains an Intel Xeon Gold 5120 with 2.20 GHz and 28 cores, combined with eight NVIDIA Geforce GTX 1080 TI GPUs.
- **argon-tesla** Uses an Intel Xeon Silver 4116 CPU with 2.10GHz and 24 cores. This platform also contains two NVIDIA Tesla P100 GPUs.
- **Piz Daint (Single Compute Node)** Contains an Intel Xeon E5-2690 v3 with 2.60 GHz and 12 cores, as well as an NVIDIA Tesla P100 GPU.

6.4 Performance Comparison between the Stencil FMM Kernels and the Interaction List FMM Kernels

Here, we are comparing the runtime we can achieve with the stencil FMM kernels with the runtime we can achieve using the interaction list FMM kernels. We use a series of tests and platforms to achieve this. In Test 1 we will take a look at overall single core performance using different kernel combinations. In Test 2 we slowly increase the number of CPU cores to examine the node-level scaling. In Test 3 we look again at the performance of the different kernels, but this time using all available ressources on each platform. We conclude this section with a small summary.

6.4.1 Method for Collecting Runtime Data

To compare the runtime between the stencil FMM kernels and the old interaction list FMM methods, we use command line arguments for Octo-Tiger to switch between them. By doing this, we ensure, that we use the same toolchain for both versions. We cannot merely use a commit at the beginning of this work for comparison since it would work neither with HPX 1.1.0, nor be capable of being compiled with Clang. The other way around this holds true as well, since a downgrade to HPX 1.0.0 would involve breaking changes that would influence the performance comparison. For example, on some machines, the switch to Clang alone brought a performance boost of up to 10% for both versions.

Since we are using the same version of Octo-Tiger no matter what kind of FMM kernels we run, we are still calling the methods to convert the AoS data to SoA data. However, for the comparison between the kernels itself, this is rather useful since the conversion is meant to be a temporary measure anyway and will be removed once more of Octo-Tiger gets converted to the SoA data

Method	Fraction of runtime
m2mp stencil kernel with angular corrections	19.51%
p2p stencil kernel	16.06%
reconstruct	14.50%
m2mp stencil kernel without angular corrections	10.24%
p2m stencil kernel with angular corrections	3.52%
p2m stencil kernel without angular corrections	1.42%
Overall percentage of time spent with the FMM	**50.75%**

Table 6.1: Most compute intensive parts of Octo-Tiger when using the stencil FMM kernels on a Haswell Processor.

structure. In our analysis of the hotspots of Octo-Tiger when using the stencil kernels, the conversion methods do not show up at all, as can be seen in the table 6.1. While the overall time we spent with the computation of the FFM algorithm is about 51% (including the conversion methods), the runtime is being used in the methods calculating the interactions. The only method in this list of hotspots, which has nothing to do with the FMM part of Octo-Tiger, is `reconstruct` which is related to the creation of the octree.

6.4.2 Comparison Test 1: Measuring Single Core Speedup

In our first test we compare the total runtime of Octo-Tiger using either the stencil FMM kernels, or the old interaction list FMM kernels. Additionally, we show the speedup we achieve by using the stencil-based kernels. We use scenario one to collect this data. The results can be seen in table 6.2.

Platform (one core used)	Using interaction list	Using stencil	Speedup
Intel Core i7-7500U CPU with 2.70 GHz	1076s	821s	1.31
Intel Core i7-4770K with 3.50GHz	1127s	916s	1.23
Intel Xeon CPU E5-2660 v3 with 2.60GHz	1568s	1308s	1.20
Intel Xeon Gold 5120 with 2.2GHz	2114s	815s	2.59
Intel Xeon Gold 6148 with 2.4GHz	1687s	643s	2.62
Intel Xeon Phi 7210 with 1.3GHz	8317s	3463s	2.40

Table 6.2: Per core performance using the refactored Octo-Tiger.

We achieve a speedup on all platforms. The ones supporting AVX512 benefit the most from using the stencil FMM kernels.

In table 6.3 we can see how Octo-Tiger performs when we are mixing the interaction list FMM kernels with the stencil FMM kernels. For simplicity, we do not differentiate between the kernels including angular corrections, and the respective versions that do not. We always consider both at the same time. In the upper table, we can see the raw runtimes. For the runtimes in column two, we use only the interaction list FMM kernels. In the second column, we exchange the interaction list

multipole-multipole kernel, as well as the multipole-monopole kernel, with their respective version of stencil FMM kernels (m2mp kernel). For the remaining kernels, we still use the interaction list kernels. This allows us to see the speedup we get by using the stencil m2mp kernel. Similarly, we can calculate the speedup that we can achieve by using the stencil monopole-monopole kernel (p2p). To do this calculation, we use the runtimes in column three and compare them with the interaction list runtimes in column two.

The most beneficial stencil kernel is the combined multipole kernel (m2mp), followed by the monopole-monopole stencil kernel (p2p). The monopole-multipole (p2m) stencil kernel yields the worst performance of the three. Multipole to monopole interactions are comparatively rare, so we hit the early exit condition in this kernel very often. As we can see in table 6.1 the p2m interactions have the least impact on the overall performance. In the lower part of the table 6.3 we can directly see the speedups each stencil kernel accomplishes on its own over the runtime when only using the interaction list kernels. Each of these speedups is in relation to the runtimes when we are using only the interaction list FMM kernel. Therefore, they do not add up to the overall speedup. We can calculate the overall speedup using column two and six in the upper table 6.3. This results in the same values as in table 6.2.

Platform (one core used)	**All interaction list**	**m2mp**	**p2p**	**p2m**	**All stencil**
Intel Core i7-7500U CPU	1076s	862s	1049s	1073s	821s
Intel Core i7-4770K	1127s	943s	1101s	1128s	916s
Intel Xeon CPU E5-2660 v3	1568s	1359s	1482s	1601s	1308s
Intel Xeon Gold 5120	2114s	995s	1939s	2107s	815s
Intel Xeon Gold 6148	1687s	761s	1575s	1687s	643s
Intel Xeon Phi 7210	8317s	4207s	7873s	8031s	3463s

Platform	**Speedup stencil m2mp**	**Speedup stencil p2p**	**Speedup stencil p2m**
Intel Core i7-7500U CPU	1.24	1.03	1.00
Intel Core i7-4770K	1.19	1.02	0.999
Intel Xeon CPU E5-2660 v3	1,14	1.09	0.98
Intel Xeon Gold 5120	2.12	1.09	1.00
Intel Xeon Gold 6148	2.21	1.07	1.00
Intel Xeon Phi 7210	1.97	1.06	1.04

Table 6.3: Runtime for various kernel combination. Either we use only the interaction list kernels (left), we use one of the stencil kernels and use the interaction kernels for the rest (middle), or only the stencil-based kernels (right).

