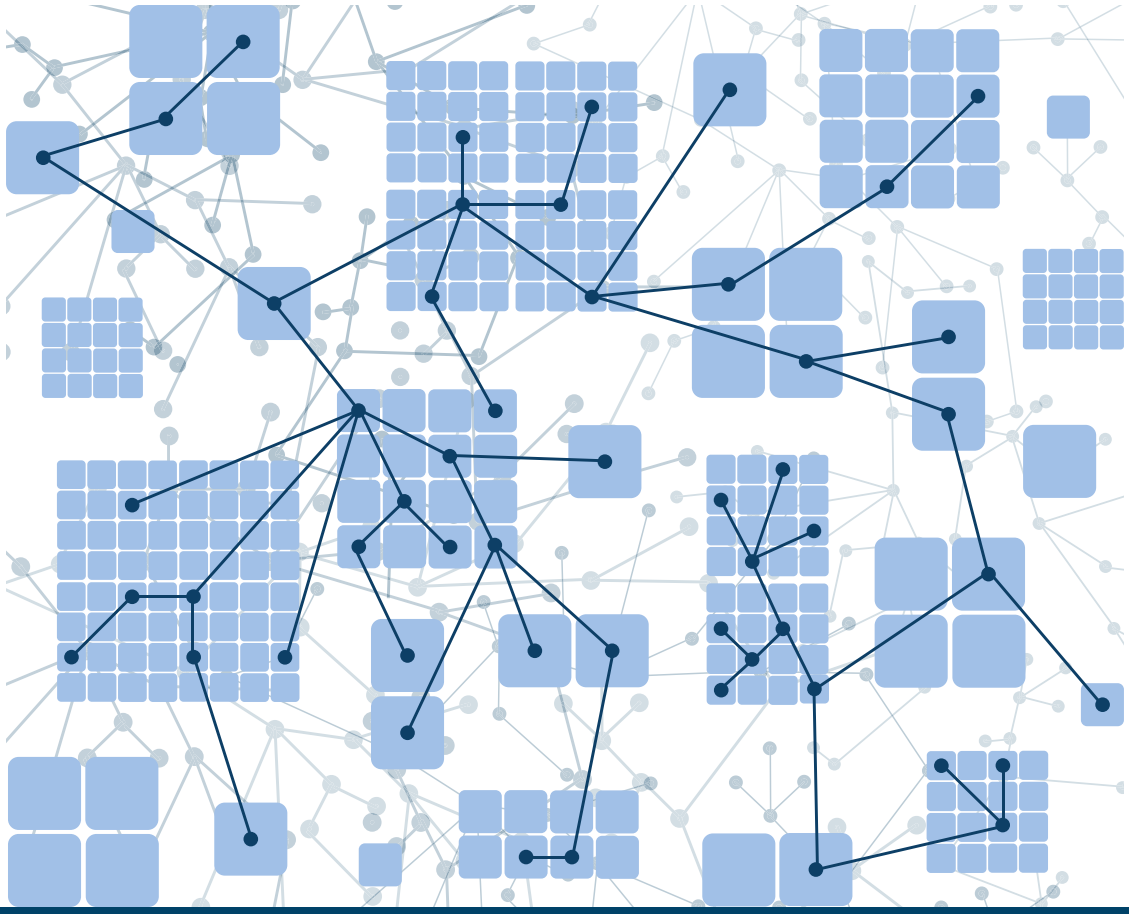


Eventify Meets Heterogeneity: Enabling Fine-Grained Task-Parallelism on GPUs

Laura Morgenstern

IAS Series
Band / Volume 63
ISBN 978-3-95806-765-3

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
Forschungszentrum

Schriften des Forschungszentrums Jülich
IAS Series Band / Volume 63

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

Eventify Meets Heterogeneity: Enabling Fine-Grained Task-Parallelism on GPUs

Laura Morgenstern

Schriften des Forschungszentrums Jülich
IAS Series

Band / Volume 63

ISSN 1868-8489

ISBN 978-3-95806-765-3

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

Herausgeber und Vertrieb:	Forschungszentrum Jülich GmbH Zentralbibliothek, Verlag 52425 Jülich Tel.: +49 2461 61-5368 Fax: +49 2461 61-6103 zb-publikation@fz-juelich.de www.fz-juelich.de/zb
Umschlaggestaltung:	Grafische Medien, Forschungszentrum Jülich GmbH
Druck:	Grafische Medien, Forschungszentrum Jülich GmbH
Copyright:	Forschungszentrum Jülich 2024

Schriften des Forschungszentrums Jülich
IAS Series, Band / Volume 63

DE-Ch1 (Diss. Fakultät für Informatik, Techn. Univ. Chemnitz, 2023)

ISSN 1868-8489
ISBN 978-3-95806-765-3

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

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

ALU Arithmetic Logic Unit

ATG Asynchronous Task Graph

CAS Compare-And-Swap

CCD Core Chiplet Die

CoD Cluster-on-Die

CRTP Curiously Recurring Template Pattern

CSP Communicating Sequential Processes

CU Compute Unit

DAG Directed Acyclic Graph

DSL Domain Specific Language

FFT Fast Fourier Transform

FIFO First In First Out

FMM Fast Multipole Method

FPU Floating Point Unit

GPC GPU Processing Cluster

GPGPU General Processing Graphics Processing Unit

GROMACS GROningen MAchine for Chemical Simulations

GTL GPU Template Library

HPC High Performance Computing

ITS Independent Thread Scheduling

JSC Jülich Supercomputing Centre

L2L Local to Local

L2P Local to Particle

LAMMPS Large-scale Atomic/Molecular Massively Parallel Simulator

LIFO Last In First Out

M2L Multipole to Local

M2M Multipole to Multipole

MA Micro-Architecture

MD Molecular Dynamics

MHQ Multiple Hierarchical Queues

MIMD Muliple-Instruction Stream, Multiple-Data Stream

MISD Multiple-Instruction Stream, Single-Data Stream

MPI Message Passing Interface
MPMC Multi-Consumer Multi-Producer
MPSC Multi-Producer Single-Consumer
MQ Multiple Multi-Producer Single-Consumer Queues
NUMA Non-Uniform Memory Access
OOO Out Of Order
P2M Particle to Multipole
P2P Particle to Particle
PB Processing Block
PE Processing Element
PGAS Partitioned Global Address Space
PME Particle Mesh Ewald
PRAM Parallel Random Access Machine
PT Persistent Threads
PU Processing Unit
RDMA Remote Direct Memory Access
SFINAE Substitution Failure Is Not An Error
SFU Special Function Unit
SIMD Single-Instruction Stream, Multiple-Data Stream
SIMT Single Instruction, Multiple Threads
SISD Single-Instruction Stream, Single-Data Stream
SM Streaming Multiprocessor
SMT Simultaneous Multithreading
SP Stream Processor
SPMD Single Programme, Multiple Data
SPPEXA Priority Programme Software for Exascale Computing
SPSC Single-Consumer Single-Producer
SQ Single Multi-Consumer Multi-Producer Queue
SU SIMD Unit
TBB Threading Building Blocks
TMP Template Meta Programming
TPC Texture Processing Cluster
UAM Uniform Architecture Model
UEM Uniform Execution Model
ULT User Level Threads
UMA Uniform Memory Access

Notation

d_i	Data element i
i_i	Instruction i
D_l	Data stream l
I_l	Instruction stream l
D_{grid}	Grid dimension
D_{block}	Thread block dimension
N_{thread}	Total number of threads in grid
t_i^l	Thread with local thread ID i
t_i^g	Thread with global thread ID i
B_i	Thread block with block index i
m_i	Master thread of thread block B_i
G	Grid as N_{thread}-tuple of its threads
M	All master threads of a grid as D_{grid}-tupel
m	Single, arbitrary master thread, $m \in M$
$d_{\max}$	Maximal tree depth
p	Multipole order
ω	Multipole moment
μ	Local moment
d	Tree depth
$r_{\min}$	Minimal runtime
$t_{r_{\min}}$	Number of threads for which $r_{\min}$ is reached
$R_{\max}$	Maximal runtime ratio
R_{eff}	Effective runtime ratio

Abstract

Many scientific computing algorithms barely provide sufficient data-parallelism to exploit the ever-increasing hardware parallelism of today's heterogeneous computing environments. The challenge is to fully exploit the parallelization potential of such algorithms. To tackle this challenge, diverse task-parallel programming technologies have been introduced that allow for the flexible description of algorithms along task graphs. For algorithms with dense task graphs, however, task-parallelism is still hard to exploit efficiently since it is programmatically complex to describe and imposes high dependency resolution overheads on the execution model. This becomes especially challenging on GPUs which are not designed for synchronization-heavy applications.

The research objective of this thesis is an execution model that enables fine-grained task parallelism on GPUs. To reach this objective, the contributions of the thesis are five fold. Firstly, it refines the stream interaction model behind Flynn's Taxonomy as uniform foundation for concurrency in architectures and programming models. Secondly, it analyzes the quantitative trends in CPU and GPU architectures and examines their influence on programming models. Thirdly, it introduces an execution model that enables threading, efficient blocking synchronization and queue-based task scheduling on GPUs. Fourthly, it ports the task-parallel programming library Eventify to GPUs. And fifthly, it examines the performance and sustainability of this approach with the task graph of a fast multipole method as use case. The results show that fine-grained task parallelism improves execution time by an order of magnitude in comparison to classical loop-based data parallelism.

Chapter 1

A new generation of software libraries and algorithms are needed to effectively use the high performance computing environments in use today.
Jack Dongarra

Introduction

1.1 Motivation

Processors become fatter, not faster. For decades, each new transistor generation provided smaller transistors that could switch faster than ever before. This enabled new processors to operate at clock frequencies never seen before. In the past, this continuous increase in clock frequencies was the main driver of compute performance growth. For software, this resulted in the convenient effect of shorter execution times without any code changes.

Around 2005, however, the striving for higher clock frequencies approached several physical limitations that led to excessively high power consumption and heat generation. Hence, the trend of continuously increasing clock frequencies started to stagnate (see [56, p. 45]). This begs the question, how compute performance growth can be sustained further. The answer to this question are SMT, multi- and many-core architectures. Hence, hardware parallelism became and continues to be the new driver of compute performance growth. To benefit from this performance growth, software has to be build upon parallel programming models and languages. Software developers have to determine the parallelization potential of applications and algorithms as well as understand the pitfalls and bottlenecks of concurrency (see [98]).

Scientific software aims for strong scaling, not weak scaling. Scientific simulations have become a vital research method in many fields such as climate modeling, materials science and biochemistry. The problem size of scientific simulations is typically determined by physical properties such as spatial resolution, number of particles or size of molecules. As soon as simulations reach realistic system sizes, there is little scientific value in increasing problem sizes further. Accordingly, we cannot employ weak scaling to exploit the increasing amount of hardware parallelism. For this reason, scientific simulations typically aim at strong scaling (see [35]).

A few rare algorithms are inherently parallel and exhibit no or very few data-dependencies. Thus, they provide sufficient data-parallelism that can be exploited via loop-based parallelism with OpenMP [84] or OpenACC [83]. Most algorithms employed by scientific applications, however, provide only a limited amount of data-parallelism. The challenge is to exploit the full parallelization potential of such algorithms. To tackle this challenge, the concept of task-parallelism was introduced. Task-parallelism allows for the flexible description of an algorithm along task graphs, i.e. along work units and dependencies between these work units. Meanwhile, task-parallelism is provided by several parallel programming technologies such as HPX [62], Intel TBB [90] and Kokkos [39]. Consequentially, the concept became also part of OpenMP and CUDA [78], which are the prevalent on-node programming models on current supercomputers.

For algorithms with overly many data-dependencies, however, task-parallelism is still hard to

exploit efficiently. First, since algorithms with many data-dependencies require relatively fine-grained tasks to provide sufficient parallelism. This, however, leads to complex task graphs that are hard to describe with common task-parallel programming models. Second, since it imposes high dependency resolution overheads on the execution model. Event-based task-parallelism [49] was introduced to ease the description of complex task graphs and to enable low-overhead dependency resolution.

Sustainable software requires a uniform code base, not architecture-dependent derivatives. In addition to processors getting more parallel, the heterogeneity of compute nodes increases. Current compute nodes exhibit multi- and many-core CPUs side by side with GPUs. However, different processor types exhibit different hardware properties; CPUs and GPUs vary in latency, throughput, memory hierarchy and the type of parallelism they provide. Since CPUs are supposed to run an operating system and a highly diverse set of applications as efficiently as possible, they consist of only a few but very sophisticated compute cores. GPUs, on the other hand, consist of thousands but very light-weight cores to provide a huge amount of parallelism. To fully exploit the performance of all processor architectures a compute node provides, software has to adapt to these differences. Hardware and software vendors as well as open standard committees acknowledge this fact by aiming at uniform programming models that enhance the programmability of a wide range of applications on heterogeneous systems. Examples include, but are not limited to OpenCL [65], Intel oneAPI [58], OpenMP and OpenACC.

Despite the unifying ambitions behind those models, scientific software rarely relies on only one of them. Instead, it combines multiple parallel programming technologies to hand-tune performance on each and every architecture from each and every vendor. This results in overly complex code bases that are hard to maintain. In the worst case, diverging software derivatives exist, each of which supports a different architecture. To provide a consistent user experience, all versions should provide the same set of functional features. However, keeping those features in sync is costly in two respects. First, it multiplies the implementation and maintenance efforts since every feature has to be implemented for multiple architectures. Second, these efforts are made at the charge of new functional features since time, staff and funds are limited. Accordingly, scientists and software developers strive for sustainable scientific software that solves the *3P challenge* – the trade-off between performance, portability and programmability.

1.2 Research Objective

This work tackles the 3P challenge by contributing to a uniform programming model for event-based task-parallelism on heterogeneous, massively parallel hardware. Research objective and solution strategy are oriented towards the insights on sustainable software for science provided by [35], and by the *Exascale Software Study* provided by [10]. Therefore, this work contributes to closing the identified “gap between computer science research in areas such as code abstractions […] and high-level scientific application software”[35].

Concisely, the vision, research objective and research questions of this work are stated as follows:

Vision A sustainable uniform programming model to enhance the scalability and execution time of irregular algorithms on heterogeneous systems.

Objective A sustainable execution model that enables task-parallelism for irregular algorithms on GPUs.

Q1: What are the hardware architecture trends for parallelism on CPUs and GPUs?

Q2: What are the parallel programming model trends for CPUs and GPUs?

Q3: Can event-based task-parallelism in comparison to loop-based data-parallelism enhance the scalability and execution time of the FMM on CPUs?

Q4: How can event-based task-parallelism be enabled on GPUs?

Q5: Can event-based task-parallelism in comparison to loop-based data-parallelism enhance the scalability and execution time of irregular algorithms on GPUs?

Subsequently, we introduce the terms *sustainability*, *scalability* and *heterogeneous system* as used throughout this work. Furthermore, the FMM is motivated as demonstrator for algorithms with dense data dependency graphs.

1.2.1 Heterogeneous System

We refer to a heterogeneous system as a compute node that exhibits any combination of x86-compatible multi- and many-core CPUs as well as Nvidia's GPUs. Due to their availability in current HPC systems, this covers multi-core CPUs as Intel's Xeon Scalable Processors, many-core CPUs as AMD's EPYC Rome and GPUs as Nvidia's Tesla V100. The focus of this work is on on-node parallelization in the sense of shared memory and GPU programming.

1.2.2 Sustainability

Scientific software requires sustainability to enable the adding of new features as the domain science advances and the porting to new architectures as HPC systems evolve. This work refers to sustainability as follows:

Definition 1.1. *Sustainability A long-living software system is sustainable if it can be cost-efficiently maintained and evolved over its entire life-cycle.[68]*

Following [68] further, a software system is long-living if it must be operated for more than 15 years. This is typically true for scientific simulation software due to a relatively stable problem set, large user communities and heavy performance optimization efforts. MD simulation codes such as GROMACS [4] or LAMMPS [88] were first released in the 90s and meanwhile are in use for over 25 years.

For scientific software in HPC, this work considers sustainability to cover programmability and performance portability. In this context, we differentiate between two developer roles *library developers* and *application developers*. Library developers design and implement abstractions for parallel and heterogeneous hardware and provide these abstractions to application developers via a high-level parallel programming model. Based thereon, application developers express the parallelism behind their application algorithms independently of low-level hardware properties such as core counts, NUMA properties or warp sizes. The developed uniform programming model should not only enable the application developer to write sustainable software but should also be sustainable itself. This ensures the adaptability of the uniform programming model to future architectures and accordingly the sustainability of the application software.

1.2.3 Scalability

Adapting [69], the scalability of a parallel application on a parallel architecture is a measure of its capacity to efficiently utilize an increasing number of parallel processing elements. Here, with *parallel processing elements* being CPU-cores or streaming processor cores of a GPU. In this work, scalability is defined as strong scaling efficiency, i.e. the problem size is kept constant while the number of processing elements is increased. Based thereon, an application is considered to exhibit optimal scalability if its speedup corresponds to the number of processing elements the application is executed on. In this work, *scalability* and *runtime* are the underlying performance metrics to evaluate performance portability.

1.2.4 Event-Based Task-Parallelism

Event-based task parallelism, as introduced by Haensel et al. [50], combines event-based programming with the task-parallel algorithm model. The concept was developed to describe fine-grained task-parallelism for algorithms with a high amount of data-dependencies in a convenient way and resolve dependencies as efficiently as possible. In comparison to the classical top-down, recursive task-spawning approach, event-based task parallelism allows for a bottom-up creation of tasks with

user-defined granularities and priorities. *Eventify* [50] is a C++ implementation of this concept for CPUs. This work extends the programming and execution model of Eventify to GPUs.

Event-based or event-driven programming is a programming paradigm that is used for interrupt handling in operating systems, for synchronization in concurrent programming and as *Observer Pattern* in the context of object-oriented design patterns (see [40]). The control and data flow of an event-based program is determined by the (non-)occurrence of specific events. Eventify relies on *data events*. Data events are a special type of events that *reflect the progress of the operations achieved on the data* [50]. Section 5.3.4 elaborates on the event-based API of Eventify and its task-parallel execution model.

1.2.5 Fast Multipole Method for Molecular Dynamics

MD is a simulation method for the simulation of the physical movement of molecules and atoms in a system. The most time-consuming part of MD simulations is the computation of all pairwise long-range interactions between N particles[61]. A naive computation of all pairwise interactions would lead to a computational complexity of $\mathcal{O}(N^2)$.

The FMM is a fast summation technique that reduces the computational complexity to $\mathcal{O}(N)$. The FMM is based on a hierarchical space decomposition and therefore operates on a tree-based data structure. In addition to the dependencies represented by the edges of the tree, the FMM exhibits level-wise, horizontal data-dependencies between tree nodes. In order to generate sufficient concurrency, this leads to a high amount of per-task dependencies in the task graph. Hence, the FMM is used as demonstrator for irregular algorithms that exhibit dense task graphs.However, the concept of event-based task-parallelism is directly transferable to further tree-based algorithms with similar data dependency patterns such as Barnes-Hut tree codes and multi grid methods.

FMSolvr [1] is a C++-implementation of the FMM for CPUs that can be integrated into MD codes like GROMACS as header-only library. It was developed within the scope of the SPPEXA project GROMEX (GROMACS on the Exascale) [61]. The CPU parallelization of *FMSolvr* is based on *Eventify*.

1.3 Solution Strategy

Figure 1.1 provides an overview of the solution strategy to reach the research objective and answer the research questions **Q1** to **Q5**. The definition of the research objective is followed by a problem analysis that leads to a solution approach and its implementation. The subsequent evaluation covers reference implementations for a demonstrator application and their performance analysis.

The problem analysis is separated into two interacting paths; the analysis of hardware properties and the analysis of programming models.

First, we outline the classification of computer architectures with a focus on parallelization. CPU and GPU architectures are described with a focus on similarities and differences between both architectures. Subsequently, we conduct a comparative, quantitative analysis of hardware architecture trends for CPUs and GPUs; this answers research question **Q1**.

Second, we describe data- and task-parallelism as parallel algorithm models. Based thereon, we outline the application developer interfaces of OpenMP, OpenACC and CUDA, which are the most widely used on-node parallel programming technologies on current HPC systems. In addition, we provide an overview of OpenCL as parallel programming technology for heterogeneous systems. Its platform model provides a uniform terminology for CPU and GPU architectures that is employed to compare CPU and GPU features and serves as basis for the uniform architecture model introduced in this work. This is followed by a qualitative analysis of parallel programming model trends with a focus on unification efforts and task-parallelism; this provides insights on research question **Q2**.

The solution approach is based on the observed hardware properties and trends as well as the observed programming model trends. This helps to extrapolate the direction of future hardware and programming models and hence is supposed to develop a *sustainable* programming model. Hardware trends are incorporated by the uniform architecture model, while programming model trends are incorporated by the uniform execution model. The core part of this work extends the

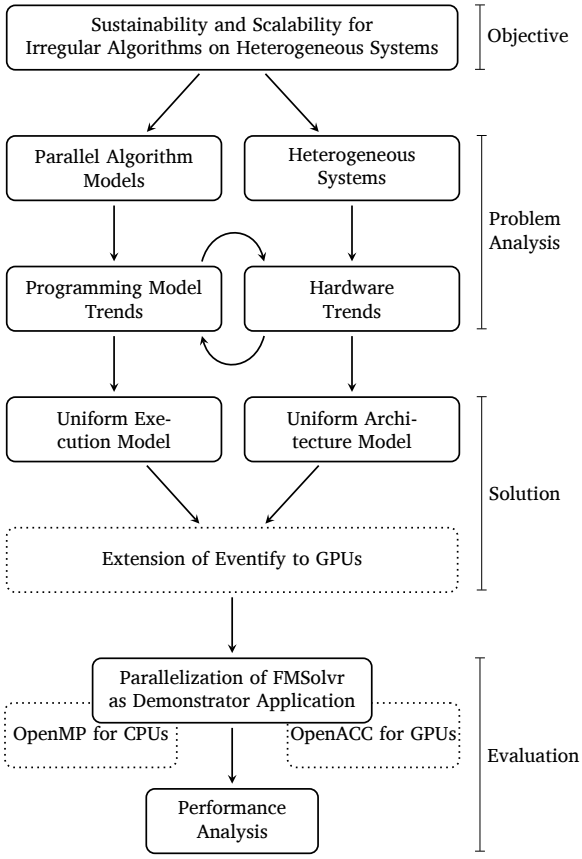


Figure 1.1: Solution strategy

event-based execution principle behind Eventify to GPUs. This includes the requirements and software architecture analysis, the development of several GPU queuing schemes and synchronization algorithms. The uniform architecture model and the uniform execution model are combined to retrieve a parallel programming model that provides a uniform view on heterogeneous hardware for application developers; this provides the answer to research question **Q4**.

The subsequent evaluation covers the following reference implementations of FMSolvr:

- Loop-based data-parallel OpenMP version for CPUs
- Loop-based data-parallel OpenACC version for GPUs
- Event-based task-parallel Eventify version for CPUs
- Event-based task-parallel Eventify version for GPUs

In a comparative performance analysis, research questions **Q3** and **Q5** are discussed.

Chapter 2

We are like dwarfs sitting on the
shoulders of giants.
Bernard of Chartres

Literature Review

The literature review is subdivided into an outline of parallel programming technologies that are most widely used in HPC today, and an overview of related work.

2.1 State of the Art

OpenMP (Open Multi-Processing) is a shared memory programming API specification for C, C++ and Fortran. It provides several compiler directives, runtime library routines and environment variables that enable the parallelization of application codes. Due to its convenient directive-based API, it is the de-facto standard for shared memory programming in HPC. With OpenMP 4.0 (2013), OpenMP started to support vectorization-based data-parallelism with the `simd` construct and accelerators with the `target` construct.

OpenACC (Open Accelerators) is an API specification that supports the parallel programming of heterogeneous systems in C, C++ and Fortran. It originated from the efforts to extend OpenMP to accelerators but was finally released as separate standard in 2011. Similar to OpenMP, it provides compiler directives, runtime library routines and environment variables that enable the parallelization of application codes.

The most widely used GPU programming technology on current HPC systems is Nvidia's CUDA. This is due to the high availability of Nvidia GPUs in current TOP500[97] systems; 139 out of 500 systems exhibit Nvidia GPUs. Thus, the reference implementation of the execution model for Eventify on GPUs is written in CUDA for reasons of compatibility and performance on current HPC systems. Furthermore, this work analyzes the effect of latest Nvidia GPU features, such as independent thread scheduling, on the adaption of task-parallelism to GPUs.

2.2 Related Work

2.2.1 Notions of Concurrency and Parallelism

First theoretical notions of concurrency and parallelism in programming can be found in the works of Dijkstra and Hoare on mutual exclusion [34], the Dining Philosophers Problem [32] [54] and Communicating Sequential Processes [54]. To the best of our knowledge, the earliest attempt to clearly distinguish both terms is formulated by Dijkstra: "parallelism [refers to] rather identical components, progressing 'in parallel', i.e. in rather strict synchronism. The term 'concurrency' only refers to the (possibility of) simultaneous activity" [33].

The subsequent definitions incorporate these ideas. Following [77, p. 288], "concurrency is a property of a program (at design level) where two or more tasks can be in progress simultaneously", while "parallelism is a run-time property where two or more tasks are being executed simultaneously". This notion is derived from [20, p. 266], defining concurrency as "[t]he capability of having more than one computation in progress at the same time. These computations may

be on separate cores or they may be sharing a single core by being swapped in and out by the operating system at intervals." Based thereon, [20, p. 3] conceives parallelism as a subset of concurrency. The fundament for these ideas is the abstraction of concurrency provided in [17, p. 18 ff.]. Even though these definitions are widely accepted [77, p. 288], they contain the language-owed ambiguity of "in progress" and "being executed".

Further, there are the definitions of parallelism and concurrency in the practical context of programming languages such as Go: "[C]oncurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations. Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once." [87]. This definition considers both, concurrency and parallelism, as execution properties while [20, p. 3] considers concurrency as design time and parallelism as execution time property. In addition to differentiating both terms based on their timing behaviors, it considers whether processes are independent or related to each other. This work, however, conceives concurrency and parallelism as conceptually separate from any (not timing related) process dependencies.

All of these definitions evolve around the concept of simultaineity. Simultainety is, in turn, "closely bound up with [the concept] of spatial separation" [55, p. 233] since two simultaneous actions physically cannot happen at the same location. This is also embraced by Flynn's taxonomy which classifies computer architectures based on process interactions that take place in *space- or time-multiplex* [41, p.948].

This work also relies on *simultaneity* as the core concept to distinguish parallelism from concurrency; see Definitions 3.2 and 3.4. The proposed definitions circumvent the language-owed ambiguity of *in progress* and *being executed*. Further, they avoid implementation-specific terms such as *core*, *operating system* and *programme* that would restrict their applicability to the highest level of parallel programming models only.

2.2.2 Classification of Parallel Architectures and Programming Models

Computer architectures can be classified by means of different properties such as concurrent processing capabilities, instruction level parallelism and memory organization. The Erlangen Classification System [51] is a quantitative classification that considers parallelism at the levels of program control units, ALUs and elementary logic circuits. Flynn's taxonomy [41] is a technology-independent classification of parallel computer architectures following their concurrent processing capabilities. Duncan's taxonomy [36] aims at a holistic view on parallel architectures and programming paradigms. Being based on the hard- and software reality of the 1980s, it introduces the classes *Synchronous*, *MIMD* and *MIMD paradigm*, each with subclasses for specific architectures or programming paradigms[1].

Similar to the objective of Duncan's taxonomy, this work aims at a uniform taxonomy for parallel soft- and hardware. Instead of introducing separate classes for both worlds, however, it aims to describe concurrent processing capabilities on all levels of software and hardware based on the same property, i.e. with the same classes of a taxonomy. Therefore, it revives the stream concept behind Flynn's taxonomy as a generalization that is abstract enough to describe concurrency not only in computer architectures (as is common practice) but also in parallel programming paradigms.

2.2.3 CPU-Managed Task-Parallelism on Heterogeneous Hardware

Task parallelism emerged as an alternative to tackle the limitations of loop-based data parallelism. The main goals are the reduction of sequential regions, synchronization phases and load imbalances by expressing an algorithm along its tasks and task dependencies, instead of artificially introduced loop-patterns. A taxonomy of task-parallel programming technologies for HPC is introduced by [105]. The taxonomy classifies task-parallel programming technologies based on four main

[1] As an example: [36, p. 6] explicitly introduces the subclasses *SIMD* and *vector* as part of *synchronous* to integrate pipelined vector processors into the taxonomy since it considers them "difficult to accomodate" in Flynn's taxonomy. However, this is redundant since pipelined vector processors are already covered by Flynn's taxonomy as time-multiplexing SIMD machines [41, p. 954, "The Pipelined Processor"].

Table 2.1: Task-parallel programming technologies in HPC according to the classification by [105, Table 1]. Only categories, aspects and technologies that are relevant for this work are included. Technologies marked with an "*" have been added for this work; values marked with "**" have been updated. The communication model is either shared memory ("smem") or global adress space ("gas"). Marker "i" refers to implicit support through the technology, while marker "e" refers to explicit, user-specified support. Possible supported graph structures are directed acyclic graphs ("dag"), trees ("tree") and arbitrary graphs ("graph"). Technological readiness is determined by means of the technology readiness levels as defined by the European Commission for HORIZON2020 projects [24]; levels range from "1 - basic principles observed" to "9 - actual system proven in operational environment". The implementation type is either library ("lib") or language extension ("ext").

	Communication Model	Distributed Memory	Heterogeneity	Graph Structure	Task Partitioning	Worker Management	Work Mapping	Synchronization	Technological Readiness	Implementation Type
OpenMP	smem	✗	i	dag	✗	e	i	i/e	9	ext
CUDA*	smem	✗	e	dag	✓	e	i	i/e	9	ext
HPX	gas	i	e	dag	✓	i/e	i/e	e	6	lib
Intel TBB	smem	✗	e**	graph**	✗	i	i	i	8	lib
Legion	gas	i	e	tree	✓	i	e**	i**	4	lib
Kokkos*	smem	✗	i	dag	✓	i	i/e	i	7	lib
Eventify*	smem	✗	✗	dag	✗	i	i/e	i	4	lib
Technology	Architectural			Task System		Management			Engineering	

categories that describe *architectural*, *task system*, *management* and *engineering* properties. Each category consists of multiple aspects. Table 2.1 provides an overview of the categories, aspects and technologies relevant for the classification and distinction of this work.

A task-parallel programming technology is considered as related work if it is compatible with scientific software written in C++. Furthermore, considered technologies must support heterogeneous systems as demanded by the research objective and allow for shared memory programming consistent with the scope of this work. For reasons of compatibility with different application codes and compilers, only library-based approaches are considered. Additionally, the language extensions CUDA and OpenMP are examined since both are the de-facto standards for parallel programming in HPC and used for reference implementations in this work. Task-parallel programming technologies are described with a focus on their GPU capabilities. The following paragraphs of this section extend the state of the art section provided in [50].

Originally, OpenMP supported loop-based data-parallelism on CPUs only. Hence, it was mainly applied to applications that exhibit highly regular parallelism, such as matrix- or vector-oriented computations. With increasing hardware parallelism and the advent of the first task-parallel programming technologies, such as Cilk [19] and Chapel [22], this was perceived as limitation. In OpenMP 3.0 (2008), the `task` and `taskwait` constructs were introduced to allow the definition of tasks; OpenMP 4.0 (2013) added support for task dependencies. Due to considerable task management overheads and programmability challenges, however, OpenMP's loop-based approach is still the most common parallelization pattern for scientific software.

CUDA supports task-parallelism by means of CUDA asynchronous task graphs. This allows for the definition of coarse-grained tasks in form of CUDA kernels and the definition of dependencies between these kernels.

HPX is a parallel runtime system that provides a user-friendly API for task-parallel programming on shared and distributed memory systems. With HPXCL [31], HPX also supports the asynchronous execution of tasks on Nvidia GPUs. However, GPU tasks have to be explicitly defined by the user in form of CUDA kernels. Kernel launch parameters such as grid size and thread block size also have to be explicitly defined by the user. Since the optimal values of these parameters are not necessarily independent from the GPU architecture, this may hurt performance portability.

Intel TBB is a C++-library that supports data and task-parallel programming on shared memory systems; it is the successor of Cilk and Cilk Plus [86]. In Intel TBB task-parallelism is expressed via the *flow graphs* interface that enables the description of arbitrary graphs, e.g. data-flow graphs or dependency graphs. Parallelism in flow graphs is exploited by a task scheduler that relies on split-join patterns. This introduces additional scheduling overhead and increases memory usage since tasks are created even if they are not ready to execute. By means of *asynchronous nodes* flow graphs can be used in combination with offloading to execute tasks in form of CUDA or OpenCL kernels on GPUs. Similar to HPXCL, this requires the user to write explicit GPU code and define kernel launch parameters.

Kokkos is a C++-library that provides a performance portable, user-friendly programming model for parallel programming on CPUs and GPUs. It provides CUDA, HPX, OpenMP and Pthreads as backend programming models. According to [37], Kokkos reaches 90% of the performance of application-specific parallelization approaches. Due to its general applicability, Kokkos does not take highly application-specific knowledge such as critical paths or customized task priorities into account. Kokkos does not only support offloading of GPU-tasks, but also enables dynamic task spawning on the GPU; see Section 2.2.4.

Legion is a data-centric programming model that enables the implementation of HPC applications for distributed, heterogeneous systems. In contrast to Intel TBB, HPX, Kokkos, OpenMP and Eventify, Legion does not require the application developers to explicitly express parallelism, e.g. in form of task graphs. Instead, application developers describe data properties in form of *logical regions* via a relational data model; based thereon, Legion implicitly extracts parallelism. Its execution model relies on recursive task-spawning. In contrast to any other considered technology, Legion furthermore requires the application developer to explicitly specify the mapping of logical regions onto target hardware via *mappers*. Hence, Legion is abstraction-wise diametrical to other task-parallel programming technologies.

2.2.4 GPU-Managed Task-Parallelism

OpenMP, Intel TBB, HPX and Legion rely on GPU tasks in the form of kernel functions that are offloaded to the GPU for execution. Therefore, they rely on the CPU for the management of GPU tasks. Launch parameters are configured on the CPU, dependencies are defined and resolved on the CPU and finally tasks are enqueued for execution on the GPU. This offloading-based approach is well-suited for task graphs with coarse-grained tasks and a small number of dependencies per task; especially, since all considered approaches provide asynchronous tasks to overlap dependency resolution and task execution. For task graphs with fine-grained tasks and a high number of dependencies, however, kernel launch overheads and dependency resolution overheads prevail. With a best case launch time of 5 µs for an empty kernel, this would allow for the execution of 200 empty tasks per millisecond. MD simulations, however, require 1000 to 10000 of compute tasks to be executed in the same time to achieve reasonable simulation times.

To overcome the limitations of offloading, the *PT* concept is researched in different application areas. In [73] PT are used to implement parallel programming primitives, [110] introduces a PT-based FFT, and [8] presents PT-based ray tracing. Gupta et al. [48] are the first to introduce a common definition of the PT paradigm. Furthermore, the authors identify four main use cases: load balancing, producer-consumer schemes, global synchronization and CPU-GPU synchronization. Load balancing, producer-consumer schemes and global synchronization are vital for GPU-managed task-parallelism to allow dynamic task generation and management. On heterogeneous systems, CPU-GPU synchronization is required to coordinate task scheduling between both processor types.

Tzeng et al. [108] are the first to describe PT-based task-parallelism for irregular workloads on

GPUs. The authors introduce a worksharing scheme with distributed queues for Reyes rendering as use case.Since Reyes rendering is based on a pipeline, each task is required to dynamically generate at most one new task, namely the task for the next pipeline stage. However, the approach does not support multiple, arbitrary dependencies between tasks.

Reference [107] is a follow-up work by the same authors and presents a task-parallel programming model that allows for complex task dependencies. They introduce a general-purpose dynamic scheduling approach that operates on a single, lock-based queue. Task dependencies are described via a look-up table that holds each task's dependencies to other tasks. For dependency resolution, each task is provided with a dependency counter that describes its outstanding dependencies. As soon as a task is finished, it decrements the dependency counters of its dependent tasks. If a task's dependency counter reaches zero, the task is enqueued for execution.

The dynamic scheduling approach is evaluated by applying it to the intra frame prediction in H.264 encoding, and to N-queens backtracking with $15 \leq N \leq 18$. The former exibits a task graph with maximally four dependencies per task, while the latter exhibits at least 15 dependencies per task due to N as branching factor. In comparison to a multi-threaded and vectorized CPU-encoder, the task-parallel GPU implementation decreases runtime by a factor of 3.6. The task-parallel N-queens backtracking, on the other hand, increases runtime on average by a factor of 1.7 in comparison to a multi-threaded and vectorized CPU-backtracking. Hence, the dynamic scheduling approach is well-suited for irregular work loads with sparse dependency graphs but introduces considerable dependency resolution overheads for work loads with dense dependency graphs.