6.4.3 Comparison Test 2: Measuring Node-Level Scaling

In the last test, we showed that the stencil FMM interaction kernels are faster on all platforms if we are only using one core. Now we test how well these results scale up if we are using more than one core. To do this, we slowly increase the amount of HPX worker threads and record the runtime and parallel efficiency for both the interaction list FMM kernels, and the stencil FMM kernels.

Since the i7-7500U CPU only has two cores and the i7-4770k only has four cores, we do not consider these platforms for this test. We instead focus on the other four platforms, since they include more cores, enabling us to get a clearer picture how the parallel efficiency is affected.

- **Intel Xeon CPU E5-2660 v3**: On this Haswell platform, we achieve almost exactly the same parallel efficiency with both variants of the kernel, as we can see in figure 6.1. At the maximum number of cores used, the parallel efficiency using the stencil kernels if even a bit higher than the other one. Thus, the runtime speedup stays roughly the same, even when utilizing the whole 20 cores.
- **Intel Xeon Gold 5120** The results for this platform can been seen in the top section of table 6.3. Using the Xeon Gold 5120, the parallel efficiency drops quickly on both platforms. However, while they are dropping in a similar pattern, the stencil kernels always yield a poorer parallel efficiency than the interaction FMM kernels.
- **Intel Xeon Gold 6148** This time the results can be found in the bottom section of table 6.3. We see a similar pattern with on the other Intel Xeon Gold 5120, albeit less extreme.
- **Knights Landing Xeon Phi 7210** On the Knights Landing Platform we can see a similar pattern in figure 6.4. There is still a runtime speedup using the stencil FMM kernels, however the parallel efficiency drops more quickly.

6.4.4 Comparison Test 3: Measuring Total Speedup per Platform

In the first test we only measured the runtime and the speedup for a single core. While this eliminated any scaling differences that might occur, it is also an unrealistic scenario. Usually we want to use all of the processing power we can. Therefore, we use all available CPU cores in this test run by launching as many HPX worker threads as possible.

As we can see in table 6.4, the speedup obtained when we use the whole CPU is on the Haswell platform, it even increased slightly from 1.31 to 1.33. However, on the other three platforms we can see the effect of the decreased parallel efficiency that we measured last test. While the stencil kernels still achieve a significant speedup over the interaction list kernels, this speedup has dropped noticeably.

As we can see in table 6.5, the influence the different stencil FMM kernels have on the runtime stays roughly the same if we use the complete CPU on each platform. Due to the decrease in parallel efficiency using the AVX512 platforms, we achieve a smaller speedup using the stencil kernels. However, the m2mp kernel is still the most influential one.

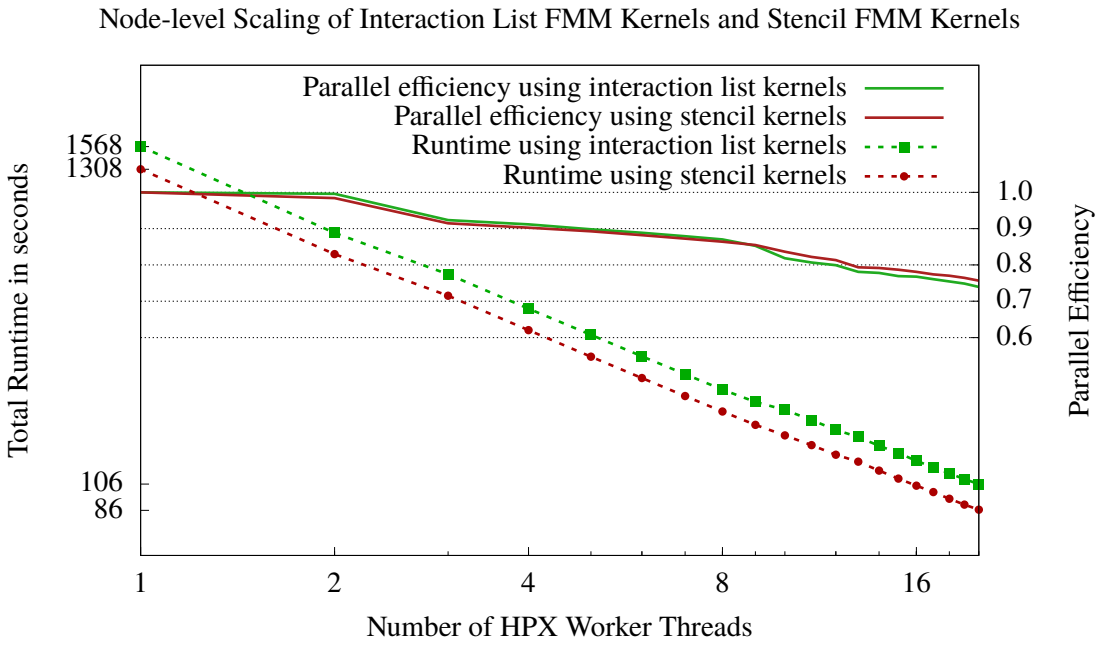

Figure 6.1: Comparison of the node-level scaling on an Intel Xeon CPU E5-2660 v3.

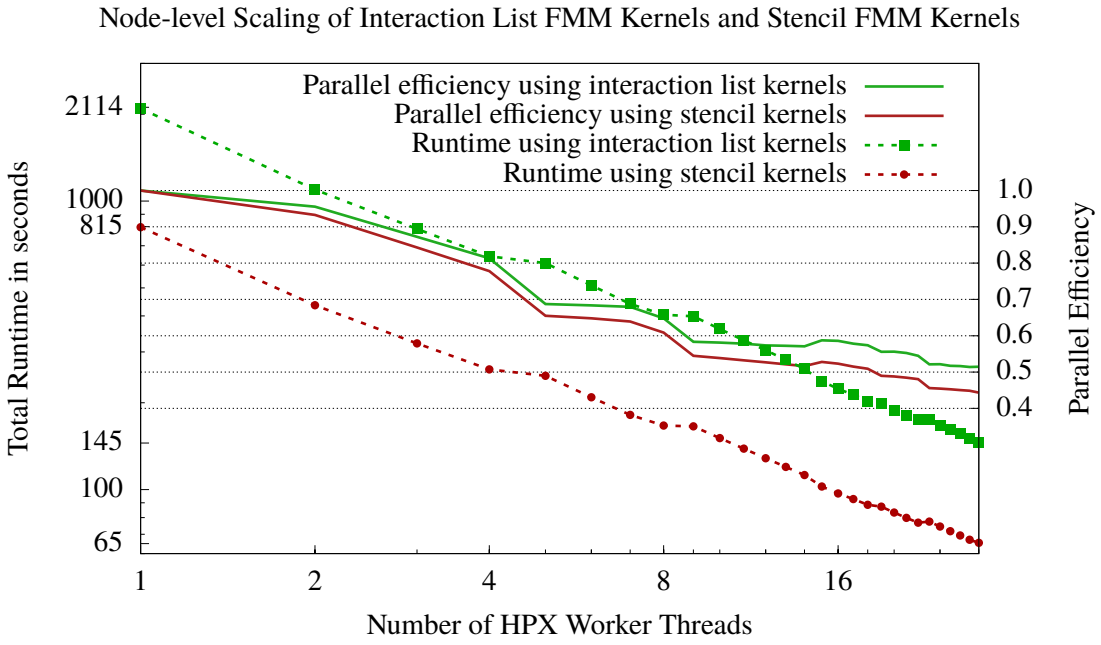

Figure 6.2: Comparison of the node-level scaling on an Intel Xeon Gold 5120.

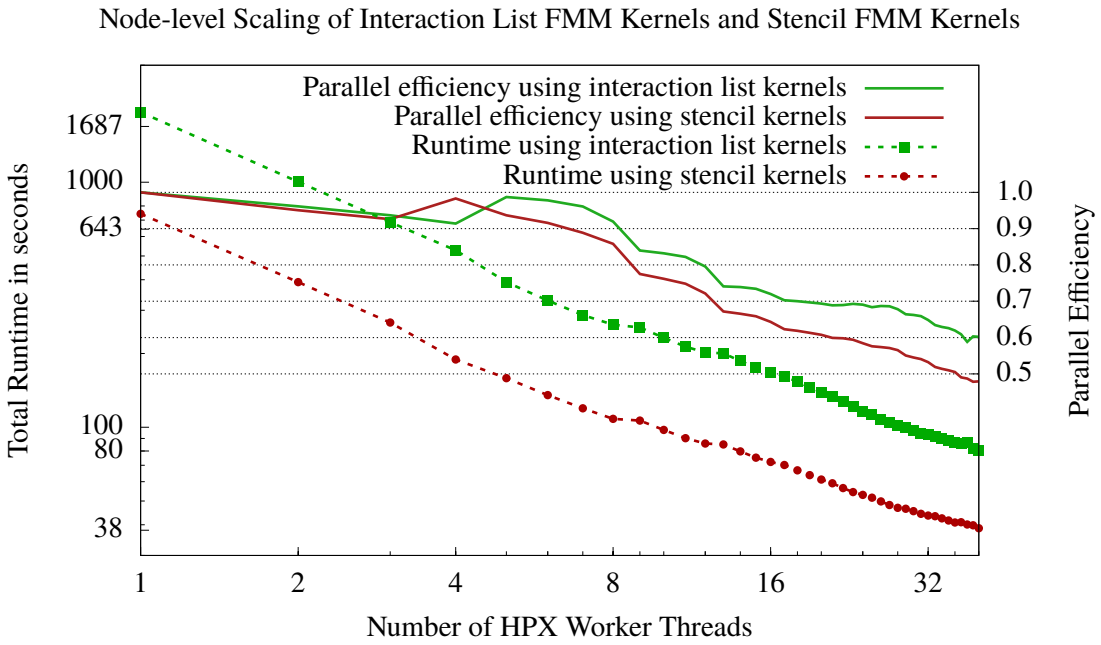

Figure 6.3: Comparison of the node-level scaling on an Intel Xeon Gold 6148.

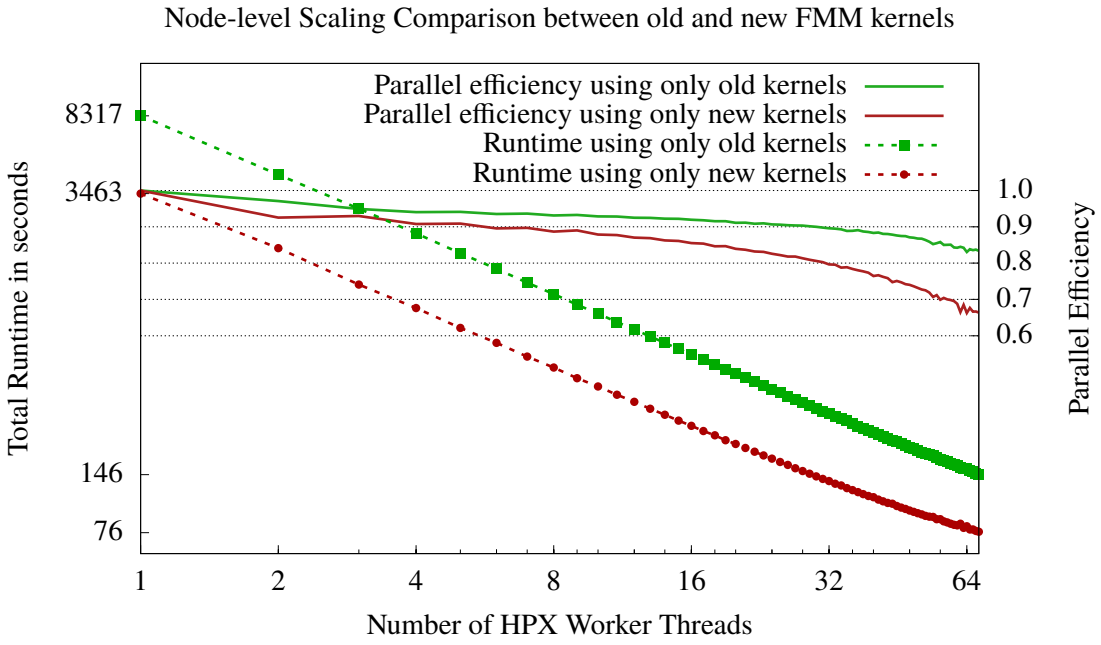

Figure 6.4: Comparison of the node-level scaling on a Xeon Phi 7210.

Platform	Using interaction list	Using stencil	Speedup
Intel Core i7-7500U CPU with 2.70 GHz	681s	513s	1.33
Intel Core i7-4770K with 3.50GHz	370s	274s	1.35
Intel Xeon CPU E5-2660 v3 with 2.60GHz	106s	86s	1.23
Intel Xeon Gold 5120 with 2.2GHz	145s	65s	2.22
Intel Xeon Gold 6148 with 2.4GHz	80s	38s	2.07
Intel Xeon Phi 7210 with 1.3GHz	146s	76s	1.90

Table 6.4: Speedup obtained by using stencil FMM kernels over the interaction list FMM kernels.