PT-based task-parallelism is also provided by Kokkos (see [38]). Kokkos differentiates between sequential tasks *TaskSingle* and parallel tasks *TaskTeam*. With TaskSingle being executed by a single thread of a GPU warp only, and TaskTeam being executed in a data-parallel manner by all threads of a warp. Hence, performant code must be written warp-aware to avoid idling of the majority of threads. Tasks are generated via dynamic task spawning. Similar to the approach provided by [107], newly generated tasks are enqueued into a single, global task queue. Enqueueing and dequeueing of tasks is done by a single, dedicated thread per warp.

Whippletree [96] is the only technology that provides fine-grained task-parallelism on GPUs that is available open source. However, Whippletree does not support generic task graphs but only task graphs with one in- and one out-dependency per task. Therefore, this work considered extending Whippletree to support multiple dependencies per task as required by the FMM. However, this resulted in race conditions on current GPU architectures due to a change in the forward progress behaviour of latest GPU architectures.

Reference [29] provides a broad study on the performance portability of five mini applications, written in five parallel programming models across twelve architectures. The findings reveal that Kokkos provides the best performance portability for four out of five applications. However, OpenMP provides the best performance portability for the miniFMM[2]; therefore, the event-based approach considered in this work is compared against an OpenMP-based version of FMSolvr. The study reveals further that today "no robust and efficient task-parallel programming model exists for entirely on-GPU execution"[29]. Therefore, this work extends the event-based approach to GPUs and analyzes its performance.

2.2.5 Parallel Fast Multipole Methods

Due to the wide application of the FMM in MD, plasma physics and astrophysics, its parallelization is heavily researched on shared memory, distributed memory and heterogeneous systems. However, a comparison of this broad range of FMM applications, especially regarding performance and scalability, remains an open research question. This is due to the fact that diverse mathematical and technical variants of the FMM operators exist, which lead to different accuracy and performance behavior. For classification of the present work, we nevertheless provide a short outline on parallel FMM implementations with a focus on task-parallelism and GPU-versions. This section extends the state of the art section provided in [50].

Since OpenMP's loop-level parallelism is only applied locally to specific loops or loop nests, it is not aware of global algorithmic structures that may potentially provide more parallelism. Hence,

it introduces unnecessary sequential regions and load imbalances. According to Amdahl's Law, the parallel speedup of an application is limited by its sequential part. Therefore, unnecessary sequential regions are particularly critical when aiming at strong scaling. That such limitations apply for the FMM is shown in [14],[100] and [5]. This work confirms these findings for *FMSolvr* through the implementation and performance analysis of a loop-based OpenMP-parallelization.

ExaFMM [111] is an open source FMM-library for astrophysics that supports shared memory systems via OpenMP, distributed memory systems via MPI as well as GPUs via CUDA. It aims at scaling large particle simulations with billions of particles to exascale systems. For a simulation with 10^8 particles the authors report a strong scaling efficiency of 93% on 2048 processes. This result is highly promising for the FMM in particular, as well as hierarchical algorithms in general, to reach excellent scalability on exascale systems. In [3] the task-parallel programming approaches Cilk Plus, Intel TBB and OpenMP Tasks are applied to parallelize ExaFMM. The performance analysis reveals that Intel TBB perfectly scales up to 64 cores on Intel's Knights Landing Xeon Phi for 10^8 particles. In [71] a data-driven CPU-implementation of ExaFMM with the runtime system QUARK [71] is described. For particle ensembles with 10^7 particles the approach leads to linear speed-up on 16 cores.

ScalFMM [7] is a parallel, C++-based FMM-library. It is a kernel independent FMM, while FMSolvr is specialized on kernels with spherical harmonic expansions. Since this may have an impact on the parallelization approach and its performance, this complicates a direct comparison of both implementations. Similar to the research objective of this work, the main objectives of ScalFMM's software architecture are maintainability and understandability. A lot of research about task-based and data-driven FMMs is based on ScalFMM. The authors devise the parallel data-flow of the FMM for shared memory architectures in [7] and for heterogeneous systems in [6]. For the StarPU-based ScalFMM in [7] strong scaling with a parallel efficiency of 91% is reached for a sufficiently large particle ensemble with $2 \cdot 10^8$ particles on CPUs. As follow-up work, [7] extends the StarPU-based ScalFMM to GPUs by providing highly-tuned CUDA kernels for the two most compute-intensive steps of the FMM. Furthermore, it provides a dynamic scheduling strategy to distribute work between CPU cores and multiple GPUs; scheduling and dependency resolution are done by the CPU only.

The idea of breaking the stages of the FMM into smaller tasks to improve load balancing is furthermore applied and analyzed in [112], [14] and [7]. HPX-based implementations of the FMM for CPUs are described in [112] and [64].

All considered works focus on the efficient computation of large, often inhomogeneous, particle ensembles with millions of particles. This leads to particular challenges regarding memory footprint, scheduling policies and communication patterns for the applied parallelization approaches. In this context, the considered works show that task-parallelism with recursive task spawning is efficient for compute-bound simulations on CPUs. The focus of this work, however, is on the efficient computation of small, homogeneous particle ensembles. Hence, the overhead of the applied parallelization approaches cannot be hidden through computational work, but must be reduced to a minimum. On GPUs, this is especially hard to achieve since their architecture heavily relies on latency hiding. This work analyzes whether event-based task-parallelism can help to reach similar scalability results for small particle ensembles on CPUs and GPUs.

Full GPU-implementations of the FMM, i.e. implementations that compute near and far field on the GPU, are rare. A full GPU-implementation of FMSolvr is provided by [66], and a full GPU-implementation of the miniFMM is provided by [2]. Both implementations are based on CUDA and execute all steps of the FMM on the GPU, instead of the most compute-intensive or inherently data-parallel ones only. Even though OpenACC is widely used to parallelize scientific software, there is no OpenACC-based implementation of the FMM up until now. This work introduces a loop-based OpenACC implementation of FMSolvr for comparison against the event-based approach on GPUs.

To the best of our knowledge, GPU-managed task-parallelism for the FMM was researched by [2] only. The authors propose a Kokkos-based implementation of the miniFMM [2] and compare it to a CUDA-based implementation. Based on runtime measurements for a simulation with 10^7 particles, the Kokkos-based implementation is 2.8× slower. Kokkos tasks require 200 registers per thread, while each CUDA kernel requires only 80 registers per thread. Based thereon, the authors

assume high register pressure as one reason for the performance difference. Further potential bottlenecks are Kokko's restriction to a single thread block per streaming multiprocessor and the usage of a single, global queue.

Similarily to Kokkos, this work follows a PT-based approach. Instead of relying on recursive task spawning, however, this work extends the programming and execution model of event-based task-parallelism to GPUs.

Chapter 3

Concurrency is the next major
revolution in how we write software.
Herb Sutter

Concepts of Concurrency

This chapter introduces definitions for *concurrency* and *parallelism* and relates both terms with each other. Further, it proposes the *stream interaction model* as common foundation for the description of concurrency in software and hardware. Additionally, it outlines the general steps of parallelization. This work combines these concepts to derive consistent, complementary and applicable definitions of the particular components of parallel programming models in theory and practice. This allows for the coherent description of commonalities, differences and dependencies between:

1. Models of *different categories*, e.g. to answer questions like *Which parallel algorithm models does the parallel programming model OpenACC support?* (see Section 5.5.4) or *Which requirements must be fulfilled by an execution model to support loop-based algorithm models?* (see Section 5.2).

2. Models of the *same category*, e.g. to answer questions like *What do the SMT mechanism on CPUs and the latency-hiding mechanism on GPUs have in common?* (see Section 4.4.4) or *What is the qualitative difference between loop-based parallelism and vectorization-based parallelism?* (see Section 5.1).

This supports a mapping between high-level algorithmic and low-level architectural parallelism that is consistent with the technical specifications of the applied programming models. In Chapter 4, it enables the description of CPU and GPU architectures in a similar way and allows to draw analogies between both. The same holds true for the description of parallel programming models in Chapter 5. Based thereon, it allows for the transfer of the task-parallel programming model from CPUs to GPUs in Chapter 6.

To reach a common foundation for the definition of parallel computing models, this work relies on the following two hypotheses:

1. Concurrency in algorithms and architectures can be expressed in the same manner by refining the stream-based model behind Flynn's Taxonomy.

2. Parallel programming models are based on the general steps of parallelization as described in [27, p. 97] and [46, p. 85].

3.1 Parallelism and Concurrency

As outlined in Section 2.2.1, the definitions of the terms *parallelism* and *concurrency* typically depend on the considered literature and application context. And, especially in practice, might be defined rather informally and even be used interchangeably. This work, however, requires

a clear distinction between those (and related) concepts. The subsequent paragraphs state the assumptions about *processes* and *systems* that hold true for all proposed definitions.

A *process* is considered in the general sense of a sequential stream of operations that transforms some input into some output. In the context of programming models this maps i.a. to operating system processes, threads, MPI ranks, SIMD lanes or warp lanes. Other from that, no further assumptions are made about process properties, i.e. it is neither specified whether two processes are data dependent or independent, nor whether a process exhibits a finite or infinite execution time. This is reasonable since processes with different sets of properties are considered throughout this work, but the notion of execution orders is required to be the same for all of them.

A *system* consists of processes, execution units and schedulers. The execution order of processes in a system is subject to the available execution units and the effective scheduler. Exemplary, a system might be a single process with multiple kernel threads which is executed on a multicore processor and subject to the scheduler of the operating system; or, a system might be a CUDA kernel on a grid of CUDA threads which is executed on a GPU and subject to the thread engine and warp schedulers.

This work considers *consecutiveness*, *concurrency*, *interleaving* and *parallelism* as properties of the execution order of exactly two processes A and B (in contrast to a set with any number of processes). This is reasonable since different processes within the same system, e.g. CPU and GPU threads, might exhibit different execution orders with respect to each other. Nevertheless, the provided definitions are straightforwardly applicable to systems with more than two processes by considering all unordered pairs of processes in the system. As an example, consider the processes X, Y and Z. X, Y and Z are said to be parallel if all permutations (i.e. XY, XZ, YZ) are parallel.

Based on these preliminaries, consecutiveness is defined as follows:

Definition 3.1. ***Consecutiveness.*** *A scheduler guarantees* consecutive *execution of two processes A and B iff the execution of B must not start before the termination of A, and vice versa.*

Informally speaking, A and B are executed strictly one after another. Consecutiveness prohibits the execution of multiple processes within overlapping time intervals. Concurrency, to the contrary, allows for the execution of multiple processes within overlapping time intervals:

Definition 3.2. ***Concurrency.*** *A scheduler permits* concurrent *execution of two processes A and B iff the execution of B may start before the termination of A, and vice versa.*

Informally speaking, two processes are concurrent if they are executed within overlapping time intervals. This can be achieved via two mechanisms: interleaved execution or parallel execution. Therefore, both mechanisms are conceived as special cases of concurrency, which is reflected by their subsequent definitions.

Definition 3.3. ***Interleaving.*** *A scheduler implements the* interleaved *execution of two concurrent processes A and B iff A and B are executed alternatingly on the same execution unit.*

Informally speaking, interleaving refers to multiple processes executing in turns on a single execution unit. Therefore, interleaving can be considered as concurrency via time-multiplexing.

Definition 3.4. ***Parallelism.*** *A scheduler implements the* parallel *execution of two concurrent processes A and B iff A and B are executed simultaneously on different execution units.*

Informally speaking, parallelism refers to multiple processes being executed at the same time, which requires a dedicated execution unit per process. Consequentially, parallelism can be considered as concurrency via space-multiplexing, which is also further considered in Section 4.1.

3.2 General Steps of Parallelization

In general, the parallelization of an algorithm can be subdivided into four coarse steps [27, p. 97]: decomposition, assignment, orchestration and mapping.

3.2.1 Decomposition

Decomposition is the division of an algorithm into smaller work units in order to induce concurrency. The decomposition of an algorithm into work units depends on its *data dependency graph*.

Definition 3.5. ***Data Dependency Graph.*** *A data dependency graph of an algorithm represents the transformation of data elements in the course of this algorithm, with nodes representing data elements, and edges representing operations.*

For an example of a data dependency graph, please refer to Section 7.3.

Task Graphs

Decompositions can be expressed in form of task graphs that model the computational dependencies between tasks. In general, a *task* is an arbitrarily defined unit of work [27, p. 96]. As defined by Robert [92], a *task graph* is the partitioning of an algorithm into tasks, with nodes representing tasks and edges representing dependencies. The decomposition of an algorithm into tasks is restricted by its data dependency graph. However, the correspondence between an algorithm and its task graph is not necessarily bijective; usually, multiple partitionings of an algorithm into tasks exist. For one thing, because the algorithm itself or the employed data structures may be varied, which leads to varying tasks and task dependencies. For another, because the definition of a task depends on the required task granularity as well as the inherent parallelism of the task.

Regarding inherent parallelism, we distinguish between sequential and parallel tasks. A *sequential task* is executed by a single instruction stream, while the work of a *parallel task* is executed concurrently by multiple instruction streams.

Quantifying the notions of [89, p. 4] and [46, p. 89], the subsequent definition of *granularity* is proposed:

Definition 3.6. ***Granularity.*** *The granularity g is the relation between the number and size of tasks a computation is decomposed into. It holds $1 \leq g \leq N$, with N being the number of instructions of the computation. A decomposition A is* fine-grained *in comparison to a decomposition B if its granularity is higher. A decomposition A is* coarse-grained *in comparison to a decomposition B if its granularity is low.*

A set of tasks is *fine-grained if its tasks contain only a few instructions relative to the overall workload* [75, p. 14]. Analogously, a set of tasks is *coarse-grained if its tasks contain many instructions relative to the overall workload* [75, p. 14]. Hence, a task can be as large as a function or as small as a single instruction.

Being related to granularity, the *degree of concurrency* is a metric that quantifies the parallelization potential of an algorithm. Slightly adapting the definition of the maximum degree of concurrency provided by [46], the degree of concurrency is defined as follows:

Definition 3.7. ***Degree of Concurrency.*** *"The […] number of tasks that can be executed simultaneously in a parallel program at any given time is known as its […] degree of concurrency."[46, p. 89 f.]*

In naturally parallel algorithms, such as vector additions, the maximum degree of concurrency corresponds to the total number of possible tasks, i.e. a single task per scalar addition. Except for this class of algorithms, however, the task graph of an algorithm contains computational dependencies between the tasks. Therefore, the maximum degree of concurrency is typically less than the total number of tasks. With increasing granularity (i.e. finer-grained decomposition), the degree of concurrency increases since more tasks, which can be executed concurrently, are generated. However, the degree of concurrency cannot be increased indefinitely since it is limited by the amount of operations an algorithm consists of. Furthermore, the practically reasonable degree of concurrency is not necessarily the maximum degree of concurrency since a higher degree of concurrency also increases synchronization overheads. Hence, if the granularity is too high (i.e. tasks are too small) the parallel efficiency is decreased.

Decomposition Techniques

To generate task graphs that induce concurrency, several decomposition techniques are available. Common decomposition techniques in scientific simulations are *recursive decomposition*, *data decomposition* and *functional decomposition*. In order to parallelize an algorithm efficiently, the decomposition technique must be chosen in alignment with the prevalent type of parallelism that the algorithm exhibits, i.e. *data parallelism*, *recursive parallelism*, *functional parallelism* or *task parallelism*, a mixture thereof. Further, decomposition techniques can be implemented via different parallel algorithm models (see Section 5.1) that have to be chosen in alignment with the available hardware architectures.

Recursive decomposition

Recursive decomposition is used to induce concurrency in algorithms that are based on a divide-and-conquer strategy [46, p. 95], i.e. algorithms that exhibit *recursive parallelism*. This decomposition technique recursively subdivides a problem into similar sub-problems that can be solved concurrently. Recursive decomposition is the decomposition technique behind recursive task spawning as applied by diverse task-parallel programming technologies (see Section 2.2).

Data decomposition

Data decomposition is used to induce concurrency in algorithms that are based on large, regular data sets [46, p. 97], i.e. algorithms that exhibit static *data parallelism*. By applying the data decomposition technique, a data set is partitioned into chunks of data. From the resulting data partitioning, a partitioning of computations into tasks is derived. Based thereon, data decomposition describes the concurrent execution of identical operations on multiple data elements. Typical examples include dense matrix multiplication, scalar multiplication and vector addition.

Data-parallelism can be applied via diverse parallel algorithm models such as loop-level parallelization and SIMD vectorization. In parallelism-wise more complex algorithms, the parallel control pattern *fork-and-join* (see [72, p. 88]) can be applied to exploit data parallelism. For this purpose, an algorithm is subdivided into data-parallel phases that are intermitted by synchronization phases.

Functional decomposition

Functional decomposition is used to induce concurrency in algorithms that require the computation of multiple, independent functions, i.e. algorithms that exhibit *functional parallelism*.

Task-Graph decomposition

If the efficient parallelization of an algorithm requires a combination of any of these approaches, the algorithm is generally said to exhibit *task parallelism*. Lastly, any type of parallelism can be expressed by means of task graphs, e.g. trivial data-parallelism and functional parallelism as edgeless graphs, and recursive parallelism as tree.

3.2.2 Assignment

Assignment refers to the assignment of tasks to logical execution streams, e.g. threads, processes or SIMD-lanes. In this step of parallelization, the goal is to distribute the workload between execution streams as equally as possible. Equal workload distribution supports the reduction of synchronization-induced idle times and hence improves parallel efficiency. To achieve this, load balancing and scheduling approaches in terms of work sharing and work stealing are applied.

The interpretations of *work sharing* and *work stealing* as used in this work are derived from [18]; even though, this work originally refers to process scheduling instead of task scheduling. Here, work sharing is a static load balancing method in which a load balancer assigns tasks to a specific execution stream based on a static schedule. Work stealing, on the other hand, is a dynamic load balancing method in which idle execution streams steal tasks from busy execution streams.

3.2.3 Orchestration

Orchestration is the fundamental concept behind process interaction. Orchestration enables the coordination of data exchange between processes and hence is the basis for cooperation

and communication. Cooperation refers to implicit data exchange via shared variables, while communication refers to explicit data exchange via message passing.

3.2.4 Mapping

Mapping refers to the mapping of execution streams to execution units, i.e. processors or cores. Which system component is responsible for mapping, depends on the applied programming model. Mapping can either be done implicitly by the operating or runtime system via the process scheduler, or be specified explicitly by the software developer or user via mechanisms like thread or process pinning.

Dependent on the hardware architecture, mapping may influence the communication or memory latency between execution streams. Hence, mapping schemes influence parallel efficiency. This holds true for threads on different NUMA-nodes regarding CPU architectures as well as for thread blocks on different SMs on GPU architectures.

3.3 Stream Interaction Model

The stream interaction model aims for the same concurrent processing capabilities (just on a different level of the model hierarchy) to be explained with the same concept. This section revives and extends the stream concept behind Flynn's Taxonomy to classify concurrent systems in general, and not only hardware architectures in particular. Originally, Flynn's Taxonomy [42] was introduced as a technology-independent classification of computer organizations based on their concurrent processing capabilities. In order to quantify these capabilities, the so-called stream concept was introduced in [42, p. 1902] and refined in [41, p. 948]. Following the latter and adapting it to the terminology of this work, a stream can be considered as "a sequence of items (instructions or data) as executed or operated on by a [processing unit]". Based thereon, [41, p.948] classifies computer organizations *by the magnitude (either in space or time multiplex) of interactions of their instruction and data streams*. This definition immediately leads to the following four classes of computer organizations:

- Single-Instruction Stream, Single-Data Stream (SISD)
- Single-Instruction Stream, Multiple-Data Stream (SIMD)
- Multiple-Instruction Stream, Single-Data Stream (MISD)
- Muliple-Instruction Stream, Multiple-Data Stream (MIMD)

Please note the term *stream* in all of these classes. In literature canon, this term is occasionally omitted. This, however, is a source for confusion (cf. [99, p. 588], [89, p. 13]) since it neglects the fact that a data stream is typically not a single data element but a sequence of multiple data elements. This differentiation is vital for the powerfulness of Flynn's Taxonomy and the subsequent considerations.

Up until today, Flynn's Taxonomy is the most widely used approach for the classification of computer organizations regarding hardware concurrency. This work, however, aims to classify concurrency in algorithms and architectures via the same taxonomy to enhance the alignment of parallel software to hardware.

With minimal adaptations to and formalization of the stream definition and the classification criterion, the abstractness of the stream model allows for both - the classification of architecture models, and the classification of algorithm models. In order to adapt the definition of instruction and data streams, an understanding of the relation between the terms *instruction*, *data element*, *operation* and *process* is required. Generally, an operation is considered as an operator that is applied to a (possibly empty) set of operands. Following this notion, an operation is here considered as a container that consists of an instruction (the operator) and a set of data elements (the operands). Based thereon and as stated in Section 3.1, a process is a sequence of operations. With this understanding in mind, the following definitions are proposed:

Table 3.1: Stream interaction schemes based on Flynn's Taxonomy. For each stream interaction scheme the number N of instruction streams, the number M of data streams and the resulting number $I = \max(N, M)$ of stream interactions is provided.

Interaction Scheme	N	M	I
SISD	1	1	1
SIMD	1	n	n
MISD	n	1	n
MIMD	n	n	n

Definition 3.8. ***Instruction Stream.*** *An instruction stream I_l is a sequence of instructions $i_1, i_2, \dots, i_k$.*

Regarding architectural models, an instruction stream can accordingly be interpreted a sequence of instructions that is executed by a processing unit. Regarding algorithm models, on the other hand, an instruction stream can be seen as a sequence of instructions that is contained in subsequent operations of a program.

Definition 3.9. ***Data Stream.*** *A data stream D_l is a sequence of data elements $d_1, d_2, \dots, d_k$.*

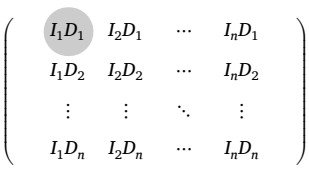
Similarly, from the architectural perspective, a data stream is a sequence of data elements as operated on by a processing unit. And, from the algorithmic perspective, a sequence of data elements as contained in subsequent operations of a program.

It is vital to note that both definitions do not impose any restrictions on the origin of the data elements or instructions in a stream. Therefore, a data stream must not necessarily consist of independent data elements but can also consist of data elements that are derived from a data element of the same stream through a previous instruction. The concept of derived streams was en passant mentioned in the context of MISD architectures in [42, p. 1908] and partially specified further in [41, p. 949]. For reasons of consistency, however, the concept of derived streams should not only be applicable to MISD (for whose understanding it is vital, as outlined below) but to all classes of the taxonomy. Further, it is the notion of derived streams that allows for output values to be used as input values. Without the concept of derived streams, the stream interaction model would exclude the modeling of concurrency in iterative programs since they rely on accumulation variables whose output value is their successive input value.

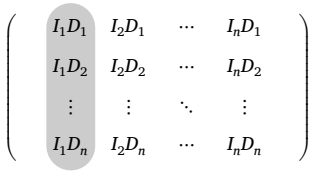
Based on the multiplicity of interactions between instruction and data streams, the stream interaction schemes provided in Table 3.1 are used for the classification of concurrent systems. Each stream interaction scheme consists of a multiplicity of data streams and a multiplicity of instruction streams, with multiplicities noted as *single* or *multiple*). The retrieved stream interaction schemes correspond to Flynn's classes of computer organizations since the generalization of the stream concept does, as required, not change the number of possible multiplicity-stream combinations. Here, the number of instruction streams is denoted as N, the number of data streams is denoted as M and the resulting number of stream interactions is defined as $I = \max(N, M)$.

$I = \max(N, M)$ dissents Flynn's original work that implicitly allows $I > \max(N, M)$ and others that explicitly define $I = N \times M$. For instance, a MIMD architecture is considered quantifiable by "specifying [...] the number of instruction streams per data stream, or vice versa."[41, p. 949], what implies that there could potentially be more than one instruction stream per data stream, i.e. $I > \max(N, M)$. This assumption, however, prohibits the description of pairwise disjoint classes and hence violates the orthogonality criterion of classes in a taxonomy. As a consequence, MIMD would have to generally model the properties of SIMD and MISD in terms of expecting - potentially every - instruction stream to operate on every data stream or vice versa. In this work, this would especially complicate the separate description of SIMD-capabilities on MIMD-architectures as required in Chapter 4.

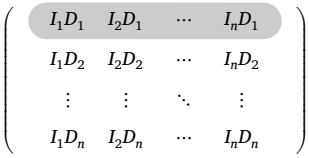
Figure 3.1 shows the interaction matrix between $N = M$ interaction and data streams for each interaction scheme. For each interaction scheme the according interactions are highlighted. Non-highlighted interactions do actually not exist in the according schemes and are just shown for



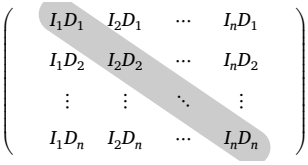
(a) SISD: Each single entry corresponds to a SISD system.



(b) SIMD: Each column vector corresponds to a SIMD system.



(c) MISD: Each row vector corresponds to a MISD system.



(d) MIMD: The vector of diagonal elements corresponds to a MIMD system.

Figure 3.1: Matrix of all possible interactions between $N = M$ interaction and data streams. For each interaction scheme an example set of interactions is highlighted. Non-highlighted interactions are just shown for visual comparability between schemes.

comparability between schemes regarding interaction multiplicity and interaction types. SIMD and MISD, for example, exhibit the same number of stream interactions. Qualitatively, however, SIMD and MISD exhibit different interaction types. This is reflected by the interaction matrix in which SIMD appears as a column vector and MISD appears as a row vector.

In a SISD scheme, there is only a single interaction. Namely, the interaction between instruction stream I_1 and data stream D_1.

A SIMD scheme consists of n interactions. Namely, the interactions of a single instruction stream I_1 that operates on n different data streams $D_1, D_2 \dots, D_n$.

A MISD scheme consists of n interactions and is hence similar to a SIMD scheme regarding interaction multiplicity. In contrast to SIMD, however, MISD covers the execution of n different instruction streams $I_1, I_2, \dots, I_n$ on a single data stream D_1. Please note that this does by no means imply that different instructions have to be executed in parallel on the same data element. Instead, the instruction streams $I_1, I_2, \dots, I_n$ operate on derived data elements of data stream D_1. Meaning, I_1 operates sequentially on the data elements $d_1, d_2, \dots, d_k$ resulting in $d_1', d_2', \dots, d_k'$ and I_2 operates on the data elements $d_1', d_2', \dots, d_k'$ resulting in $d_1'', d_2'', \dots, d_k''$. In literature, the concept of derived streams is mostly omitted. This, however, leads to the conclusion that MISD architectures are a theoretical construct that does not exist in practice (cf. [89, p. 13], [99, p. 587]). Based on the above notion of derived data elements, however, MISD architectures do indeed exist. Please see Section 4.1 for examples.

The MIMD interaction scheme consists of n interactions. Namely, the interactions between n instruction streams $I_1, I_2, \dots, I_n$ and n data streams $D_1, D_2, \dots, D_n$, with each instruction stream operating on exactly one data stream. Hence, the MIMD scheme corresponds to the diagonal elements of the interaction matrix.

The stream interaction model is applied to classify parallel architecture models in Section 4.1 and parallel algorithm models in Section 5.1 since these are the lowest and highest levels of abstraction within a parallel programming model.

Chapter 4

Now that Moore's Law is delivering less, it's time for the architects and software engineers to start delivering more.
Steve Furber

Computer Architectures

This section outlines the classification of computer architectures based on the stream interaction model introduced in Section 3.3. Further, it briefly describes the architectural properties of CPUs and GPUs. Conclusive, it elaborates on the similarities and differences between both processor types. These considerations serve as foundation for the uniform architectural model in Chapter 6.1.

4.1 Architecture Models

This section classifies computer architectures based on their concurrent processing capabilities following the stream interaction model and hence in accordance with Flynn's taxonomy. Here, the theoretical number of concurrent stream interactions a hardware architecture supports is considered. First, a mapping between the stream interaction model and the main components of common computer architectures is derived.

Since architectures can be considered on different abstraction levels, concurrency also appears on different abstraction levels:

- **Cluster Level (C)**: A cluster consists of compute nodes.
- **Node Level (N)**: A compute node consists of processors and accelerators.
- **Processor Level (P)**: A processor consists of compute cores, cache and memory.

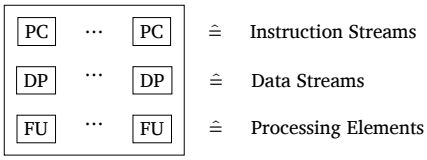


Figure 4.1: Mapping between the ISA level architecture model of a hypothetical processor (left) and the terminology of the stream interaction model (right). To maintain an instruction stream, an architecture requires at least a program counter. To maintain a data stream, it further requires at least a data path, i.e. the set of input registers that store operands. In order to process the interaction between an instruction and a data stream, it must at least exhibit a single functional unit.

- **Instruction Set Architecture Level (ISA)**: A core consists of control units, functional units and registers, which are connected via data, address and control busses; these elements are directly or indirectly controllable by the software developer via the instruction set.
- **Micro-architecture Level (MA)**: Functional units consist of logic circuits.

The stream interaction model can be applied on each particular level as can be seen from the examples in Table 4.1. As can be seen from the level descriptions, the higher levels are based on the lower levels. Hence, the overall concurrent processing capability of a computer system, i.e. the number of stream interactions it supports, is the product of the numbers of stream interactions each of its levels supports. As an example: imagine a cluster with 10 compute nodes. On the cluster level, the cluster can execute 10 processes in parallel. If each compute node, however, consists of 2 processors, the cluster can process $10 \cdot 2 = 20$ processes in parallel at the node level. If further each processor covers 8 cores, the cluster can handle $10 \cdot 2 \cdot 8 = 160$ processes at the processor level.

Based on the considered abstraction levels, we have to identify the smallest functional elements of hardware architectures that are required to maintain a) an instruction stream, b) a data stream and c) a stream interaction in the sense of the stream interaction model. The subsequent considerations to identify a), b) and c) refer to the ISA level, unless explicitly stated otherwise. As interface between hardware and software, the ISA is the basis for higher level programming languages. Therefore, it is the most suitable level for mapping hardware concurrency to software concurrency.

Figure 4.1 shows the ISA level architecture model of a hypothetical hardware architecture. The model comprises only the common components of CPUs and GPUs that are relevant for the mapping between the stream interaction model and those architecture features that are reflected by the ISA, i.e. are implicitly or explicitly exposed to the software developer.

Each architecture exhibits a set of registers that stores the processing state of a program. This includes an instruction register that holds the current instruction as well as a program counter that holds the address to the next instruction. Following the notions of [99, p. 587] on Flynn's Taxonomy, the program counter is the smallest functional element of an architecture that is required to maintain an instruction stream[1]. Thus, the number N of instruction streams in the stream interaction model corresponds to the number of program counters.

Furthermore, an architecture consists of data registers and one or multiple functional units. Each functional unit is connected to a number of data registers via a dedicated data path. A data path is the connection to a set of input registers that contain the source operands for a single scalar operation (either stand-alone or as part of a vector operation). Hence, a data path is considered as the smallest functional element of an architecture that is required to maintain a data stream. The number of data paths, in turn, corresponds to the number M of data streams in the stream interaction model.

Here, a functional unit is considered as the logic circuit that processes a single scalar operation (either stand-alone or as part of a vector operation). As functional units are the hardware implementation of an instruction and are supplied with data via data paths, they are the physical interaction points between instruction and data streams. Hence, a *functional unit* processes stream interactions and accordingly corresponds to a *processing element* in terms of the stream interaction model. Thus, the number F of functional units determines the number of stream interactions that can take place in parallel.

In the context of architectures as physical instances of the stream interaction model, the time- and space-multiplexing aspect of stream interactions should be revisited to consider *when* and *where* interactions are processed.

[1] Following Flynn's original work, instruction units [42, p. 1908] would correspond to instruction streams. However, both terms are rather broad and may consist of components that exhibit concurrency themselves, e.g. a control unit may feature an instruction pipeline, an out-of-order unit or even control multiple program counters (see warp scheduler of the Nvidia V100 in Section 4.3). Due to these peculiarities, this work classifies concurrent processing capabilities for every abstraction level separately.

Table 4.1: Classification of architecture models following the stream interaction model.

Interaction scheme	Multiplexing	Model				Example (Level)
		N	M	I	F	
SISD	n/a	1	1	1	1	Single-core CPU (P)
SIMD	Space	1	n	n	n	Array Processor (ISA) Associative Processor (ISA) CPU with vectorization (ISA)
	Time	1	n	n	1	Pipelined Vector Processor (ISA)
MISD	Space	n	1	n	n	Hard-wired rendering-pipeline (P) FPGA (MA) Pipelined Vector Processor (MA) Instruction Pipelining (MA)
	Time	n	1	n	1	Intrinsic Multiprocessing (C)
MIMD	Space	n	n	n	n	Multi-core Processor (P) Multi-processor node (N) Multi-node cluster (C) Superscalar execution (MA) GPU with universal shaders (P)
	Time	n	n	1	1	SMT (MA)

Time-multiplexing refers to stream interactions proceeding consecutively at the same functional unit. Time-multiplexing can be used to enable the sharing of compute resources between multiple processes. If single processes do not provide enough work load to use functional units to the full it allows to increase the utilization of resources and hence increases overall throughput. While this setup enables the *interleaved* (Definition 3.3) execution of processes, it does not support *parallel* execution (Definition 3.4).

Space-multiplexing, on the other hand, refers to stream interactions proceeding in parallel at different functional units. Space-multiplexing enables an architecture to execute multiple processes in parallel and hence allows to decrease overall runtime. Figure 4.2 provides an overview of all possible architecture schemes for time- and space-multiplexing.

Table 4.1 provides an overview of the architecture models behind each interaction scheme. For each interaction scheme, it is considered whether interactions take place in a time-multiplexing or in a space-multiplexing manner. Furthermore, the number N of instruction streams (program counters), the number M of data streams (data paths) and the resulting number I of interactions is provided in accordance with Table 3.1. Dependent on the multiplexing mode, the number F of functional units is provided. For space-multiplexing, it corresponds to the number of I of interactions. For time-multiplexing, it corresponds to 1 since all interactions are processed at the same functional unit. For each interaction scheme, typical architecture examples are given.

Subsequent sections elaborate on the properties of the examples provided in Table 4.1. They demonstrate that the stream interaction model can be applied to classify concurrent processing capabilities on different architecture levels. Hence, the stream interaction model allows for the *same principle* (just on a different level of the model hierarchy) to be explained with the *same concurrency* concept.

4.1.1 SISD

The SISD architecture model describes the operating principle of sequential computer models such as the von-Neumann or Harvard architecture. It covers a single functional unit that executes instructions from a single instruction stream on the data elements of a single data stream. Since there is only one stream interaction to process, there is no distinction between time- and space-multiplexing. Regarding the core level, desktop computers - as single-core uniprocessor machines

- were a classical example for SISD machines until the introduction of the first dual-core desktop CPUs in 2005. More recent examples for SISD architectures are most of today's microcontrollers and single-core processors of small single-board computers, such as the Raspberry Pi Zero. While these processors do not exhibit parallelism at the core level, they may still exhibit concurrent processing capabilities at the ISA or MA level, e.g. in the form of instruction pipelining or superscalarity. This is in accordance with [43, p. 692] that classifies single-core architectures as SISD and nevertheless allows them to exhibit instruction level parallelism.

4.1.2 SIMD

The SIMD architecture model consists of several processing elements that each execute the same instruction stream on multiple data streams concurrently. This includes array processors as introduced by the Solomon project [93] (1962) and its successor ILLIAC IV [15] (1968), as well as early pipelining-based vector processors such as the CRAY-1 [91] and the Connection Machines [53]. Array processors are the archetype of space-multiplexing SIMD processors since they execute a single instruction stream on multiple data elements at different functional units simultaneously (see [41, p. 954]). As variation of the array processor, associative processors belong in this category, too. In an associative processor, the functional units process an operation only, if a general condition is satisfied for their data stream, otherwise, they stay idle. Vector processors, on the other hand, are the archetype of time-multiplexing SIMD processors since they execute the instructions of a single instruction stream on multiple data elements consecutively at the same functional unit (see [41, p. 954]). Hence, in order for vector processors to reduce parallel runtime, they have to provide some form of parallelism on the MA level (see Section 4.1.3).

4.1.3 MISD

The MISD architecture model consists of several functional units that each execute a different instruction stream on the same (derived) data stream. From the space-multiplexing point of view, MISD architectures can be considered as pipelines in which (derived) data elements are passed from one functional unit to the next (see [46, p. 74]). Firstly, pipelining is the form of MA level parallelism provided by Pipelined Vector Processors. Considering the MA level, they can hence be considered as space-multiplexing MISD architectures. Secondly, this principle is reflected by GPUs that exhibit a hard-wired rendering-pipeline in which data elements flow through multiple stages: from the geometry shader to the vertex shader right through to the fragment shader and the frame buffer. Thirdly, this principle can also be found in data-flow architectures such as FPGAs, when considering a single path through a hardware-configured data flow graph (see [43, p. 695]). On such a path, a data stream flows from operator to operator. From the time-multiplexing perspective, intrinsic multiprocessing (IMP) systems can be seen as MISD architectures (see [42, p. 1908]). An intrinsic multiprocessing system is a cluster of multiple sequence control units and a number of time-shared execution units (see [13, p. 81]). IMP systems exploit the advantage of independent processes to increase resource utilization. This is achieved by time-sharing data paths and execution units that would be used only sporadically by a single process[2].

4.1.4 MIMD

In a MIMD architecture model, each processing element executes a different instruction stream on a different data stream. From the space-multiplexing perspective, a cluster with multiple nodes, a node with multiple processors as well as a processor with multiple cores are typical examples for MIMD architectures since all of these architectures allow for the execution of multiple stream interactions in parallel. SMT can be considered as time-multiplexed MIMD on the ISA level. While the states (program counter, instruction register, stack) of multiple hardware threads can be managed in parallel, their stream interactions are processed consecutively since hardware threads time-share specific functional units.

[2] An often mentioned example of MISD architectures is the space shuttle primary computer system [95]. Examining this system in more detail, however, it becomes apparent that it fulfills the requirements for a MIMD architecture. Indeed, it does not execute multiple instruction streams on the same (derived) data stream, but execute different instances of the same instruction stream on different instances of the same data stream; i.e. it handles multiple independent (but redundant) pairs of instruction and data streams.

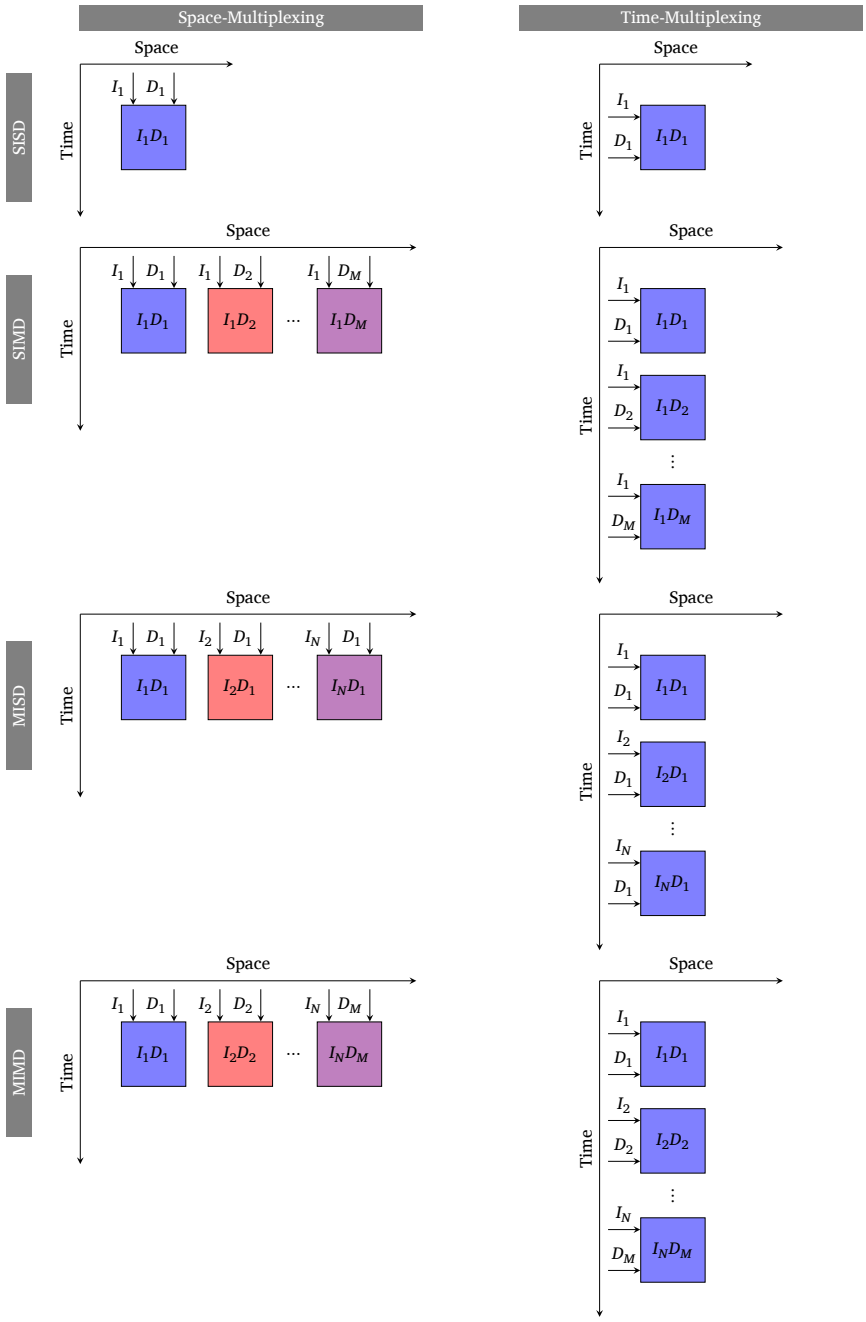


Figure 4.2: Space- and time-multiplexing versions of the SISD, SIMD, MISD and MIMD architectures.

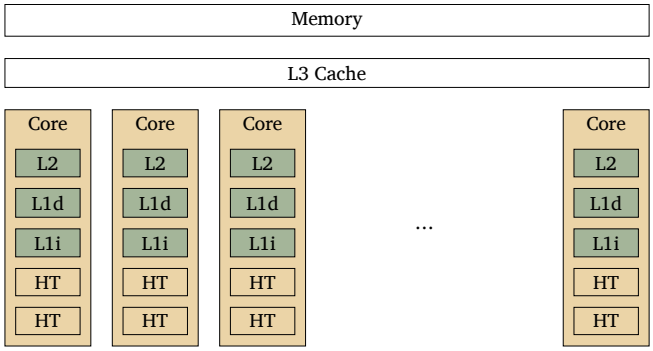


Figure 4.3: Block diagram of a multi-core CPU based on the Intel Xeon Processor Scalable Family. A multi-core CPU consists of multiple compute cores. Each core consists of two hardware threads that share an L1 instruction cache, an L1 data cache and a combined L2 cache. All cores share a combined L3 cache and memory. Cores (yellow) provide the control unit for fetching and dispatching instructions for execution by the hardware threads.

4.2 CPU Architecture

This section provides a short high-level introduction to CPU architectures. Regarding multi-core CPUs, this work follows the terminology of Intel as used in the Intel Architectures Software Developer's Manual [57].

Figure 4.3 provides the schematic view of a typical multi-core CPU such as Intel's Xeon Gold 6148 (Skylake) processor that is applied for the performance analysis in Section 8. It consists of several compute cores, a shared memory and a shared L3 data and instruction cache. Each compute core consists of multiple logical cores, so-called hardware threads. The CPU in Figure 4.3 provides two hardware threads per core, i.e. it exhibits 2-way SMT. All hardware threads of a compute core share a combined L2 cache, an L1 instruction cache and an L1 data cache.

4.3 GPU Architecture

Regarding GPU hardware, this work follows the terminology of Nvidia as used in architecture white papers and the CUDA documentation [78]. Nevertheless, the introduced models and concepts are developed to be effortlessly transferable to GPU architectures by AMD and Intel via OpenCL or SYCL. For mappings between the terminologies used by different platforms and vendors please refer to Table A.1.

Figure 4.4 shows the block diagram of a GPU architecture based on the Nvidia data center GPUs which are used for the performance analysis; see [82] for K40 (Kepler), [80] for P100 (Pascal) and [81] for V100 (Volta). Since these microarchitectures are qualitatively broadly similar, Figure 4.4 provides a schematic view for all of them.

A GPU architecture consists of several execution units referred to as *SMs*, a shared L2 cache, multiple DRAM stacks, a thread engine as well as a host interface. All SMs are connected to a shared L2 cache in which data loaded from DRAM is implicitly cached. DRAM itself is in turn connected to an interface for communication with the host CPU. Groups of instruction streams are scheduled onto SMs by means of a thread engine.

Figure 4.7b shows the architecture of an SM based on the Volta microarchitecture. A Volta-SM consists of an L1 instruction cache, a configurable L1 data cache/shared memory and four equal processing blocks (PBs). In contrast, a Pascal-SM consists of an L1 instruction cache, a dedicated L1 data cache, a dedicated shared memory and two equal PBs. A Kepler-SM, in turn, exhibits an L1 instruction cache, a configurable L1 data cache/shared memory like Volta and two equal processing blocks like Pascal. Additionally, Kepler provides a dedicated read-only data cache.

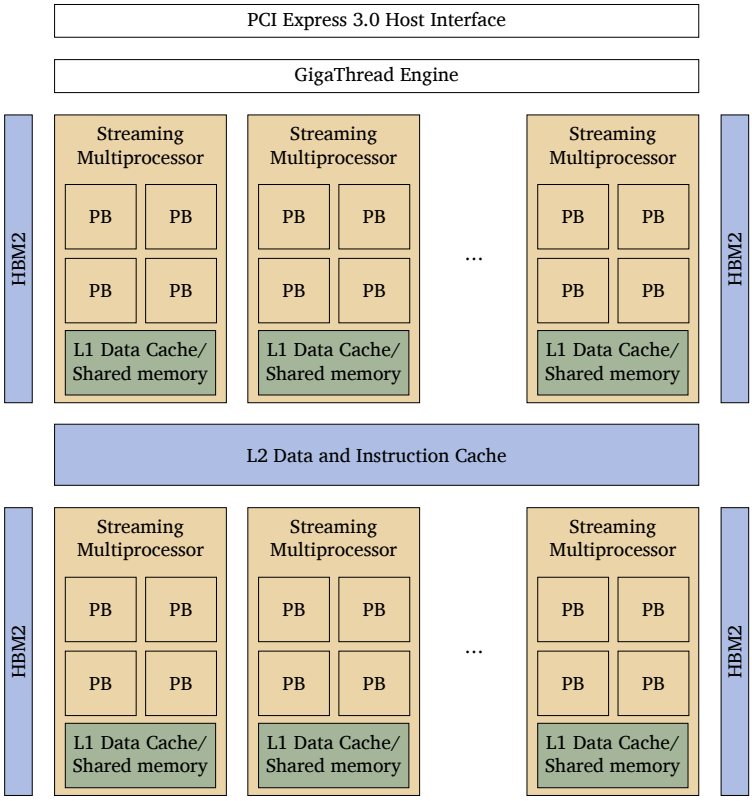


Figure 4.4: Block diagram of a GPU architecture based on Nvidia Tesla V100 [81]. A GPU consists of a grid of streaming multiprocessors. These streaming multiprocessors have shared access to several memory modules and an L2 instruction and data cache. Each streaming multiprocessor consists of a configurable L1 cache/shared memory and, dependent on the architecture, two or four PBs. From a software developer's point of view, the memory modules and the L2 cache (all blue blocks) form a unit called global memory, while the L1 cache/shared memory modules (green blocks) can be used as software-managed caches referred to as shared memory. Streaming multiprocessors (yellow) provide the compute and scheduling hardware for the execution of thread blocks. Other parts (white blocks) are not exposed to the programmer.

For Volta, each PB consists of an L0 instruction cache, a warp scheduler, a dispatch unit as well as a register file and diverse functional units such as INT32 ALUs, FP32/64 ALUs, SFUs and tensor cores. We refer to the latter as *Stream Processors (SPs)*; commonly, SPs are also referred to as *CUDA Cores* or *shading units*. In contrast, Kepler and Pascal exhibit two dispatch units per PB or, to be more precise, per warp scheduler.

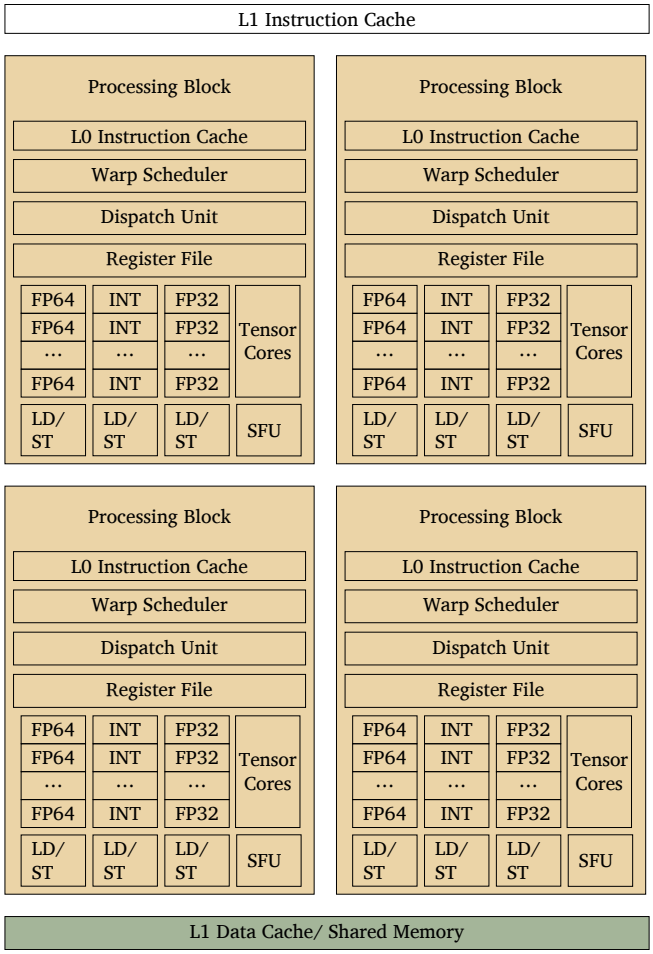


Figure 4.5: Block diagram of a streaming multiprocessor based on the Nvidia Tesla V100 Streaming Multiprocessor[81]. A streaming multiprocessor consists of a configurable L1 cache/Shared memory and, dependent on the specific architecture, two or four PBs . Each PB covers an L0 instruction cache, a warp scheduler and a dispatch unit for obtaining, scheduling and mapping instructions. Furthermore, each PB covers a register file as well as diverse execution units. These comprise single/double precision ALUs (FP32/64; also known as *CUDA cores*), integer ALUs (INT), special function units, load/store units and tensor cores.

4.4 CPU vs. GPU Architecture

This section outlines the optimization goals behind CPU and GPU architectures. Further, the influence of these optimization goals on concurrent processing capabilities is outlined for the following criteria:

- SIMD capabilities
- MISD capabilities

Table 4.2: Comparison of size, theoretical peak performance and compute density of a CPU and a GPU architecture.

Property	CPU	GPU
Chip	AMD EPYC 7702	AMD MI100
Die Size [mm^2]	592	750
Cores	64	128
Base Clock [GHz]	2.0	1.0
IPC	16 (2× AVX2 FMA)	64 (32× FMA SPs)
FP64 Peak Performance [GFLOPS]	2048	8192
Compute Density [GFLOPS/mm^2]	3.46	10.9

- MIMD capabilities

4.4.1 Optimization Goals

CPU and GPU architectures are designed towards different optimization goals. CPU architectures are latency-driven; they are designed to minimize instruction and memory latencies. GPU architectures, in contrast, are throughput-driven; they are designed to maximize instruction and memory throughput.

Each CPU core consists of multiple features that help a single operation stream to be executed as fast as possible. This covers sophisticated control logic such as OOO units and branch predictors as well as large low-latency caches. Accordingly, chip space is invested into few, sophisticated cores instead of many, simpler cores.

On GPU architectures, in contrast, chip space is invested into many, simpler cores instead of few, sophisticated ones. Therefore, GPUs do not exhibit OOO units or branch predictors. Instead, they exhibit a global control unit (Nvidia *GigaThread Engine* or AMD *Command Processor*) that is shared between all of those cores, in addition to basic control logic that is shared by multiple functional units (see Figure 4.7b). This allows GPUs to exhibit a multitude of functional units and hence a multitude of hardware concurrency in comparison to CPUs.

Figure 4.6[3] shows the die shots of an AMD EPYC 7702 CPU CCD (Zen 2) with 8 cores and an AMD MI100 GPU (Arcturus) with 8 arrays à 16 compute units. The figure illustrates that the relative amount of chip space occupied by compute logic on GPUs is larger than on CPUs. As exemplified for MI100 and EPYC 7702 in Table 4.2, the GPU can execute three times more FLOPS per square millimeter than a CPU. Please note that this comparison is only valid since both processors are fabricated with the same process size; here, the 7nm MOSFET process of TSMC for both chips.

These different architectural optimization goals are further reflected by a difference in clock speeds. As outlined in Figure 4.7a, GPUs have in general a lower clock speed than CPUs. Considering the processor chips in Table 4.3, the clock speed of CPUs is on average 3 times higher than the clock speed of GPUs.

4.4.2 SIMD Capabilities

CPUs and GPUs both cover space-multiplexed SIMD mechanisms. CPUs support SIMD in terms of SSE and AVX units for vectorization. Typically, GPUs are considered as SIMD architectures since they execute instructions on multiple lanes of a warp in lockstep. Examining this further, lanes in a warp diverge at conditional statements by masking lanes dependent on the execution

[3]Credit for the underlying high-resolution die shots goes to [44] for the CPU and [101] for the GPU. Due to the different distribution of compute logic on CPUs and GPUs, comparing core/compute unit die shots only is not sufficient. Therefore, this work annotates the underlying CCD die shot by combining the annotations provided by AMD in a core die shot of the Zen 2 architecture [104] with a die shot of one CCD of an EPYC 7702 CPU[44]. Please note that the latter consists of 8 such CCDs, exhibiting 64 cores overall. This does, however, not distort the ratio between compute and control logic, which is relevant for a fair comparison to the GPU architecture. For the GPU die shot, it combines the details of the compute unit die shot [12] and the full-chip die shot [11].

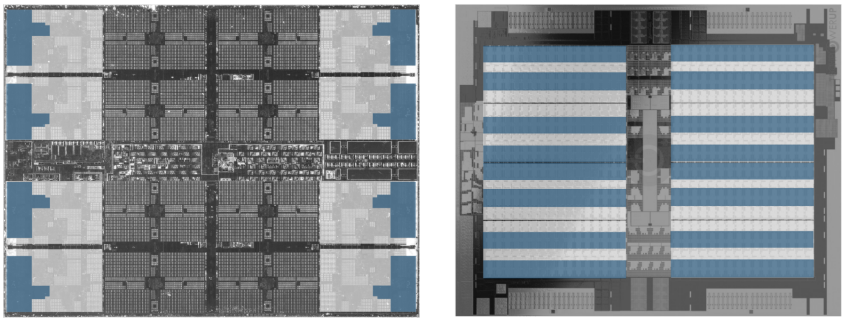


Figure 4.6: Comparison of a CPU die shot (left) and a GPU die shot (right). The CPU die shot belongs to an AMD EPYC 7702 CPU CCD. The CCD consists of 8 Zen 2 cores (transparent white) and 4 L3 cache slices. The compute logic (blue) on each core includes all FP/SIMD units and the ALU. The full 64-core AMD EPYC 7702 CPU consists of 8 such CCDs. The GPU die shot belongs to an AMD MI100, which consists of 8 arrays (transparent white) à 16 compute units, i.e. the vertical slices of an array. The compute logic (blue) on each array includes all FP/SIMD units and ALUs.

path they take. Therefore, GPUs can be considered as a variation of associative array processors. Current Nvidia GPUs (Volta and onward), however, support independent thread scheduling, which introduces a dedicated program counter per warp lane and mitigates the negative impacts of strict lockstepping, such as performance-costly branch divergence and warp-synchronous deadlocks.

4.4.3 MISD Capabilities

CPUs and GPUs are both scalar processors in the sense that both provide instruction pipelining, which corresponds to space-multiplexed MISD at the MA level. On multicore CPUs, each core exhibits a control unit that implements a pipelined fetch-decode-execute cycle on the MA level. On GPUs, each SM exhibits multiple warp schedulers and dispatch units that implement instruction pipelining on whole warps. More important, regarding MISD capabilities on GPUs, however, are the deep execution pipelines.

4.4.4 MIMD Capabilities

Current CPUs and GPUs both exhibit forms of space-multiplexed MIMD. On a multi-core CPU, multiple threads can run independently in parallel on different cores, and flexibly communicate and synchronize with each other via shared memory. Due to this, thread- and SPMD-based parallelization are the prevalent parallel algorithm model on CPUs. On a GPU, multiple thread blocks can run independently in parallel on different PBs since each PB covers a dedicated warp scheduler and dispatch unit. However, synchronization between threads from different thread blocks is either impossible (for AMD GPUs or pre-Pascal Nvidia GPUs) or considered costly. So far, this prevents irregular applications that do not exhibit sufficient SIMD data-parallelism from running efficiently on GPUs.

Furthermore, CPUs and GPUs both exhibit forms of time-multiplexed MIMD. Multi-core CPUs in terms of SMT and GPUs in terms of interleaved multithreading. Both mechanisms have in common that the execution contexts of multiple operation streams are kept in registers to allow for fast context switching. This enables the control units or the warp scheduler to issue an instruction from a different ready-to-execute operation stream in every cycle. Therefore, both mechanisms lead to interleaved execution of operation streams and reduce data dependency stalls on the execution pipelines (see Section 4.4.3). On CPUs, this is typically implemented for two or maximally four

hardware threads per core. On GPUs, in contrast, this is done for 16 up to 32 resident warps per PB or accordingly 64 resident warps per SM.

4.5 Quantitative Architecture Trends

The content of the subsequent subsections is closely based on a previously published work [74].

Based on the concurrent processing capabilities of both architectures, this work derives a mapping between CPU and GPU features that allows for a quantitative comparison of both architectures over time. This mapping equates:

- each SIMD lane and each FPU on a CPU with an SP on a GPU, and refers to both as *Processing Element (PE)*;
- a core on a CPU with an SM on a GPU, and refers to both as *Compute Unit (CU)*.

4.5.1 Methods

To quantify architectural trends over the years, this work compares high-end CPUs and GPUs from Intel, Nvidia and AMD since the beginning of the dual-core era int 2005 and the advent of the first GPGPUs in 2006. Table 4.3 lists the specific processors and microarchitectures that are used as data basis for the analyzes in the subsequent sections. The data basis is retrieved from the GPU Specs Database [103] and the CPU Specs Database [102]. The time points that serve as basis for the charts correspond to release dates.

A CPU chip is only included in the data basis if it was used in Intel server CPUs that target the HPC market. Hence, architectural properties are retrieved from Intel's product specifications for the Intel Xeon Processors and Intel Xeon Scalable Processors [59]. Following the UAM, the number of compute units #CUs corresponds to the number of cores. This does not include SMT since multiple SMT-threads share a SIMD unit and therefore do not operate in parallel, but only in an interleaving manner. The number of processing elements #PEs corresponds to the number of SIMD-lanes on an entire CPU. It is derived from the `FP32` SIMD width and the overall number of SIMD units.

A GPU chip is only included in the data basis if it is used in dedicated HPC GPUs, i.e. if it is part of the Nvidia Data Center (former Tesla) series or AMD Radeon Instinct (former FirePro S) series. For each microarchitecture, the chip with the highest #CUs is considered. However, dual-GPU designs are excluded from this approach since they do not reflect technical progress in the fabrication process and are rather comparable to dual-socket CPU systems, which are also excluded from the data basis. Therefore, the provided #CUs and #PEs refers to GPU chips, instead of specific GPU models. The number of compute units #CUs for all GPU chips is retrieved from the GPU Specs Database [6] and corresponds to its parameter *SM Count*. The number of processing elements #PEs per CU is determined from the ratio of the parameters *Shading Units* and *SM Count*. Hence, this can be considered as the size of a CU.

For both processor types, stated clock frequencies refer to base frequencies instead of turbo frequencies. This allows for a fair comparison between current hardware that supports a turbo frequency, and past hardware that does not. Further, it eliminates the influence of core utilization and SIMD usage on the clock frequency.

4.5.2 Clock Frequency Trend

With the beginning of the 21st century, clock frequencies started to stagnate due to their thermal design power. The latter rendered the continuation of the ever-increasing clock frequency trend of earlier decades impossible. Figure 4.7a shows the development of base clock frequencies of CPUs and GPUs subsequent to this period. Since around 2005, CPU clock frequencies stagnate between 2.3 to 3.1 GHz.

In principle, GPU clock frequencies follow a similar trend just with an offset in time and the maximal base clock frequency. Subsequent to the introduction of dedicated general purpose computing GPUs in 2006, GPU clock frequencies steadily increased. Around 2016, however, GPU clock frequencies began to stagnate at 1.0 to 1.2 GHz due to thermal design power, too.

Table 4.3: Considered GPU and CPU Chips

ID	Year	Nvidia GPU Chips	AMD GPU Chips	Intel CPU Chips
0	2005			Pentium D840 (Smithfield)
1	2006	G80 (Tesla)		
2	2010	GF100 (Fermi)		X7560 (Nehalem)
3	2011			E7-8870 (Westmere)
4	2012		Tahiti (GCN 1)	E5-2687W (Sandy Bridge)
5	2013	GK180 (Kepler)	Hawaii (GCN 2)	
6	2014			E7-8890 v2 (Ivy Bridge)
7	2015	GM200 (Maxwell)	Fiji (GCN 3)	E7-8890 v3 (Haswell)
8	2016	GP100 (Pascal)	Ellesmere (GCN 4)	E5-2699A v4 (Broadwell)
9	2017	GV100 (Volta)		8180M (Skylake)
10	2018		Vega 20 (GCN 5.1)	
11	2020	GA100 (Ampere)	Arcturus (CDNA 1)	8380HL (Cooper Lake)
12	2021			8380 (Ice Lake)
13	2022		Aldebaran (CDNA 2)	

4.5.3 Compute Unit Trend

To enable further growth in hardware performance despite the stagnation of clock frequencies, hardware vendors increased the number of CUs per processor. This trend can be seen in 4.7b that shows the development of the number of CUs (*#CUs*) per processor between 2005 and 2022. In this period, *#CUs* per CPU or GPU increased regardless of vendors. For CPUs, *#CUs* doubled roughly every four years, which led to a twenty fold increase overall.

Considering Nvidia GPUs, *#CUs* was constant till 2014. In the subsequent two years, *#CUs* more than quadrupled from 15 to 60 CUs per GPU. In roughly the same period, AMD introduced its first GPGPU chip (Tahiti) with 32 CUs and doubled this number to 64. Having a similar number of CUs at the start of 2016, Nvidia continued the increase in CUs.

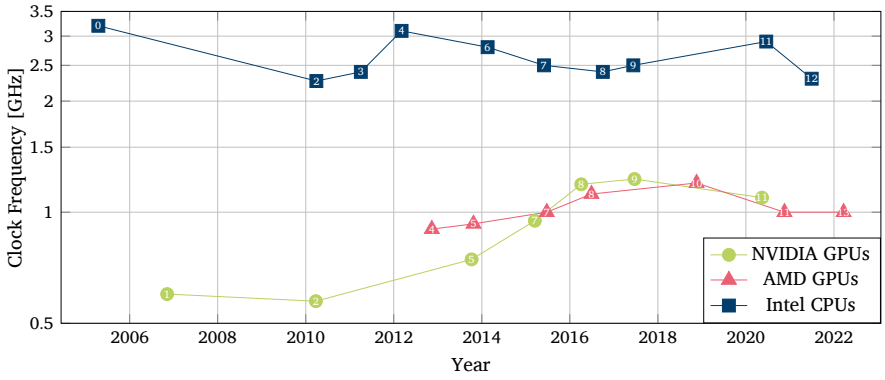
AMD, however, decreased the number of CUs with the introduction of its GCN 4 micorarchitecture, which is compatible with GCN 3 but aims at higher clock frequencies by using a smaller fabrication process size. In fact, this is a data selection artifact that hides the fact that AMD continued to produce GCN 3 based GPUs with 64 CUs, i.e. continuing the trend of an increase in *#CUs* just as Nvidia. Hence, all vendors heavily expanded the MIMD capabilities of their CPUs and GPUs between 2005 and 2022.

4.5.4 Processing Element Trend

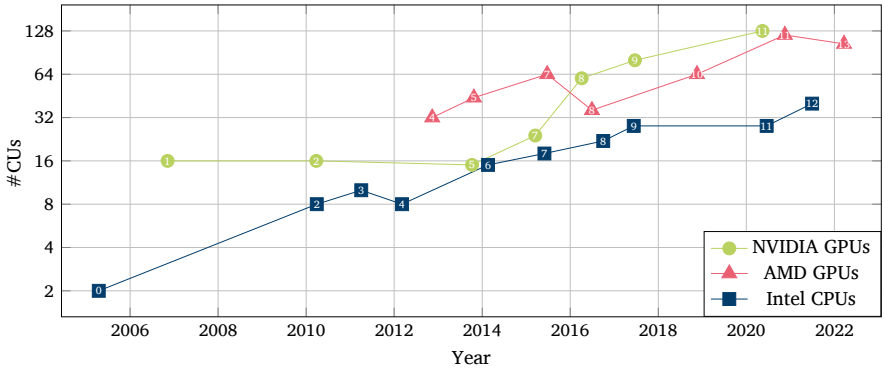
Figure 4.7c depicts the size of CUs between 2005 and 2022. The size of a CU corresponds to the ratio of the number of processing elements *#PEs* per CU. Hence, this ratio can be considered as the amount of hardware parallelism a CU provides.

On a CPU, *#PEs* per CU resembles the SIMD width. Hence, the chart shows that the SIMD width is step wise increasing. This reflects the successive introduction of SSE, SSE2 (Sandy Bridge), AVX (Haswell) and AVX2 (Skylake). Further, the steps in the trend reflect Intel's tick-tock production model, with the *tick* being a new fabrication process, and the *tock* being a new microarchitecture.

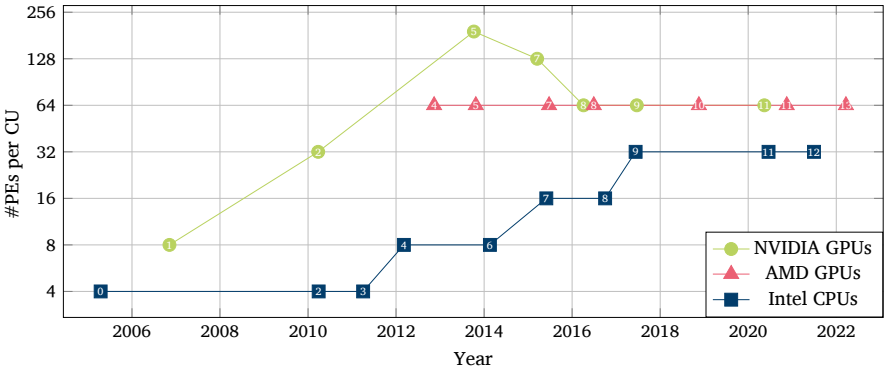
For AMD GPUs, the CU size is constant over time. For Nvidia GPUs, however, the CU size increases between 2006 and 2013. Comparing Figures 4.7c and 4.7b, this increase overlaps with a stagnation in the amount of CU per GPU. Hence, both vendors increased the overall amount of concurrent processing capabilities, but with two different approaches. Finally, the Maxwell and Pascal architectures introduced a trend reversal that led to a decrease in CU size and a leveling at 64 *#PEs/CU*; remarkably, at the same size as AMD CUs.



(a) Clock Frequency Trend



(b) Number of compute units per processor over time.



(c) Size of compute units over time.

Figure 4.7: Development of CPU and GPU architecture properties over time.

Chapter 5

Redesigning your application to run multithreaded on a multicore machine is a little like learning to swim by jumping into the deep end.
Herb Sutter

Parallel Programming Models

In this work, a parallel programming model is considered as a set of abstractions (instructions, functions, directives, runtimes) that enable the development and execution of concurrent software.

As stated in Section 3.2, the parallelization of an algorithm can be subdivided into four coarse steps: decomposition, assignment, orchestration and mapping. Based thereon, a parallel programming model should provide front-end abstractions or back-end mechanisms or both for all of these theoretical steps. Considering the technical specifications of the parallel programming models CUDA, OpenACC and OpenMP, the following commonalities can be derived.

First, each parallel programming model supports one or several *partitioning models* that allow for the *decomposition* of an algorithm into work units and the *assignment* of these work units to processes. Second, it consists of an *execution model* that describes a set of processes and determines *orchestration* capabilities between these processes. Third, it provides an *architectural model and a memory model* that serve as the foundation of the execution model and describes the *mapping* of processes and interactions to hardware resources.

5.1 Classification

This section classifies parallel programming models based on the architectures and partitioning models they support. Regarding supported architectures, this work considers:

- CPU-only programming models (see Eventify in Section 5.3)
- GPU-only programming models (see CUDA in Section 5.4)
- Heterogeneous programming models with:
  - Compiletime heterogeneity (see OpenACC in Section 5.5): enables execution of uniform parallel code sections either on CPU or GPU, with the architecture being selectable at compile time.
  - Runtime heterogeneity (see OpenMP in Section 5.2): enables execution of architecture-dependent parallel code sections on CPUs and GPUs simultaneously.

Partitioning models describe the decomposition of an algorithm into tasks and the assignment of these tasks to processes or threads. Hence, partitioning models are on the one hand related to the type of concurrency that algorithms exhibit, and on the other hand to the specific parallelization techniques that the underlying execution and architectural models support. Firstly, this allows for a clear distinction between parallelization approaches that can be applied to implement diverse

Table 5.1: Classification of partitioning models based on the stream interaction model.

Interaction Scheme	Partitioning Model	Example (Programming model)
SIMD	Vectorization-based Warp-based	SIMD intrinsics (C++) Warp-aware data access (CUDA)
MISD	Data-Flow-based Pipeline-based	Data flow modelling (VHDL) Software pipelining (CUDA)
MIMD	Thread-based: • Loop-based • Task-graph-based SPMD-based	 Parallel loops (OpenMP) Task dependencies (OpenMP) Multiprocessing (MPI)

types of data- and task-parallelism. And secondly, it allows for a mapping between partitioning models and architecture models.

Table 5.1 provides an overview of partitioning models classified by means of the stream interaction model. The concurrent processing capabilities of a partitioning model with stream interaction scheme *X* correspond to an architecture model that (on any of its levels) supports the same stream interaction scheme *X* and vice versa. Similar to the considerations on architecture models, partitioning models are not mutually exclusive and can be combined, e.g. multiprocessing and multi-threading can be combined with vectorization.

5.2 OpenMP

This section briefly covers the architecture, memory and execution model behind OpenMP. Based thereon, it describes the algorithm models OpenMP supports as well as the OpenMP C++ directives and environment variables applied in this work.

5.2.1 Architecture Model

OpenMP is a parallel programming model for heterogeneous hardware. It supports CPU and GPU architectures from different vendors [84, p.], which can jointly be used during the execution of a single OpenMP program. Therefore, OpenMP differentiates between *host* and *target* devices. The host device is the CPU on which the execution of the OpenMP program begins. Target device, on the other hand, is a relative term; it refers to a device that another device (typically, the host device) offloads work to.

5.2.2 Memory Model

OpenMP provides a shared-memory model with relaxed consistency and support for offloading. Due to the latter, the API provides directives to transfer data between `host` and `device` memory.

OpenMP differentiates between `private` and `shared` variables. Private variables can be accessed by a single thread only. Shared variables, in contrast, can be accessed by all threads of the team. The access type of a variable can either be implicitly defined by its scope or explicitly specified via an OpenMP data-sharing clause.

5.2.3 Execution Model

Each OpenMP program is executed as an *OpenMP process*, that is, a set of *OpenMP threads* and *OpenMP address spaces* that can potentially be located on multiple, different target devices. The execution model behind such a process is based on fork-join parallelism [84, p. 22]. Therefore, each OpenMP process consists initially of a single, sequential thread that runs on the host device. This *initial thread* spawns additional threads once it encounters a `parallel`, `target` or `teams` construct. If the OpenMP implementation supports nested parallelism, these threads may also spawn further threads.

The `parallel` construct (see Listing 5.1, Line 1) creates a *team of threads* that execute a parallel region [84, p. 92] concurrently. The work encompassed by the parallel region is distributed

LISTING 5.1: OPENMP for CONSTRUCT WITHIN parallel REGION

```
#pragma omp parallel [clause[ [,] clause] ... ]
    #pragma omp for [clause[ [,] clause] ... ]
        loop-nest
```

LISTING 5.2: COMBINED OPENMP CONSTRUCT parallel for

```
    #pragma omp parallel for [clause[ [,] clause] ... ]
        loop-nest
```

between these threads via work sharing constructs such as partitioned `for`-loops or independent `sections`.

The `teams` construct, however, creates a *set of teams* and the initial thread in each team executes the region [84, p. 100]. Since the initial thread of each team can spawn further threads, the `teams` construct allows for hierarchical parallelism as provided by GPU architectures. Therefore, the `teams` construct is only applicable within a `target` region.

The `target` construct enables offloading work and data to an accelerator. The `target` construct spawns an initial thread on the target device [84, p. 22] and this thread executes the target region; commonly, by spawning further threads via `parallel` or `teams`.

5.2.4 Partitioning Models

In OpenMP, algorithm models are expressed via work sharing constructs that allow the distribution of work between multiple threads. OpenMP supports vectorization-based parallelization via the `simd` construct, loop-based parallelization via the `for` construct and task-graph-based parallelization via the `task` construct in combination with the `depend` clause.