Platform	All interaction list	m2mp	p2p	p2m	All stencil
Intel Core i7-7500U CPU	681s	547s	651s	681s	513s
Intel Core i7-4770K	370s	322s	326s	369s	274s
Intel Xeon CPU E5-2660 v3	106s	93s	97s	108s	86s
Intel Xeon Gold 5120	145s	72s	139s	144s	65s
Intel Xeon Gold 6148	80s	42s	79s	79s	38s
Intel Xeon Phi 7210	146s	93s	135s	142s	76s

Platform	Speedup stencil m2mp	Speedup stencil p2p	Speedup stencil p2m
Intel Core i7-7500U CPU	1.24	1.05	0.997
Intel Core i7-4770K	1.14	1.13	1.00
Intel Xeon CPU E5-2660 v3	1,14	1.09	0.98
Intel Xeon Gold 5120	2.00	1.04	1.00
Intel Xeon Gold 6148	1.86	1.01	1.01
Intel Xeon Phi 7210	1.56	1.08	1.03

Table 6.5: Runtime for various kernel combination. Either we use only the interaction list kernels (left), one stencil FMM kernel (middle), or only the stencil-based kernels (right)

6.4.5 Test Summary

Using the stencil FMM kernels, we achieve a speedup over the old interaction list kernels in each of our tests. On platforms using AVX512 the difference between the kernels is significant, which is the result we expected when first introducing the stencil FMM kernels to Octo-Tiger. The speedup is less severe on AVX2 platforms, yet it is still noticeable.

We also notice a decrease in parallel efficiency using the stencil kernels over the interaction list kernels. However, if we examine the parallel efficiency at specific runtimes, we can see that the stencil kernels achieve a similar parallel efficiency at similar runtimes. For example, using the interaction list kernels on the Knights Landing platform with all 68 cores, we achieve a parallel efficiency of 0.83 and a runtime of 146 seconds. Using the stencil kernels, we can achieve this runtime with only 29 cores. At this point, we still have a parallel efficiency of 0.81. The same holds true when using the Xeon Gold 6148. Using the interaction list kernels, we achieve a runtime of 80

seconds and a parallel efficiency of 0.6 if we use all cores. Using the stencil kernels, we achieve a parallel efficiency of 0.66 at a runtime of 79 seconds. To reach this runtime, we only require 14 cores. The stencil kernels seem to run out of work faster, which decreases the parallel efficiency further down the line.

6.5 GPU Tests

In this section, we take a look at the speedup we can accomplish when we start to use NVIDIA GPUs as coprocessors to help out with the execution of the stencil FMM kernels. In the second test, we take a look at the impact the number of used Cuda streams has on the runtime. We also compare it with the runtime, when we use the same number of overall Cuda streams, but split them onto multiple GPUs. In the last test, we use Piz Daint to simulate a large scenario Octo-Tiger on multiple nodes. We use 60 compute nodes and thus 60 NVIDIA P100 GPUs for this run.

6.5.1 GPU Test 1: Speedup when using both CPU/GPU

We tested the implementation with three different scenarios, using three different platforms. We use scenario two, three and four. In each run, we first collected the total runtime data using only the CPU of the respective platform. After that, we added all available GPUs of the platform to assist the CPU. We use 24 Cuda streams for each GPU, meaning each of them can process 24 octree nodes simultaneously. On the CPU we use all available cores for every run.

For scenario 2 the results are in table 6.6, for scenario 3 in table 6.7, and for scenario 4 the results can be found in table 6.8. For all of these tests we achieve a similar speedup.

Platform	CPU-only	CPU and GPU	Speedup
argon-gtx	108s	81s	1.32
argon-tesla	140s	108s	1.29
Piz Daint (single node)	176s	126s	1.39

Table 6.6: Runtimes of scenario 2 using either CPU only or CPU/GPU combined.

Platform	CPU-only	CPU and GPU	Speedup
argon-gtx	242s	189s	1.28
argon-tesla	331s	222s	1.40
Piz Daint (single node)	424s	311s	1.37

Table 6.7: Runtimes of scenario 3 using either CPU only or CPU/GPU combined.

Platform	CPU-only	CPU and GPU	Speedup
argon-gtx	143s	102s	1.40
argon-tesla	321s	223s	1.44
Piz Daint (single node)	230s	158s	1.46

Table 6.8: Runtimes of scenario 4 using either CPU only or CPU/GPU combined.

6.5.2 GPU Test 2: Varying the Number of Cuda Streams

In the last test we used a fixed number of Cuda streams for each GPU. However, the theoretical limit of concurrent kernel execution is higher than 24. An NVIDIA Geforce GTX 1080 Ti supports 32 concurrent kernels, an NVIDIA Tesla P100 GPU supports 128. In this test, we slowly increase the number of Cuda streams per GPU used for each run and measure the runtime.

In figure 6.5 we can see the result of this test on the argon-tesla platform. Even one P100 supports 128 concurrent kernel executions. Therefore, we would expect the runtime to be optimal if we are using 128 Cuda streams. However, this is not the case. After 24 Cuda streams, the runtime starts to increase again. We can still benefit from using a second P100 though. In this case, we split the streams between the GPUs. The first 24 Streams are being run on the first P100. The remaining streams will be run on the second P100.

We see a similar behavior using the platform argon-gtx in figure 6.6. If we use only one GPU, the runtime starts to increase again if we use more than 22 streams. If we use two GPUs, this does not happen. In this case, the scheduler switches to the second GPU at 16 streams. We achieve the best runtime in this test when we are using four GPUs, each of them with eight Cuda streams.

Since the argon-gtx contains eight GPUs, we repeat this test multiple times using all of them. However, we vary the maximum number of streams of each GPU. As we can see in figure 6.7, we achieve a similar running time in the end. If we are only using 16 streams on each GPU, the scheduler will use all GPUs sooner. For example, at 32 streams it would already use two GPUs. In the run with 32 streams per GPU, it would only use one. We thus achieve a similar running time faster, by using fewer streams per GPU. The best overall running time was obtained with 24 streams per GPU, resulting in an overall number of 192 streams. However, using 16 streams per GPU we reach a runtime almost as good with only 128 streams.

6.5.3 GPU Test 3: Distributed Run

In figure 6.8 we can see how Octo-Tiger scales to multiple comute nodes on Piz Daint. We use scenario 5. This scenario is too large to fit on a single compute node, so we therefore start with twelve compute nodes. We do each run twice, one time with the P100 GPUs of the compute nodes, and one time without them. If we use the P100 GPUs in those twelve compute nodes we achieve a speedup of 1.33 over the runtime without the GPUs. Using the GPUs, we achieve almost the same parallel efficiency as without them. Thus we can maintain this speedup, even if are scaling up to 60 compute nodes.

6.5.4 GPU Test Summary

In all our GPU tests, we achieve a speedup if we add one or more GPUs to assist the CPU. Furthermore, we achieve the same distributed parallel efficiency with or without using the GPUs on Piz Daint. Thus the speedup is also applicable to larger scenarios than the ones we tested here.

However, the speedup we achieve depends on how many Cuda streams we use on each GPU. On the one hand, if we are not using enough streams, we merely get the overhead of using a GPU without a noticeable runtime improvement. On the other hand, the runtime improves only up to 24 Streams

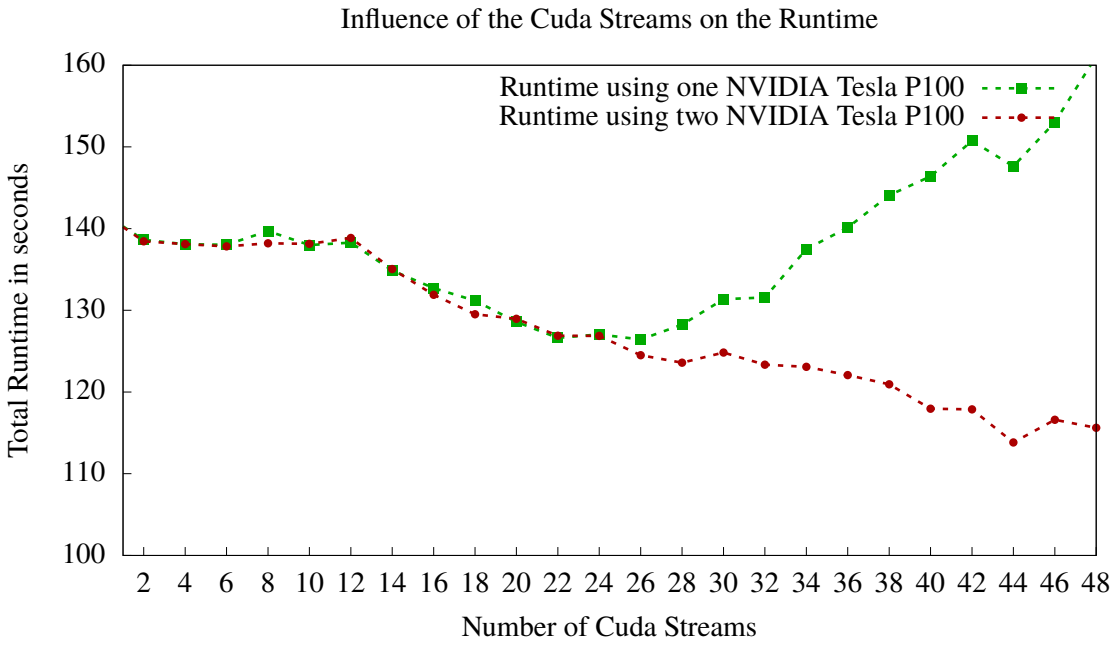

Figure 6.5: Comparison of the runtime for a different number of Cuda streams using either one or two P100 GPUs. We use scenario 2 and the platform argon-tesla for this test.

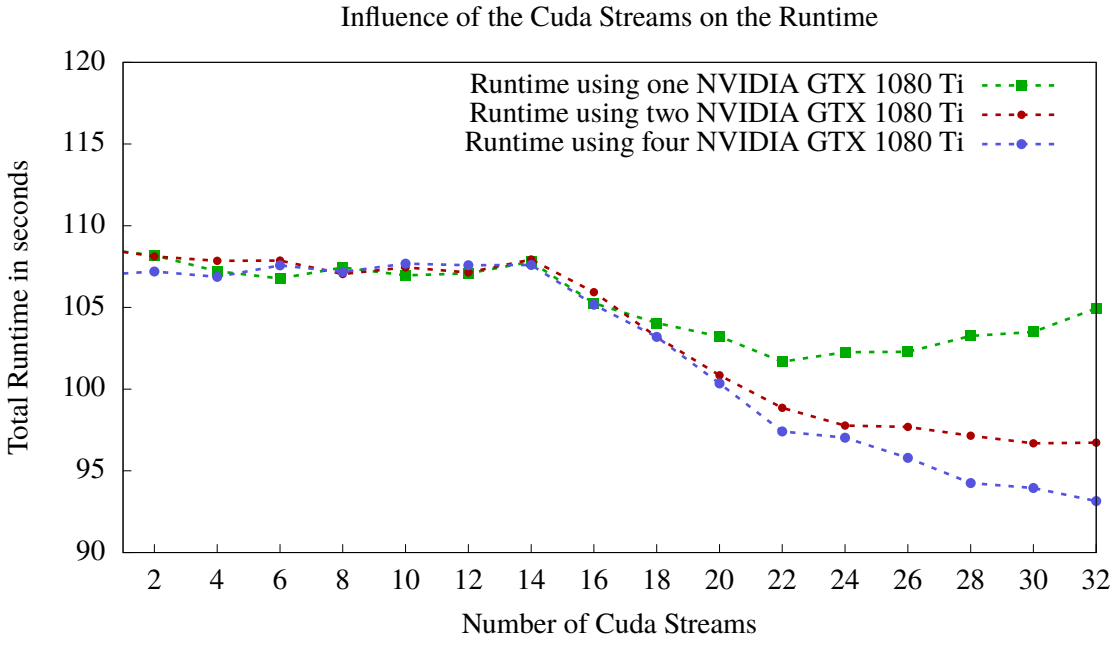

Figure 6.6: Comparison of the runtime for a different number of Cuda streams using either one, two or four GTX 1080 Ti GPUs. We use scenario 2 and the platform argon-gtx for this test.