Here, only loop-based parallelization as applied within the OpenMP version of *FMSolvr* in Section 7.4.1 is outlined further. In order to parallelize a for-loop with OpenMP, the for-loop and the OpenMP `for` construct must be contained in a parallel region. This can either be achieved by nesting a `for`-construct within a `parallel` construct as shown in Listing 5.1 or by using a combined construct as shown in Listing 5.2. The `for` construct then subdivides the iteration space into chunks, which are distributed to all t threads of the parallel region for execution.

Since distribution and size of those chunks may have a severe impact on performance, they can be specified through the `schedule` clause via `schedule(type, chunk_size)`. While `type` determines the applied scheduling policy, `chunk_size` determines the number of iterations per chunk. If no `chunk_size` is specified, the iteration space is divided into at most t chunks that are approximately equal in size [84, p. 129].

The schedule-type `static` indicates that chunks are distributed to threads following a static round-robin policy, which is especially advantageous if chunks contain equal amounts of compute load. The schedule-type `dynamic` denotes that threads request chunks from a work pool dynamically, which improves load balancing if chunks contain different amounts of compute load. The schedule-type `runtime` indicates that the scheduling type is chosen via the environment variable `OMP_SCHEDULE` at runtime.

5.3 Eventify

Eventify is a low-overhead, task-parallel programming library that targets strong scaling and performance portability of applications with many, tiny, dependent tasks [50]. Its source code is published at www.fmsolvr.org under the open source license LGPL v2.1.

5.3.1 Architecture Model

So far, Eventify supports CPU architectures only. This work contributes to the extension of *Eventify* to GPUs.

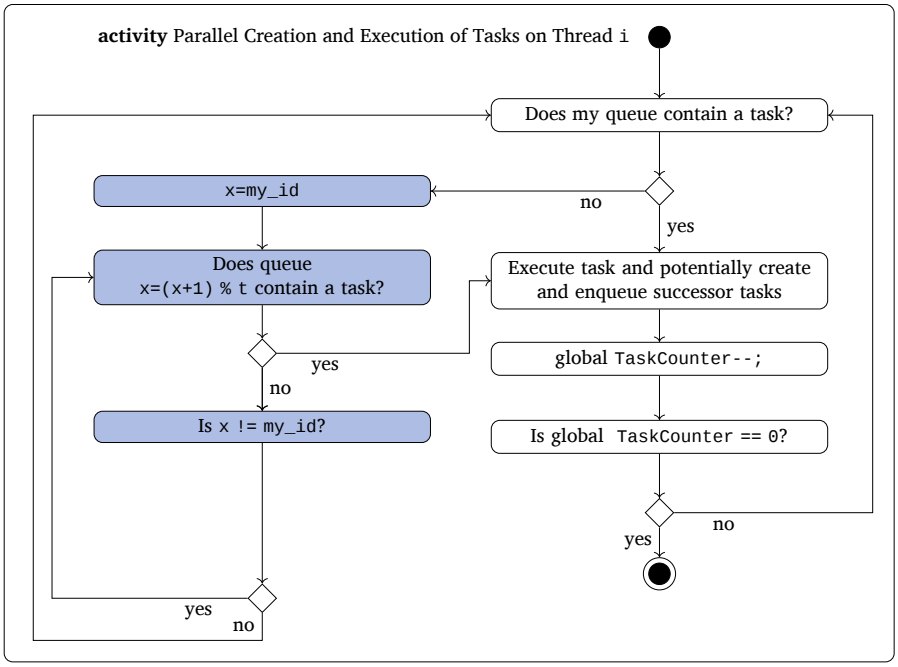


Figure 5.1: UML activity diagram of Eventify's task execution engine for a single thread, as depicted in [75].

5.3.2 Memory Model

Since Eventify relies on `std::threads`, it expects applications to adhere to the memory model of C++11 [25, p. 6f] or above. Further, it provides a NUMA layer that enhances data locality by reusing static work distribution functionalities for thread pinning, i.e. threads that operate on the same data, should be located on the same NUMA node.

5.3.3 Execution Model

The execution model of Eventify is based on tasks that are executed by C++'s `std::threads`. Each thread owns a task queue and all threads follow the same task execution routine as shown in Figure 5.1.

At startup, a set of initial tasks is statically assigned to each thread. By definition, initial tasks do not exhibit any incoming dependencies, which allows for Eventify's *bottom-up* task creation.

The load balancing approach of Eventify combines three techniques, namely *static work assignment*, *work sharing* and *work stealing*. During execution, each thread first checks its own queue for a task. If the queue contains a task, it is executed and its dependencies are resolved. Otherwise, the thread follows a work stealing approach and checks other queues for work. If it finds a task, the task is executed and its dependencies are resolved. If the dependency resolution generates new tasks that are ready-to-execute, the thread follows a work sharing approach; based on the static work distribution scheme, it inserts the new tasks into the respective task queues. Following this approach, only tasks that are ready to execute enter a task queue.

5.3.4 Partitioning Models

Eventify enables software developers to specify parallelism in form of event-based task parallelism. In Eventify, a task consists of a processor object and a data object. Hence, the execution of a task

LISTING 5.3: EVENTIFY SYNTAX FOR TASK DEFINITION

```
template <typename... Args>
struct CTask : public AbstractProcessor<CTask, Args...>
{
    typedef AbstractProcessor<CTask, Args...> Base;
    using Base::AbstractProcessor;
    template <typename DataType>
    void run_computation(DataId data_id)
    {
        compute_CTask(data_id, this->data_structure);
    }
};
```

LISTING 5.4: SYNTAX FOR CONFIGURATION OF THE STATIC EVENT DISPATCHER

```
using EventDispatcher =
    EventListenerContainer<
        EventListener<DataEventAlpha, Handler<Create_ATask>>,
        EventListener<DataEventAlpha, Handler<Create_BTask>>,
        EventListener<DataEventBeta, Handler<Create_CTask>>,
        EventListener<DataEventGamma, Handler<Create_CTask>>,
        //...
    >;
```

is the execution of a processor object on a specific data object. Instead of defining and declaring each and every task and its dependencies separately, tasks are described in form of *task types* that specify the dependency pattern and compute operation for a group of tasks. Dependency patterns are defined upon user-defined data structures (which are anyway part of any sequential software) via *event listeners* that are used to configure a static event dispatcher. The compute operation is specified as part of the processor object. Overall, this leads to a "convenient way to describe recurring task dependency patterns in large task graphs" [50].

To parallelize a sequential application with Eventify, a software developer has to provide:

- **Processor definitions**: Task-type specific processors are defined via inheritance from the `AbstractProcessor` class and implementation of its `run_computation()` function by calling a user-defined compute function. As an example, Listing 5.3 shows the definition of a task of type `CTask`.
- **Event dispatcher definition**: The event dispatcher defines dependencies between task types. It consists of event listeners that *statically* define dependencies between data events and event handlers *at compile-time* (see Listing 5.4). As soon as a specific data event is triggered, its event handler is called to initiate the creation and enqueueing of the respective task. The connection between an event source, e.g. the computation of a specific data element, and the triggered event is established by calling the `dispatch()` method of the event dispatcher (see Listing 5.5).
- **Multi queue definition**: In *Eventify*, each thread owns a multi queue to store its tasks. As the name implies, a multi queue consists of multiple sub-queues; one for each task type. With the multi queue definition, the user determines the prioritization of task types. Listing 5.6 and corresponding Figure 5.2 show a configuration in which tasks of type `ATask` are of higher priority then tasks of type `BTask` and `CTask`, i.e. task types are ordered with decreasing priority.

Listing 5.5: Eventify syntax for dependency resolution via dispatch call

```
compute_CTask(DataId data_id, DS& data_structure)
{
  // Compute a data element alpha
  data_structure[data_id].alpha = ... ;
  // Resolve dependencies
  EventDispatcher::dispatch<DataEventAlpha>(data_id);
  // Event dispatcher calls respective event handlers Create_ATask and Create_BTask
}
```

Listing 5.6: Syntax for Configuration of the Multi Queue

```
using multi_queue =
    MultiQueue<ATask, BTask, CTask>;
```

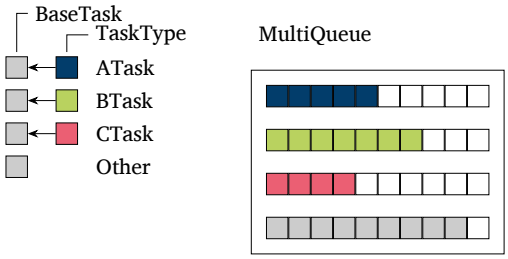


Figure 5.2: Configuration of Eventify's multi queue.

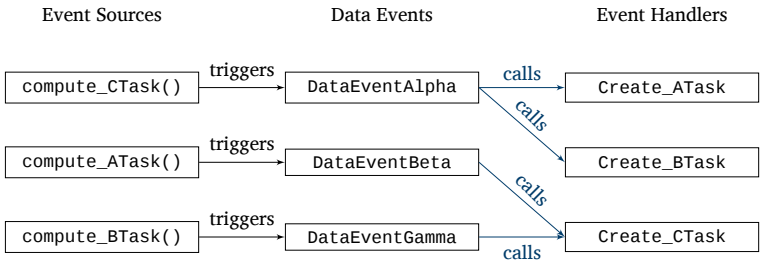


Figure 5.3: Concept and configuration of Eventify's static event dispatcher. The internal connections (blue) of the static event dispatcher are configured at compile time. This example configuration corresponds to Listing 5.4.

5.4 CUDA

CUDA is a parallel programming model that enables general-purpose computations on Nvidia GPUs. CUDA supports C, C++ and Fortran by means of language extensions and built-in variables. It consists of the CUDA Runtime API and the CUDA Driver API, with the architecture, memory and execution models behind both being the same. This section provides an overview on the concepts of the CUDA Runtime API for C++ that are relevant for this work.

5.4.1 Architecture Model

From a software developer's point of view, many of the architectural details provided in Section 4.3 are hidden by the architecture model in Figure 5.4. The architecture model consists of a host CPU that drives one or multiple Nvidia GPUs. The global memory module abstracts the HBM2

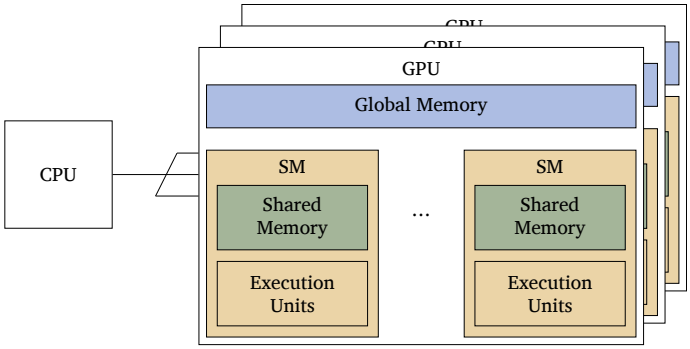


Figure 5.4: The CUDA architecture model consists of a host CPU and one or multiple Nvidia GPUs. The depicted GPU architecture model is an abstraction of the detailed GPU architecture provided in Figure 4.4.

modules and the L2 data and instruction cache (see Figure 4.4). In the architecture model, a GPU consists further of multiple SMs. Each SM covers a shared memory module that can explicitly be used as software-managed cache, but does not expose the L1 data cache (see Figure 4.4) to the software developer. Further, each SM consists of multiple execution units. Neither SMs nor execution units are explicitly exposed to the software developer, i.e. it is not possible to pin specific threads to specific hardware resources. Instead, the execution model abstracts hardware resources via logical threads and thread blocks.

5.4.2 Memory Model

To start with, CUDA assumes a physical separation between *device* and *host* memory space. Therefore, the CUDA runtime provides memory management functions that enable the host-driven allocation and deallocation of device memory as well as data transfer between both memory spaces. With the Kepler architecture, the concept of unified memory was introduced to provide a single, common address space that is accessible from host and device without extensive manual data management. Starting with the Pascal architecture, unified memory supports automatic page migration between device and host memory space in case of a page fault.

CUDA exhibits a hierarchical memory model in which threads have access to multiple memory regions. Each thread has access to a private *local memory* region; local memory cannot be explicitly addressed via the CUDA Runtime API. Each thread block has access to a *shared memory* region that is visible to all threads of the block. Allocation of shared memory is configured in the course of the kernel configuration and initialized within kernel scope via the `__shared` prefix. Further, all threads in the grid have access to *global memory*.

5.4.3 Execution Model

From a macroscopic view, a CUDA program consists of a host program and *kernel* functions. The host program is a sequential or multi-threaded program that runs on the host CPU. It defines the execution context and controls the execution of kernel functions on (potentially multiple) GPUs.

Kernel functions describe parallel computations for execution on GPUs and are written in CUDA C++. CUDA C++ is a C++-dialect that allows the definition of functions that are called once but executed in parallel by multiple GPU threads:

Definition 5.1. ***Kernel Function.*** A kernel function is an implicitly parallel subroutine that executes under the CUDA execution and memory model.*[78]*

Kernel functions are prefixed with the execution space specifier `__global__` and can be called from host and device context.

Further, a CUDA program consists of `__host__` functions that can be called from host code only and `__device__` functions that can be called from kernel functions only. Functions prefixed with both decorators are callable from both, host and device code.

CUDA's execution model supports hierarchical parallelism and therefore differentiates between threads, thread blocks and grids as units of execution:

Definition 5.2. ***Thread.*** A thread is an instruction stream that executes a kernel function.*[78]*

Definition 5.3. ***Thread Block.*** A thread block is a group of threads which execute the same kernel function on a single streaming multiprocessor.*[78]*

Definition 5.4. ***Grid.*** A grid is a group of thread blocks that execute the same kernel function.

Following from the CUDA memory model, all threads of a thread block have access to shared data located in the shared memory of the SM. However, thread blocks cannot access the data located in shared memory of other thread blocks.

Following from the architecture model, SMs provide the compute and scheduling hardware for the execution of thread blocks. The number and configuration of threads and thread blocks in the grid are determined by the software developer within the limits of hardware resources such as shared memory and registers. However, the mapping of thread blocks onto SMs cannot be explicitly configured by the developer. The mapping of thread blocks onto SMs is determined by the GigaThread Engine. All threads of a thread block are mapped to the same SM and reside on this SM for their total execution time. Multiple thread blocks can reside on one SM. The number of threads per thread block and the number of thread blocks per SM depends on the number of hardware resources a kernel function uses and the number of hardware resources an SM provides. The maximum number of threads per thread block and the maximum number of thread blocks per SM depends on the compute capability of the GPU and can be found in Table 8.2 for all GPUs considered in this work.

For execution, thread blocks are subdivided into *warps* as schedulable units. Each warp consists of W_s threads or *lanes*, where W_s is typically 32. Each thread block of size D_{block} is partitioned into $ceil(D_{\text{block}}/W_s)$ warps, with the first warp containing thread t_0 being followed by consecutive, increasing thread IDs [78, p. 104]. Warps are scheduled for execution by means of warp schedulers. Specific scheduling strategies are architecture-dependent and are outlined in Section 5.6.

5.4.4 Partitioning Models

Considering the execution of kernel code, CUDA implements the SIMT model. SIMT combines thread-based programming in terms of MIMD with warp-based programming in terms of the SIMD model. Typically, the SIMT model is used to express data-parallelism in the form of kernel functions that are executed on a grid of threads. It is more flexible then vectorization-based models since it allows for branching and barely requires the programmer to express parallelism explicitly.

Threads are ordered within blocks and grids and made available in the API via a 3-dimensional indexing scheme. The CUDA runtime API provides the built-in variables `blockDim.{x,y,z}` to retrieve the number of threads in a block along the *x*-, *y*- and *z*-dimension. In the same manner, it provides the `gridDim.{x,y,z}` variables to retrieve the number of thread blocks in a grid along the *x*-, *y*- and *z*-dimension. The built-in variables `threadIdx.{x,y,z}` return the thread index of the calling thread in the specified dimension. In the same manner, the variables `blockIdx.{x,y,z}` return the block index of the calling thread in the specified dimension. Figure 5.5 depicts this principle for a 2-dimensional grid. To configure the execution context of a kernel function by means of this indexing scheme, a kernel function is called as `compute_kernel<<<gridDim, blockDim>>>(...)`.

For GPUs with compute capability ≥ 3.5 (Kepler or newer), CUDA further supports dynamic parallelism, which allows calling kernel functions from the device context.

CUDA further supports task-parallelism with kernel-level granularity in form of *ATGs*. ATGs allow for the expression of dependencies between kernel functions and the automatic derivation of a concurrent execution order.

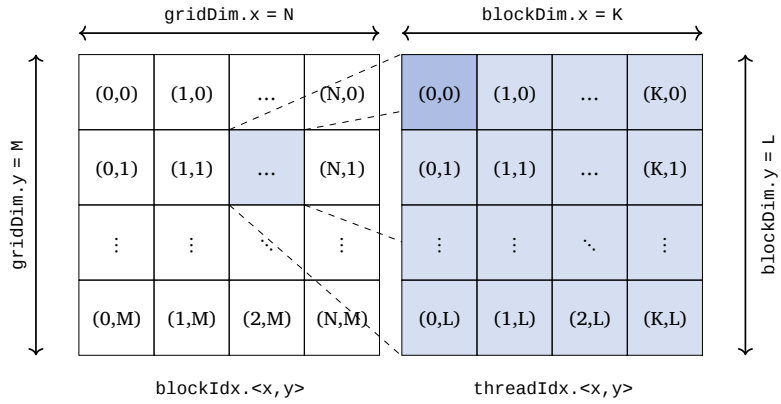


Figure 5.5: CUDA indexing scheme for a 2-dimensional grid. The left grid represents the indexing scheme of blocks in the grid. The blue colored tile represents a single thread block. The right grid represents the indexing scheme of threads in a block. The dark blue tile corresponds to a single thread.

5.5 OpenACC

OpenACC is an offloading-based parallel programming model that combines the ideas behind OpenMP and CUDA. It can be applied to C, C++ and Fortran programs by means of directives, functions and environment variables. This section provides an overview on the concepts of OpenACC along with its architecture, memory and execution model. Based thereon, the algorithm models OpenACC supports are described.

5.5.1 Architecture Model

OpenACC differentiates between *host* and *accelerator* devices. It supports x86- and Power-compatible CPUs as hosts and devices, as well as GPUs from AMD and Nvidia as accelerators[83, p. 137]. In contrast to OpenMP, however, CPUs and GPUs cannot be used flexibly together during the execution of a single OpenACC program in the sense of executing some regions on the host, and offloading other regions to an accelerator. Instead, OpenACC differentiates between *host-only* and *host+accelerator* programs. The former refers to an OpenACC program that is compiled to execute parallel regions on the multi-core host (`-ta=host`), while the latter is compiled to execute parallel regions on the accelerator device (`-ta={gpu_arch}`).

5.5.2 Memory Model

OpenACC supports heterogeneous systems with the following physical relations between host and accelerator memory: *discrete* host and accelerator memories, *shared* host and accelerator memory as well as *discrete and shared* memories [83, p. 34]. This work considers the first system setup only since it is the most common scenario in typical HPC systems and is further the most generic one, i.e. if an OpenACC program operates correctly on a machine with physically separated memories it will operate correctly on a machine with shared memory (under the assumption of similar memory coherence and consistency models).

OpenACC incorporates the physical separation between host and accelerator memory mainly via implicit, compiler-managed data transfers, which aims to free software developers from providing all of the boilerplate memory management code that low-level GPU programming models demand. This does, however, not hold for more complex, pointer- and reference-based data structures as commonly used in object oriented software. For such cases, OpenACC provides the `data enter` and `data exit` directives for the manual creation of deep copies. This work applies the following clauses to the `data` directive:

LISTING 5.7: STRUCTURE OF AN OPENACC DIRECTIVE

```
1    #pragma acc directive-name [clause-list]
```

- `create(vars)`: Allocates memory for `vars` on the accelerator.
- `delete(vars)`: Deallocates memory for `vars` on the accelerator.
- `copyin(vars)`: If `vars` are not present on the accelerator, the clause allocates device memory and copies `vars` from the host to the device when entering a region that requires `vars`[83, p. 48].

5.5.3 Execution Model

For *host-only* programs, the execution model behind OpenACC is similar to the thread-based execution model of OpenMP. Indeed, OpenACC can be seen as a precursor of future OpenMP features. For *host+accelerator* programs, the execution model behind OpenACC is akin to the kernel-based execution model of CUDA and OpenCL, and even makes use of their runtimes. In either scenario, the execution is orchestrated by a host thread. What varies between all of these models, is the terminology and API to access the execution model as outlined in Section 5.5.4.

Since OpenACC targets heterogeneous accelerators, it supports hierarchical parallelism. For this purpose, OpenACC introduces the parallelism levels `vector`, `worker` and `gang`. `vector` models SIMD parallelism, while `worker` models fine-grained parallelism and `gang` models coarse-grained parallelism [83, p. 10]. Please note that OpenACC does not specify how these levels of parallelism map to actual hardware architectures since an architecture does not necessarily exhibit all levels of parallelism. Therefore, references provide diverse views on mappings between OpenACC and other programmings models such as CUDA (cf. [79], [85]).

5.5.4 Partitioning Models

For *host-only* programs, OpenACC relies on the execution of parallel regions in terms of work sharing loops. For *host+accelerator* programs, OpenACC generates kernel code automatically following directive-based compiler hints, instead of expecting software developers to write explicit kernel functions. Hence, OpenACC syntax allows for the generic expression of data-parallelism for both program types. In contrast to OpenMP and CUDA, however, OpenACC does not provide any directives that allow for the explicit expression of task parallelism.

The general syntax of an OpenACC directive is shown in Listing 5.7[83, p. 25]. To define a parallel region, OpenACC provides the directives `kernels` and `parallel`.

When the `kernels` directive is applied to a structured block, the structured block is automatically split into a sequence of compute kernels, which is then scheduled for execution on the device. According to [83, p. 31], each loop nest in the block will typically be transformed into a separate kernel. Since this approach requires the compiler to auto-detect loop dependencies, it might lead to the overcautious parallelization of inner loops only, which hurts performance.

By means of the `parallel` directive, a structured block is executed in parallel on the current device [83, p. 28]. The level of parallelism to apply can be specified by means of the `vector`, `worker` and `gang` clauses, and the according sizes with the `vector_length`, `num_workers` and `num_gangs` clauses. This work uses the `parallel` directive in combination with the `loop` clause, which is similar to the combined OpenMP construct described in Section 5.2.4. Listing 5.8 shows the architecture-independent parallelization of a loop with OpenACC. Please note that, in contrast to the combined `parallel for` construct of OpenMP, this construct must be applied to each loop of a loop nest separately or in combination with the `collapse` clause. This might further require the use of `loop seq` on inner loops, which declares a loop as non-parallelizable to the compiler.

LISTING 5.8: COMBINED OPENACC CONSTRUCT `parallel loop`

```
1     #pragma acc parallel loop [clause-list]
2         for loop
```

5.6 Programming Model Trends

This section outlines how the hardware architecture trends and properties observed in Chapter 4 are reflected by the latest features of parallel programming models. The focus of this section is to analyze how these features contribute to the effectiveness of task parallelism in heterogeneous systems and in particular on GPUs.

5.6.1 Enhancing Flexibility

The trend of stagnating CU sizes on CPUs and GPUs accounts for the limited amount of inherent SIMD-style data-parallelism in many HPC applications. In fact, many HPC applications exhibit irregular work loads and complex parallelization potentials that require more flexible ways of expressing concurrency. Therefore, increasing the size of CUs would not be effective to achieve further scaling. Instead, a general increase of the number of CUs per processor can be observed. This leads to an increase in the number of independently executable execution streams and hence to an increase in the MIMD capabilities of CPUs and GPUs.

Considering the origin of GPGPU programming in graphics processing, which is the paragon of data-parallelism, GPGPUs were designed for heavily data-parallel applications. Multicore CPUs, on the other hand, do not only provide data-parallelism through SIMD vectorization but also support MIMD parallelism via multi-threading, which is the basis for task-parallelism. The observed trends, however, indicate a gradual convergence of both architectures and accordingly pave the way for a more uniform and flexible description of concurrency in heterogeneous systems.

On GPUs, the increase in MIMD capabilities enhances the number of compute kernels that can be executed simultaneously, e.g. via CUDA streams, without any programming model changes. However, the increase in MIMD capabilities also inspires diverse qualitatively new features in GPU programming models that allow to express less regular work loads by means of task parallelism. These features are briefly outlined in the following subsections.

Dynamic Parallelism

In 2012, CUDA 5.0 introduced dynamic parallelism for Kepler and newer architectures (compute capability $\geq$ 3.5). Dynamic parallelism enables a kernel function to spawn a grid of threads without yielding control to the CPU. This reduces the necessity of device-host synchronizations and enables recursive task-parallelism on GPUs. By allowing a kernel function to spawn child kernels the latter facilitates the expression of concurrency in algorithms that exhibit nested parallelism or make use of hierarchical data structures.

Cooperative Groups

In 2017, CUDA 9.0 introduced the cooperative groups concept for Kepler and newer architectures (compute capability $\geq$ 3.5). The concept allows to define groups of threads in a grid beyond classical thread blocks. This enables finer grained control of synchronizations within thread blocks and between thread blocks. Both mechanisms are of particular interest for task parallelism on GPUs since they allow for the implementation of producer-consumer schemes. In addition to the basic cooperative groups concept supported by Kepler, Pascal enables grid-wide synchronization without yielding control to the host CPU. Volta goes yet a step further and allows to group threads at sub-warp level. In task-parallel applications, this can be employed to fine tune task granularity.

Asynchronous Task Graphs

With the introduction of *ATGs* in CUDA 10.0 in 2018, CUDA started to support task-parallelism explicitly. ATG enables task-parallelism with kernel-level granularity since it allows for the expression of dependencies between kernel functions in the host code. This enables the description

of a static task graph that allows for the automatic derivation of a concurrent execution order. Hence, it frees software developers from defining complex synchronization patterns via CUDA streams and events manually.

Scheduling

GPU execution models do not usually provide any formal forward progress guarantees. However, they are empirically observed to exhibit *warp-synchronicity* and *non-preemptive* thread block scheduling which both hinder forward progress. Warp-synchronicity means that threads are grouped together in schedulable units, with all threads of a schedulable unit sharing a single instruction counter and accordingly executing in lock-step. This effect may lead to warp-synchronous deadlocks if threads in the same warp compete for a mutex since not all lanes of the warp can acquire the mutex simultaneously. Since deadlock-free algorithms require some execution stream (here, any lane of the warp) to make progress, deadlock-free algorithms are not guaranteed to terminate on GPUs. Non-preemptive thread block scheduling, in turn, means that oversubscription leads to thread blocks not being executed till other thread blocks are terminated. Since starvation-free algorithms require every execution stream (here, thread block) to make progress, starvation-free algorithms are not guaranteed to terminate on GPUs.

The issue of warp-synchronicity is resolved with the introduction of *ITS* with the Volta architecture in 2018. Since ITS, each lane has got a dedicated program counter and call stack to retain its state of execution. This allows threads of the same warp to compete for a mutex without risking a deadlock. Hence, it enables the implementation of deadlock-free algorithms on GPUs. In contrast to the warp-synchronous scheduling model, this allows for fine-grained synchronization and concurrency between threads of the same warp. In fact, ITS is the basis for cooperative groups at sub-warp level.

In order for an architecture to support starvation-free algorithms, however, the scheduler must ensure that every thread in the grid – and not only threads in resident warps, can make progress independent of the progress of other threads. This, however, requires a scheduler to be preemptive. Whether or not the thread block scheduler of the CUDA runtime is preemptive is subject to scientific debate. [94] states that the scheduler is non-preemptive and therefore cannot provide general forward progress guarantees. Nvidia, however, claims in [81] that GPUs with Volta or newer microarchitectures support starvation-free algorithms. The CUDA programming guide, however, does not provide any relative forward progress guarantees for threads from different warps. Since the absence of non-preemptive scheduling only hinders starvation-freedom in the case of oversubscription, this work relies on grid sizes that are no larger than the amount of hardware resources.

5.6.2 Enhancing Uniformity

In addition to the outlined means to express concurrency more flexibly, the convergence of CPUs and GPUs is further underpinned by concepts to unify the programmability of both architecture. Firstly, uniform programmability is enhanced by the introduction of unified address space concepts, such as CUDA's unified memory, that hide the physical separation between host and device memory. Secondly, uniform programmability is further enhanced by the conformation of CUDA to C++, e.g. with the introduction of `libcu++` and the adoption of the C++ memory model.

5.6.3 Conclusion

Considering the fact that the described quantitative hardware trends are observed for AMD and Nvidia GPUs alike, OpenCL introduced similar concepts. For instance, OpenCL and CUDA both support dynamic parallelism, and OpenCL's *shared virtual memory* is similar to CUDA's *unified memory*. However, OpenCL and non-Nvidia GPUs do not yet support independent thread scheduling, grid-wide synchronization and starvation-free forward progress guarantees due to missing hardware requirements; the same holds true for OpenACC and OpenMP. Since these features are expected to enable the effectiveness of task-parallel programming approaches on GPUs, the implementations introduced in this work are written in CUDA.

Chapter 6

Synchronization is bad, but if you must ...
Olivier Giroux

Event-based Task Parallelism on GPUs

The objective of this chapter is to extend the concept of event-based task parallelism to GPUs. The key properties of event-based task parallelism as implemented by CPU-Eventify are scalability and sustainability, as empirically evaluated in [50] and [76]. Hence, GPU-Eventify aims similarly at scalability and sustainability.

For scalability the task engine of CPU-Eventify relies on three main concepts:

- *Work sharing* and *work stealing* via a dedicated priority queue per thread.
- *Ready-to-execute tasks* via event-driven dependency resolution and task generation
- *Dependency patterns* that follow user-defined data structures and task types.

In terms of sustainability, these concepts and their implementation could ideally directly be ported to GPUs. However, this is technically not feasible since the execution model behind them relies on MIMD threading and the availability of synchronization mechanisms, both of which are only available to a limited extent in the SIMD-dominated execution model of GPUs. Therefore, this chapter introduces a uniform architecture model and derives a GPU execution model that serves as a basis to implement event-based task parallelism on GPUs. This is a step towards software sustainability since it lays the foundation for the reuse of task-parallel CPU code on GPUs, and removes the necessity to force irregularly parallel applications into data-parallel structures and thereby risk losing parallelization potential.

6.1 Uniform Architecture Model

In this section, a Uniform Architecture Model (UAM) is derived from the comparison of CPU and GPU architectures in Chapter 4. The content of this section is closely based on a previously published work [74].

Terminology-wise, the presented UAM borrows from the platform model of OpenCL since this allows to describe an abstract architectural model that exhibits hierarchical concurrency. In the OpenCL platform model [65, p. 18] each compute device is subdivided into CUs, which are further subdivided into PEs. The mapping of CUs and PEs to actual hardware components is not determined by the OpenCL standard but specified by the software developer dependent on the actual hardware and parallelization scheme. This work derives such a mapping from the comparison of the architectural properties of CPUs and GPUs on the ISA level. The mapping considers components on the ISA level only since these are controllable by the software developer.

Following from the MIMD capabilities of CPUs and GPUs, the UAM equates a core on a CPU with an SM on a GPU. Hence, the UAM refers to CPU cores and SMs equally as CUs.

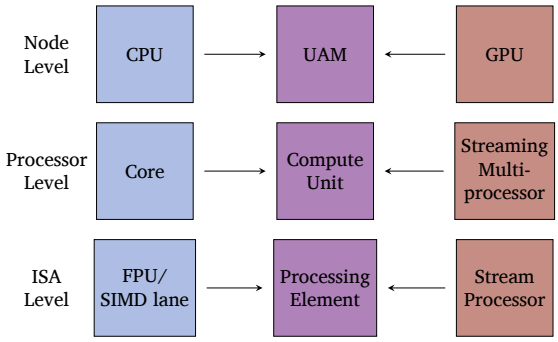


Figure 6.1: Mapping of CPU features and GPU features to components of the UAM on different architecture levels.

Following from the SIMD capabilities of both processor types, the UAM equates a SIMD lane on a CPU with an SP on a GPU. Therefore, the UAM denotes both components as *PE*, which is consistent with the stream interaction model and the OpenCL platform model. A CPU core exhibits SIMD units that execute an operation on multiple data elements simultaneously. For comparability of CPU and GPU architectures, we consider only single-precision floating point (FP32) SIMD operations in terms of `MUL`-, `ADD`- and `FMA`-units. Based thereon, each FPU and each SIMD-lane are referred to as a single PE. Regarding GPUs, we refer to each SP as a single PE.

Figure 6.1 shows the UAM by depicting the mapping between CPU and GPU features. Dependent on the scheduling strategy (see Section 5.6.1) that a GPU architecture supports, this principle does either lead to SIMD or MIMD concurrency. If SPs are operated in lockstep, this is similar to the SIMD parallelism as observable for CPU SIMD units. If, however, the scheduler supports ITS, this leads effectively to MIMD concurrency. Please note that this does not necessarily lead to an improvement in performance but enables the feasibility of starvation-free algorithms on GPUs.

6.2 Uniform Execution Model

The objective of the UEM is to bridge the gap between CPU and GPU execution models as foundation to approach fine-grained task parallelism on GPUs with the same concepts as on CPUs. To reach this objective, the UEM must emulate the concurrent processing capabilities of CPUs on GPUs based on the features that GPU programming models support. For this purpose, we adduce the comparison of CPU and GPU programming models as outlined in Chapter 5.

As outlined in 5.4.4, the SIMT programming model of GPUs supports both, SIMD and MIMD concurrency. While SIMD concurrency is naturally supported and aligns well with the architecture, MIMD concurrency on GPUs is less flexible and efficient than on CPUs. Hence, the UEM cannot treat each GPU thread the same as a fully independent CPU thread. This poses two challenges. Firstly, from a programming model perspective, threads are implicitly retired at kernel completion instead of executing the next kernel. This resembles a classical fork-join threading approach and hence impedes the efficiency of fine-grained parallelism. Secondly, the SIMT execution model of GPUs does not generally guarantee forward progress. Therefore, blocking algorithms – as employed by CPU Eventify – are currently considered inefficient and can even restrict liveness on GPUs.

This work addresses the first challenge by employing a persistent threads approach, and the second by suggesting appropriate mutual exclusion mechanisms.

6.2.1 Persistent Threads

This work employs the *persistent threads* paradigm on the GPU-side to emulate the behavior of CPU threads.

Definition 6.1. ***Persistent Threads.*** *Persistent threads are a GPU programming paradigm that launches a kernel with a grid size that corresponds to the maximal number of threads that can be active on a GPU simultaneously. Each thread in this grid executes a loop and is called a persistent thread.*

This is different from the classical GPU programming approach that maximizes the number of *resident* (instead of *active*) threads to leverage latency hiding and increase throughput. In fact, the persistent threads paradigm bypasses the hardware scheduler and instead employs a software-based load balancing approach [48].

Especially in the context of tasking on GPUs, the PT paradigm is typically used in conjunction with megakernels [96] [108]. A megakernel is a single, large kernel function that contains all code that an application executes on a GPU. While this approach in fact is closest to the threading model on CPUs, it is considered inefficient [70]. Firstly, since it increases the risk of control flow divergence which in turn decreases performance due to the SIMT execution model. And secondly, it increases register usage which decreases the number of resident warps and accordingly decreases the latency-hiding capabilities of the GPU. Instead of employing a single megakernel to execute all tasks, this work employs a hybrid approach and proposes the usage of one kernel per task type.

6.2.2 Thread Safety

This section introduces lock-based mutual exclusion mechanisms that can be used to ensure thread safety on GPUs. Firstly, a spin-lock implementation for GPUs in CUDA is described, and its guarantees regarding mutual exclusion and liveness are examined. Secondly, the `libcu++` -based lock implementations provided by the *freestanding* library [45] are stated to employ them for a comparative performance analysis in Section 8.4.2.

Eventify GPU Lock

Classical CAS-based spin-locks are proven to be correct and deadlock free under the following assumptions [52]:

1. **Sequential consistency** to ensure that “two memory accesses by the same thread, even to separate variables, take effect in program order”[52, p. 143];
2. **Memory coherence** to ensure that concurrent threads observe the same state of the lock;
3. **Forward progress guarantees** to ensure that a thread that holds a lock eventually releases it.

Subsequently, the validity of these assumptions on GPUs is assessed.

Sequential consistency

According to [78, p. 132], CUDA is based on a weakly-ordered memory model, i.e. it does not provide sequential consistency. Regarding atomic functions the documentation further explicitly states that [78, p. 152], “[a]tomic functions do not act as memory fences and do not imply [...] ordering constraints for memory operations”. Without sequential consistency, however, data-independent instructions could be reordered. Since acquiring the lock and executing an operation within the critical section are data-independent, a thread might enter the critical section before having acquired the lock. Following from this, CAS-based spin-locks on GPUs must use memory fences to enforce sequential consistency. This specification-based argumentation supports the empirical findings in [9] that observe the occurrence of race conditions for PTX-based spin-locks without memory fences.