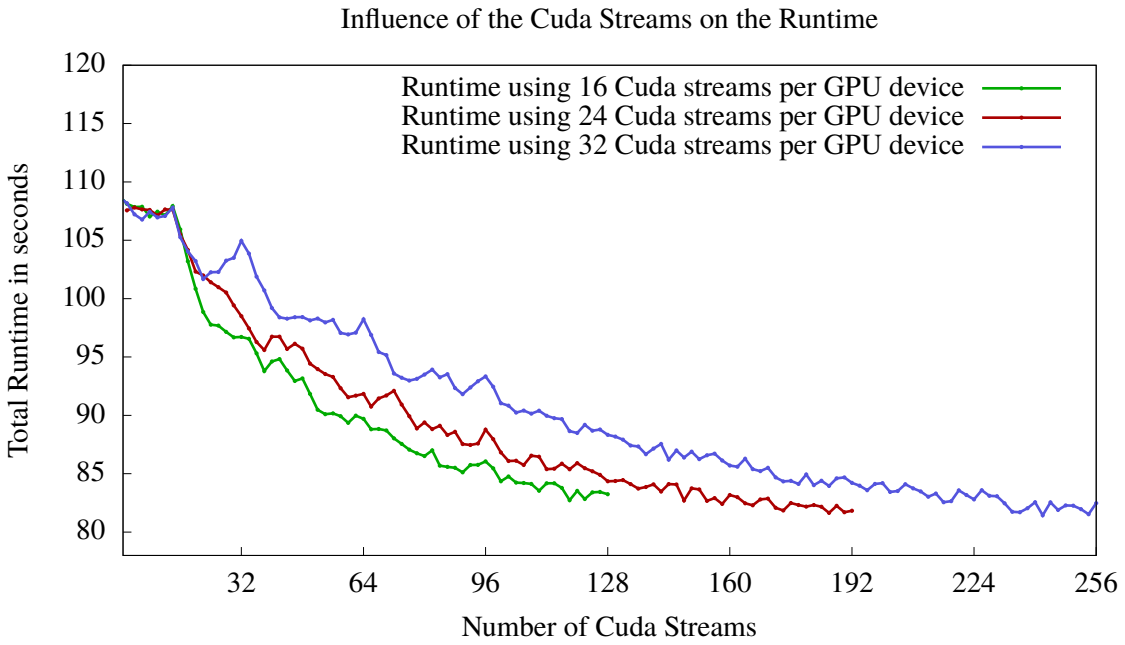

Figure 6.7: Comparison of the runtime using a different number of Cuda streams per GPU. We use scenario 2 and the platform argon-gtx for this test. Each of the three runs will eventually use all eight GPUs, albeit with a different number of streams per GPU.

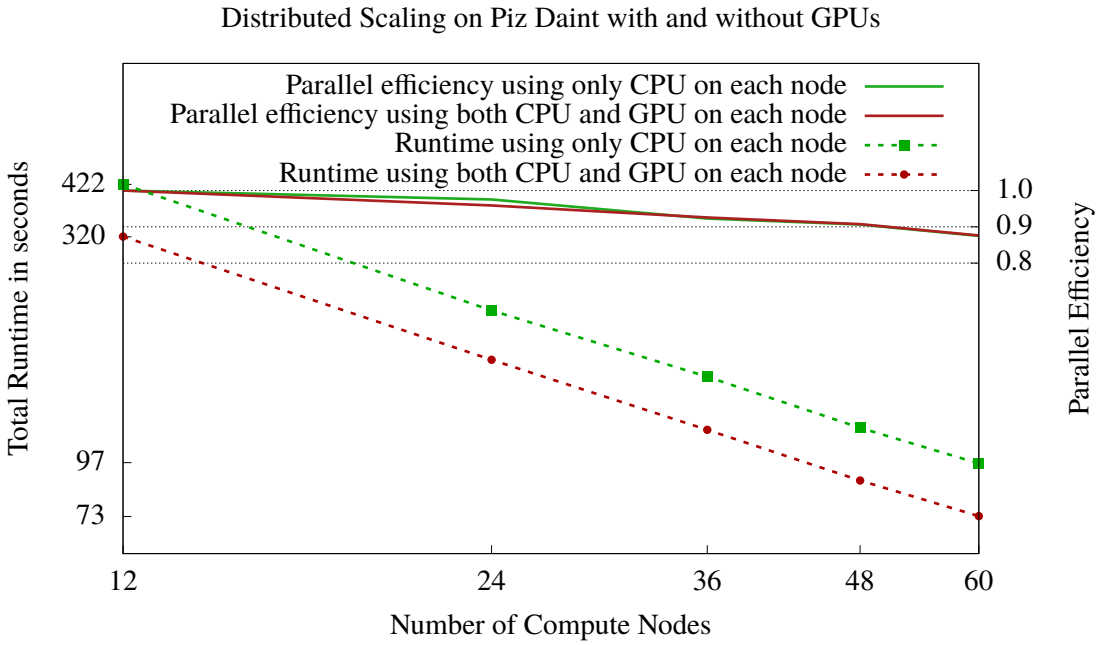

Figure 6.8: Distributed run of Octo-Tiger with scenario 5. We use 12 to 60 compute nodes, each with or without their P100.

per GPU. More than that yield no runtime improvement. In theory, both GPUs we considered in our test should support more than 24 concurrent streams. The Geforce GTX 1080 Ti supports 32, the P100 128 concurrent kernel launches. We also do not run out of work. As we can see in test 2, splitting the same amount of streams onto multiple GPUs instead of just one, improves the runtime significantly. Using the NVIDIA profiler, we can also see that more than 24 FMM kernels get launched concurrently on a GPU. However, we can also observe that the runtime per kernel increases when more of them are running in parallel. Thus, we are most likely facing a bandwidth problem.

7 Conclusion and Future Work

In this work, we introduced the Struct of Array data structure and five stencil FMM interaction kernels into Octo-Tiger. Using our implementation of the five new interaction kernels, we achieved a consistent speedup over the interaction list FMM kernels that Octo-Tiger was used before. We can see the most significant speedup on platforms supporting AVX512. Here, the new interaction kernels are especially beneficial. Using all 40 cores of an Intel Xeon Gold 6148, we achieve a speedup of 2x using the stencil FMM kernels. On all 68 cores of an Intel Xeon Phi 7210, the speedup was 1.9x. On platforms only supporting AVX2 we still experience a speedup, albeit less so.

We implemented these stencil kernels in a way so that they are also executable by NVIDIA GPUs. By using templated functions and lambdas functions, we can call the same cell to cell interaction methods on both the CPU and the GPU. This way we can minimize code duplication and increase the maintainability of the code. We further introduced a way of using the GPU as a coprocessor without hindering the CPU. To accomplish this, we made use of HPX Compute and Cuda streams. Using a CPU/GPU scheduler, we avoid starving the CPU of work, while still being able to offload the execution of some interaction kernels to the GPU. Using both the CPU and the GPU, we achieve a consistent speedup compared to the runtime when we are using only the CPU. For the platforms and scenarios tested, this speedup ranged from 1.28 to 1.46. Since we are only offloading work regarding the FMM to the GPU, the theoretical speedup we can obtain is limited. In our prior work [PDM+18] the fast multipole method was responsible for about 50% of the runtime, which would limit the maximum speedup we can expect to two. This way of interleaving CPU and GPU work is also not specific to Octo-Tiger. As long as HPX and Cuda are being used, other applications could use a very similar approach. To an extent, they could even reuse the scheduler introduced in this work.