Since PTX is the ISA behind CUDA, the CUDA-based spin-lock proposed in Listing 6.1 is derived from the PTX-based spin-lock presented in [9]. The lock makes use of CUDA’s `__threadfence()`[78, pp. 132]. This memory fence ensures that:

- for the calling thread, all reads from memory before the fence are ordered before all reads from memory after the fence, and
- for any thread in the grid, no writes to memory after the fence are observed as occurring before any write to memory before the fence.

For these ordering guarantees to hold, threads must access values in memory and not in cache or registers [78, p. 133], which leads us to the consideration of memory coherence on GPUs.

Memory Coherence

Memory coherence is required to ensure that threads do not read stale values of the lock state. In comparison to CPU architectures, GPU architectures do not implement a cache protocol and hence do not guarantee memory coherence. However, memory coherence can be enforced via the `volatile` type qualifier, or by the consistent use of atomic loads and stores. The GPU lock shown in Listing 6.1 relies on the latter since classical spin-locks in any case require the use of atomic operations.

Forward Progress Guarantees

As outlined in Section 5.6.1, GPUs do not generally provide forward progress guarantees due to a not necessarily preemptive scheduler and warp-synchronicity. The first limitation to forward progress cannot be lifted by the lock implementation itself but only at execution model level. The second limitation, on the other hand, can either be lifted at the architecture level via ITS, via warp-primitives as outlined in [63], or by the synchronization algorithm as outlined in this work.

The PT-based execution model circumvents the question of whether or not the scheduler is preemptive by spawning only as many threads as can be executed simultaneously. This ensures that only active threads can ever hold the lock and therefore guarantees forward progress at the warp-level.

Regarding warp-synchronicity, a case differentiation between GPU architectures with compute capability < 7.0 and ≥ 7.0 is required. The latter support ITS, which prevents warp-synchronous deadlocks at the architecture level and therefore guarantees forward progress also at sub-warp level.

On GPUs that do not support ITS, however, forward progress must be ensured at the programming model level. This work bypasses warp-synchronous deadlocks by requiring threads competing for a lock to belong to different warps. In contrast to locking mechanisms that rely on warp-primitives to avoid warp-synchronous deadlocks, this might limit the use cases in which the lock can be used but in return ensures portability.

CAS-based Spin-Lock in CUDA

Based on the considerations on consistency, coherence and progress, this work proposes the CUDA-based spin-lock in Listing 6.1 as solution to mutual exclusion on GPUs. The lock is implemented as a class `Mutex` that consists of a private counter variable `mutex` and the public methods `lock()` and `unlock()`. If `mutex == 0`, the mutex is unlocked. If `mutex == 1`, the mutex is locked. Initially, the mutex is unlocked.

In order to acquire the mutex by calling `lock()`, all threads follow the same busy-wait approach: check whether the `mutex` is unlocked; if it is unlocked, set it to locked; otherwise, repeat. Checking and updating the state of the counter variable is implemented atomically by CUDA's `atomicCAS(...)`. Firstly, this ensures that threads do not update the state of the mutex concurrently. And secondly, it ensures that threads do not read stale lock states due to incoherent memory. To ensure that the lock is acquired before any instruction within the critical section is executed, a memory fence is called directly after the CAS instruction.

If a thread holds the lock and calls `unlock()`, `atomicExch(...)` is called to free the lock by setting `mutex = 0`. This is preceded by another memory fence to guarantee that the lock is only released after the all operations within the critical section are completed.

Following from the preceding considerations on consistency, coherence and progress, this approach unconditionally guarantees mutual exclusion on all CUDA-capable GPUs. Liveness is

LISTING 6.1: GPU SPIN-LOCK

```
class Mutex
{
    public:
        __inline__ __device__ void lock()
        {
            while (atomicCAS(&mutex, 0, 1) != 0);

            __threadfence();
        };

        __inline__ __device__ void unlock()
        {
            __threadfence();
            atomicExch(&mutex, 0);
        };

    private:
        int mutex = 0;
};
```

examined under the assumption of a PT-based execution model that naturally guarantees warp-level forward progress since all threads in the grid are executed in parallel. [1] Hence, only sub-warp progress conditions are considered further to examine liveness. GPU architectures with compute capability ≥ 7.0 guarantee sub-warp forward progress. Hence, the proposed locking mechanism guarantees liveness on these architectures. On GPU architectures with compute capability < 7.0, however, liveness is only guaranteed if threads in the same warp do not compete for the same lock.

GPU Locks of Library freestanding

The *freestanding*[45] library implements a subset of the C++ STL. It is the predecessor of CUDA's C++ STL `libcu++` but meanwhile relies on the functionalities of `libcu++` itself. Based thereon, *freestanding* provides implementations of diverse locking mechanisms. Since `libcu++` adheres to the C++ memory model, these locking mechanisms rely on the properties of C++ atomics which are remarkably different from classical CUDA atomics. The main difference is that C++ atomics implicitly contain a memory barrier, and that this memory barrier can be configured to adhere to a specific memory consistency model. *freestanding* provides the following mutual exclusion mechanisms:

- **Spin lock**
- **Semaphore lock**
- **Ticket lock**

This work uses these locking mechanisms for a comparative performance analysis with the proposed Eventify GPU lock.

6.3 Eventify Execution Model on GPUs

This section outlines how the UEM is used to port Eventify's core concept – work sharing queues – to GPUs. Firstly, this section illustrates the conceptual differences between queue-based task scheduling on CPUs and GPUs. Secondly, the underlying queue data structure implementation is described. Thirdly, a taxonomy to describe hierarchical queueing schemes in a comparable way is introduced. And finally, the developed queueing schemes are presented.

[1] In theory, the guarantee for parallel execution of all persistent threads strictly follows from Definition 6.1. In practice, however, it requires adherence to kernel configuration limitations at runtime, i.e. there must not be more threads in the grid than can be active at the same instant such that no oversubscription of actual hardware resources takes place.

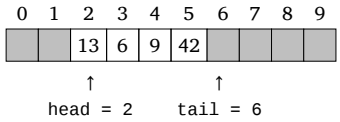


Figure 6.2: Queue based on a fixed-size array that is operated as ring buffer. New elements are inserted into the queue via `enqueue(e)` at `tail`, while elements are deleted from the queue via `dequeue(n)` at `head`. Read access to an element at position $head + i$ of the queue is granted via `front(i)`.

6.3.1 Comparison of CPU and GPU Queueing Principles

Just as CPU-Eventify, GPU-Eventify relies on queues to store tasks that are ready to execute. However, there are major differences in the data structure design and application that stem from the architecture-driven differences in the GPU execution model.

Firstly, GPU-queues hold tasks of a single type only; in contrast to CPU-queues they are accordingly not multi-queues. This is due to the fact that each PT kernel executes tasks of a single type only, which alleviates the performance loss that would be induced by a megakernel approach.

Secondly, a GPU-queue is owned by a group of threads, meaning that multiple threads consume tasks from the same queue. This is reasonable since GPU-threads are executed within schedulable units, e.g., warps or half-warps. Even if a GPU architecture provides ITS, threads should not be considered to execute fully independent for reasons of performance.

These differences hold true for all queueing schemes introduced in Sections 6.3.4ff.

6.3.2 Data Structure

Figure 6.2 illustrates the base queue and its terminology as used throughout this work. All described task queues rely on a statically allocated array of size N that is operated as a ring buffer; this approach is based on [26, p. 234]. `head` is a pointer that points to the first element of the queue, while `tail` is a pointer that points to the next free location, i.e., the address directly after the current last element in the queue. Initially, *head* and *tail* point to index 0 of the array. The queue provides the modifying operations `enqueue(e)` and `dequeue(n)`:

- **`enqueue(e)`** inserts element e at position *tail* into the queue and sets $tail = (tail + 1) \mod N$.
- **`dequeue(n)`** deletes n elements at positions $head, head + 1, \cdots, head + (n - 1)$ from the queue and sets $head = (head + n) \mod N$

Furthermore, it provides the non-modifying queries `front(i)` and `size()`:

- **`front(i)`** returns a reference to the element at position $head + i$ of the queue.
- **`size()`** returns the number of elements in the queue.

For reasons of performance, the queue implementation does not safe-guard against overflowing. If the queue is full, i.e. if *tail* overtakes *head*, the element at *tail* will be overwritten by `enqueue(e)` with e.

6.3.3 Taxonomy

We propose a taxonomy for the classification of task queues that serves as a uniform basis for the subsequent description of queueing schemes.

Properties

The taxonomy covers the following properties:

Memory Location describes where the queue resides. Considering the memory hierarchy of GPUs as exposed by the programming model, a task queue can be located either in *shared memory* or *global memory*.

Memory Allocation describes through which mechanism the memory for the queue is acquired. This may be done via *static*, *dynamic* or *custom* allocation.

Read/Enqueue/Dequeue Synchronization states which thread safety mechanism is applied to guard read/enqueue/dequeue access to the queue. If no thread safety mechanism is required, the value of this property is *none*. If synchronization is implicitly achieved through the algorithmic access pattern, the value of this property is *lock-less*. If synchronization is explicitly achieved through a mechanism for mutual exclusion, the value of this property is *mutex*. The latter may be supplemented by a specific mechanism such as spin-lock, semaphore or ticket mutex. Further values, e.g., *lock-free* or *wait-free*, are possible but not considered in this work since they are already vigorously examined elsewhere [63] [21] [106].

The **Read/Enqueue/Dequeue Access Scope** describes which entities of parallel execution are allowed to read elements from, enqueue elements to or dequeue elements from the queue, respectively. In the context of the GPU programming model these entities may be all or specific threads of a *block*, or all or specific threads of the whole *grid*.

Read/Enqueue/Dequeue Access Parallelism states the entities of parallel execution from a determined access scope that are allowed to access the queue for reading, enqueueing or dequeueing in parallel.

The read/enqueue/dequeue synchronization, access scope and access parallelism are each provided as tuple (r, e, d).

Notation

In order to describe task queueing along the UEM in a consistent manner, the following notation is introduced (as derived from Section 5.4):

- D_{grid}, the total number of thread blocks in grid G, i.e. the grid dimension,
- D_{block}, the total number of threads per thread block in grid G, i.e. the block dimension,
- N_{thread}, the total number of threads in grid G with $N_{\text{thread}} = D_{\text{grid}} \cdot D_{\text{block}}$,
- t_i^{l}, a thread with local thread ID i,
- t_i^{g}, a thread with global thread ID i,
- B_i, a thread block with block index i as D_{block}-tupel $(t_0^{\text{l}}, \dots, t_{D_{\text{block}}-1}^{\text{l}})$ of its threads,
- m_i, the master thread of the thread block B_i,
- G, a grid as N_{thread}-tuple $(t_0^{\text{g}}, \dots, t_{N_{\text{thread}}-1}^{\text{g}})$ of its threads,
- M, all master threads of a grid as D_{grid}-tupel $(m_0, \dots, m_{D_{\text{grid}}-1})$,
- m, a single, arbitrary master thread, $m \in M$.

For conciseness, all symbols are defined relative to a single grid G. Where necessary, the executed kernel function k is added as index to G to distinguish different grids, e.g. G_{computeA} and G_{computeB}.

This notation allows to address threads, thread blocks and master threads in a one-dimensional grid G. In fact, all introduced concepts rely on one-dimensional index spaces only due to the PT approach. Following the CUDA programming model, local thread IDs are accordingly coextensive with their local thread index and global thread IDs are coextensive with their global thread index.

6.3.4 Queueing Schemes

The subsequently introduced queueing schemes are described by specifying the properties described in Section 6.3.3 for each type of queue in a queueing scheme; in addition, the quantity of queues is provided for each queue type.

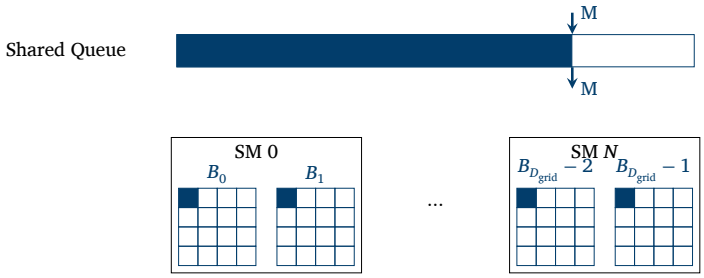


Figure 6.3: Block diagram of the task queueing scheme with a single MPMC queue.

Single MPMC Queue (SQ)

The most basic task queueing scheme considered in this work consists of a single MPMC-queue as shown in Figure 6.3. In combination with the queue usage scenario in Listing 6.2, this results in the characteristics provided in Table 6.1.

As can be seen from Figure 6.3, the queue is located in global memory. This way, it is accessible to all threads of the grid G. Memory is allocated statically since the queue is based on a fixed-size array.

All threads of the grid G are allowed to access the queue via non-modifying queries, while only master threads M are allowed to modify the queue with `dequeue()` and `enqueue()` operations. This leads to an access scope of (G, M, M).

Even though all threads of the grid G are allowed to read from the queue, only threads of the same block B_i are allowed to read from the queue in **parallel**. For parallel reading, each thread of block B_i calls `front(k)` at its thread index k. Thus, each thread reads a single task from the queue. `enqueue()` and `dequeue` operations, however, are executed sequentially by the master thread m_i of block B_i. This leads to an access parallelism of (B_i, m_i, m_i) as shown in Table 6.1.

The characterization as MPMC-queue is based on the fact that *multiple* master threads *produce* tasks by enqueueing them to the queue and that *multiple* master threads *consume* tasks by dequeuing them from the queue; it does not acknowledge the parallel execution of tasks by threads of the same thread block. Hence, the name reflects modifying operations on the queue (as it would for CPU queues) but does not reflect the execution concurrency of tasks on thread block level.

Applied synchronization mechanisms can be derived from Listing 6.2. Modifying operations are protected by a mutex since all thread block masters access the queue concurrently. Non-modifying queries are also protected by a mutex to avoid inconsistent query results, even though these would not invalidate the queue itself. Only threads of the same block B_i read from the queue in parallel. The mutex for this operation is acquired by the master thread m_i of B_i. Hence, only master threads compete for the mutex. First, this allows for the warp-synchronous deadlock-free use of the mutex implementation provided in Listing 6.1. Second, it reduces lock contention by a factor of D_{block} since only one thread per thread block competes for the mutex.

The implementation of this task-pool-based approach serves as a proof-of-concept that the Eventify-like management of fine-grained tasks is possible on GPUs. However, the approach suffers from a synchronization bottleneck that hurts performance. The mutex is applied to protect a rather long critical section (Lines 6 to 13 in Listing 6.2) covering task execution and dependency resolution instead of the modifying queue operations only. This enables intra-block parallelism since it allows all threads of a block to read and execute tasks from the queue in parallel. However, it impedes inter-block parallelism since at each point in time only one thread block executes tasks, while all other blocks are waiting to acquire the global mutex. To resolve this synchronization bottleneck, we propose the subsequently introduced queuing scheme with multiple shared queues.

Table 6.1: Task queueing scheme with a single, shared MPMC-queue.

	MPMC Queue
Quantity	1
Memory Location	global memory
Memory Allocation	static
(r,e,d) Access Scope	(G, M, M)
(r,e,d) Access Parallelism	(B_i, m_i, m_i)
(r,e,d) Synchronization	(mutex, mutex, mutex)

LISTING 6.2: KERNEL FOR ACCESS TO SINGLE SHARED QUEUE

```
1  produce_and_consume<T>()
2  {
3    while(!finished())
4    {
5      block_master:
6        mutex.lock();
7      syncthreads();
8      block:
9        queue<T>.front(threadIdx.x).execute();
10     syncthreads();
11     block_master:
12       solveDependencies();
13       mutex.unlock();
14     syncthreads();
15   }
16 }
```

Multiple MPSC Queues (MQ)

With the multiple MPSC queueing scheme, we aim to reduce lock contention and enable inter-block parallelism. As shown in Figure 6.4, the scheme employs one task queue per thread block. In combination with the queue usage scenario in Listing 6.3, this results in the characteristics provided in Table 6.2.

As can be seen from Figure 6.4, all D_{grid} MPSC queues are located in global memory. This is required since thread blocks do not only require access to their own queue, but also to other queues for reasons of load balancing. Similarly to the single MPMC queue, memory for each queue is allocated statically since the queues are based on a fixed-size array each.

The characterization of each queue as MPSC-queue is based on the fact that *multiple* master threads *produce* tasks by enqueueing them to a queue Q_i but only a *single* master thread, namely m_i of block B_i, *consumes* tasks by dequeuing them from Q_i.

Each thread block B_i owns a queue Q_i. Each thread block B_i consumes tasks from Q_i only; it does not consume tasks from any other queue and no other block consumes tasks from Q_i. Hence, read access to Q_i is required by the threads of B_i only. Enqueue access, on the other hand, is granted to all master threads M of the grid for work sharing. Dequeueing is done by the master thread m_i of B_i only. This leads to an access scope of (B_i, M, m_i).

Since the intra-block parallelism from the single MPMC approach should be preserved, all threads of B_i can read and consequently execute tasks from Q_i in parallel. Considering a single queue Q_i, enqueueing sequentializes since only one master thread m can enqueue a task to Q_i at each point in time due to mutex protection. Considering a single queue, there is also no dequeuing parallelism since a master thread m_i dequeues tasks from its own queue only. This leads to an access parallelism of (B_i, m, m_i). Hence, this scheme does not provide more per-queue access parallelism than the MPMC approach with an access parallelism of (B_i, m_i, m_i). Nevertheless, the overall amount of access parallelism is increased due to the splitting of the single MPMC into D_{grid} block-owned queues. In comparison to the SQ scheme this allows all blocks of the grid to execute tasks in parallel, which increases inter-block parallelism by a factor of D_{grid}. As can be seen in

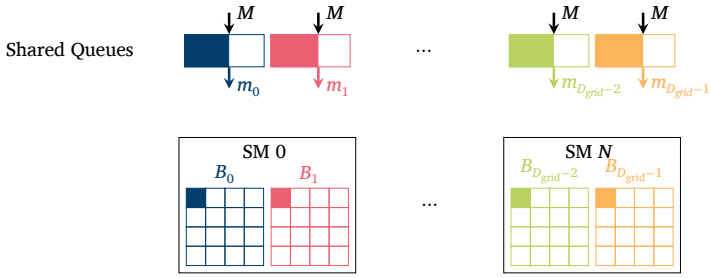


Figure 6.4: Block diagram of the queueing approach with multiple MPSC queues.

Table 6.2: Task queueing scheme with multiple MPSC queues

	MPSC Queue
Quantity	D_{grid}
Memory Location	global memory
Memory Allocation	static
(r,e,d) Access Scope	(B_i, M, m_i)
(r,e,d) Access Parallelism	(B_i, m, m_i)
(r,e,d) Synchronization	(lock-less, mutex, lock-less)

Listing 6.3: Kernel for Access to Multiple Shared Queues

```
1  produce_and_consume<T>()
2  {
3    while(!finished())
4    {
5      block:
6        shared_queues<T>[blockIdx.x].front(threadIdx.x).execute();
7      syncthreads();
8      block_master:
9        mutex[blockIdx.x].lock();
10       solveDependencies();
11       mutex[blockIdx.x].unlock();
12     syncthreads();
13   }
14 }
```

Listing 6.3 this also reduces the size of the critical section (Lines 9 to 11) in comparison to the critical section in the SQ scenario. In contrast to SQ, it covers the resolution of task dependencies only.

The tasks of a queue are consumed by threads of the same thread block only, with each thread accessing a distinct task based on its local thread ID t_i^1. Therefore, read access is granted lock-less. Hence, the read access in Line 6 of Listing 6.3 is not protected by a mutex. Enqueue access, on the other hand, is protected by a mutex since multiple master threads can enqueue tasks concurrently. Dequeue access is granted lock-less since only the master thread is allowed to dequeue tasks. Under the assumption of load balance, this reduces the overall contention in the system by a factor of D_{grid} since contention is distributed to D_{grid} queue mutexes.

Multiple Hierarchical Queues (MHQ)

The MHQ scheme extends the MQ scheme by a second level of block-owned queues that are fully private to each thread block, i.e. tasks of these queues can be read, enqueued and dequeued by threads of this block only. Therefore, the scheme consists of two queues per thread block; one MPSC queue as described by Table 6.2 and one SPSC queue as described by Table 6.3. Figure 6.5

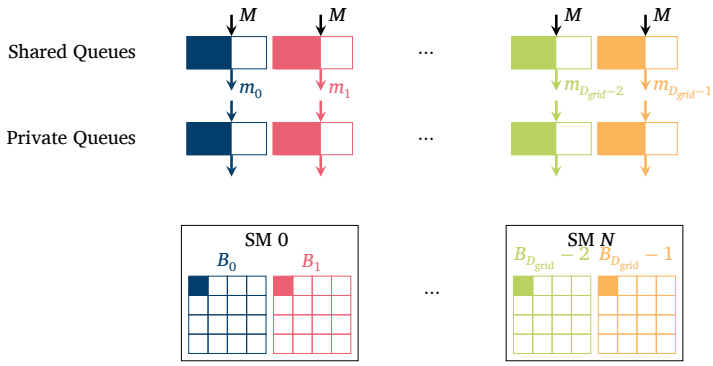


Figure 6.5: Block diagram of the queueing approach with multiple, hierarchical queues.

Table 6.3: SPSC queue of task queueing scheme with multiple, hierarchical queues.

	SPSC Queue
Quantity	D_{grid}
Memory Location	global memory
Memory Allocation	static
(r,e,d) Access Scope	(B_i, m_i, m_i)
(r,e,d) Access Parallelism	(B_i, m_i, m_i)
(r,e,d) Synchronization	(lock-less, lock-less, lock-less)

provides an overview of this queueing scheme and Listing 6.4 shows the corresponding access algorithm.

Regarding queue properties, this section focuses on the description of the private SPSC queues since the MPSC queues are adopted from the MQ scheme. The MHQ scheme consists of D_{grid} MPSC queues and D_{grid} SPSC queues, all of which are statically allocated and located in global memory. Based thereon, each thread block B_i consumes tasks from its private queue Q_i^{p} and its shared queue Q_i^{s}. For reading and dequeuing, both queues are accordingly only accessed by the threads of B_i. Regarding enqueue access, however, the private queue Q_i^{p} varies from the shared queue Q_i^{s}. Only the master thread m_i of B_i can enqueue tasks to Q_i^{p}, while all master threads M can enqueue tasks to the shared queue Q_i^{s}. Accordingly, the (r,e,d) access scope of the private queue is (B_i, m_i, m_i).

Due to the limited access scope and parallelism of the SPSC queue, all queue operations can be performed lockless. Hence, the (r, e, d) synchronization is described as (lock-less,lock-less,lock-less).

The purpose of the second level SPSC queues is twofold. Firstly, they are used for the management of initial tasks, which makes them comparable to the back-fill queues employed by CPU-Eventify. Secondly, they are used by the master thread of the owning block to enqueue tasks for its threads directly. This reduces contention on their MPSC counterparts since the master thread does not compete with other master threads to access its own queue. The extent to which contention is reduced through this principle depends on the static task partitioning of the application algorithm. In the best case scenario, in which a task and all its successors are executed by the same thread block, this prevents contention fully. In the worst case scenario, in which none of a tasks successors is executed by the same thread block, contention is equally as high as in the MQ scheme.

LISTING 6.4: KERNEL FOR ACCESS TO MULTIPLE HIERARCHICAL QUEUES

```
produce_and_consume<T>()
{
  while(!finished())
  {
    block:
      if(private_queue<T>.size() > threadIdx.x)
        private_queue<T>.front(threadIdx.x).execute();
      else if(shared_queues<T>[blockIdx.x].size() > threadIdx.x-private_queue<T>.size())
        shared_queues<T>[blockIdx.x].front(threadIdx.x).execute();
    syncthreads();
    block_master:
      mutex[blockIdx.x].lock();
      solveDependencies();
      mutex[blockIdx.x].unlock();
    syncthreads();
  }
}
```

6.4 Implementation

This section briefly outlines the implementation of Eventify on GPUs. It provides an overview of the software architecture and outlines the general workflow. For the exact implementation, please refer to `https://code.fmsolvr.fz-juelich.de/ATML-SE/eventify-GPU`.

6.4.1 Software Architecture

Eventify for GPUs is based on the core concepts of Eventify for CPUs. For reasons of sustainability, this work aims to adjust the smallest number of classes as possible with only the minimal code changes that are required to port the functionality to CUDA. This involves:

- Finding substitutes for STL functions and data structures either by considering `thrust` (std::pair → thrust::pair) and `libcu++` or by reimplementing a reduced version that at least matches the requirements of Eventify (e.g. `std::vector` → `gtl::vector`, `std::tupel` → `gtl::tupel`, `std::forward` → `gtl::forward`),
- Circumventing `C++` language features that are not yet supported by CUDA, such as constructor inheritance via `using`,
- Adding `__host__` and `__device__` decorators, and
- Reimplementing functionality that does not fit the execution model of GPUs, especially, substituting Eventify's multi-queue by the queueing schemes outlined in Section 6.3.4.

This procedure is applied to classes that are vital for the execution of fine-grained tasks only. Classes that are concerned with the implementation of Eventify's dependency pattern DSL are beyond the scope of this work.

Figure 6.6 shows the UML diagram of GPU Eventify.

6.4.2 Workflow

Algorithm 1 outlines the workflow of Eventify on GPUs. Functions highlighted in blue are executed on the GPU.

The workflow commences by reading in the user-defined block and grid dimensions. Since these are highly dependent on the application, they cannot be automatically determined by Eventify. Afterwards, `InitData` is instantiated and transferred to the GPU. Next, the `create_global_objects` kernel is called to initialize `InitData` and the dependency counters in a user-defined `DependencyStructure`. The latter corresponds to the idea of dependency patterns as outlined in Section 5.3 and is the prototypical substitute of the DSL on GPUs. Since each `produce_and_consume` kernel allows for the execution of one task type only at a time, a user-defined number of CUDA streams is defined to allow for overlapping. Finally, the `print_result` kernel is called to output the results.

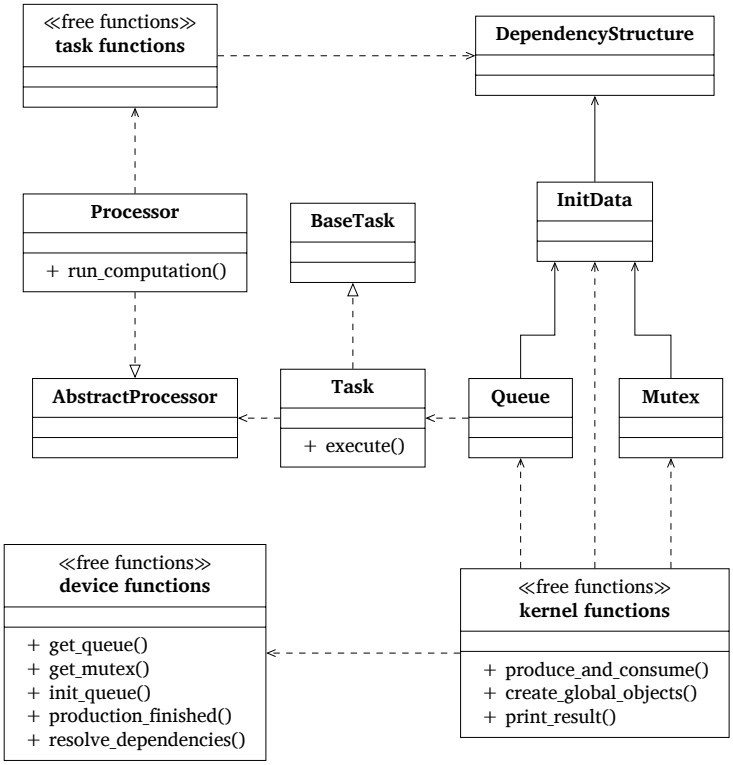


Figure 6.6: UML class diagram of the software architecture of Eventify on GPUs.

Algorithm 1 Workflow of Eventify

1: Read in user defined kernel configuration D_{block} and D_{grid}
2: Instantiate `InitData` on the host and create device pointer
3: Allocate device-side memory for members of `InitData`
4: Transfer `InitData` from host to device
5: `cudaDeviceSynchronize()`
6: Call `create_global_objects` to initialize `InitData` and `DependencyStructure`
7: `cudaDeviceSynchronize()`
8: Create CUDA streams for concurrent kernel execution asynchronously
9: Launch `produce_and_consume` kernels for event-based task execution
10: `cudaDeviceSynchronize()`
11: Launch `print_result` kernel
12: `cudaDeviceSynchronize()`

Chapter 7

We can solve [the software crisis in parallel computing], but only if we work from the algorithm down to the hardware — not the traditional hardware first mentality.
Tim Mattson

Use Case: Fast Multipole Method

This chapter provides a short overview of the N-body problem of electrostatics and how it can be solved with the FMM. It introduces *FMSolvr*, a C++-implementation of the FMM for molecular dynamics, and its software architecture. Based thereon, we introduce a data-parallel OpenMP-version of *FMSolvr* for CPUs, a data-parallel OpenACC-version for GPUs as well as task-parallel *Eventify*-versions for CPUs and GPUs. We examine the influence of these parallelization approaches on the software architecture of *FMSolvr*.

7.1 N-Body Problem of Electrostatics

In biochemistry and materials science MD simulations are used to analyze the evolution of particle ensembles over time. According to [61], the most expensive part of MD simulations is the computation of pairwise long-range interactions such as Coulomb interactions.

Derived from the definition of the numerical N-body problem provided in [47], we refer to the N-body problem of electrostatics as *the determination of all pairwise forces (or potentials) for a fixed configuration* of N particles that interact electrostatically. Following [16], the N-body problem of electrostatics requires the evaluation of the Coulomb potential

$$\Phi(\boldsymbol{x}_j) = \sum_{\substack{i=1 \\ i \neq j}}^{N} \frac{q_i}{r_{ij}}, \tag{7.1}$$

as well as the electrostatic field

$$\boldsymbol{E}(\boldsymbol{x}_j) = \sum_{\substack{i=1 \\ i \neq j}}^{N} q_i \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{r_{ij}^3}, \tag{7.2}$$

and the Coulomb force

$$\boldsymbol{F}(\boldsymbol{x}_j) = q_j \sum_{\substack{i=1 \\ i \neq j}}^{N} q_i \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{r_{ij}^3}, \tag{7.3}$$

acting on each particle j. The location of particle j is denoted as $\boldsymbol{x}_j$, while r_{ij} corresponds to the Euclidean distance between $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$. The charge of particle j is denoted as q_j.

MD simulations are used to determine trajectories for all particles in the system. Therefore, each time step of the simulation covers the computation of the Coulomb potential $\Phi(\boldsymbol{x}_j)$, electrostatic

field $\boldsymbol{E}(\boldsymbol{x}_j)$ and Coulomb force $\boldsymbol{F}(\boldsymbol{x}_j)$ for each particle in the system. Hence, the direct computation of all pairwise interactions via a classical Coulomb solver results in double sums and accordingly a computational complexity of $\mathcal{O}(N^2)$. Since realistic systems contain millions of particles, this direct approach is not feasible. Therefore, more efficient methods, such as the FMM[47] and PME [28], have been developed.

7.2 Sequential Fast Multipole Method

The FMM is a hierarchical fast summation method for the evaluation of Coulomb interactions in MD simulations. It computes the Coulomb force $\boldsymbol{F}_j$ acting on each particle j, the electrostatic field $\boldsymbol{E}_j$ and the Coulomb potential Φ_j in each time step of the simulation. From the Coulomb force $\boldsymbol{F}_j$ the particle's position in the next time step is computed. The FMM reduces the computational complexity of classical Coulomb solvers from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$ by use of multipole expansions for the computation of long-range interactions.

7.2.1 Input Parameters

The input data set to set up the FMM tree consists of location $\boldsymbol{x}$ and charge q of each particle in the system as well as multipole order p, maximal tree depth d_{max} and well-separateness criterion ws as parameters. The multipole order p, the maximal tree depth d_{max} and the well-separateness criterion ws influence the time to solution and the precision of the results.

7.2.2 Hierarchical Space Decomposition

The FMM starts out with a hierarchical space decomposition of the simulation space to group particles. This is done by recursively bisecting the simulation box in each of its dimensions. For a 1D simulation space this yields a binary tree, for a 2D simulation space a quad-tree, and for a 3D simulation space an octree. The developing tree structure is referred to as FMM tree, and consists of $d_{max} + 1$ levels $d = 0, \dots, d_{max}$. Subsequently, the relations between the boxes of the FMM tree are introduced referring to [16] as:

- **Parent-child relation**: A box b_p is parent box of box b_c if b_p and b_c are directly connected when moving towards the root of the tree.
- **Near neighbor**: Two boxes are near neighbors if they are at the same refinement level d and share a boundary point; a box is a near neighbor of itself.
- **Interaction set**: The interaction set of a box b_c is the set consisting of the children of the near neighbors of b_c's parent box b_p which are well separated from b_c.
- **Well separateness**: Two boxes are said to be well separated if they are at the same refinement level d and are not near neighbors. Only well separated boxes interact in the far-field via multipoles.

7.2.3 Workflow

The sequential workflow of the FMM referring to [60] is stated in Algorithm 2; blue steps compute the near field interactions, green steps compute the far field interactions. A direct solver that follows Equations 7.1, 7.2 and 7.3 is used to evaluate the pair-wise near field interactions between particles and the particles in their near neighborhood. Multipole and local Taylor-like expansions are used to approximate the far field interactions between distant clusters of particles.

The workflow is described from a data structural and data dependency perspective since the actual FMM operations, such as multipole and Taylor expansion shifts and translations, are not relevant in the scope of this work. Therefore, these operations are not considered further but handled as black boxes. This is reasonable since the efficient implementation of FMM operators is architecture dependent and would therefore distort the analysis of the performance and scaling behavior of the proposed task-parallel programming approach. Please refer to [75] for a high-level introduction to the FMM operations, or to [60] for a deeper understanding of the operators, their derivation, complexity and error-control.

Algorithm 2 Fast Multipole Method

Input: Location $\boldsymbol{x}$ and charge q of each particle in the system
Output: Electrostatic field $\boldsymbol{E}$, Coulomb force $\boldsymbol{F}$ and Coulomb potential Φ for each particle

Set up FMM tree:
Group particles by means of hierarchical space decomposition
Particle to Multipole (P2M):
Expansion of particles in each box on the lowest level $d_{\max}$ of the FMM tree into multipole moments ω relative to the center of their box.
Multipole to Multipole (M2M):
Accumulative upwards-shift of the multipole moments ω to the centers of the parent boxes.
Multipole to Local (M2L):
Translation of the multipole moments ω of the boxes covered by the interaction set of box b_i into a local moment μ for b_i.
Local to Local (L2L):
Accumulative downwards-shift of the local moments μ to the centers of their child boxes.
Local to Particle (L2P):
Translation of the local moment μ of each box b_i on the lowest level $d_{\max}$ to each particle $\boldsymbol{x}_i$ of this box.
Particle to Particle (P2P):
Evaluation of the interactions between the particles contained by the near neighbors of a box b_i for each box on the lowest level $d_{\max}$ via a classical *ws* range-limited Coulomb solver.

7.2.4 Assumptions

The input data set, parameters, boundary conditions, operator complexity and implementation have a vital impact on the runtime and accuracy of an FMM implementation. Therefore, this section discusses which properties are relevant for performance and scaling in the context of tasking, which of them are kept constant, or are abstracted to ensure comparability. For the remainder of this work, the following assumptions hold true:

- **Simulation space**: if not explicitly stated otherwise, a three-dimensional simulation space is used for theoretical considerations and performance measurements alike since it is the most commonly used dimensionality in MD simulations.
- **Boundary conditions**: periodic boundary conditions are applied throughout to exclude edge cases.
- **FMM tree depth**: $d_{\max}$ is varied to vary the task graph size.
- **Well separateness**: $ws = 1$ applies constantly because it reduces the number of direct interactions to a minimum and resembles the most common use case for the FMM [60, p. 28].
- **Operators**: all operator-related properties are disregarded since each multipole and Taylor expansion is substituted by a scalar addition to model minimal computational load so that bottlenecks in the tasking approach (rather then the algorithmic optimizations of the FMM) can be identified.

7.3 Parallel Fast Multipole Method

As outlined in Section 3.2.1, the degree of concurrency of an algorithm depends on its data dependency graph. Therefore, this work introduces the data dependency graph of the FMM to estimate the theoretical parallelization potential of task-parallelism in comparison to data-parallelism.

7.3.1 Data Dependency Graph

Figure 7.1 shows the data dependency graph of the FMM with periodic boundary conditions for a one-dimensional simulation space[1]. The base structure of the data dependency graph is a binary FMM tree. For clarity, the tree is plotted on a radial map, with its root node in the centre, and inner nodes and leaf nodes arranged on concentric circles. Each node in the graph represents a box in the FMM tree, with the node color representing different data types, e.g. a box with either particles, multipoles ω or local moments μ. Each directed edge (u, v) represents an FMM operation with the data elements of box b_u as input, and the data elements of box b_v as output.

The data dependency graph represents data dependencies between boxes, which represents the finest level of granularity that is considered in this work since particle granularity would lead to the data dependency graph being dependent on the input data set. Therefore, in 3D the FMM operations are defined as follows:

- P2M: operates on a single box on the lowest level by expanding all particles in the box into a multipole.
- M2M: translates the multipole moments ω of eight child boxes into a multipole at the center of the parent box.
- M2L: executes an M2L operation for all boxes in its interaction list.
- L2L: translates the local moment μ of a parent box into local moments at the centers of its eight child boxes.
- L2P: shifts the local moment *mu* of a box to all the particles in the according box.
- P2P: computes the near field interactions for all particles in a box with respect to its well separated neighbors.

The data dependency graph is composed of six structures, one per pass of the FMM. Following the critical paths through the graph, the subsequent structures are evident:

- a P2P edge between each particle-type node and its near neighbors,
- a P2M edge from each particle-type node to each ω-type leaf node,
- an in-tree[2] of M2M dependencies connecting ω-type nodes along the parent-child relation,
- level-wise M2L dependencies from the nodes in the M2M tree to the nodes in the L2L tree following the interaction set,
- an out-tree of L2L dependencies connecting μ-type nodes along the parent-child relation,
- an L2P edge from each μ-type leaf node to each particle-type node.

Next, the theoretical degree of concurrency is determined for a data-parallel FMM in comparison to a task-parallel FMM with different levels of granularity.

[1] Please note that the one-dimensional simulation space has only been chosen since the high density of data dependency graphs for higher dimensional simulation spaces is not reasonably visualizable.

[2] In graph theory, *in-tree* denotes a rooted tree in which all edges are directed towards the root, and *out-tree* denotes a rooted tree in which all edges are directed away from the root.

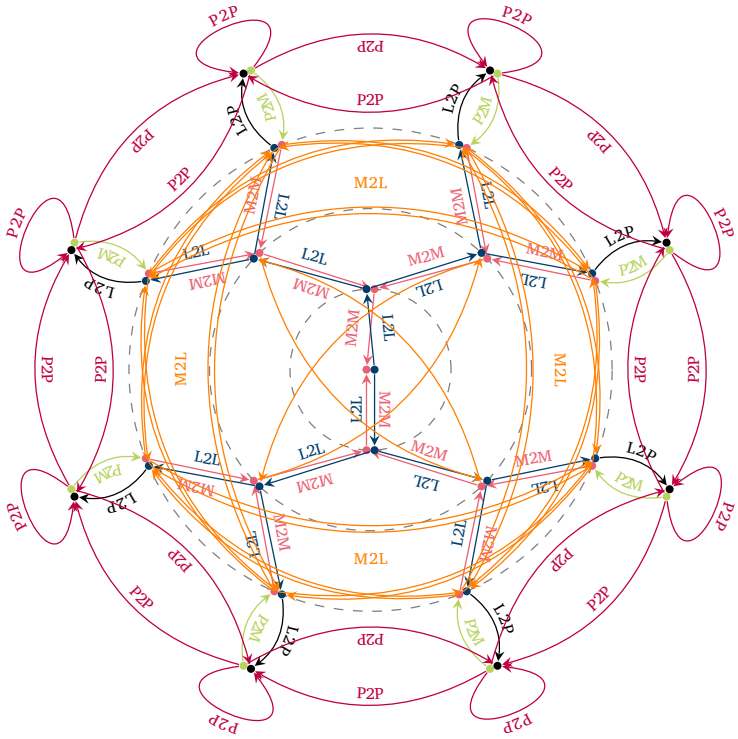


Figure 7.1: FMM data dependency graph for a 1D simulation space.

7.3.2 Data-Parallel FMM

The classical loop-based data-parallel approach adheres to the sequential workflow of the FMM but processes each pass via a fork-join model. Therefore, even data independent passes cannot overlap and the degree of concurrency is limited by the degree of concurrency in each pass. If a pass operates on multiple levels, then the maximal degree of concurrency is reached on the lowest (and therefore widest) level that the pass operates on.

In general, the degree of concurrency in full trees decreases with branching factor b per level when traversing the tree bottom-up, i.e. lower levels in the tree exhibit more parallelism than higher levels. Hence, the level-wise degree of concurrency can be modeled as a geometric series

$$\sum_{i=0}^{d_{\max}-1} b^i \,. \tag{7.4}$$

For the FMM in three dimensions $b = 8$ holds.

For P2M, this results in $8^{d_{max}}$ loop iterations, one per box on the lowest level, that can be executed simultaneously. This holds similarly for P2P and L2P.

Due to the in-tree structure of M2M, only M2M operations on the same level can be executed simultaneously. This results in $8^{d_{max}-1}$ as degree of concurrency since level $d_{max} - 1$ is the lowest level on which M2M operates. This holds similarly for L2L due to its out-tree structure.

In contrast to M2M and L2L, M2L does not exhibit inter-level dependencies since it only operates horizontally in the tree. The fork-join approach for the data-parallel FMM further guarantees that M2M is completed on all levels before M2L commences execution. Due to these preconditions, all M2L operations in the tree can be executed simultaneously, which leads to a maximal degree of concurrency $\sum_{i=0}^{d_{\max}} 8^i$.

7.3.3 Task-Parallel FMM

Following from Algorithm 2, six task types can be derived, namely P2M, M2M, M2L, L2L, L2P and P2P. However, dependent on the chosen task granularity, different task graphs can be derived from the data dependency graph.

Pass granularity as the most coarse granularity is reached when each pass is considered as a single sequential task. Hence, a task graph with pass granularity consists of exactly six tasks. While P2M, M2M, M2L, L2L, L2P must be executed in order, only P2P can be executed simultaneously. With respect to pass-size tasks, the maximal degree of concurrency in this scenario is 1(Far Field) + 1(Near Field) = 2.

Level granularity leads to medium granularity and is achieved when all operations of the same type that are located on the same level are considered as a single sequential task. In addition to the overlapping of passes as supported by pass granularity, this also allows for overlapping within passes, e.g. the overlapping of the M2L task on level d_{max} with the M2L task on level $d_{max} - 1$ under the assumption that M2M on level $d_{\max} - 1$ is already completed. The latter can be guaranteed since the number of M2L operations on level $d_{\max}$ is $8^{d_{max}}$ and the number of all M2M operations is only $\sum_{i=0}^{d_{\max}-1} 8^i$. An M2L operation further consists of $189 \cdot \mathcal{O}(p^3)$ operations, while an M2M operation consists of $8 \cdot \mathcal{O}(p^3)$ operations only. Hence, the number of M2L operations on the lowest level is strictly larger than the number of all M2M operations. Under the assumption of comparable prefactors for M2M and M2L, this implies that the compute time of all M2M operations can be fully hidden by M2L from the lowest level only. Concluding, this leads to a maximal degree of concurrency of 1(P2P) + 1(M2M) + 1(M2L) = 3.

Box granularity is the finest granularity considered in this work, since it is the finest level of granularity leading to a static task graph. Box granularity allows for the degree of concurrency of data parallelism, in addition to the overlapping of tasks from different levels and passes as enabled by level granularity. Following from the fact that P2P on level $d_{\max}$, M2M on level $d_{\max} - 1$ and M2L on level $d_{\max}$ can be executed simultaneously, the maximal degree of concurrency at any point in time in terms of FMM operations is $8^{d_{\max}} + 8^{d_{\max}-1} + 8^{d_{\max}} = 2 \cdot 8^{d_{\max}} + 8^{d_{\max}-1}$, which is strictly larger than the maximal degree of concurrency of a data-parallel FMM.

Algorithmically, FMM task graphs that exhibit even finer levels of granularity are conceivable. For instance, a task graph with *operator granularity* would correspond to an edge-vertex transformation of the data dependency graph in Figure 7.1 with additional edges for each direct particle-to-particle interaction. However, the latter are dependent on the input data set and therefore are not representable by the static data flow dispatcher behind Eventify. Independent from this work, active research is undertaken to enable a hybrid approach for the representation of static and dynamic dependencies in Eventify to allow for even finer granularity in the future.

7.4 FMSolvr

FMSolvr is an open source `C++`-implementation of the FMM for MD on CPUs. The sequential version of FMSolvr is the starting point for the data-parallel CPU and GPU implementations presented in this work. The Eventify-based version is the starting point for the task-parallel GPU version. The source code of FMSolvr can be found at `http://code.fmsolvr.org/`.

Please note that the implementation of FMSolvr on GPUs is not a full-fledged FMM but only models the task graph of the FMM to allow for a meaningful evaluation and optimization of the overhead induced by Eventify on GPUs. This is a worst case scenario for tasking frameworks since it is entirely driven by dependency resolution and task queueing and cannot use the trivially

parallelizable computations in the near field to hide bottlenecks in the far field. From hereon, the task graph implementation of FMSolvr with minimal workload is referred to as miniFMSolvr. All presented versions of miniFMSolvr can be found at `https://code.fmsolvr.fz-juelich.de/ATML-SE/eventify-GPU`. The correctness of miniFMSolvr is ensured by testing it against the statically known number of dependencies resolved per task as outlined in 7.3.3.

The UML class diagram provided in Figure 7.2 depicts the schematic software architecture of FMSolvr. It covers classes and functions which are relevant for this work to describe the parallelization schemes applied to FMSolvr in an object-oriented way. Even though it conveys the architectural design, it does not necessarily preserve the exact naming of classes and functions nor contain each and every component of them for reasons of clarity and visualization.

In addition to the member functions and the collection of free functions in `FMMPasses`, the subsequent compute kernel functions are defined as:

- `kernel_P2M(Particle& particle, Multipole& omega)`
- `kernel_M2M(Multipole& child, Multipole& parent, Scratch& s)`
- `kernel_M2L(Multipole& omega, Local& mu, Scratch& s)`
- `kernel_L2L(Local& parent, Local& child, Scratch& s)`
- `kernel_L2P(Particle& particle, Local& mu)`
- `kernel_P2P(Box& target, Box& source)`

For FMSolvr, each kernel function follows exactly its specification as outlined in Algorithm 2. For miniFMSolvr, however, only the dependency structure is considered. Therefore, the execution of the FMM operators is substituted with an accumulation operation to follow and count algorithmic dependencies.

7.4.1 Data-Parallel Implementations

OpenMP-FMSolvr for CPUs

This section provides an overview of a data-parallel, loop-based parallelization of FMSolvr with OpenMP for CPUs. OpenMP-FMSolvr supports $\mathcal{O}(p^3)$ as well as $\mathcal{O}(p^4)$ operators for the kernel functions of M2M, M2L and L2L. For conciseness, the parallelization scheme of each FMM-pass is described in C++-like, object-oriented pseudo-code. The pseudo-code preserves the semantic loop-structure only, meaning that the implementation of this loop-structure can be syntactically different. As an example, the loop on Line 9 in pseudo-code Listing 7.1 appears in the source code of FMSolvr as triply-nested loop and is accordingly equipped with an OpenMP `collapse` clause. Since such implementation details do not influence the exploitation of the algorithmic parallelization potential, they are not considered further.

Listing 7.1 shows the parallelization of pass P2M with OpenMP. The outermost loop iterates over all $8^{d_{max}}$ boxes on the lowest level d_{max} of the tree. By means of OpenMP's parallel work sharing loop this outermost loop is subdivided into chunks, which are distributed to all t threads of the team for parallel execution. Size and distribution of those chunks are specified through the `schedule` clause. The schedule-type `runtime` indicates that the scheduling type is chosen via the environment variable `OMP_SCHEDULE` at runtime. This allows to flexibly interchange scheduling policies to adapt to specific input data sets or efficiently perform comparative runtime measurements. If the scheduling policy `static` is applied, each thread executes at most one chunk, with each chunk covering approximately $8^{d_{max}}/t$ iterations.

As can be seen in Listing 7.2, pass M2M operates on all levels of the FMM-tree following a bottom-up scheme. Since the M2M operations on different levels are not fully independent of each other, the outermost loop cannot be parallelized via a parallel work sharing loop. Instead, each level is considered per se and the M2M-kernel is applied to its boxes in parallel. This level-wise parallelization scheme introduces an inherent synchronization bottleneck by relying on the implicit

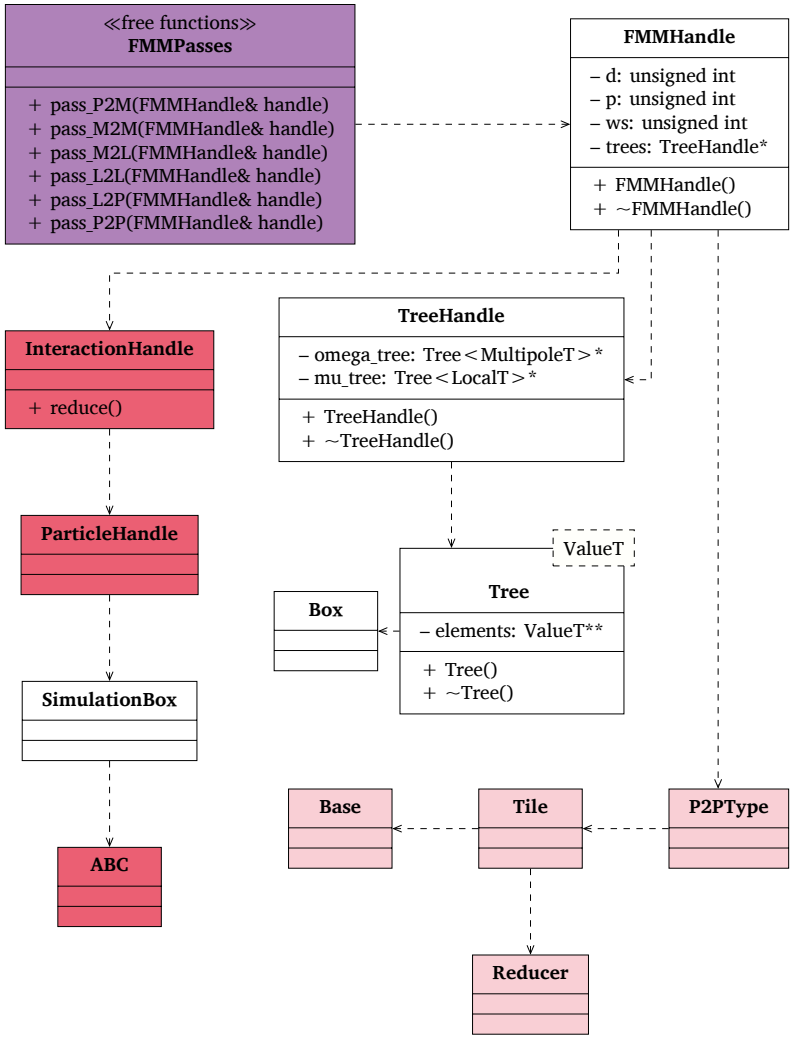


Figure 7.2: UML class diagram of FMSolvr. Violet signifies that a component required changes for both parallelization approaches, OpenMP and OpenACC. Red signifies that a component required changes for parallelization with OpenACC, but not for parallelization with OpenMP. Light red implies that components were removed and their functionality transferred to other components due to the restraint of dynamic memory allocation in OpenACC parallel sections.

barrier at the end of the `parallel` construct. Due to this, the amount of parallel work is reduced by a factor $1/8$ for each iteration of the outermost loop. For $d = 0$ this results in full sequentialization.

In contrast to the parallelization of P2M, we do not apply a combined parallel work sharing loop but a parallel construct (Line 6) followed by a separate work sharing loop construct (Line 9). This allows for the thread-private allocation and reuse of scratch memory (Line 8) as used in the

LISTING 7.1: OPENMP-VERSION OF PASS P2M

```
1  void pass_P2M(FMMHandle& handle)
2  {
3      Tree& tree = handle.tree();
4      unsigned int d = handle.d();
5      #pragma omp parallel for schedule(runtime)
6      for (box : tree.boxes_on_level(d))
7      {
8          for (particle : box.particles())
9          {
10             kernel_P2M(particle, box.omega());
11         }
12     }
13 }
```

LISTING 7.2: OPENMP-VERSION OF PASS M2M

```
1  void pass_M2M(FMMHandle& handle)
2  {
3      Tree& tree = handle.tree();
4      for (level : tree.levels_from_leaves())
5      {
6          #pragma omp parallel
7          {
8              scratch_type scratch(p);
9              #pragma omp for schedule(runtime)
10             for (box : level.boxes())
11             {
12                 for (child : box.children())
13                 {
14                     kernel_M2M(child.omega(), box.omega(), scratch);
15                 }
16             }
17         }
18     }
19 }
```

sequential version to improve memory efficiency.

Listing 7.3 covers the loop-based OpenMP-parallelization of pass M2L. The parallelization scheme of M2L is similar to the parallelization scheme of M2M; the outermost loop iterates over levels sequentially and the M2L-kernel is applied to all 8^d boxes on a level d in parallel. This is possible since the sequential implementation follows a target-centric approach. In contrast to a source-centric approach, this avoids the concurrent accumulation of local moments and hence the necessity of critical sections. Since the M2L operations on different levels are fully independent of each other, the outermost loop of M2L provides further parallelization potential. However, this is not exploited here since it does not provide a sustainable solution to the fundamental tree-induced synchronization bottlenecks of passes M2M and M2L.

Listing 7.4 shows the OpenMP-parallelization of pass L2L. Parallelization-wise, pass L2L is the counterpart to pass M2M. Hence, it operates on all levels of the FMM-tree following a top-down scheme. Due to the parent-child relation L2L operations are not fully independent of each other. Therefore, the outermost loop, which iterates over levels, cannot be parallelized with a parallel work sharing loop. Instead, the L2L-kernel is applied to all boxes of a level in parallel via a parallel construct (Line 6) and a separate worksharing loop (Line 9).

Pass L2P operates on the lowest level d_{max} only. Accordingly, its parallelization scheme follows the parallelization of pass P2M. As can be seen in Listing 7.5, a parallel worksharing-loop is applied to the outermost loop. Hence, each thread executes a chunk of boxes, while chunks are processed by multiple threads in parallel.

Listing 7.6 shows the OpenMP-parallelization of pass P2P. Similar to P2M and L2P, P2P operates only on the lowest level d_{max} of the tree. Therefore, a parallel worksharing-loop is applied, which subdivides the outermost loop into chunks of boxes that are processed by multiple threads in

LISTING 7.3: OPENMP-VERSION OF PASS M2L

```
void pass_M2L(FMMHandle& handle)
{
    Tree& tree = handle.tree();
    for (level : tree.levels())
    {
        #pragma omp parallel
        {
            scratch_type scratch(p);
            #pragma omp for schedule(runtime)
            for (box : level.boxes())
            {
                for (i : box.interaction_set())
                {
                    kernel_M2L(i.omega(), box.mu(), scratch);
                }
            }
        }
    }
}
```

LISTING 7.4: OPENMP-VERSION OF PASS L2L

```
void pass_L2L(FMMHandle & handle)
{
    Tree& tree = handle.tree();
    for (level : tree.levels_from_root())
    {
        #pragma omp parallel
        {
            scratch_type scratch(p);
            #pragma omp for schedule(runtime)
            for (parent : level.boxes())
            {
                for (child : parent.children())
                {
                    L2L(parent.mu(), child.mu(), scratch);
                }
            }
        }
    }
}
```

LISTING 7.5: OPENMP-VERSION OF PASS L2P

```
void pass_L2P(FMMHandle& handle)
{
    Tree& tree = handle.tree();
    OutputHandle& output = handle.output_handle();
    unsigned int d = handle.d();
    #pragma omp parallel for schedule(runtime)
    for (box : tree.boxes_on_level(d))
    {
        for (particle : box.particles())
        {
            kernel_L2P(particle, box.mu(), output);
        }
    }
}
```

parallel.

As can be seen from Figure 7.2, the OpenMP-parallelization of *FMSolvr* is minimal-invasive as it only affects the compute functions in `FMMPasses`.

OpenACC-miniFMSolvr for GPUs

The parallelization scheme of OpenACC-miniFMSolvr corresponds to the parallelization approach of OpenMP-FMSolvr. Since OpenACC allows to take the hierarchical parallelism of GPUs into

LISTING 7.6: OPENMP-VERSION OF PASS P2P

```
void pass_P2P(FMMHandle& handle)
{
    Tree& tree = handle.tree();
    OutputHandle& output = handle.output_handle();
    #pragma omp parallel
    {
        #pragma omp for schedule(runtime)
        for (target : tree.boxes_on_level(d_max))
        {
            for (source : target.near_neighbors())
            {
                kernel_P2P(target, source);
            }
        }
    }
}
```

LISTING 7.7: OPENACC-VERSION OF PASS P2M

```
void pass_P2M(FMMTree& tree)
{
    #pragma acc parallel loop gang worker vector num_workers(block_dim) vector_length(1) num_gangs(
        grid_dim)
    for (box : tree.boxes_on_level(d_max))
    {
        #pragma acc loop seq
        for (particle : box.particles())
        {
            kernel_P2M(particle, box.omega());
        }
    }
}
```

LISTING 7.8: OPENACC-VERSION OF PASS M2M

```
void pass_M2M(FMMTree& tree)
{
    for (level : tree.levels_bottom_up())
    {
        #pragma acc parallel loop gang worker vector num_workers(block_dim) vector_length(1) num_gangs(
            grid_dim)
        for (box : level.boxes())
        {
            #pragma acc loop seq
            for (child : box.children())
            {
                kernel_M2M(child.omega(), box.omega());
            }
        }
     }
}
```

account, the type of loop parallelism can be specified by providing `worker`, `vector` and `gang` dimensions. Worker and gang dimension are specified at runtime in order to facilitate the search for the optimal kernel configuration. The vector dimension is statically set to 1 since this allows OpenACC to comprise warps of the worker threads flexibly. Listings 7.7 to 7.12 outline all passes of miniFMSolvr as parallelized with OpenACC loops.

7.4.2 Task-Parallel Implementation

The fine-grained task-parallel implementation of miniFMSolvr on GPUs uses Eventify. Figure 7.3 shows the integration of Eventify in miniFMSolvr based on the software architecture described in Section 6.4.1. From the user perspective, the integration involves the subsequent steps that follow from the software architecture and the workflow of Eventify as outlined in Section 6.4.2:

LISTING 7.9: OPENACC-VERSION OF PASS M2L

```
void pass_M2L(FMMTree& tree)
{
    for (level : tree.levels())
    {
        #pragma acc parallel loop gang worker vector num_workers(block_dim) vector_length(1) num_gangs(
            grid_dim)
        for (box : level.boxes())
        {
            #pragma acc loop seq
            for (i : box.interaction_set())
            {
                kernel_M2L(i.omega(), box.mu());
            }
        }
    }
}
```

LISTING 7.10: OPENACC-VERSION OF PASS L2L

```
void pass_L2L(FMMTree & tree)
{
    for (level : tree.levels_top_down())
    {
        #pragma acc parallel loop gang worker vector num_workers(block_dim) vector_length(1) num_gangs(
            grid_dim)
        for (parent : level.boxes())
        {
            #pragma acc loop seq
            for (child : parent.children())
            {
                L2L(parent.mu(), child.mu(), scratch);
            }
        }
    }
}
```

LISTING 7.11: OPENACC-VERSION OF PASS L2P

```
void pass_L2P(FMMTree& tree)
{
    #pragma acc parallel loop gang worker vector num_workers(block_dim) vector_length(1) num_gangs(
        grid_dim)
    for (box : tree.boxes_on_level(d_max))
    {
        #pragma acc loop seq
        for (particle : box.particles())
        {
            kernel_L2P(particle, box.mu());
        }
    }
}
```

- Definition of two CUDA streams; one for the near field pass, and another for the in-order execution of all far field passes.
- Reimplementation of the `Tree` data structure for GPUs based on `gtl::vector`.
- Definition of `AbstractProcessor`, `Task` type and queue structures for all six task types of the FMM.
- Equipping `InitData` with a pointer to a `Tree` for compute data, and a pointer to a `Tree` for dependency counters. The latter substitutes the placeholder class `DependencyStructure` of Eventify.
- Definition of function template specializations for `get_queue()`, `get_mutex()`, `init_queue()`,

LISTING 7.12: OPENACC-VERSION OF PASS P2P

```
void pass_P2P(FMMTree& tree)
{
    #pragma acc parallel loop gang worker vector num_workers(block_dim) vector_length(1) num_gangs(
        grid_dim)
    for (target : tree.boxes_on_level(d_max))
    {
        #pragma acc loop seq
        for (source : target.near_neighbors())
        {
            kernel_P2P(target, source);
        }
    }

}
```

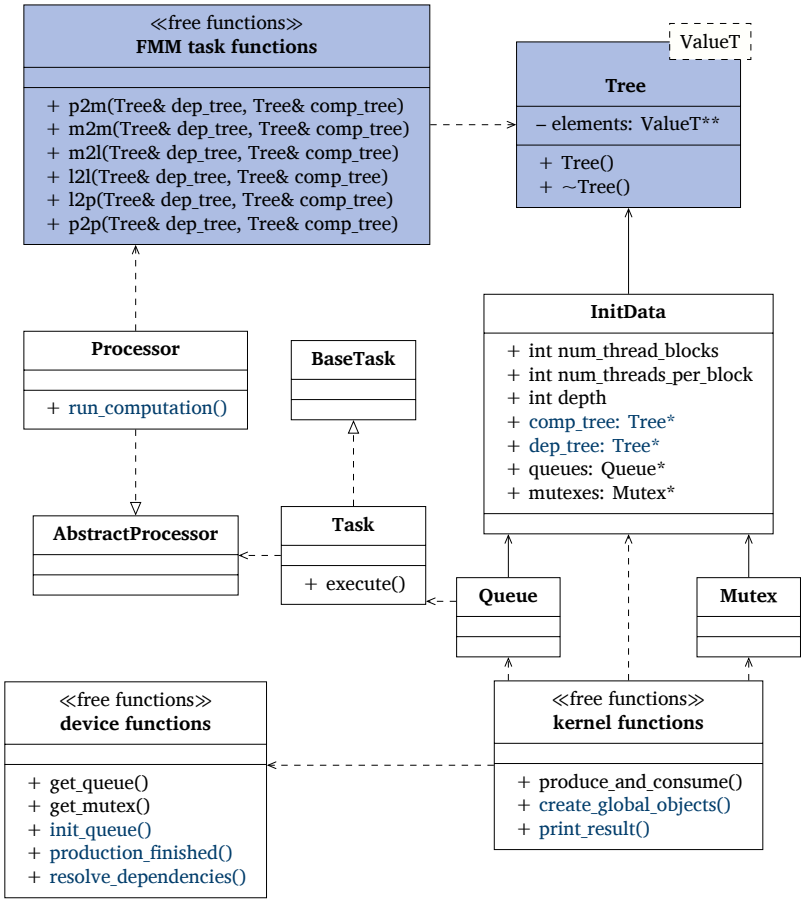


Figure 7.3: UML class diagram of the integration of miniFMSolvr and Eventify.

`production_finished()` and `resolve_dependencies()` for each task type.

Chapter 8

Evaluation

The objective of this chapter is to evaluate the performance and sustainability of fine-grained task-parallelism in comparison to data-parallelism on CPUs and GPUs. The chapter commences by determining quantitative metrics for performance. Afterwards, the methodology to conduct measurements of these metrics is outlined. Based thereon, the measurement results for all parallel implementations of FMSolvr are presented, analyzed and evaluated against each other. Conclusive, threats against validity of this work are outlined.

8.1 Metrics

The focus of this work is on the comparison of different parallel programming approaches. Hence, it requires metrics that do not only quantify the behavior of a single program but allow for the comparison of multiple programs.

8.1.1 Scalability

In this work, scalability is defined as follows:

Definition 8.1. ***Scalability.*** *The scalability of a parallel algorithm on a parallel architecture is a measure of its capacity to efficiently utilize an increasing number of [processing elements] [69].*

In order to measure scalability, the "capacity to efficiently utilize an increasing number of processing elements" must be quantified. To quantify this capacity, runtime is used as underlying performance metric since it is the main concern of scientific simulations.

Definition 8.2. ***Runtime.*** *The runtime r_A of a program A is the absolute amount of time it takes to execute A.*

r_{min} denotes the minimal runtime of a parallel program, while $t_{r_{\text{min}}}$ denotes the number of threads for which this minimal runtime is reached.

To evaluate runtime under an increasing number of processing elements, two properties of a parallel program can be analyzed: *strong scaling* and *weak scaling*. Strong scaling describes the efficiency of a parallel application under a constant problem size and an increasing number of processing elements, while weak scaling describes the efficiency of a parallel application under a problem size that is direct proportional to the number of processing elements. This work considers strong scaling only since the problem size of scientific simulations is typically determined by physical properties, and there is little scientific value in increasing problem sizes beyond the size of realistic systems. Therefore, it is not sufficient to target weak scaling in order to utilize increasing hardware parallelism. This work employs the *efficiency* [89] of a parallel program to quantify strong scaling.

Definition 8.3. ***Efficiency***. *The efficiency of a parallel program is defined as*

$$E = \frac{T_s}{t \cdot T_t}, \tag{8.1}$$

where T_s denotes the runtime of the best sequential implementation and T_t the runtime of the parallel implementation on t threads.

Based on these considerations, an application exhibits optimal scalability if its speedup corresponds to the number of processing elements it is executed on, i.e. if $E = 1$ assuming the absence of super-linear speedup.

With *relative* efficiency, this work describes the scaling on GPUs relative to the grid size for a fixed block size to take hierarchical parallelism into account. This allows to analyze which block size provides the best scalability.

This work is not only concerned with the performance behavior of a single implementation but with a comparison of different implementations. To quantify the performance change an implementation provides relative to another implementation, the *runtime ratio* (derived from the *performance improvement* metric introduced in [75]) is applied.

Definition 8.4. ***Runtime Ratio***. *The runtime ratio $R_{A,B}$ of an implementation A in comparison to a baseline implementation B is the ratio $\frac{r_B}{r_A}$.*

Based thereon, the arithmetically averaged runtime ratio $\overline{R}$ is defined. $\overline{R}$ averages the runtime change over all numbers of threads t for which $R_{A,B}$ has been determined. Further, there is a maximal runtime ratio R_{max}. Since the best runtime is not necessarily reached for the same number of threads in implementation A and implementation B, the effective runtime ratio R_{eff} is defined. It denotes the practical relevant runtime change, i.e. the ratio between the minimal runtime of the baseline B and the minimal runtime of implementation A.

8.1.2 Sustainability

In order for a parallel programming approach to be considered *sustainable*, this work requires the approach to provide good programmability and enhance portability of applications. Programmability is qualitatively assessed based on the software architectures outlined in Section 7.4. While quantitative metrics such as the relative number of modified classes could be considered they neglect the complexity of the involved changes and their effects on maintainability. Portability is in this case trivially to evaluate since it only determines whether an application or programming model support multiple hardware architectures.

8.2 Methodology

This section outlines the methodology to obtain measurement results for the performance and sustainability evaluation. For reproducibility, please find the source code for all parallel versions of FMSolvr and miniFMSolvr along with compilation and execution scripts at `https://code.fmsolvr.fz-juelich.de/ATML-SE/eventify-GPU`.

8.2.1 Hardware

Table 8.1 outlines the properties of the compute node used for all measurements on CPUs. The node consists of 4 Intel Xeon E7-4830 v4 (Broadwell) CPUs with 14 cores and 2-way SMT each. Hence, it exhibits 56 (non-SMT) cores overall.

Table 8.2 outlines the properties of the Nvidia GPU used for all measurements on GPUs. The Tesla V100 was selected for two reasons. Firstly, it provides a high SM count and hence allows for an extensive scalability analysis. And secondly, it supports ITS and hence enables an evaluation of the influence of ITS on the performance of fine-grained tasking.

Table 8.1: Device properties of the CPU node used for the performance analysis.

Property	Broadwell Node
Processor	Intel Xeon E7-4830 v4
Sockets	4
Cores per socket	14
SMT threads per core	2
Clock frequency (Turbo Boost disabled)	2.00 GHz
L1d/L1i Cache	1.8 MiB
L2 Cache	14 MiB
L3 Cache	140 MiB

Table 8.2: Device properties of the GPU used for the performance analysis.

Property	Tesla V100
SM count	80
Max threads per SM	2048
Max threads per block	1024
Warp size	32
Global memory	16 GB
L2 cache size	6 MB
Shared memory per SM	96 KB
Shared memory per block	48 KB
Registers per SM	65536
Registers per block	65536

Listing 8.1: Flags to build executable FMSolvr_eventify

```
1 -std=c++11 -O3 -march=native -W -Wall -pthread -lnuma
```

Listing 8.2: Flags to build executable FMSolvr_omp

```
1 -std=c++11 -O3 -march=native -W -Wall -fopenmp
```

8.2.2 CPU Runtime Measurements

All CPU runtime measurements are performed on the compute node described in Table 8.1.

In order to compile Eventify-FMSolvr for CPUs, the `g++` compiler (version 10.2.1) is used with the flags shown in Listing 8.1.

For OpenMP-FMSolvr for CPUs, `g++` is used with the following flags outlined in Listing 8.2.

Time measurements are performed via `std::chrono::high_resolution_clock`. Each execution of the program covers the workflow of FMSolvr for a single time-step of the simulation excluding instantiation and memory allocation of data structures. The latter would only be done once in a realistic simulation with multiple time steps and hence is irrelevant in terms of runtime. To ensure stable measurements, each run is repeated 1000 times for $d_{\max} = 3$, 100 times for $d_{\max} = 4$ and 10 times for $d_{\max} = 5$ and $d_{\max} = 6$.

Intel's Turbo Boost technology is disabled for all measurements. This is necessary since Turbo Boost leads to varying clock frequencies depending on the workload and number of cores used. Hence, it can distort scaling plots due to the clock frequency being automatically increased for the single-thread implementation employing one core only. According to [23], Turbo Boost leads on average to a 6% reduction in runtime for computationally-intensive programs by accelerating sequential phases.

Since Eventify provides built-in NUMA-awareness, the thread affinity policy `OMP_PLACES=cores`

Listing 8.3: Flags to build executable miniFMSolvr_OpenACC

```
pgc++ -O4 -std=c++11 -Mlist -acc -Minfo=accel -ta=tesla:cc70,managed -DNDEBUG -DUSE_UPPER  -I./include -I
    ./tables -I./helpers  fmmtest.cpp -o miniFMSolvr_OpenACC
```

Listing 8.4: Flags to build executables miniFMSolvr_Eventify

```
nvcc -arch=compute_70 -std=c++11 -rdc=true -Xcompiler -Wall  -O3  main.cu -o miniFMSolvr_Eventify
```

`OMP_PROC_BIND=close` is employed for the OpenMP measurements to reduce the influence of NUMA on OpenMP, too.

8.2.3 GPU Runtime Measurements

The runtime measurements for OpenACC-miniFMSolvr and Eventify-miniFMSolvr on GPUs are performed on the Nvidia V100 GPU described in Table 8.1.

For the compilation of OpenACC-miniFMSolvr for GPUs, the `pgc++` compiler (version 20.7-0) is used as shown in Listing 8.3.

For the compilation of Eventify-miniFMSolvr for GPUs, the `nvcc` compiler (version 11.5) is used with the flags provided in Listing 8.4.

Time measurements are performed via `std::chrono::high_resolution_clock` in the very same way as on the CPU. Each execution of the program covers the workflow of miniFMSolvr for a single time-step of the simulation excluding instantiation and memory allocation of data structures. Since preliminary experiments showed that the runtime measurements on the GPU are stable, each run is repeated 10 times only.

8.3 Performance Analysis on CPUs: Data-Parallelism vs. Task-Parallelism

This section provides an initial performance analysis of FMSolvr on CPUs to evaluate whether the theoretical performance gains of task parallelism as anticipated in Section 7.3 can be realized in practice. For this purpose, Figure 8.1 shows the runtime, parallel efficiency and runtime ratio of data-parallel OpenMP-FMSolvr in comparison with task-parallel Eventify-FMSolvr. The light blue backgrounds represent the four sockets of the Broadwell machine described in Table 8.1 in order to illustrate NUMA effects.

The sequential runtime of OpenMP-FMSolvr is 1.01 s, while the sequential runtime of Eventify-FMSolvr is 1.29 s. Hence, the execution of Eventify-FMSolvr with $t = 1$ takes 28 % longer than the execution of OpenMP-FMSolvr. This is expected since Eventify introduces task management overheads due to fine-grained synchronization, dependency resolution and task queueing times. When executed on a single thread, however, this does not amortize since tasks are executed sequentially anyhow. OpenMP `parallel for` on the other hand does not introduce any overheads for sequential execution. Due to this, OpenMP-FMSolvr serves as best sequential implementation to compute parallel efficiency for both implementations. Therefore, the initial efficiency of Eventify-FMSolvr is 22 % lower than the efficiency of OpenMP-FMSolvr.

Considering parallel execution, the runtime ratio plot shows that the overheads introduced by Eventify amortize for $t = 5$. From this point on, Eventify-FMSolvr consistently provides a lower runtime and higher parallel efficiency than OpenMP-FMSolvr. For $t \leq 28$ the runtime of Eventify-FMSolvr exhibits the same slope as the ideal runtime with an overhead of 33% on average.