Lastly, we were able to maintain the distributed scalability of Octo-Tiger. On Piz Daint we used 60 compute nodes to simulate a large scenario. Using all 60 nodes, we still achieved a speedup of 1.33 when adding all 60 of the P100 GPUs to assist with the computation.

Future Work

In our prior work, we experienced high cache pressure in the FMM kernels, which lead to performance problems on devices with a small L2 cache. We intend to develop better algorithms for the kernels to improve that. This could also help us to achieve better kernel runtimes on the GPU since memory bandwidth and the number of available registers is limited here. If we minimize the data we touch we could launch more threads concurrently per Streaming Processor, due to the reduced register usage.

We could accomplish this by splitting the most compute-intensive stencil FMM kernel, the m2mp kernel, into multiple parts. If we sort the computations within this kernel in the right way and split it, we do not have to touch all 20 multipole moments for each cell to cell interaction. Instead, we would have multiple kernels, each only touching a subset of the 20 multipoles coefficients per cell. This way we could reduce both cache pressure, as well as register usage on the GPU, while still maintaining the stencil approach.

We also intend to offload more work to the GPU. So far we can only execute the FMM kernels on the GPU. However, now that the scheduler is in place, it is simpler to start offloading other methods as well.

Bibliography

[BB98] H. A. Bethe, G. Brown. “Evolution of binary compact objects that merge”. In: *The Astrophysical Journal* 506.2 (1998), p. 780 (cit. on p. 13).

[BBCP90] W. Benz, R. Bowers, A. W. Cameron, W. Press. “Dynamic mass exchange in doubly degenerate binaries. I-0.9 and 1.2 solar mass stars”. In: *The Astrophysical Journal* 348 (1990), pp. 647–667 (cit. on p. 14).

[BCD+16] T. Barnes, B. Cook, J. Deslippe, D. Doerfler, B. Friesen, Y. He, T. Kurth, T. Koskela, M. Lobet, T. Malas, et al. “Evaluating and optimizing the NERSC workload on knights landing”. In: *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), International Workshop on*. IEEE. 2016, pp. 43–53 (cit. on p. 35).

[BH86] J. Barnes, P. Hut. “A hierarchical O (N log N) force-calculation algorithm”. In: *nature* 324.6096 (1986), p. 446 (cit. on p. 19).

[CGH+07] G. C. Clayton, T. R. Geballe, F. Herwig, C. Fryer, M. Asplund. “Very Large Excesses of 18O in Hydrogen-deficient Carbon and R Coronae Borealis Stars: Evidence for White Dwarf Mergers”. In: *The Astrophysical Journal* 662.2 (2007), p. 1220 (cit. on p. 13).

[Cla96] G. C. Clayton. “The R Coronae Borealis Stars”. In: *Publications of the Astronomical Society of the Pacific* 108.721 (1996), p. 225. URL: http://stacks.iop.org/1538-3873/108/i=721/a=225 (cit. on p. 13).

[CT02] C. Cutler, K. S. Thorne. “An overview of gravitational-wave sources”. In: *General Relativity and Gravitation*. World Scientific, 2002, pp. 72–111 (cit. on p. 13).

[Deh00] W. Dehnen. “A Very Fast and Momentum-conserving Tree Code”. In: *The Astrophysical Journal Letters* 536.1 (2000), p. L39. URL: http://stacks.iop.org/1538-4357/536/i=1/a=L39 (cit. on p. 19).

[Deh02] W. Dehnen. “A hierarchical O (N) force calculation algorithm”. In: *Journal of Computational Physics* 179.1 (2002), pp. 27–42 (cit. on p. 26).

[Den74] J. B. Dennis. “First version of a data flow procedure language”. In: *Programming Symposium*. Springer. 1974, pp. 362–376 (cit. on p. 32).

[EFGL14] P. Estérie, J. Falcou, M. Gaunard, J.-T. Lapresté. “Boost. simd: generic programming for portable simdization”. In: *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*. ACM. 2014, pp. 1–8 (cit. on p. 33).

[ET09] W. Even, J. E. Tohline. “Constructing synchronously rotating double white dwarf binaries”. In: *The Astrophysical Journal Supplement Series* 184.2 (2009), p. 248 (cit. on p. 18).

[Fog13] A. Fog. “C++ vector class library”. In: *URL: http://www. agner. org/optimize/vectorclass. pdf* (2013) (cit. on p. 33).

[GR87] L. Greengard, V. Rokhlin. “A fast algorithm for particle simulations”. In: *Journal of Computational Physics* 73.2 (1987), pp. 325–348. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(87)90140-9. URL: http://www.sciencedirect.com/science/article/pii/0021999187901409 (cit. on p. 19).

[GSS+07] G. R. Gao, T. Sterling, R. Stevens, M. Hereld, W. Zhu. “Parallex: A study of a new parallel computation model”. In: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE. 2007, pp. 1–6 (cit. on p. 31).

[Hac86] I. Hachisu. “A versatile method for obtaining structures of rapidly rotating stars. II-Three-dimensional self-consistent field method”. In: *The Astrophysical Journal Supplement Series* 62 (1986), pp. 461–499 (cit. on p. 18).

[HDB+17] T. Heller, P. Diehl, Z. Byerly, J. Biddiscombe, H. Kaiser. “HPX–An open source C++ Standard Library for Parallelism and Concurrency”. In: (2017) (cit. on p. 31).

[HKD+16] T. Heller, H. Kaiser, P. Diehl, D. Fey, M. A. Schweitzer. “Closing the performance gap with modern c++”. In: *International Conference on High Performance Computing*. Springer. 2016, pp. 18–31 (cit. on p. 31).

[HKI13] T. Heller, H. Kaiser, K. Iglberger. “Application of the ParalleX execution model to stencil-based problems”. In: *Computer Science-Research and Development* 28.2-3 (2013), pp. 253–261 (cit. on p. 31).

[HKSF13] T. Heller, H. Kaiser, A. Schäfer, D. Fey. “Using HPX and LibGeoDecomp for scaling HPC applications on heterogeneous supercomputers”. In: *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. ACM. 2013, p. 1 (cit. on p. 31).