As can be seen from the runtime plot, the runtime of OpenMP-FMSolvr increases considerably for $t > 14$, which is exactly the first NUMA-border. Minor runtime increases are also apparent from $t = 28$ to $t = 29$, and from $t = 42$ to $t = 43$, which are also NUMA borders. These effects occur even though OpenMP's close core pinning strategy is applied to prevent the operating

system scheduler from moving threads and associated data between cores, which reduces data transfers between NUMA nodes. In contrast to OpenMP, Eventify provides automatic thread pinning and customized NUMA-aware memory allocation and load balancing strategies to alleviate NUMA-induced performance bottlenecks even further. A detailed analysis of NUMA effects in FMSolvr and their resolution via Eventify can be found in [75].

The highest runtime improvement of Eventify over OpenMP amounts to $R_{\max} = 6$ and is reached for $t_{R_{\max}} = 56$ threads. This is, however, not the effective performance improvement in practice since OpenMP-FMSolvr's best runtime is reached for $t = 14$ instead of $t = 56$. The lowest runtime of Eventify-FMSolvr amounts to 0.05s and is reached for 56 threads. The lowest runtime for OpenMP-FMSolvr is 0.16s and is reached for 14 threads. Hence, Eventify-FMSolvr is $R_{\text{eff}} = 3.2$ times faster than OpenMP-FMSolvr.

There are three main reasons why Eventify-FMSolvr scales better and reaches better execution times than OpenMP-FMSolvr for an increasing number of threads. First, Eventify-FMSolvr exhibits a higher degree of concurrency due to task overlapping as theoretically derived in Section 7.3. Second, Eventify provides a work stealing approach which accounts for load imbalances. And third, Eventify alleviates the NUMA-effects as evident at the NUMA-borders in Figure 8.1.

8.4 Performance Analysis on GPUs: Data-Parallelism vs. Task-Parallelism

This section evaluates the performance of OpenACC-miniFMSolvr and Eventify-miniFMSolvr on GPUs to evaluate whether the theoretical performance gains of task-parallelism as anticipated in Section 7.3 can be realized in practice.

8.4.1 OpenACC

Figure 8.2 shows the runtime and parallel efficiency of data-parallel OpenACC-miniFMSolvr on GPUs. Runtime and efficiency have been determined for grid sizes $D_{\text{grid}} = 1, 2, 4, 6, \dots, 160$. The maximum configurable grid size is 160 since this is the maximum grid size without leading to a CUDA out of memory error for the MHQ approach. For each grid size, eight different thread block sizes $D_{\text{block}} = 1, 2, 4, 8, 16, 32, 64, 128$ have been considered.

The sequential runtime of OpenACC-miniFMSolvr is 2.7s. For $D_{\text{block}} = 1$, the runtime decreases with increasing grid size, and the lowest runtime $r_{\min} = 0.261$s is reached with 152 thread blocks. Hence, by increasing the grid size only, runtime can be improved by a factor of 10.3. Since runtime barely decreases for $D_{\text{grid}} > 32$, this accounts to a parallel efficiency of 7% only. This is similar to the efficiency behavior of data-parallel FMSolvr on CPUs for $t > 28$ as evident in Figure 8.1.

Figure 8.2 shows further that the runtime also decreases by increasing the number of threads per block. For grid size $D_{\text{grid}} = 1$, the runtime decreases with a factor of 1.63 to 1.1 for increasing block sizes from $D_{\text{block}} = 1, \dots, 128$. The optimal runtime for $D_{\text{grid}} = 1$ is reached for $D_{\text{block}} = 128$, which corresponds to a parallel efficiency of 8%. Hence, the scaling behavior of OpenACC-FMSolvr is the same independent of the fact whether grid size or block size is increased. This is expected since OpenACC-FMSolvr, similarly to OpenMP-FMSolvr, cannot (trivially) exploit the hierarchical parallelism that the GPU provides due to inner loop dependencies as outlined in Section 7.4.1.

There are two options to alleviate this bottleneck. A combined approach of threading and vectorization as outlined for FMSolvr in [67] could be chosen. Alternatively, fine-grained task parallelism could be used to allow all lanes of a warp to execute different tasks simultaneously. This work explores the latter since it aims for a general approach to exploit hierarchical parallelism in irregular algorithms, and not specifically for a highly optimized FMM implementation.

8.4.2 Eventify

In this section, the performance and sustainability of Eventify-miniFMSolvr on GPUs are evaluated with a focus on the comparison with OpenACC-miniFMSolvr.

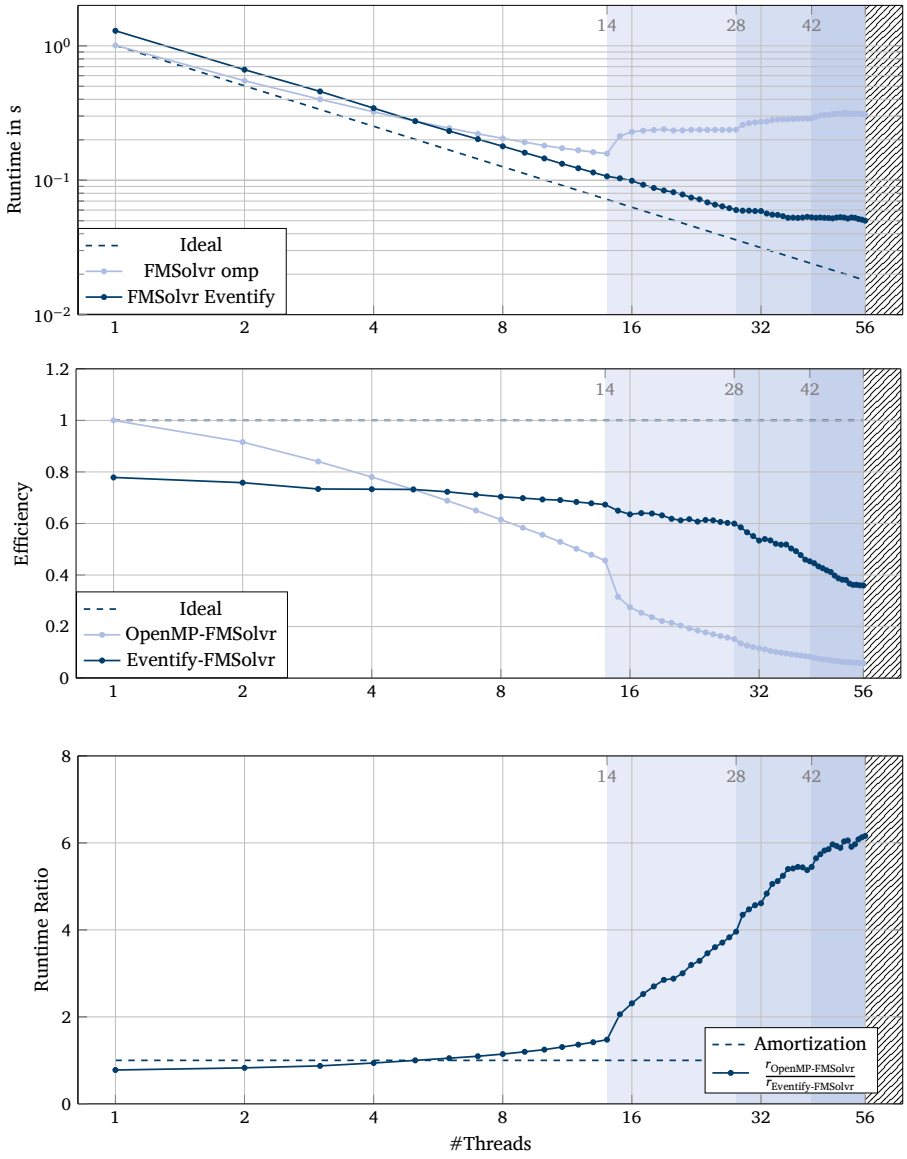


Figure 8.1: Runtime, efficiency and runtime ratio of data-parallel OpenMP-FMSolvr in comparison with Eventify-FMSolvr for tree depth $d_{\max} = 5$ and multipole order $p = 4$ with 100000 particles on a CPU.

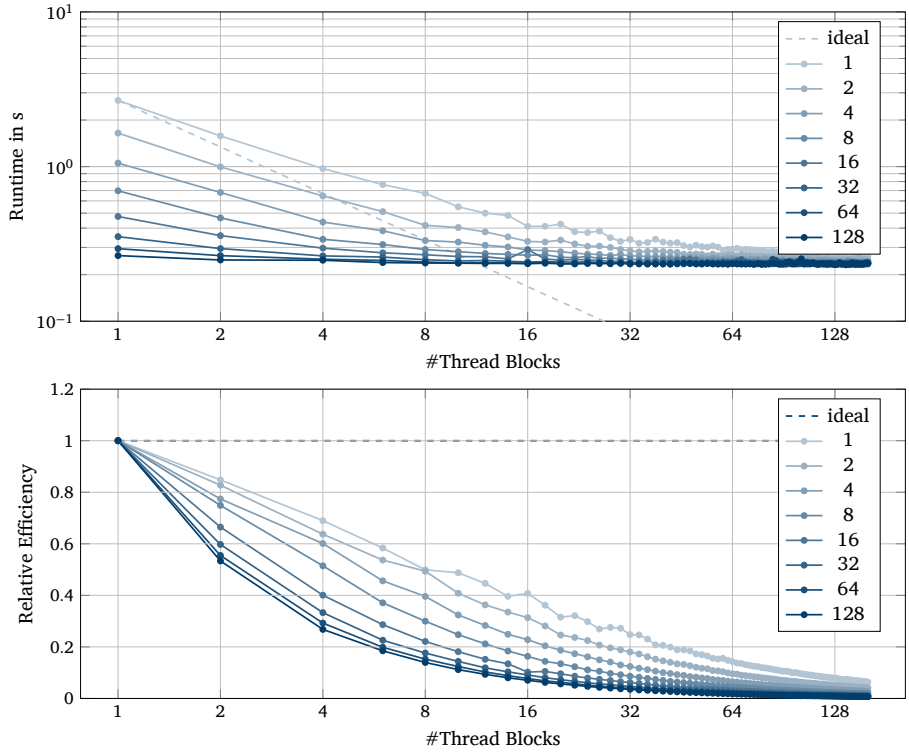


Figure 8.2: Runtime and efficiency of data-parallel OpenACC-miniFMSolvr on a GPU for tree depth $d_{\max} = 5$.

GPU Locks

The locking mechanism is a key factor for the efficiency of tasking frameworks. Therefore, a performance analysis of different mutex implementations under high contention is conducted separately to determine which mechanism to apply for all subsequent measurements with Eventify-miniFMSolvr. Figure 8.3 shows the runtime of Eventify-miniFMSolvr for passes M2M, M2L and L2L for $d_{\max} = 6$ with the mutex implementations of Eventify and `libcu++` as described in Section 6.2.2. It is evident, that the Eventify mutex leads to the best runtime independent of the chosen kernel configuration. Hence, all further runtime measurements are based on the Eventify mutex.

SQ: Single Multi-Producer Multi-Consumer Queue

Figure 8.4 shows the runtime and relative efficiency of Eventify-miniFMSolvr on GPUs with a single multi-producer multi-consumer queue as introduced in Section 6.3.4.

The sequential runtime of SQ-Eventify-miniFMSolvr is 6.5s. For a constant grid size, the runtime decreases when the block size is increased. For a constant block size, the runtime increases with increasing grid size. While the latter might appear surprising at first, this behavior reflects exactly the limited access parallelism and large critical section of the SQ scheme. Since only the thread block master has write access to the global queue, all write accesses to the queue serialize and hence no MIMD inter-block parallelism can be achieved. Due to contention, the runtime does not stay constant but even increases when more thread blocks are used. Following this rationale, SQ-Eventify-miniFMSolvr reaches its best runtime $r_{\min} = 3.1$s for $D_{\text{grid}} = 1$ and $D_{\text{block}} = 128$,

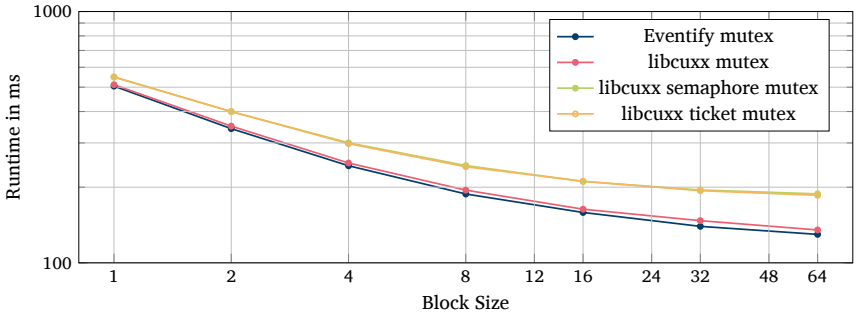


Figure 8.3: Runtime of Eventify-miniFMSolvr for passes M2M, M2L and L2L for $d_{\max} = 6$ with different mutex implementations on a GPU.

which is an order of magnitude above the best runtime of OpenACC-miniFMSolvr.

Eventify-miniFMSolvr and OpenACC-miniFMSolvr reach their best runtime for the exact same kernel configuration. While OpenACC-miniFMSolvr, however, can exploit the loop parallelism in all passes without introducing any parallelization overhead, Eventify-miniFMSolvr introduces considerable overheads due to task generation, locking, task queueing and dependency resolution.

Considering the relative efficiency plot, it becomes apparent that Eventify and OpenACC lead to considerably different scaling behavior. OpenACC-miniFMSolvr exhibits its best scalability for $D_{\text{block}} = 1$, while the scalability of Eventify-miniFMSolvr is equivalent for all tested block sizes. Qualitatively, all efficiency curves of Eventify-miniFMSolvr correspond to the worst case efficiency of OpenACC-miniFMSolvr as reached for $D_{\text{block}} = 128$.

Concluding, a naive implementation of task-parallelism on GPUs does not yield any performance gains. On CPUs, in contrast, task-parallelism does reduce the runtime in comparison to a data-parallel implementation. Hence, the performance bottleneck on GPUs has to be identified. The major difference between Eventify on CPUs and GPUs is the queueing approach. While each CPU thread draws tasks from its private queue, all GPU thread blocks draw tasks from a single shared queue. To alleviate this performance bottleneck, two queueing schemes with multiple task queues have been developed and are evaluated subsequently.

MQ: Multiple MPSC Queues

Figure 8.5 shows the runtime and relative efficiency of Eventify-miniFMSolvr on GPUs with multiple multi-producer multi-consumer queues as introduced in Section 6.3.4.

The sequential runtime of MQ-Eventify-miniFMSolvr is 6.5s, which corresponds exactly to the runtime of SQ-Eventify-miniFMSolvr. This is expected since the number of queues in MQ corresponds to the number of thread blocks D_{block}; for $D_{\text{block}} = 1$, MQ and SQ exhibit accordingly the same number of queues. Since only one master thread exists, there is no contention and all tasks are simply generated and executed sequentially in both implementations.

Qualitatively, the runtime behavior with respect to block sizes is similar for SQ and MQ as well. For a constant grid size, the runtime decreases when the block size is increased. This shows that MQ does indeed preserve the SIMD intra-block parallelism provided by SQ.

The runtime behavior with respect to grid size, however, varies considerably between both implementations. For a constant block size, the runtime decreases with increasing grid size. This shows that MQ-Eventify-miniFMSolvr fulfills the objective of achieving inter-block parallelism.

Similar to SQ, all relative efficiency curves of MQ are coextensive. Hence, the qualitative scaling behavior of SQ is preserved. Quantitatively, however, the scalability of MQ is 60% higher than the scalability of SQ and OpenACC.

Concluding, MQ-Eventify-miniFMSolvr reaches its best runtime $r_{\min} = 0.021$ for $D_{\text{block}} = 128$

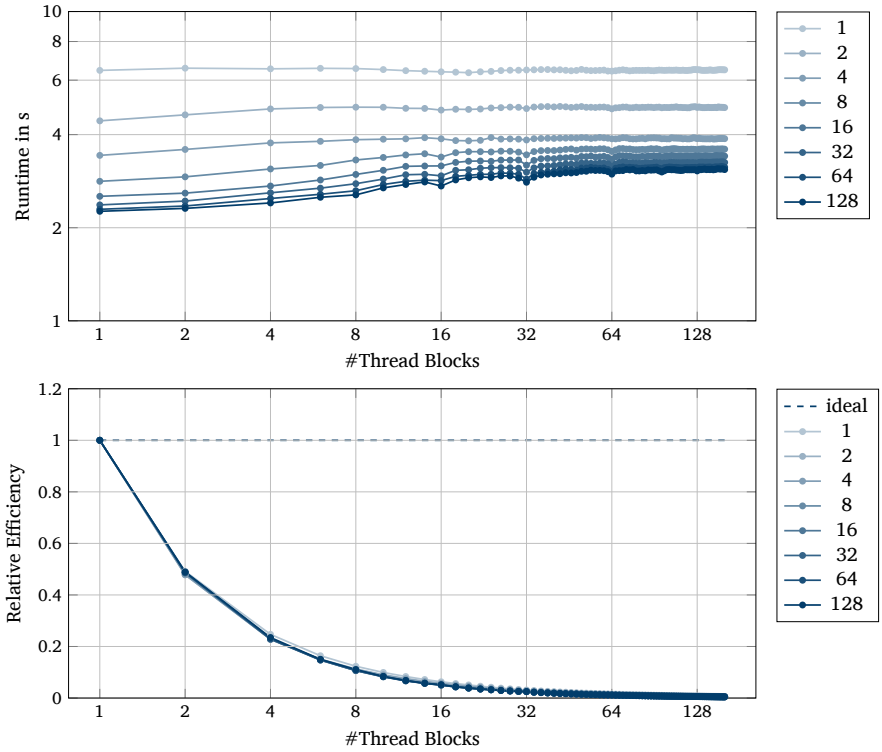


Figure 8.4: Runtime and relative efficiency of miniFMSolvr on a GPU parallelized with Eventify and SQ scheduling for tree depth $d_{\max} = 5$.

and $D_{\text{grid}} = 160$. Hence, MQ-Eventify-miniFMSolvr outperforms OpenACC-miniFMSolvr by a factor of 12.4.

MHQ: Multiple Hierarchical Queues

Figure 8.4 shows the runtime and the relative efficiency of Eventify-miniFMSolvr on GPUs with hierarchical queues as introduced in Section 6.3.4.

The sequential runtime of MHQ amounts to $r_{\min} = 6.5$s, which matches the runtime of SQ and MQ. This is unexpected, since MHQ maintains two queues instead of one queue per thread. The larger overhead should lead to an increase in runtime for sufficiently small grid sizes since the overhead cannot amortize by reducing contention on the shared queues. Reconsidering the MHQ implementation, it becomes apparent that the shared queue is initialized but never used in the sequential case since tasks are always assigned to the private queue. In addition, the onetime queue construction overhead is negligible in comparison to task generation and execution.

The runtime behavior of MHQ-Eventify-miniFMSolvr with respect to block size and grid size corresponds to the runtime behavior of MQ-Eventify-miniFMSolvr. This is due MHQ preserving the SIMD intra-block and the MIMD inter-block parallelism of MQ. MHQ reaches its best runtime $r_{\min} = 0.021$s for $D_{\text{block}} = 128$ and $D_{\text{grid}} = 160$, which corresponds exactly to the minimal runtime of MQ. Hence, the additional queue management overhead does not amortize for miniFMSolvr despite the fact that it reduces the number of mutex-protected queue accesses. This is surprising,

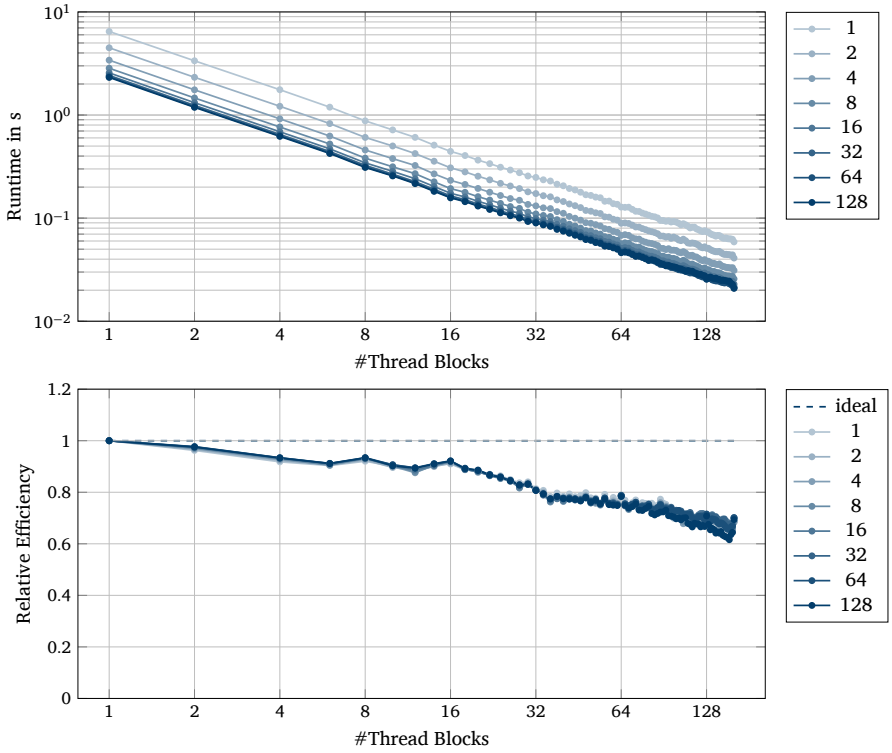


Figure 8.5: Runtime and relative efficiency of miniFMSolvr on a GPU parallelized with Eventify and MQ scheduling for tree depth $d_{\max} = 5$.

since blocking algorithms like MQ have so far been considered inefficient on GPUs. Concluding, the comparative performance analysis shows that efficient blocking algorithms are possible on current GPUs.

8.4.3 Sustainability

As evident from the UML diagrams of FMSolvr in Figure 7.2, GPU-based parallelization approaches require more and profound changes within classes, and even modifications of the software architecture itself. This holds true independently of the fact whether a data- or task-parallel programming model is applied. There are two main reasons for this. Firstly, the additional memory management required due to the separation of host and device memory. And secondly, that GPU programming models in general do not support the full range of `C++` features and the STL. Due to these complexities, the programmability of GPU programming models is poorer than the programmability of CPUs, and they worsen code maintainability. Considering portability, GPU programming models nevertheless enhance the sustainability of software since they open up an additional platform to run on. By porting Eventify to GPUs, this work accordingly contributes to the wider vision of enabling fine-grained task-parallelism on heterogeneous hardware.

Applications that employ directive-based data-parallel approaches like OpenMP and OpenACC are harder to debug and optimize since implementation details are opaque. For applications that exhibit regular data-parallelism and exhibit flat class hierarchies, however, they enable high

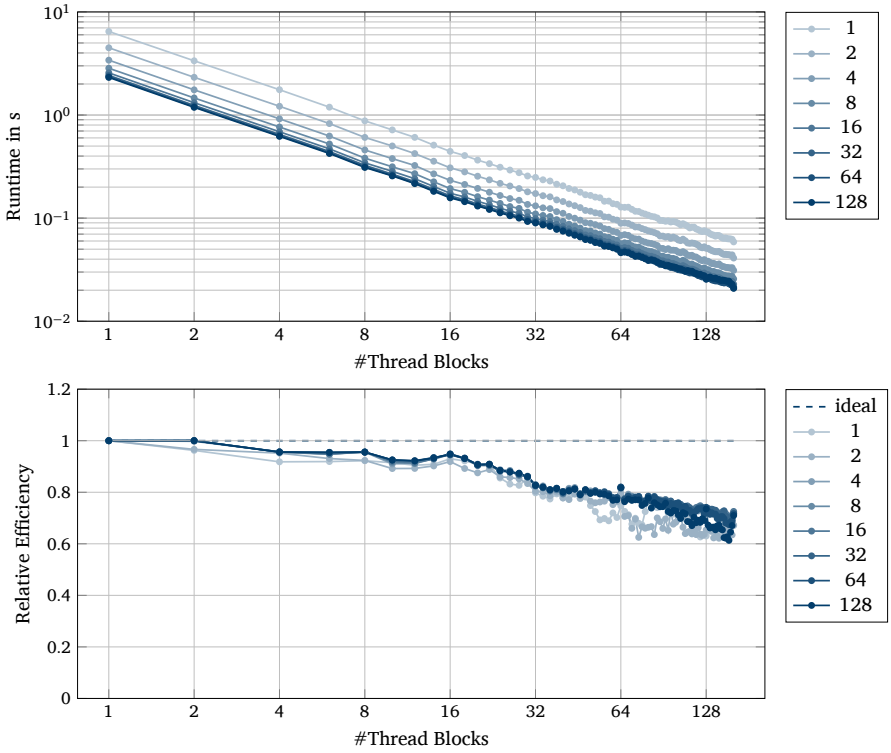


Figure 8.6: Runtime and relative efficiency of miniFMSolvr on a GPU parallelized with Eventify and MHQ scheduling for tree depth $d_{\max} = 5$.

software development efficiency.

Approaches that force software developers to explicitly describe task-parallelism pose an entirely different type of challenge since they require a deep understanding of the algorithmic data dependencies to identify and leverage loop-overarching fine-grained parallelism. If the data dependency graph of an application and its parallelization potential are known in detail, using fine-grained task-parallelism is nevertheless worth the higher development effort since it can lead to performance gains in the order of a magnitude.

8.5 Threats to Validity

The main threat to validity of this work is that it delivers a prototypical implementation of FMSolvr for GPUs only. While it allows for the processing of the complete FMM task graph it does not contain the execution of the actual FMM operations since CUDA does not yet support all of the `C++` functionalities required for this. In the course of this work prototypical implementations of many `C++` features required by FMSolvr and Eventify, such as `std::sort`, `std::forward`, `std::vector`, `std::complex`, were developed and can be found at `https://code.fmsolvr.fz-juelich.de/ATML-SE/eventify-GPU`. However, these are not optimized for performance and hence would prevent a meaningful evaluation of the effects of task-parallelism on GPUs. In fact, the optimization of these features goes beyond the scope of this work since they are application dependent and will in the foreseeable future be provided by vendor libraries or heterogeneous programming models

such as `libcu++`, SYCL or even be included in the `C++` standard itself. The main issue with the prototypicality of the GPU implementations is that the runtimes achieved on CPUs and GPUs are not comparable since they do not execute the same workload. However, this does not limit the conclusions of this work since its objective is not a performance comparison between different architectures but between different parallel programming paradigms.

Another threat to validity is that the evaluation only considers a comparison against data-parallel programming technologies but not against other task-parallel programming technologies. This is due to the scarcity of technologies that support fine-grained task-parallelism on GPUs. Whippletree [96] is the only technology that provides fine-grained task-parallelism on GPUs that is available open source. However, Whippletree does not support generic task graphs but only task graphs with one in- and one out-dependency per task.

The build-in NUMA-aware thread pinning and memory allocation of Eventify on CPUs might not be fair to compare against an OpenMP implementation which only supports thread pinning but not NUMA-aware allocation as remedy for NUMA-induced performance loss. This is secondary, however, since Eventify outperforms OpenMP on a single NUMA node already. Hence, the conclusion that task parallelism can outperform data-parallelism still holds.

Chapter 9

Conclusion

9.1 Summary

This section summarizes the findings by answering the research questions stated in Section 1.2.

9.1.1 Hardware Architecture Trends

Research Question 1 *What are the hardware architecture trends for parallelism on CPUs and GPUs?* To answer this question, Chapter 3 outlines the *stream interaction model* to derive *concurrent processing capabilities* as main comparison criterion for the architectural features of CPUs and GPUs. In Chapter 4, both processor types are then compared by means of their hardware design goals, SIMD, MISD and MIMD capabilities. Finally, the clock frequency, number of compute units and size of compute units is investigated for the years 2005 to 2022. Concluding, the data analysis reveals a trend towards the convergence of CPU and GPU architectures.

9.1.2 Parallel Programming Model Trends

Research Question 2 *What are the parallel programming model trends for CPUs and GPUs?* To answer this question, Chapter 5 examines the layers of parallel programming models based on the general steps of parallelization. Next, it applies the stream interaction model to describe the partitioning models of Eventify, OpenMP, OpenACC and CUDA. Based thereon, it identifies two major areas of interest for the sustainable implementation of task-parallelism: *flexibility* and *uniformity*. Considering flexibility, *dynamic parallelism*, *cooperative groups*, *asynchronous task graphs* and *independent thread scheduling* are the forerunners of MIMD concurrency on GPUs. Considering uniformity, unified address space concepts and the conformation of GPU programming models to `C++` are the key trends to enable a uniform code path for CPUs and GPUs.

9.1.3 Task-Parallelism vs. Data-Parallelism on CPUs

Research Question 3 *Can event-based task-parallelism in comparison to loop-based data-parallelism enhance the scalability and execution time of the FMM on CPUs?* Yes, event-based task-parallelism can enhance the runtime and scalability of the FMM on CPUs. Section 8.3 shows that fine-grained task-parallelism improves the execution time of the FMM-implementation FMSolvr by a factor of 3.2 in comparison to classical loop-based data parallelism. Further, scalability is improved by 30%.

9.1.4 Event-Based Task-Parallelism on GPUs

Research Question 4 *How can event-based task-parallelism be enabled on GPUs?* Event-based task-parallelism is a strategy to enable the execution of fine-grained tasks. Chapter 6 introduces the foundation for event-based task-parallelism on GPUs – the development of an execution model that

enables threading as flexible as on CPUs. Since fine-grained task-parallelism requires a multitude of dependency resolutions, the focus of the performance optimization strategies is on mutual exclusion and queue-based scheduling concepts.

9.1.5 Task-Parallelism vs. Data-Parallelism on GPUs

Research Question 5 *Can event-based task-parallelism in comparison to loop-based data-parallelism enhance the scalability and execution time of irregular algorithms on GPUs?* Yes, event-based task-parallelism can enhance the runtime and scalability of concurrent applications on GPUs. Section 8.4 shows that fine-grained task-parallelism can improve the execution time of concurrent applications by an order of magnitude in comparison to classical loop-based data-parallelism on GPUs. For miniFMSolvr, the runtime is improved by a factor of 12, and scalability is improved by 60%.

9.2 Future Work

This section outlines ideas for future research in several areas related to the contributions of this work.

9.2.1 Stream Interaction Model

Further research should examine the soundness of the stream interaction model for non-Von-Neumann architectures like dataflow machines, reduction machines or even quantum computers and neural processors. Under the assumption that the idea of interacting data and instruction streams is transferable to these architectures, the stream interaction model could be applied to develop parallel programming models for unconventional computing. These programming models would enable software developers to focus on concurrency instead of the peculiarities of newly emerging hardware architectures.

9.2.2 Eventify

This section considers the future development of Eventify from three different directions. Firstly, the extension of Eventify itself. Secondly, the usage of Eventify to integrate task-parallelism in further applications. And thirdly, the comparison of Eventify against other parallel programming paradigms.

Extension of Eventify

The ongoing development of Eventify should include the merging of CPU- and GPU-Eventify. This involves multiple aspects.

Firstly, the template meta programming based DSL that enables the specification of dependencies between task types should be extended to GPUs to free users from initializing individual task dependency counters.

Secondly, CUDA kernel calls and decorators should be hidden from the user via wrapper functions to enable a uniform programming interface. This would also prepare the ground for a potential port of GPU Eventify to non-Nvidia GPUs with OpenCL or SYCL, without requiring users to change their application code.

Thirdly, as soon as the STL functions and classes required by Eventify (e.g. `std::vector`, `std::complex`, `std::sort`) become available on GPUs in `libcu++` or the `C++` standard, they should replace their prototypical pendants as implemented by the GTL. Based thereon, a full-fledged version of FMSolvr can be implemented on GPUs.

Fourthly, simultaneous task scheduling and execution on CPUs and GPUs could be implemented. For a start, this could be reached at task type level by executing independent task types on both processors. This approach could further be extended to multi-GPU systems by employing CUDA streams to overlap kernel execution in a similar manner as outlined in Section 6.4.2.

Fifthly, further work should include the inter-node parallelization of Eventify. For CPUs, a first prototypical implementation is described in [49], and [109] outlines theoretical notions on optimal communication algorithms in this context.

Application to Further Use Cases

Future research should include the parallelization of further applications with Eventify to analyze its practical usability. Ongoing work in this direction is the use of Eventify to parallelize *SMILEI* [30], a C++-based Particle-In-Cell code.

Regarding further use cases, the application of Eventify to algorithms that exhibit different kinds of task parallelism is also of academic interest. So far, Eventify has only been applied to algorithms that exhibit a limited set of task types and static task graphs. Apart from this, it would be valuable to examine the effectiveness and efficiency of Eventify in algorithms that exhibit dynamic, recursive task parallelism, e.g. as provided by backtracking-based algorithms. To analyze the effects of the MHQ policy further, algorithms that exhibit load imbalances between producer and consumer threads should be considered.

Comparison against Further Concurrent Programming Models

While this work evaluates event-based task-parallelism against classical data-parallelism via established parallel programming models such as OpenMP and OpenACC, further research should undertake a comparison against other task-parallel programming approaches on CPUs and GPUs. As outlined in Section 2.2.3, HPX, Kokkos and Whippletree are especially promising candidates for a comparative study of performance and programmability.

Chapter 10

Theses

1. The stream interaction model provides a uniform classification of concurrency in architectures and programming models.
2. MISD architectures exist in practice and correspond to pipelining when taking the concept of *derived streams* into account.
3. There is a trend towards the convergence of CPU and GPU architectures.
4. GPU architectures and programming models increasingly provide features to support irregular parallelism.
5. Queue-based work sharing approaches enable efficient task parallelism on GPUs.
6. Efficient spinlock-based mutual exclusion is feasible on GPUs.
7. Task parallelism is required to fully utilize the theoretical degree of concurrency of the fast multipole method.
8. Event-based task parallelism can outperform loop-based data-parallelism by an order of magnitude.

Acknowledgments

First of all, I would like to express my sincere gratitude to my supervisor Prof. Matthias Werner, chair of the Operating Systems Group at Chemnitz University of Technology. I am deeply thankful for our fruitful discussions, his extensive conceptual knowledge and his patience to share it with me. Further, I am more than grateful for the support in terms of time and funding.

I would like to extend my deep gratitude to Dr. Ivo Kabadshow, head of the Software Engineering group at Jülich Supercomputing Centre. Ivo never got tired to share his deep understanding of the Fast Multipole Method with me. Without his foresighted and rigorous approach to the development of FMSolvr and Eventify, this work would not have been possible.

Further, I am deeply grateful to Dr. Robert Speck, head of the Division Mathematics and Education at Jülich Supercomputing Centre. Robert provided me with the opportunity to synergize theory and practice in an open-minded and exceptionally friendly department.

I would like to extend my sincere gratitude to Prof. Dr. Peter Tröger, professor for distributed systems at the Berlin University of Applied Sciences and Technology. During his time at Chemnitz University of Technology, Peter helped setting the direction for my research by sharing his experience and knowledge on GPU programming models with me.

Further, I would like to thank my colleagues at Jülich Supercomputing Centre. Special thanks go to Dr. David Haensel for his thorough work on Eventify as well as the passionate and fruitful discussions on thread safety. I would like to extend my sincere gratitude to Andreas Beckmann for sharing his knowledge on lock-less programming and software architecture patterns with me. Special thanks also go to Mateusz Zych for the stimulating discussions on concurrency and parallelism in the context of C++, and to Andreas Herten for his insights on performance optimization for OpenACC.

Special thanks go further to Nvidia's Jiri Kraus, Markus Hrywniak and Matthias Wagner for their insights on memory fencing and `libcu++`.

I would like to say a big thank you to all my supervisees who supported the implementation of my (at times rather esoteric) ideas. Special thanks go to Noé Brucy, Michael Innerberger, Janos Meny, Wojciech Nawrocki, Stefan Staude, and Theresa Werner.

Furthermore, I would like to thank Jülich Supercomputing Centre and MEGWARE for providing me with state of the art compute resources. I am much obliged to the Freestate of Saxony for generously supporting this work via the PhD scholarship *Sächsisches Landesstipendium*.

Appendix A

GPU Terminology Mapping

Table A.1: Terminology mapping between the architectural models of CUDA and OpenCL.

CUDA	OpenCL
Host	Host
Device	Device
Streaming Multiprocessor	SIMD Unit
Streaming Processor/Cuda Core	Processing Element

Table A.2: Execution Model Terminology

Nvidia/CUDA	AMD/OpenCL	Description
Thread	Workitem	Single execution stream
Warp	Wavefront	Schedulable unit that consists of an architecture-dependent number of threads
Lane	Lane	Threads in a warp
Thread block	Workgroup	One-, two- or three-dimensional group of threads that execute together
Grid	NDRange	One-, two- or three-dimensional organization of thread blocks
Kernel	Kernel	C-like function that is executed by N different threads in parallel

Table A.3: Memory Model Terminology

Nvidia/CUDA Term	AMD/OpenCL Term
Local	Private
Shared	Local
Constant	Constant
Global	Global

Appendix B

Evaluation of Additional Task Graph Sizes

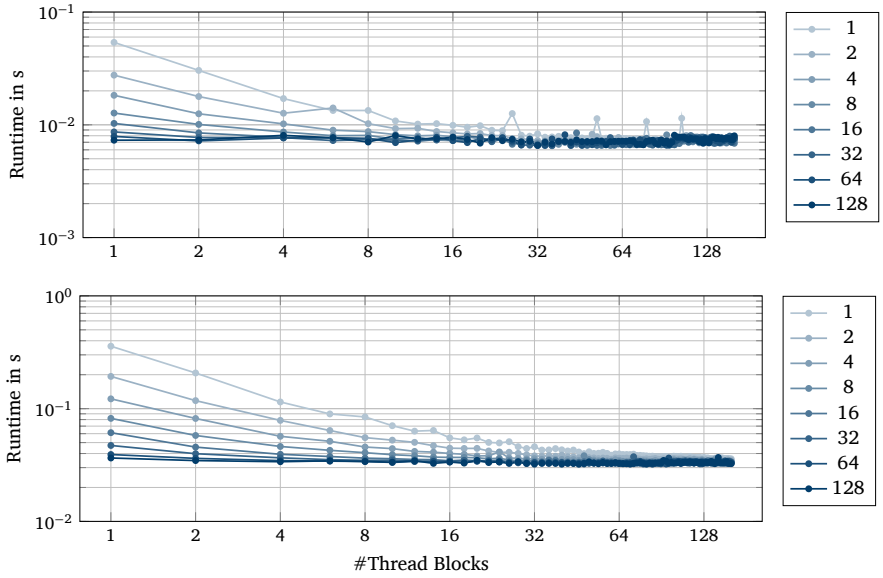


Figure B.1: miniFMSolvr on GPUs parallelized with OpenACC for tree depth $d_{\max} = 3$ and $d_{\max} = 4$.

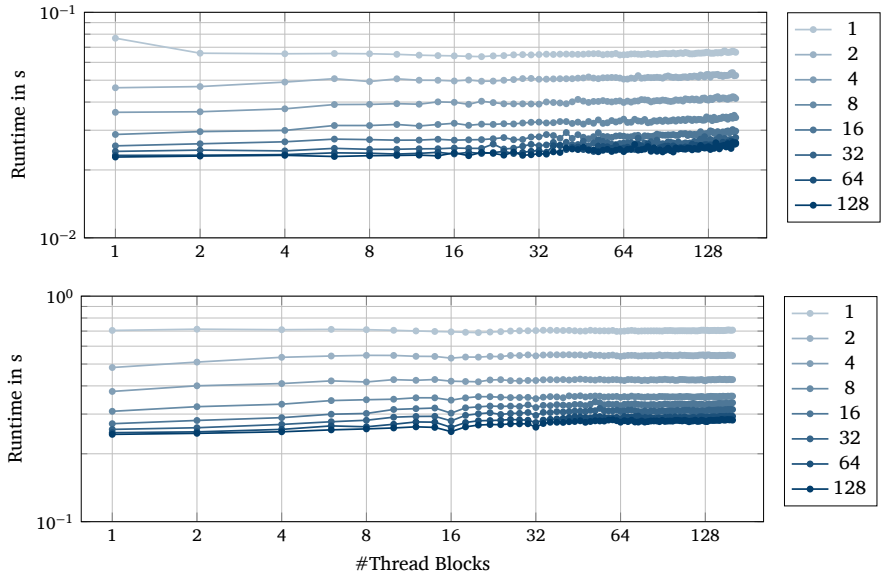


Figure B.2: miniFMSolvr on GPUs parallelized with Eventify and SQ scheduling for tree depth $d_{\max} = 3$ and $d_{\max} = 4$.

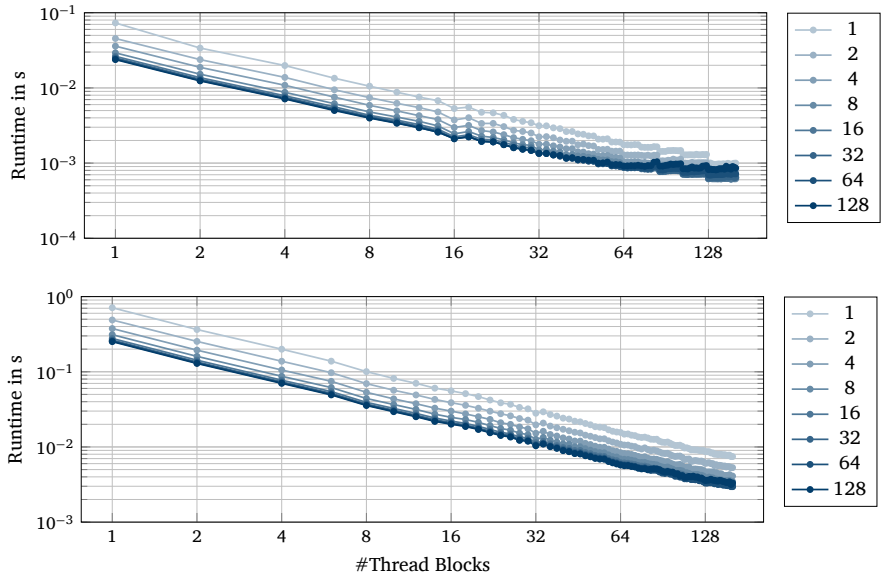


Figure B.3: miniFMSolvr on GPUs parallelized with Eventify and MQ scheduling for tree depth $d_{\max} = 3$ and $d_{\max} = 4$.

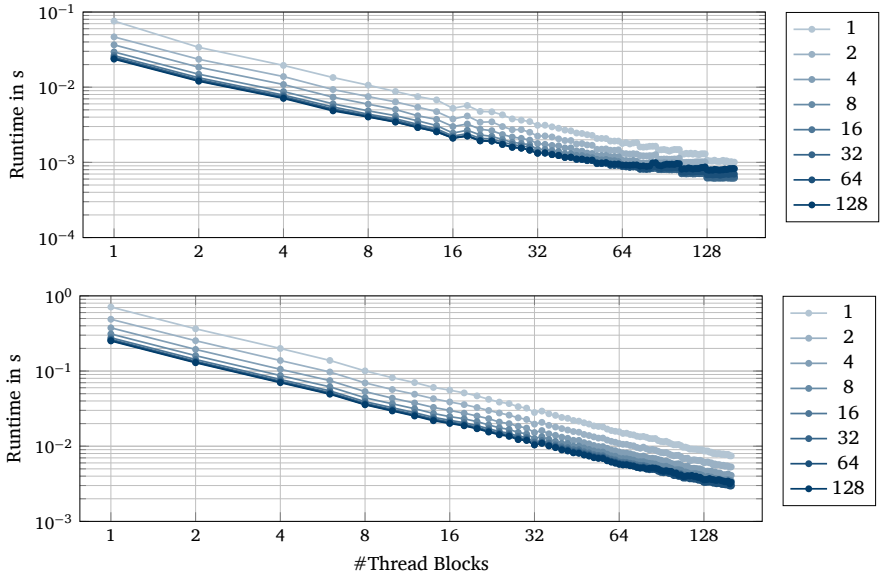


Figure B.4: miniFMSolvr on GPUs parallelized with Eventify and MHQ scheduling for tree depth $d_{\max} = 3$ and $d_{\max} = 4$.

Bibliography

[1] URL: http://www.fmsolvr.org/.

[2] URL: https://github.com/UoB-HPC/minifmm.

[3] Abduljabbar, M., Al Farhan, M., Yokota, R., and Keyes, D. "Performance Evaluation of Computation and Communication Kernels of the Fast Multipole Method on Intel Manycore Architecture." In: *Euro-Par 2017: Parallel Processing*. Ed. by Rivera, F. F., Pena, T. F., and Cabaleiro, J. C. Cham: Springer International Publishing, 2017, pp. 553–564. ISBN: 978-3-319-64203-1.

[4] Abraham, M. J., Murtola, T., Schulz, R., Páll, S., Smith, J. C., Hess, B., and Lindahl, E. "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers." In: *SoftwareX* 1-2 (2015), pp. 19–25. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2015.06.001. URL: http://www.sciencedirect.com/science/article/pii/S2352711015000059.

[5] Agullo, E., Aumage, O., Bramas, B., Coulaud, O., and Pitoiset, S. "Bridging the Gap Between OpenMP and Task-Based Runtime Systems for the Fast Multipole Method." In: *IEEE Transactions on Parallel and Distributed Systems* 28.10 (Oct. 2017), pp. 2794–2807. ISSN: 2161-9883. DOI: 10.1109/TPDS.2017.2697857.

[6] Agullo, E., Bramas, B., Coulaud, O., Darve, E., Messner, M., and Takahashi, T. "Task-based FMM for Heterogeneous Architectures." In: *Concurr. Comput. : Pract. Exper.* 28.9 (June 2016), pp. 2608–2629. ISSN: 1532-0626. DOI: 10.1002/cpe.3723. URL: https://doi.org/10.1002/cpe.3723.

[7] Agullo, E., Bramas, B., Coulaud, O., Darve, E., Messner, M., and Takahashi, T. "Task-Based FMM for Multicore Architectures." In: *SIAM Journal on Scientific Computing* 36.1 (2014), pp. C66–C93. DOI: 10.1137/130915662. eprint: https://doi.org/10.1137/130915662. URL: https://doi.org/10.1137/130915662.

[8] Aila, T. and Laine, S. "Understanding the Efficiency of Ray Traversal on GPUs." In: *Proceedings of the Conference on High Performance Graphics 2009*. HPG '09. New Orleans, Louisiana: Association for Computing Machinery, 2009, pp. 145–149. ISBN: 9781605586038. DOI: 10.1145/1572769.1572792. URL: https://doi.org/10.1145/1572769.1572792.

[9] Alglave, J., Batty, M., Donaldson, A. F., Gopalakrishnan, G., Ketema, J., Poetzl, D., Sorensen, T., and Wickerson, J. "GPU Concurrency: Weak Behaviours and Programming Assumptions." In: *SIGPLAN Not.* 50.4 (Mar. 2015), pp. 577–591. ISSN: 0362-1340. URL: https://doi.org/10.1145/2775054.2694391.

[10] Amarasinghe, S., Campbell, D., Carlson, W., Chien, A., Dally, W., Elnohazy, E., Hall, M., Harrison, R., Harrod, W., Hill, K., …, and Sterling, T. "Exascale Software Study: Software Challenges in Extreme Scale Systems." In: *DARPA IPTO, Air Force Research Labs, Tech. Rep* (2009). Ed. by Sarkar, V.

[11] AMD Community. *Arcturus/MI100 high level die shot annotations*. 2020. URL: https://forums.anandtech.com/threads/amd-cdna-compute-gpu-architecture.2577853/.

[12] AMD Community. *Compute Array comparison between Arcturus/CDNA/MI100 and Vega10/GCN5/MI25*. 2016. URL: https://on-demand.gputechconf.com/gtc/2016/webinar/openacc-course/Introduction-to-OpenACC-Course-20161102-1530-1-QA.pdf.

[13] Aschenbrenner, R. A., Flynn, M. J., and Robinson, G. A. "Intrinsic Multiprocessing." In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 81–86. ISBN: 9781450378956. DOI: 10.1145/1465482.1465495. URL: https://doi.org/10.1145/1465482.1465495.

[14] Atkinson, P. and McIntosh-Smith, S. "On the Performance of Parallel Tasking Runtimes for an Irregular Fast Multipole Method Application." In: *Scaling OpenMP for Exascale Performance and Portability*. Ed. by Supinski, B. R. de, Olivier, S. L., Terboven, C., Chapman, B. M., and Müller, M. S. Cham: Springer International Publishing, 2017, pp. 92–106. ISBN: 978-3-319-65578-9.

[15] Barnes, G. H., Brown, R. M., Kato, M., Kuck, D., Slotnick, D., and Stokes, R. A. "The ILLIAC IV Computer." In: *IEEE Transactions on Computers* C-17 (1968), pp. 746–757.

[16] Beatson, R. and Greengard, L. "A short course on fast multipole methods." In: *Wavelets, multilevel methods and elliptic PDEs* 1 (1997), pp. 1–37.

[17] Ben-Ari, M. *Principles of Concurrent Programming*. Prentice-Hall International, 1982. ISBN: 0-13-701078-8.

[18] Blumofe, R. and Leiserson, C. "Scheduling multithreaded computations by work stealing." In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 1994, pp. 356–368. DOI: 10.1109/SFCS.1994.365680.

[19] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. "Cilk: An Efficient Multithreaded Runtime System." In: *SIGPLAN Not.* 30.8 (Aug. 1995), pp. 207–216. ISSN: 0362-1340. DOI: 10.1145/209937.209958. URL: https://doi.org/10.1145/209937.209958.

[20] Breshears, C. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, Inc., 2009. ISBN: 0596521537.

[21] Cederman, D., Chatterjee, B., and Tsigas, P. "Understanding the Performance of Concurrent Data Structures on Graphics Processors." In: *Euro-Par 2012 Parallel Processing*. Ed. by Kaklamanis, C., Papatheodorou, T., and Spirakis, P. G. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 883–894. ISBN: 978-3-642-32820-6.

[22] Chamberlain, B. L. "Chapel (Cray Inc. HPCS Language)." In: *Encyclopedia of Parallel Computing*. Ed. by Padua, D. Boston, MA: Springer US, 2011, pp. 249–256. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_54. URL: https://doi.org/10.1007/978-0-387-09766-4_54.

[23] Charles, J., Jassi, P., Ananth, N. S., Sadat, A., and Fedorova, A. "Evaluation of the Intel® Core™ i7 Turbo Boost feature." In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 2009, pp. 188–197. DOI: 10.1109/IISWC.2009.5306782.

[24] Commission, E. *HORIZON 2020 – WORK PROGRAMME 2014-2015, Technology readiness levels (TRL)*. https://ec.europa.eu/research/participants/data/ref/h2020/wp/2014_2015/annexes/h2020-wp1415-annex-g-trl_en.pdf. 2014.

[25] committee, I. C. *Working Draft, Standard for Programming Language C++, N4140*. 2014. URL: https://github.com/cplusplus/draft/blob/main/papers/n4140.pdf.

[26] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[27] Culler, D., Singh, J. P., and Gupta, A. *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998. ISBN: 9780080573076.

[28] Darden, T., York, D., and Pedersen, L. "Particle mesh Ewald: An N□log(N) method for Ewald sums in large systems." In: *The Journal of Chemical Physics* 98.12 (1993), pp. 10089–10092. DOI: 10.1063/1.464397. URL: https://doi.org/10.1063/1.464397.

[29] Deakin, T., McIntosh-Smith, S., Price, J., Poenaru, A., Atkinson, P., Popa, C., and Salmon, J. "Performance Portability across Diverse Computer Architectures." In: *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 2019, pp. 1–13. DOI: 10.1109/P3HPC49587.2019.00006.

[30] Derouillat, J., Beck, A., Pérez, F., Vinci, T., Chiaramello, M., Grassi, A., Flé, M., Bouchard, G., Plotnikov, I., Aunai, N., Dargent, J., Riconda, C., and Grech, M. "Smilei : A collaborative, open-source, multi-purpose particle-in-cell code for plasma simulation." In: *Computer Physics Communications* 222 (Jan. 2018), pp. 351–373. DOI: 10.1016/j.cpc.2017.09.024. URL: https://doi.org/10.1016%2Fj.cpc.2017.09.024.

[31] Diehl, P., Seshadri, M., Heller, T., and Kaiser, H. "Integration of CUDA Processing within the C++ library for parallelism and concurrency (HPX)." In: *CoRR* abs/1810.11482 (2018). arXiv: 1810.11482. URL: http://arxiv.org/abs/1810.11482.

[32] Dijkstra, E. W. *EWD-1000*. URL: https://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1000.PDF.

[33] Dijkstra, E. W. *EWD476*. URL: https://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD476.html.

[34] Dijkstra, E. W. "Solution of a Problem in Concurrent Programming Control." In: (1965).

[35] Dubey, A., Brandt, S. R., Brower, R. C., Giles, M., Hovland, P. D., Lamb, D. Q., Löffler, F., Norris, B., O'Shea, B. W., Rebbi, C., Snir, M., and Thakur, R. "Software Abstractions and Methodologies for HPC Simulation Codes on Future Architectures." In: *Journal of Open Research Software* (2013). URL: http://doi.org/10.5334/jors.aw.

[36] Duncan, R. "A Survey of Parallel Computer Architectures." In: *Computer* 23.2 (Feb. 1990), pp. 5–16. ISSN: 0018-9162.

[37] Edwards, H. C. and Trott, C. R. "Kokkos: Enabling Performance Portability Across Manycore Architectures." In: *2013 Extreme Scaling Workshop (xsw 2013)*. Aug. 2013, pp. 18–24. DOI: 10.1109/XSW.2013.7.

[38] Edwards, H. C., Olivier, S. L., Mackey, G. E., Kim, K., Wolf, M., Stelle, G. W., Berry, J. W., and Rajamanickam, S. *Hierarchical Task-Data Parallelismusing Kokkos and Qthreads*. Tech. rep. Sandia Report SAND2016-9613, Sandia National Laboratories, Dec. 2016.

[39] Edwards, H. C., Trott, C. R., and Sunderland, D. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns." In: *Journal of Parallel and Distributed Computing* 74.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3202–3216. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.07.003. URL: http://www.sciencedirect.com/science/article/pii/S0743731514001257.

[40] Faison, T. *Event-Based Programming: Taking Events to the Limit*. 1st. USA: Apress, 2011. ISBN: 1430243260.

[41] Flynn, M. J. "Some Computer Organizations and Their Effectiveness." In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960.

[42] Flynn, M. "Very high-speed computing systems." In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909. DOI: 10.1109/PROC.1966.5273.

[43] Flynn, M. "Flynn's Taxonomy." In: *Encyclopedia of Parallel Computing*. Ed. by Padua, D. Boston, MA: Springer US, 2011, pp. 689–697. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_2.

[44] Fritzchens Fritz. *AMD Epyc 7702 ES*. 2019. URL: https://en.wikichip.org/wiki/File:AMD_Zen_2_CCD.jpg.

[45] Giroux, Olivier and Gelado, Isaac and Taylor, Paul. *A freestanding Standard C++ library for GPU programs*. 2019. URL: https://github.com/ogiroux/freestanding.

[46] Grama, A., Karypis, G., Kumar, V., and Gupta, A. *Introduction to Parallel Computing*. Second Edition. Addison-Wesley, 2003. ISBN: 9780201648652.

[47] Greengard, L. "The Numerical Solution of the N-Body Problem." In: *Computers in Physics* 4.2 (1990), pp. 142–152. DOI: 10.1063/1.4822898. eprint: https://aip.scitation.org/doi/pdf/10.1063/1.4822898. URL: https://aip.scitation.org/doi/abs/10.1063/1.4822898.

[48] Gupta, K., Stuart, J. A., and Owens, J. D. "A study of Persistent Threads style GPU programming for GPGPU workloads." In: *2012 Innovative Parallel Computing (InPar)*. 2012, pp. 1–14. DOI: 10.1109/InPar.2012.6339596.

[49] Haensel, D. "A C++ based MPI-enabled Tasking Framework to Efficiently Parallelize Fast Multipole Methods for Molecular Dynamics." PhD thesis. 2018.

[50] Haensel, D., Morgenstern, L., Beckmann, A., Kabadshow, I., and Dachsel, H. "Eventify: Event-Based Task Parallelism for Strong Scaling." In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. PASC '20. Geneva, Switzerland: Association for Computing Machinery, 2020. ISBN: 9781450379939. DOI: 10.1145/3394277.3401858. URL: https://doi.org/10.1145/3394277.3401858.

[51] Händler, W. "On Classification Schemes for Computer Systems in the Post-Von-Neumann-Era." In: *GI-4.Jahrestagung: Berlin, 9.–12. Oktober 1974*. Ed. by Siefkes, D. Berlin, Heidelberg: Springer Berlin Heidelberg, 1975, pp. 439–452. ISBN: 978-3-662-40087-6. DOI: 10.1007/978-3-662-40087-6_39.

[52] Herlihy, M. and Shavit, N. *The Art of Multiprocessor Programming, Revised Reprint*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123973375.

[53] Hillis, W. D. *The Connection Machine*. 1981. URL: https://apps.dtic.mil/dtic/tr/fulltext/u2/a107463.pdf.

[54] Hoare, C. A. R. "Communicating Sequential Processes." In: *Commun. ACM* 21.8 (Aug. 1978), pp. 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585.

[55] Hoare, C. A. R. "Towards a Theory of Parallel Programming." In: *Operating Systems Techniques, Proceedings of a Seminar at Queen's University* (1971).

[56] IEEE. *International Roadmap for Devices and Systems - Executive Summary*. 2020.

[57] Intel. *Intel Architectures Software Developer's Manual*. June 2021.

[58] Intel. *oneAPI Specification*. 2020. URL: https://spec.oneapi.com/versions/1.0-rev-3/oneAPI-spec.pdf.

[59] Intel®. *Product Specifications: Intel® Xeon® Processors*. Accessed June 10, 2022. URL: https://ark.intel.com/content/www/us/en/ark.html#@PanelLabel595.

[60] Kabadshow, I. "Periodic Boundary Conditions and the Error-Controlled Fast Multipole Method." PhD thesis. 2012.

[61] Kabadshow, I., Dachsel, H., Kutzner, C., and Ullmann, T. *GROMEX – Unified Long-range Electrostatics and Flexible Ionization*. http://www.mpibpc.mpg.de/15304826/inSiDE_autumn2013.pdf. 2013 (accessed April 27, 2017).

[62] Kaiser, H., Diehl, P., Lemoine, A. S., Lelbach, B. A., Amini, P., Berge, A., Biddiscombe, J., Brandt, S. R., Gupta, N., Heller, T., Huck, K., Khatami, Z., Kheirkhahan, A., Reverdell, A., Shirzad, S., Simberg, M., Wagle, B., Wei, W., and Zhang, T. "HPX - The C++ Standard Library for Parallelism and Concurrency." In: *Journal of Open Source Software* 5.53 (2020), p. 2352. DOI: 10.21105/joss.02352. URL: https://doi.org/10.21105/joss.02352.

[63] Kerbl, B., Kenzel, M., Mueller, J. H., Schmalstieg, D., and Steinberger, M. "The Broker Queue: A Fast, Linearizable FIFO Queue for Fine-Granular Work Distribution on the GPU." In: *Proceedings of the 2018 International Conference on Supercomputing*. ICS '18. Beijing, China: Association for Computing Machinery, 2018, pp. 76–85. ISBN: 9781450357838. DOI: 10.1145/3205289.3205291. URL: https://doi.org/10.1145/3205289.3205291.

[64] Khatami, Z., Kaiser, H., Grubel, P., Serio, A., and Ramanujam, J. "A Massively Parallel Distributed N-body Application Implemented with HPX." In: *Proceedings of the 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. ScalA '16. Salt Lake City, Utah: IEEE Press, 2016, pp. 57–64. ISBN: 978-1-5090-5222-6. DOI: 10.1109/ScalA.2016.12. URL: https://doi.org/10.1109/ScalA.2016.12.

[65] Khronos® OpenCL Working Group. *The OpenCL™ Specification v3.0.6*. 2020. URL: https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf.

[66] Kohnke, B., Kutzner, C., Beckmann, A., Lube, G., Kabadshow, I., Dachsel, H., and Grubmüller, H. "A CUDA fast multipole method with highly efficient M2L far field evaluation." In: *The International Journal of High Performance Computing Applications* 35.1 (2021), pp. 97–117. DOI: 10.1177/1094342020964857. eprint: https://doi.org/10.1177/1094342020964857. URL: https://doi.org/10.1177/1094342020964857.

[67] Kohnke, B., Ullmann, T. R., Beckmann, A., Kabadshow, I., Haensel, D., Morgenstern, L., Dobrev, P., Groenhof, G., Kutzner, C., Hess, B., Dachsel, H., and Grubmüller, H. "GROMEX: A Scalable and Versatile Fast Multipole Method for Biomolecular Simulation." In: *Software for Exascale Computing - SPPEXA 2016-2019*. Ed. by Bungartz, H.-J., Reiz, S., Uekermann, B., Neumann, P., and Nagel, W. E. Cham: Springer International Publishing, 2020, pp. 517–543. ISBN: 978-3-030-47956-5.

[68] Koziolek, H. "Sustainability Evaluation of Software Architectures: A Systematic Review." In: *Proceedings of the Joint ACM SIGSOFT Conference*. 2011. DOI: 10.1145/2000259.2000263.

[69] Kumar, V. P. and Gupta, A. "Analyzing Scalability of Parallel Algorithms and Architectures." In: *J. Parallel Distributed Comput.* 22 (1994), pp. 379–391.

[70] Laine, S., Karras, T., and Aila, T. "Megakernels Considered Harmful: Wavefront Path Tracing on GPUs." In: *Proceedings of the 5th High-Performance Graphics Conference*. HPG '13. Anaheim, California: Association for Computing Machinery, 2013, pp. 137–143. ISBN: 9781450321358. DOI: 10.1145/2492045.2492060. URL: https://doi.org/10.1145/2492045.2492060.

[71] Ltaief, H. and Yokota, R. "Data-Driven Execution of Fast Multipole Methods." In: *CoRR* abs/1203.0889 (2012). arXiv: 1203.0889. URL: http://arxiv.org/abs/1203.0889.

[72] McCool, M., Reinders, J., and Robison, A. *Structured Parallel Programming: Patterns for Efficient Computation*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123914439.

[73] Merrill, D. and Grimshaw, A. *Parallel scan for stream architectures*. Tech. rep. Technical Report CS2009-14, Department of Computer Science, University of Virginia, Dec. 2009.

[74] Morgenstern Laura an Kabadshow, I. and Werner, M. "Unparalleled Parallelism? CPU & GPU Architecture Trends and Their Implications for HPC Software." In: *Tagungsband des FG-BS Frühjahrstreffens 2021* (2021).

[75] Morgenstern, L. *A NUMA-Aware Task-Based Load-Balancing Scheme for the Fast Multipole Method*. Master Thesis, TU Chemnitz, 2017. DOI: 10.13140/RG.2.2.35575.93603.

[76] Morgenstern, L., Haensel, D., Beckmann, A., and Kabadshow, I. "NUMA-Awareness as a Plug-In for an Eventify-Based Fast Multipole Method." In: *Computational Science – ICCS 2020*. Ed. by Krzhizhanovskaya, V. V., Závodszky, G., Lees, M. H., Dongarra, J. J., Sloot, P. M. A., Brissos, S., and Teixeira, J. Cham: Springer International Publishing, 2020, pp. 428–441. ISBN: 978-3-030-50436-6.

[77] Navarro, C. A., Hitschfeld-Kahler, N., and Mateu, L. "A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures." In: *Communications in Computational Physics* 15.2 (2014), pp. 285–329. DOI: 10.4208/cicp.110113.010813a.

[78] Nvidia. *CUDA C++ Programming Guide*. 2020. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[79] Nvidia. *Introduction to OpenACC*. 2016. URL: https://on-demand.gputechconf.com/gtc/2016/webinar/openacc-course/Introduction-to-OpenACC-Course-20161102-1530-1-QA.pdf.

[80] Nvidia. "NVIDIA Tesla P100." In: (2016).

[81] Nvidia. "NVIDIA TESLA V100 GPU ARCHITECTURE." In: (2017).

[82] Nvidia. "NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110/210." In: (2014).

[83] OpenACC.org. *The OpenACC Application Programming Interface Version 3.1*. 2020. URL: https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.1-final.pdf.

[84] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 5.1*. 2020. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf.

[85] Otten, M., Gong, J., Mametjanov, A., Vose, A., Levesque, J., Fischer, P., and Min, M. "An MPI/OpenACC implementation of a high-order electromagnetics solver with GPUDirect communication." In: *The International Journal of High Performance Computing Applications* 30.3 (2016), pp. 320–334. DOI: 10.1177/1094342015626584. URL: https://doi.org/10.1177/1094342015626584.

[86] "Cilk Plus." In: *Encyclopedia of Parallel Computing*. Ed. by Padua, D. Boston, MA: Springer US, 2011, pp. 288–288. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_2339. URL: https://doi.org/10.1007/978-0-387-09766-4_2339.

[87] Pike, R. *Concurrency is Not Parallelism*. 2012. URL: https://go.dev/talks/2012/waza.slide#8.

[88] Plimpton, S. "Fast Parallel Algorithms for Short-Range Molecular Dynamics." In: *Journal of Computational Physics* 117.1 (1995), pp. 1–19. ISSN: 0021-9991. DOI: 10.1006/jcph.1995.1039.

[89] Rauber, T. and Rünger, G. *Parallel Programming for Multicore and Cluster Systems*. Second Edition. Springer-Verlag Berlin Heidelberg, 2013. ISBN: 978-3-642-37800-3. DOI: 10.1007/978-3-642-37801-0.

[90] Reinders, J. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007. ISBN: 978-0-596-51480-8.

[91] Research, C. *Cray-1 Computer Systems Hardware Reference Manual*. 1976. URL: http://bitsavers.trailing-edge.com/pdf/cray/CRAY-1/HR-0004F_CRAY-1_Computer_Systems_Hardware_Reference_Manual_May82.pdf.

[92] Robert, Y. "Task Graph Scheduling." In: *Encyclopedia of Parallel Computing* (2011). Ed. by Padua, D.

[93] Slotnick, D. L., Borck, W. C., and McReynolds, R. C. "The SOLOMON Computer." In: *Proceedings of the December 4-6, 1962, Fall Joint Computer Conference*. AFIPS '62 (Fall). Philadelphia, Pennsylvania: Association for Computing Machinery, 1962, pp. 97–107. ISBN: 9781450378796. DOI: 10.1145/1461518.1461528.

[94] Sorensen, T., Salvador, L. F., Raval, H., Evrard, H., Wickerson, J., Martonosi, M., and Donaldson, A. F. "Specifying and Testing GPU Workgroup Progress Models." In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485508. URL: https://doi.org/10.1145/3485508.

[95] Spector, A. and Gifford, D. "The Space Shuttle Primary Computer System." In: *Communications of the ACM* (1984).

[96] Steinberger, M., Kenzel, M., Boechat, P., Kerbl, B., Dokter, M., and Schmalstieg, D. "Whippletree: Task-Based Scheduling of Dynamic Workloads on the GPU." In: *ACM Trans. Graph.* 33.6 (Nov. 2014). ISSN: 0730-0301. DOI: 10.1145/2661229.2661250. URL: https://doi.org/10.1145/2661229.2661250.

[97] Strohmeier, E., Dongarra, J., Simon, H., and Meuer, M. *TOP500 List*. https://top500.org/lists/top500/list/2020/11/. Nov. 2020.

[98] Sutter, H. "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software." In: *Dr. Dobb's journal* 30.3 (2005), pp. 202–210.

[99] Tanenbaum, A. S. *Structured Computer Organization*. Upper Saddle River, NJ 07458: Pearson Prentice Hall, 2005. ISBN: 0-13-148521-0.

[100] Taura, K., Nakashima, J., Yokota, R., and Maruyama, N. "A Task Parallel Implementation of Fast Multipole Methods." In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. Nov. 2012, pp. 617–625. DOI: 10.1109/SC.Companion.2012.86.

[101] TechPowerUp. *AMD Arcturus*. 2020. URL: https://www.techpowerup.com/gpu-specs/amd-arcturus.g927#gallery-2.

[102] TechPowerUp. *CPU Specs Database*. Accessed June 10, 2022. URL: https://www.techpowerup.com/cpu-specs/.

[103] TechPowerUp. *GPU Specs Database*. Accessed June 10, 2022. URL: https://www.techpowerup.com/gpu-specs/.

[104] Teja Singh. *International Solid-State Circuits Conference*. 2020. URL: https://forums.anandtech.com/threads/amds-efforts-involved-in-moving-to-tsmcs-n7-advantages-for-going-with-chiplets-warning-many-images.2577325/.

[105] Thoman, P., Dichev, K., Heller, T., Iakymchuk, R., Aguilar, X., Hasanov, K., Gschwandtner, P., Lemarinier, P., Markidis, S., Jordan, H., Fahringer, T., Katrinis, K., Laure, E., and Nikolopoulos, D. S. "A Taxonomy of Task-Based Parallel Programming Technologies for High-Performance Computing." In: *J. Supercomput.* 74.4 (Apr. 2018), pp. 1422–1434. ISSN: 0920-8542. DOI: 10.1007/s11227-018-2238-4. URL: https://doi.org/10.1007/s11227-018-2238-4.

[106] Troendle, D., Ta, T., and Jang, B. "A Specialized Concurrent Queue for Scheduling Irregular Workloads on GPUs." In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP 2019. Kyoto, Japan: Association for Computing Machinery, 2019. ISBN: 9781450362955. DOI: 10.1145/3337821.3337837. URL: https://doi.org/10.1145/3337821.3337837.

[107] Tzeng, S., Lloyd, B., and Owens, J. D. "A GPU Task-Parallel Model with Dependency Resolution." In: *Computer* 45.8 (2012), pp. 34–41. DOI: 10.1109/MC.2012.255.

[108] Tzeng, S., Patney, A., and Owens, J. D. "Task Management for Irregular-Parallel Workloads on the GPU." In: *Proceedings of the Conference on High Performance Graphics*. HPG '10. Saarbrucken, Germany: Eurographics Association, 2010, pp. 29–37.

[109] Werner, T., Kabadshow, I., and Werner, M. "Systematic Literature Review of Data Exchange Strategies for Range-limited Particle Interactions." In: *Proceedings of the 12th International Conference on Simulation and Modeling Methodologies, Technologies and Applications - Volume 1: SIMULTECH,* INSTICC. SciTePress, 2022, pp. 218–225. ISBN: 978-989-758-578-4. DOI: 10.5220/0011144400003274.

[110] Xiao, S. and Feng, W. “Inter-block GPU communication via fast barrier synchronization.” In: *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470477.

[111] Yokota, R. and Barba, L. A. “A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems.” In: *The International Journal of High Performance Computing Applications* 26.4 (2012), pp. 337–346. DOI: 10.1177/1094342011429952. eprint: https://doi.org/10.1177/1094342011429952. URL: https://doi.org/10.1177/1094342011429952.

[112] Zhang, B. “Asynchronous Task Scheduling of the Fast Multipole Method Using Various Runtime Systems.” In: *Proceedings of the 2014 Fourth Workshop on Data-Flow Execution Models for Extreme Scale Computing*. DFM ’14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 9–16. ISBN: 978-1-4799-8095-6. DOI: 10.1109/DFM.2014.14. URL: https://doi.org/10.1109/DFM.2014.14.

Band / Volume 50
Utilizing Inertial Sensors as an Extension of a Camera Tracking System for Gathering Movement Data in Dense Crowds
J. Schumann (2022), xii, 155 pp
ISBN: 978-3-95806-624-3

Band / Volume 51
Final report of the DeepRain project
Abschlußbericht des DeepRain Projektes
(2022), ca. 70 pp
ISBN: 978-3-95806-675-5

Band / Volume 52
JSC Guest Student Programme Proceedings 2021
I. Kabadshow (Ed.) (2023), ii, 82 pp
ISBN: 978-3-95806-684-7

Band / Volume 53
Applications of variational methods for quantum computers
M. S. Jattana (2023), vii, 160 pp
ISBN: 978-3-95806-700-4

Band / Volume 54
Crowd Management at Train Stations in Case of Large-Scale Emergency Events
A. L. Braun (2023), vii, 120 pp
ISBN: 978-3-95806-706-6

Band / Volume 55
Gradient-Free Optimization of Artificial and Biological Networks using Learning to Learn
A. Yeğenoğlu (2023), II, 136 pp
ISBN: 978-3-95806-719-6

Band / Volume 56
Real-time simulations of transmon systems with time-dependent Hamiltonian models
H. A. Lagemann (2023), iii, 166, XXX pp
ISBN: 978-3-95806-720-2

Band / Volume 57
Plasma Breakdown and Runaway Modelling in ITER-scale Tokamaks
J. Chew (2023), xv, 172 pp
ISBN: 978-3-95806-730-1

Schriften des Forschungszentrums Jülich
IAS Series

Band / Volume 58
Space Usage and Waiting Pedestrians at Train Station Platforms
M. Küpper (2023), ix, 95 pp
ISBN: 978-3-95806-733-2

Band / Volume 59
Quantum annealing and its variants: Application to quadratic unconstrained binary optimization
V. Mehta (2024), iii, 152 pp
ISBN: 978-3-95806-755-4

Band / Volume 60
Elements for modeling pedestrian movement from theory to application and back
M. Chraibi (2024), vi, 279 pp
ISBN: 978-3-95806-757-8

Band / Volume 61
Artificial Intelligence Framework for Video Analytics: Detecting Pushing in Crowds
A. Alia (2024), xviii, 151 pp
ISBN: 978-3-95806-763-9

Band / Volume 62
The Relationship between Pedestrian Density, Walking Speed and Psychological Stress: Examining Physiological Arousal in Crowded Situations
M. Beermann (2024), xi, 117 pp
ISBN: 978-3-95806-764-6

Band / Volume 63
Eventify Meets Heterogeneity: Enabling Fine-Grained Task-Parallelism on GPUs
L. Morgenstern (2024), xv, 110 pp
ISBN: 978-3-95806-765-3

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

IAS Series
Band / Volume 63
ISBN 978-3-95806-765-3

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
Forschungszentrum