[HLH+18] T. Heller, B. Lelbach, K. Huck, J. Biddiscombe, P. Grubel, A. Koniges, M. Kretz, D. Marcello, D. Pfander, A. Serio, J. Frank, G. Clayton, D. Pflüger, D. Eder, H. Kaiser. “Harnessing Billions of Tasks for a Scalable Portable Hydrodynamic Simulation of the Merger of Two Stars”. In: *International Journal of High Performance Computing Applications (IJHPCA)* (2018). submitted (cit. on pp. 13, 31).

[Kat16] M. P. Katz. “White Dwarf Mergers on Adaptive Meshes”. PhD thesis. State University of New York at Stony Brook, 2016 (cit. on pp. 13, 14).

[KBS09] H. Kaiser, M. Brodowicz, T. Sterling. “Parallex an advanced parallel execution model for scaling-impaired applications”. In: *Parallel Processing Workshops, 2009. ICPPW’09. International Conference on*. IEEE. 2009, pp. 394–401 (cit. on p. 31).

[KCM+17] K. Kadam, G. C. Clayton, P. M. Motl, D. C. Marcello, J. Frank. “Numerical Simulations of Close and Contact Binary Systems Having Bipolytropic Equation of State”. In: *Proceedings of the American Astronomical Society (AAS)*. meeting 229, art. id 433.14. http://adsabs.harvard.edu/abs/2017AAS...22943314K. Grapevine, Texas, Country, 2017 (cit. on p. 13).

[KHA+14] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey. “Hpx: A task based programming model in a global address space”. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM. 2014, p. 6 (cit. on p. 31).

[KL12] M. Kretz, V. Lindenstruth. "Vc: A C++ library for explicit vectorization". In: *Software: Practice and Experience* 42.11 (2012), pp. 1409–1430 (cit. on pp. 32, 33).

[KMF+16] K. Kadam, P. M. Motl, J. Frank, G. C. Clayton, D. C. Marcello. "A Numerical Method for Generating Rapidly Rotating Bipolytropic Structures in Equilibrium". In: *Monthly Notices of the Royal Astronomical Society (MNRAS)* 462.2 (2016). http://adsabs.harvard.edu/abs/2016MNRAS.462.2237K, pp. 2237–2245. ISSN: 0035-8711. DOI: 10.1093/mnras/stw1814 (cit. on p. 13).

[Kre+15] M. Kretz et al. "Extending C++ for explicit data-parallel programming via SIMD vector types". In: (2015) (cit. on p. 33).

[KZC+16] M. P. Katz, M. Zingale, A. C. Calder, F. D. Swesty, A. S. Almgren, W. Zhang. "White dwarf mergers on adaptive meshes. i. methodology and code verification". In: *The Astrophysical Journal* 819.2 (2016), p. 94 (cit. on p. 14).

[Mag17] K. Maguire. "Type Ia Supernovae". In: *Handbook of Supernovae*. Ed. by A. W. Alsabti, P. Murdin. Cham: Springer International Publishing, 2017, pp. 293–316. ISBN: 978-3-319-21846-5. DOI: 10.1007/978-3-319-21846-5_36. URL: https://doi.org/10.1007/978-3-319-21846-5_36 (cit. on p. 13).

[Mar17] D. C. Marcello. "A Very Fast and Angular Momentum Conserving Tree Code". In: *The Astronomical Journal* 154.3 (2017), p. 92. URL: http://stacks.iop.org/1538-3881/154/i=3/a=92 (cit. on pp. 13, 19, 23).

[MCML15] E. J. Montiel, G. C. Clayton, D. C. Marcello, F. J. Lockman. "What Is the Shell Around R Coronae Borealis?" In: *Astronomical Journal (AJ)* 150.1, art. id 14 (2015). http://adsabs.harvard.edu/abs/2015AJ....150...14M. ISSN: 0004-6256. DOI: 10.1088/0004-6256/150/1/14 (cit. on p. 13).

[MKC+ew] D. C. Marcello, K. Kadam, G. C. Clayton, J. Frank, H. Kaiser, P. M. Motl. "Introducing Octo-tiger/HPX: Simulating Interacting Binaries with Adaptive Mesh Refinement and the Fast Multipole Method". In: *Proceedings of Science* (under review) (cit. on pp. 13, 17, 18, 20, 21, 23, 24).

[PAD+98] S. Perlmutter, G. Aldering, M. Della Valle, S. Deustua, R. Ellis, S. Fabbro, A. Fruchter, G. Goldhaber, D. Groom, I. Hook, et al. "Discovery of a supernova explosion at half the age of the Universe". In: *Nature* 391.6662 (1998), p. 51 (cit. on p. 13).

[PCM+16] A. Pohl, B. Cosenza, M. A. Mesa, C. C. Chi, B. Juurlink. "An evaluation of current SIMD programming models for C++". In: *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*. ACM. 2016, p. 3 (cit. on pp. 32, 33).

[PDM+18] Pfander, Daiß, Marcello, Kaiser, Pflüger. "Accelerating Octo-Tiger: Stellar Mergers on Intel Knights Landing with HPX". In: 2018 (cit. on pp. 13, 14, 19, 22, 36, 61).

[RFC+98] A. G. Riess, A. V. Filippenko, P. Challis, A. Clocchiatti, A. Diercks, P. M. Garnavich, R. L. Gilliland, C. J. Hogan, S. Jha, R. P. Kirshner, et al. "Observational evidence from supernovae for an accelerating universe and a cosmological constant". In: *The Astronomical Journal* 116.3 (1998), p. 1009 (cit. on p. 13).

[Sod15] A. Sodani. "Knights landing (knl): 2nd generation intel® xeon phi processor". In: *Hot Chips 27 Symposium (HCS), 2015 IEEE*. IEEE. 2015, pp. 1–24 (cit. on p. 35).

[Tho80] K. S. Thorne. “Multipole expansions of gravitational radiation”. In: *Rev. Mod. Phys.* 52 (2 Apr. 1980), pp. 299–339. DOI: 10.1103/RevModPhys.52.299. URL: https://link.aps.org/doi/10.1103/RevModPhys.52.299 (cit. on p. 19).

[Web84] R. Webbink. “Double white dwarfs as progenitors of R Coronae Borealis stars and Type I supernovae”. In: *The Astrophysical Journal* 277 (1984), pp. 355–360 (cit. on p. 13).

[WKH15] M. Wong, H. Kaiser, T. Heller. “Towards Massive Parallelism (aka Heterogeneous Devices/Accelerator/GPGPU) support in C++ with HPX”. In: (2015) (cit. on p. 31).

All links were last followed on March 17, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature