Patrick Pichler, BSc

Dynamic Dependencies for Task Scheduling on the GPU

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. BSc. Markus Steinberger

Institute of Computer Graphics and Vision
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Dieter Schmalstieg

Graz, September 2020

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

Date

Signature

Abstract

Algorithms implemented on the Graphics Processing Unit (GPU) become more and more complex. We want to split algorithms into simple tasks to overcome this complexity. However, by splitting an algorithm into tasks, usually dependencies between these tasks arise. These dependencies arise depending on the input of the tasks. So these tasks cannot be executed in an arbitrary order.

In this work, we present an extension to GPU task scheduling that enables the resolution of dynamic dependencies. We introduce two different approaches that handle dependencies independently: phases and individual dependencies. The first approach allows us to define an order of execution depending on the type of task. The second approach is used to define dependencies between individual tasks. These dependencies can be resolved depending on the input of the tasks. Both forms of dependency definition are necessary because, depending on the algorithm, only one of them is applicable.

We evaluated our approaches with two algorithms. The first algorithm is Reverse Cuthill-McKee (RCM), which is used to reorder a matrix into a band matrix with small bandwidth. We compared our RCM implementation with cuSolver showing speedups of up to 15x. The second algorithm is the Jacobi method, which iteratively determines a solution to a system of linear equations. Our Jacobi implementation was compared with a kernel-by-kernel implementation and achieved a speedup of up to 2x. With these implementations, we show that our framework can be used to implement a wide range of algorithms and achieve good performance.

Kurzfassung

Algorithmen, die auf einem Grafikprozessor (GPU) implementiert werden, werden immer komplexer. Um diese Komplexität zu bewältigen, wollen wir Algorithmen in einfache Tasks aufteilen. Aber durch das Aufteilen der Algorithmen in Tasks, entstehen Abhängigkeiten zwischen den Tasks. Diese Abhängigkeiten sind nicht einfach aufzulösen, da sie von den Inputs der Tasks abhängen. Die Tasks können also nicht in beliebiger Reihenfolge ausgeführt werden.

In dieser Arbeit zeigen wir eine Erweiterung von GPU Task Scheduling, welche die Auflösung von dynamischen Abhängigkeiten zwischen Tasks ermöglicht. Wir stellen zwei verschiedene Ansätze vor, die Abhängigkeiten unabhängig voneinander lösen: Phasen und individuelle Abhängigkeiten. Der erste Ansatz erlaubt es uns, eine Ausführungsreihenfolge in Bezug auf die Tasktypen zu definieren. Der zweite Ansatz wird verwendet, um Abhängigkeiten zwischen einzelnen Tasks zu definieren. Diese Abhängigkeiten können in Bezug auf den Input der Tasks aufgelöst werden. Beide Formen zur Auflösung von Abhängigkeiten sind notwendig, da je nach Algorithmus nur eine von ihnen anwendbar ist.

Wir haben unsere Ansätze mit zwei Algorithmen getestet. Der erste Algorithmus ist Reverse Cuthill McKee (RCM), welcher verwendet wird, um eine Matrix in eine Bandmatrix mit kleiner Bandbreite umzuordnen. Wir verglichen unsere RCM Implementierung mit cuSolver, wobei wir eine bis zu 15-fache Verbesserung der Ausführungszeit feststellen konnten. Der zweite Algorithmus ist die Jacobi Methode, welche iterativ eine Lösung für ein System von linearen Gleichungen bestimmt. Unsere Jacobi Implementierung wurde mit einer Kernel-by-kernel Implementierung verglichen und erreichte eine bis zu 2-fache Verbesserung der Ausführungszeit. Mit diesen Implementierungen zeigen wir, dass unser Framework zur Implementierung einer breiten Palette von Algorithmen geeignet ist, und eine gute Leistung erzielt wird.

Contents

List of Figures

1. Introduction

The developments of the recent years show a clear trend of high-performance computing towards Graphics Processing Unit (GPU) clusters. GPUs provide a large number of cores, which can be used for highly parallel execution of programs. However, it is challenging to implement and optimize algorithms for GPUs. A complex program is split into simpler parts called *task-types* to reduce the complexity and increase maintainability. This approach is called divide and conquer, and it is widely used in software development. However, by splitting an algorithm into task-types, dependencies between these tasks-types arise. These dependencies are difficult to handle since they are dependent on the input of the task-type. There are multiple options to resolve dependencies between task-types.

A commonly used technique is called *kernel-by-kernel*. The application is split into kernels. Now let us assume there are two kernels with data dependencies, i.e., kernel A requires data generated by kernel B. However, there is only a partial dependency. So parts of the input of A depends on parts of the output of B. Nevertheless, kernel B must be executed before kernel A. This order can be achieved by issuing the kernel launches in the right order since kernels can be executed sequentially. The GPU ensures the sequential execution of kernels in the same stream. This technique introduces a problem with GPU utilization. The execution of kernel A is delayed until kernel B is completed, although a part of kernel A may already be executed.

Another way to resolve dependencies but with fewer kernel launches is the CUDA graph API [Gra19]. This API is used to launch multiple kernels with a single operation. This is done by recording multiple kernel launches and CUDA API calls into a CUDA *graph*. Then the graph can be executed with a single call from the host. All the kernels and CUDA runtime functions, including the synchronization between different streams, are then performed by the device. Since the setup of the graphs is quite slow, the graph API is used to launch the same workflow multiple times. This is quite useful if the same algorithm has to be executed for many iterations or with many different inputs, for example, neural networks or particle simulations.

This work presents a way to resolve dependencies in a task scheduling framework. Task scheduling allows us to split an algorithm into *task-types*. The input parameters to these task-types are called *work items*. These work items are chunks of data that can be processed independently. The combination of a work item and a task-type forms a *task*. Task-types are separate functions of an algorithm. Each task-type defines the type of input (work item), the number of threads needed per task (*worker size*), and the amount of shared memory required. Each task-type implements an `execute` method that is processing the work item. The framework provides a queue per task-type that holds work items. Then it automatically schedules and executes tasks from the queues.

Here we show two ways of defining dependencies for task scheduling. The first kind is called *phases*, where multiple task-types are grouped together to form a phase. Then the tasks are executed according to their phase, meaning all tasks of the first phase are executed before tasks of the second phase. The other form of dependency management is to define *individual dependencies* between tasks. This technique allows a user to specify dependencies for each task separately.

The main reason to have two forms of dependency management is to make the system more flexible and versatile. A user implementing an algorithm with this framework can choose how to define the dependencies. However, there are many algorithms that only work with either phases or individual dependencies. It is generally easier to work with phases because they are statically defined for the algorithm. Individual dependencies, on the other hand, are dynamically defined between individual tasks. Since this allows us to define fine granular dependencies, the number of tasks that can be executed simultaneously is increased, which also improves performance.

1.1. CUDA

The GPU architecture is modeled as a manycore system. It consists of multiple Streaming Multiprocessors (SM), which are capable of executing multiple Collaborative Thread Arrays (CTA) each. Threads within CTAs are capable of efficient synchronization and communication. The communication between CTAs requires global memory, and thus, it is slower. CTAs and threads within CTAs have identifiers that can be one-, two- or three-dimensional.

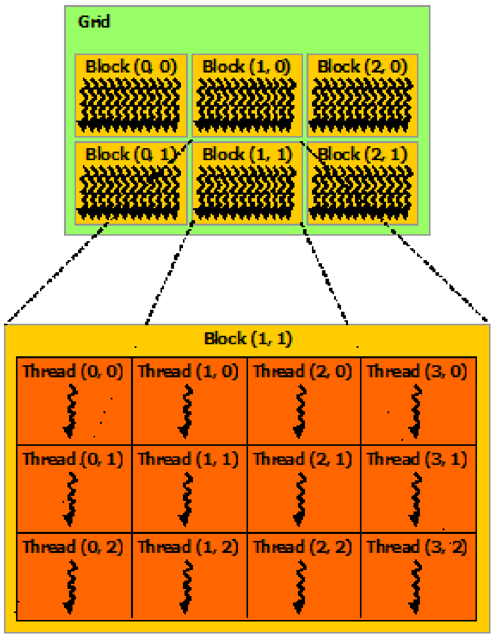

Figure 1.1.: Grid of thread blocks. [Nvi20a]

Each CTA is split into Single Instruction Multiple Thread (SIMT) groups, also called warps. Specialized warp-level primitives can be used for fast and efficient communication and synchronization. Every thread of a SIMT group has to execute the same instructions to achieve high performance. If threads within a warp diverge, the individual code paths are executed sequentially. When all code paths are completed, the threads converge. Therefore, it is important that the branching within warps is kept at a minimum.

The Volta architecture introduced independent thread scheduling. This allows threads to diverge and converge at a sub-warp granularity without sequential execution. Threads are grouped to sub-warp sized SIMT units, which enables SIMT execution with higher flexibility. But then threads within a warp are no longer executed in lockstep, which must be kept in mind when writing device code.

Another major influence in algorithm performance is memory transfer. The data transfer often surpasses the computation time. This is why CUDA has a hierarchy of memory spaces, each with different capacity and access time. The fastest and smallest memory space are registers, which are only accessible by the thread itself. The next memory space, which is larger and slower, is shared memory, which can be accessed by all threads within a CTA. The largest and slowest memory spaces are global, constant and texture memory, which are accessible by all threads of the device. Since register space is limited, there is also local memory. Local memory can only be accessed by the thread itself but has the same speed and size as global memory. If a thread uses too many registers, the excess is stored in local memory. [Nvi20a]

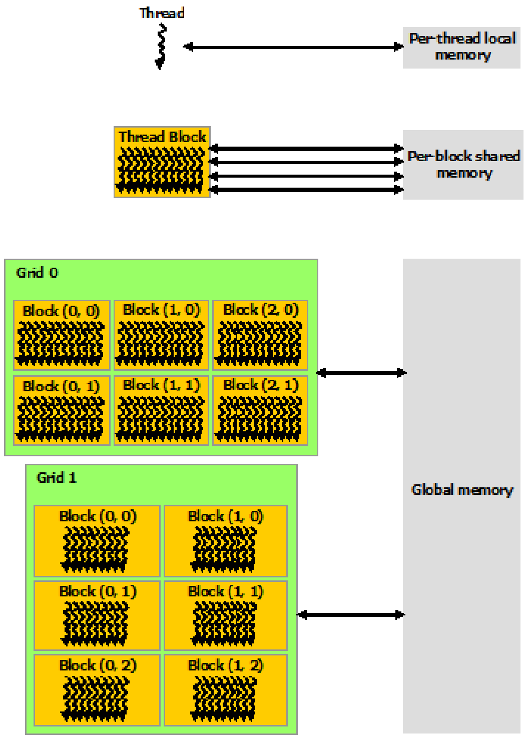

Figure 1.2.: CUDA memory hierarchy. [Nvi20a]

2. Related work

This work is built on top of the MHive task scheduling framework. MHive can schedule a wide range of algorithms. For example, graph processing, complex recursions and mesh processing. But for many complex algorithms, dependencies between tasks need to be resolved. In this work, we present a solution that gives users a simple and efficient way to handle dependencies.

2.1. Work Queue

Many algorithms require generating new work during execution dynamically. This means that there is a variable number of output data. This is no problem for CPU execution models since they are flexible enough to adjust to this. CPU models can allocate new memory and redistribute the work during execution. The GPU can also allocate new memory during kernel execution. But this is very slow and thus hardly used. Instead, usually multiple kernels are launched: first, a kernel to determine the amount of work generated and the number of threads needed, then a prefix sum to assign the work to the independent threads, and a final kernel to execute the tasks. Another approach is to manage the work in work queues [CT08]. Work queues are shared data objects to store tasks before and during execution. When a worker finishes its work, it can grab a new task from the work queue. Using these work queues, algorithms like work-stealing [Cha+13] or work-donation can be implemented on the GPU.

The work queue does not have the possibility to handle tasks with dependencies. However, it is possible to adopt the dependency models described in this thesis to the work queue. For this, the dependencies must be checked before work/threads are removed from the work queue.

2.2. Task-based Execution of Applications with Dynamic Data Dependencies

The work of Belviranli et al. [Bel+14] introduces a task-based execution framework running "all-in-GPU". The framework runs multiple worker thread blocks (TB) and a scheduler TB. Each worker TB has a private queue in global memory. There is also a global task list that is accessed by the scheduler TB. The execution follows the following steps:

1. The scheduler grabs tasks whose *dependency counter* is zero from the task list and puts them into the queues of the worker TBs. The worker TB is notified by incrementing the Input Queue Size (IQS) counter.
2. The worker TB executes the task.
3. The worker TB increments the Output Queue Size (OQS) counter to inform the scheduler that the task is finished.
4. The scheduler continuously checks the worker TBs for finished tasks. If a task is finished, the scheduler goes over the dependencies via a provided dependency matrix and checks whether new tasks can be executed.

This process repeats until there are no more tasks to process.

Since the dependency matrix is immutable, it is not possible to change the dependencies based on the input of the tasks. This is a disadvantage compared to our approach, which supports dynamically changing dependencies. Moreover, there is no way to define phases. So there are problems where this task-based execution framework does not work. Another difference is that this framework only supports one scheduling system, which uses persistent workers. If the resource requirements of the tasks are varying strongly, then persistent workers do not perform well.

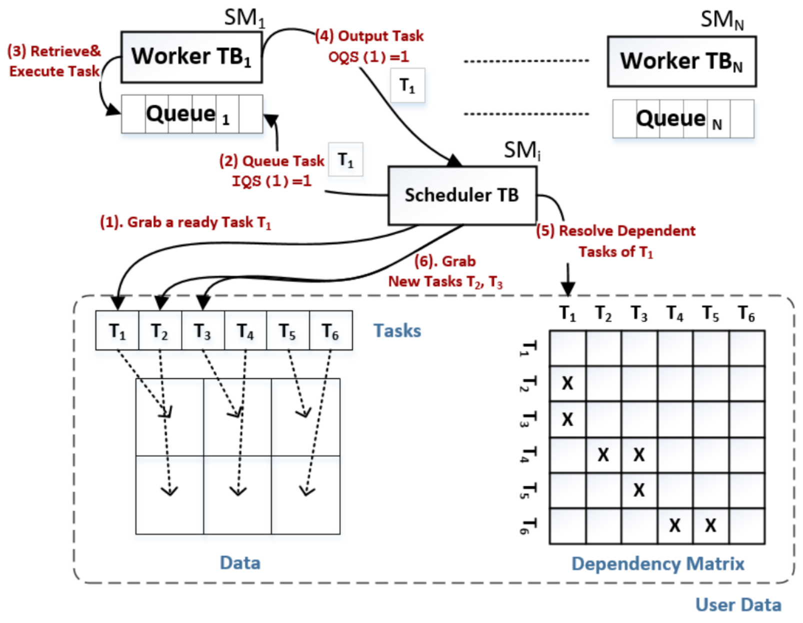
SM$_1$
(3) Retrieve& Execute Task
Worker TB$_1$
Queue $_1$
(4) Output Task OQS (1) =1
T$_1$
(2) Queue Task IQS (1) =1
SM$_i$
Scheduler TB
SM$_N$
Worker TB$_N$
Queue $_N$
(1). Grab a ready Task T$_1$
(5) Resolve Dependent Tasks of T$_1$
(6). Grab New Tasks T$_2$, T$_3$
Tasks
Data
Dependency Matrix
User Data

Figure 2.1.: Overview of the framework introduced by Belviranli et al. [Bel+14].

2.3. GPU Task-Parallel Model with Dependencies

The work of Tzeng, Lloyd, and Owens [TLO12] introduces a persistent threads model with a global task queue. The worker size is 32 threads (one warp). Each warp dequeues and executes tasks from the task queue and puts the output of the task into an output queue. Tasks may also produce new tasks wich are enqueued into the global task queue. The dependency resolution scheme ensures that a task can only be enqueued if all dependencies are resolved. The dependencies are defined using a dependency map. The framework supports multiple tasks, but the code for all tasks is in one large Megakernel. Depending on the task, one of the execution paths is selected.

The framework of Tzeng, Lloyd, and Owens [TLO12] is also similar to the Megakernel of MHive. But the worker size is fixed while MHive allows a configurable worker size. The dependency resolution is similar to our individual dependencies. However this framework does not support phases, which are required to implement many algorithms.

3. MHive

3.1. General

The MHive framework provides a queue per task-type. The scheduler decides which tasks to execute depending on the queue fill levels and the device utilization. More details on the different schedulers can be found in the chapters below. The scheduler launches workers that take elements from the queues and process them. During the execution of tasks, it is also possible to generate new work depending on the input.

3.2. Timesliced Kernel (TSK)

This scheduler runs a controller on the CPU. In the beginning, the controller copies the fill levels of the queues from the device to the host. Then the number of blocks needed for the kernel launches is calculated. After that, the kernels which are executing the tasks are launched into different streams; there is one kernel launch per task-type. If the number of blocks is zero, no kernel is launched.

3.3. Megakernel (MK)

This scheduler starts a kernel with enough blocks to fill the whole device when the application is started. The number of blocks depends on the number of SMs as well as the number of resident blocks on a SM. These blocks contain the worker threads, which are executing the tasks. Each block is independently dequeueing and executing work-items from the queues. All worker blocks run until the end of the application. The tasks are not necessarily of the same size as the worker blocks. So one block is usually executing multiple tasks at once. Since not all tasks are completed at the same time, there must be a way for the threads of the worker block to communicate with each other. This communication is done via status variables in shared memory. There is also a counter in global memory to know how many blocks are still running. If this counter reaches zero, all blocks are out of work, and the current phase is finished.

3.4. Persistent Device Controller (PDC)

Similar to TSK, this scheduler features a controller, which is scheduling and launching tasks. The main difference is that the controller is executed on the GPU. The controller performs the following steps in a loop: Each controller thread checks the fill level of its queue and determines whether a launch would be efficient. If this is the case, the controller performs the kernel launch to execute the tasks. The controller is also increasing a global variable that tracks the number of active warps on the device. Each warp has to decrease this count before it terminates. When this *warp count* reaches zero, the controller finishes execution.

4. Phases

The notion of phases is described using a simple example of building a table. To build a table, one must first make the individual parts: the legs and the tabletop. So we have two task-types to create the individual parts: *CreateLeg* and *CreateTabletop*. There is also a task-type to assemble the table called *AssembleTable*. We can define the following phases to ensure that all tasks are executed in the correct order:

1. Phase: *CreateLeg*, *CreateTabletop*
2. Phase: *AssembleTable*

It does not matter if the tabletop or the legs are created first so the corresponding tasks are in the same phase. However, the table can only be assembled if the individual parts are finished, so *AssembleTable* is in the second phase.

4.1. Overview

The entire execution is split into separate phases, where each phase is a group of task-types. The next phase is executed after all tasks of the previous phase are finished. An example program using phases to resolve the dependencies is visualized in Figure 4.1.

The phases of the application are defined using a phase queue. Each phase consists of one or more task-types. The order in which these phases are defined corresponds to the order of execution. All tasks of the specified task-types are executed in the respective phase. If all tasks of the current phase are finished, the next phase is executed. It is also possible to specify task-types, which do not belong to any phase. Tasks of these task-types can be executed at any time. An example of queues grouped into phases is given in Figure 4.2.

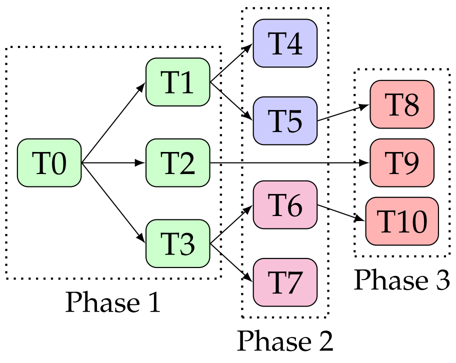

Figure 4.1.: Example program using phases as dependencies. The arrows between the tasks show enqueue operations. The color of the tasks show the task-type.

4.2. Cyclic execution

Many algorithms require an iterative execution of tasks. This means that after the last phase was executed, the first phase should be executed again. So all phases are executed in specified order until all queues are empty. An example of cyclic execution is shown in Figure 4.3. If a phase has no tasks it is skipped. This can be used to handle some iterations differently. For example, the initialization of data is only done in the first iteration.

4.3. Phase Queue

The work items of tasks are stored in queues. There is a separate queue for each task-type. Each queue is given an index in the order that the queues are created. This index is automatically assigned at the declaration of each task-type. Since the task-types which belong to the same phase are declared together, they have successive indices. This fact can be used to create an efficient way to check whether a task-type belongs to the current phase. A

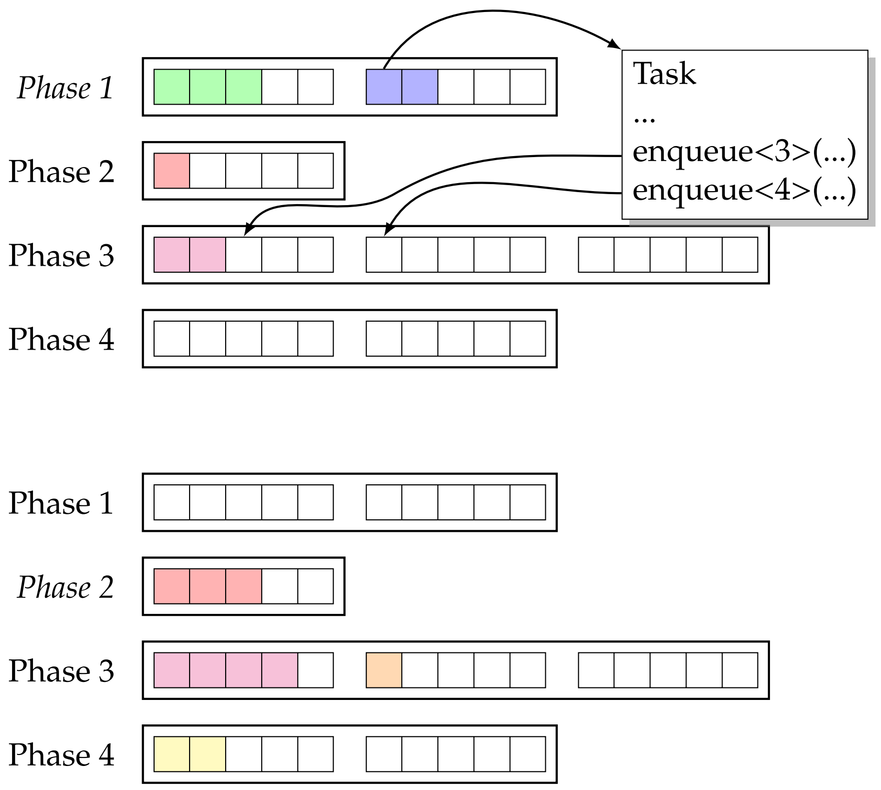

Figure 4.2.: THis overview figure shows how the queues of task-types are grouped into phases. The current phase is shown in italic. To advance from phase 1 to phase 2, all queues of phase 1 must be empty.

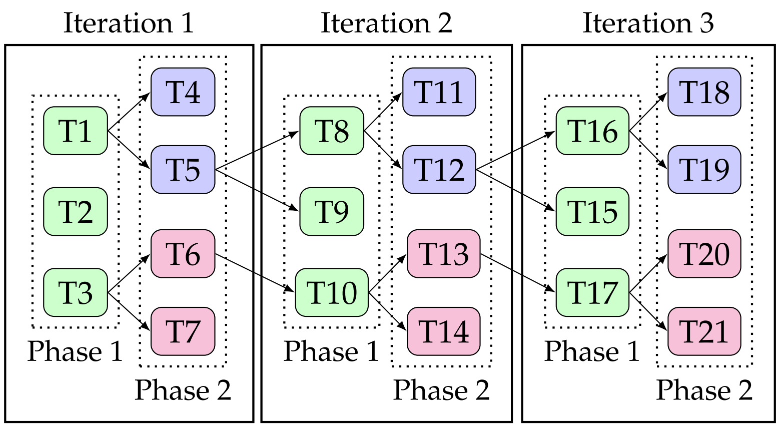

Figure 4.3.: Example of cyclic execution with phases. The arrows between the tasks show enqueue operations. The color of the tasks shows the task-type.

Phase	index	start	end	Task-type	index
Phaseless	-1	0	0		
Phase1	0	0	2	TaskType1	0
				TaskType2	1
Phase2	1	2	5	TaskType3	2
				TaskType4	3
				TaskType5	4

Table 4.1.: Calculated indices for example in Listing 4.1.

start and end index for each phase can be calculated. For example, the indices of the queues for the first phase start with zero and end with the number of task-types in this phase. The framework automatically creates a phase called *phaseless*. This phase is used to handle tasks that do not belong to any phase. The indices for the phases in Listing 4.1 are listed in Table 4.1. The example consists of two phases, with five task-types in total.

The calculated indices are then used to determine the phase of a given task-type. The scheduler keeps track of the current phase and can determine whether a task belongs to the current phase or not. There is also a special index for phaseless task-types.

The definition of phases and queues in C++ is done using variadic templates. These are templates that take a variable number of arguments. There is a template class called `Phase`, which takes an arbitrary number of task-types. These *phases* are then given to another template class called `PhaseQueue`. The order of execution depends on the order of the arguments of the `PhaseQueue` class. An example of how to declare phases can be found in Listing 4.1.

Listing 4.1: Example of how to declare phases

```cpp
using Phase1 = Phase<TaskType1, TaskType2>;
using Phase2 = Phase<TaskType3, TaskType4, TaskType5>;
using Queue = PhaseQueue<Phaseless<>, Phase1, Phase2>;
```

The `PhaseQueue` class just holds the parameters like the queue configuration and the phases. The implementation is in the `PhaseQueueImpl` template class. This class template is instantiated once for each phase parameter given to the

`PhaseQueue`. Each `PhaseQueueImpl` knows the class (instantiated template) and instance of the next template instantiation. This is used to perform actions on all phases or task-types. Each method of the `PhaseQueueImpl` performs an action and then calls the same method of the next class (template instantiation). The class `PhaseQueue` is also implementing the first instantiation of `PhaseQueueImpl`. A visual representation of this can be found in Figure 4.4

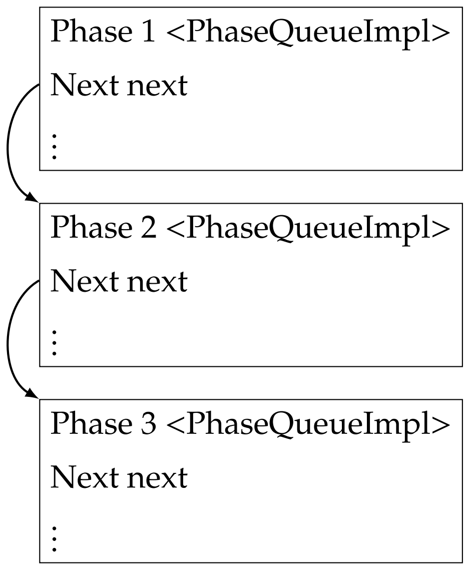

Figure 4.4.: Visual representation of phase queue structure. This example shows three phases.

5. Individual Dependencies

Let us come back to the example of building a table described before to understand the benefits of individual dependencies. Now we want to build many tables quickly and efficiently. As stated before, we need to make the legs and tabletop and then assemble the table. Let us assume there are three kinds of stations needed: a lathe for the legs, a mill for the tabletop and a workbench to assemble the table. If we resolve the dependencies using phases, then there are always idle stations. The workbench is idle while we make the parts. While the table is assembled, the lathe and mill are idle. A better solution is to define the dependencies for each table individually. Then the parts for a table can be created while another table is assembled. This increases the throughput of the workshop compared to the approach using phases.

5.1. Overview

This form of resolving dependencies, allows the framework to handle different dependencies. Each task can specify other tasks that must be finished before it can be executed. So while phases define task-type relations, which is very coarse, with individual dependencies, we can define fine-grained relations on the tasks itself. When using phases, a single long-running task delays the execution of all successive phases. With individual dependencies on the other hand, only a few dependent tasks are delayed. In such cases, individual dependencies provide better performance compared to phases.

This form of dependency management is also required for algorithms that can not be split into phases. Some algorithms might only use one task-type, which is executed many times with different input data. Then it is not possible to define phases since the dependencies depend on the input data and there is only one task-type. The basic operations of individual dependencies using the MK scheduler are shown in Figure 5.1.

1. A task T1 that depends on other tasks is created and inserted into the waiting storage.
2. Other workers (e.g. worker 2) execute tasks that resolve dependencies of T1.
3. If all dependencies of T1 are resolved, then the task is moved from the waiting storage to the queue. This is done by the task that resolves the last dependency and thus reduces the dependencies to zero (e.g. task executed by worker 3).
4. The workers continuously grab and execute tasks from the queue.

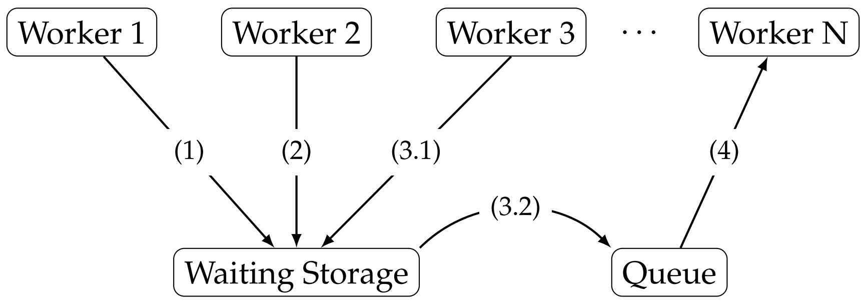

Figure 5.1.: Overview figure showing the basic principles of individual dependencies using the MK scheduler. (1) Enqueue Task T1 with dependencies. (2) Reduce dependencies of T1. (3.1) Reduce dependencies of T1 to zero. (3.2) Move T1 to Queue. (4) Grab and execute tasks from the queue.

We have to know on how many tasks it depends on to know whether a task may be executed. This number of dependencies is stored in an atomic counter. If a task T that this task depends on is finished, then task T reduces the counter by one. As soon as the counter reaches zero, the corresponding task can be executed. The initial value of this counter is zero. Since the value of the counter is checked after reducing it, it can only reach zero after the number of dependencies have been set by enqueueing the corresponding task. For example task X is generated that has data dependencies on tasks Y and Z. After tasks Y and Z have created the data that task X requires, they each reduce the number of dependencies of task X by one. Since the dependency counter of task X is back to zero, it can be executed.

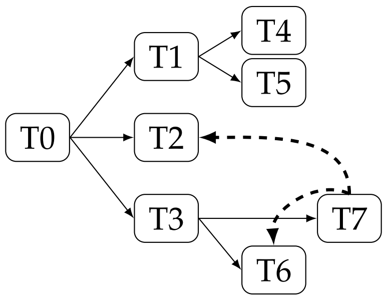

Figure 5.2.: Example of tasks with individual dependencies. The thin arrows between the tasks show enqueue operations. The thicker dashed arrows show dependencies. In this example, T7 depends on T2 and T6. Thus it is executed after T2 and T6 are finished.

5.2. Waiting Storage

The work items which are to be executed are stored in a queue. Queues are not suited for random access, which is required for individual dependencies. So we introduced an array which buffers tasks with dependencies. Henceforth this array will be called *waiting storage*.

There are two types of operations to the waiting storage: writing (insert tasks) and reading (extract tasks). The insertion of tasks into the waiting storage is a two-step process. The first step is the reservation of an empty slot (see Algorithm 5.1). An empty slot is selected by generating a random value that lies within the size of the array. But the policy to insert new tasks is encapsulated and configurable as a parameter of the waiting storage. This allows one to write an application-specific insert policy that possibly yields better performance. The reservation is made using atomic operations on the status variable *reserved*, which reserves a spot for writing. The variable is initialized with zero. If a thread wants to reserve a position, it sets *reserved* to one using an `atomicCAS` operation. If the spot is already taken, the next spot is checked. After a slot has been reserved, a task can be written to it in the second step (see Algorithm 5.2). For this step, no atomic operations are necessary, since only one thread accesses a slot after reservation. There is a second status variable called *filled*, which is set after the work item was written to the slot. The steps necessary to insert tasks are hidden by a facade to reduce complexity for the user.

The extraction of tasks from the waiting storage is done with a single function (see Algorithm 5.3). First, the location of the data is read from the dependency. Second, the work item is read from this location. Then the work item is enqueued. After that the variables are set back to zero in reverse order. So first *filled* is set to zero, and then *reserved*.

Algorithm 5.1 Reserve spot in waiting storage

```
function RESERVE($data$)
    $start \leftarrow$ insertPolicy($data$)
    $idx \leftarrow start$
    while true do
        if atomicCAS($reserved[idx], 0, 1$) $= 0$ then
            return $idx$
        end if
        $idx \leftarrow idx + 1$                                ▷ linear probing
        if $idx > STORAGE\_SIZE$ then
            $idx \leftarrow 0$
        end if
    end while
end function
```

Algorithm 5.2 Write to spot in waiting storage

```
function WRITE($data, spot, \&dependency$)
    $storage[spot] \leftarrow data$
    $filled[spot] \leftarrow 1$
    $dependency.location \leftarrow spot$
end function
```

5.3. Dependency

The dependencies between tasks are managed by objects of a class called `Dependency`. These objects include the atomic counter to store the number of dependencies as well as the storage location of the corresponding task. The atomic counter is initialized to zero. The `Dependency` object provides a simple interface to manage the dependencies of a task. The primary method

Algorithm 5.3 Handle tasks where all dependencies are resolved

```
1: function HANDLEREADYTASK($dependency$)
2:     $spot \leftarrow dependency.location$
3:     $data \leftarrow storage[spot]$
4:     enqueue($data$)
5:     $filled[spot] \leftarrow 0$
6:     $reserved[spot] \leftarrow 0$
7: end function
```

used to manage the dependencies of a task is called `reduce`, which reduces the number of dependencies by one. The `reduce` method also moves the task from the *waiting storage* to the queue, if the number of dependencies is reduced to zero. It is also possible to reduce the number of dependencies of a dependency object before the corresponding task is enqueued.

Now let us consider an example. A task X depends on two other tasks Y and Z. The task X is enqueued with the number of dependencies set to three. It is set to the number of tasks it depends on $+1$ to avoid race conditions. After task X was written to the waiting storage, the number of dependencies is reduced to two. The dependency object is stored in global memory such that tasks Y and Z can access it. Then task Y is executed and at the end of the execution it reduces the dependencies of X to one. After that, task Z is executed and reduces the dependencies of X to zero. By reducing the dependencies to zero, task Z also removes task X from the waiting storage and enqueues it. Since task X is now in the queue, it can be executed.

Now let us consider a similar example but with different order of events. We have again task X that depends on tasks Y and Z. Now task Y is executed first and reduces the dependencies of X to -1. Next, task Z is executed and reduces the dependencies of X to -2. After that, task X is enqueued. Since the dependencies are already resolved ($-2 + 2$ dependencies $= 0$), task X is written directly to the queue instead of the waiting storage.

5.4. Comparison to phases

We can now compare individual dependencies and phases. Both systems have benefits and drawbacks depending on the algorithm to implement. Some algorithms are easily split into phases, while for others, individual dependencies must be defined. This means that neither one of these systems is generally better. Both of these systems are required so the user can pick the one that fits the problem.

6. Scheduling

The three execution models discussed here were already implemented in MHive. However, to increase performance and ensure compatibility with the newly implemented dependencies, they were modified. The following sections describe these modifications.

It is not necessary to make changes to the scheduler to support individual dependencies. Tasks with dependencies are stored in the waiting storage. As soon as the dependencies of the tasks are resolved, they are moved to the queue.

6.1. Timesliced Kernel (TSK)

The scheduler executes kernel launches to execute tasks from the queues. If a task-type is not in the current phase, the number of required blocks for this kernel launch is set to zero. This ensures that no kernel is launched for this task-type. Moreover, a variable indicating the current phase was added. This variable is initialized to zero and incremented when all queues of the current phase are zero and no warps are running.

6.2. Megakernel (MK)

Several changes were necessary to support the dependency model described in this work. Task-types that are not in the current phase are skipped. A variable in global memory was added to keep track of the current phase. This variable is incremented if all queues of the current phase are empty and no blocks are executing tasks. Since the scheduling is done decentralized, only the first block writes to this variable to avoid race conditions. The original implementation used only the number of running blocks as exit condition. This was changed to additionally incorporate the fill levels of the queues, because we use the number of running blocks to update the current phase variable.

6.3. Persistent Device Controller (PDC)

Several changes were made to implement our dependency management but also increase performance. Although the previous implementation used multiple controller threads (as many threads as queues), only the first thread was performing the kernel launches. This was changed such that each controller thread launches the kernel for its queue. The queues for task-types which are not in the current phase are skipped to adhere to the dependencies. The current phase is stored in shared memory. Only the first controller thread increments the variable to avoid race conditions. The other threads may only read from the variable. If all queues of the current phase are empty and no warps are running (other than the controller), then the current phase is finished. Then the current phase variable is incremented.

Although kernels are launched from the device, there is still a launch overhead. So it is important that there is not a large number of small kernel launches. The scheduling implemented here uses metrics to prioritize efficient launches. So if there are controllers, which can fill multiple blocks, then only those controllers launch additional workers. Otherwise, all controllers that have non-empty queues may launch workers. If the GPU is sufficiently oversubscribed, then no additional kernels are launched. If the *warp count* falls below a threshold, additional workers are launched. The number of warps needed to sufficiently oversubscribe the device is calculated as:

$$\text{max\_warps} = \frac{\#\text{SMs} \cdot \text{max\_resident\_threads} \cdot 1.5}{\text{warp\_size}}$$

This yields a good device utilization independent of the used device.

7. Test Programs

7.1. Generic Test

This test program provides a toolset to create different test scenarios. It enables one to define an execution tree. So it is possible to define task-types and transitions between task-types. The transitions have a configurable probability. The task-types execute a configurable number of floating-point operations, which serve as a synthetic workload of the tasks. It is not possible to define cyclic transitions, because the task-types must be defined before a transition to it can be defined.

Figure 7.1 shows two execution trees with the corresponding scheduling plots. In this example, all transitions have a probability of 100%. The scheduling plots show how the dependencies are uphold for both examples. The scheduling plot for phases shows, that all tasks of task-type 0 are executed before tasks of task-types 1 and 2. The same holds for tasks of task-type 3, which are executed after all tasks of phase 2 are finished. If we compare this to the scheduling plot for individual dependencies we see, that tasks of task-type 3 may be executed as soon as its dependencies are resolved, so tasks of task-type 3 may run concurrent with tasks of task-types 1 and 2. The example with individual dependencies is still slower because of the additional overhead to manage the dependencies.

Figure 7.2 shows a larger and more complex example. The scheduling plot for phases shows a strict separation of the phases. In this example, the execution of all tasks in phase 1 is delayed, because some tasks of phase 0 are not scheduled efficiently. The plot for individual dependencies on the other hand, shows no separation. All tasks are executed as soon as their individual dependencies are resolved. But the overhead of individual dependencies increases the runtime of the tasks.

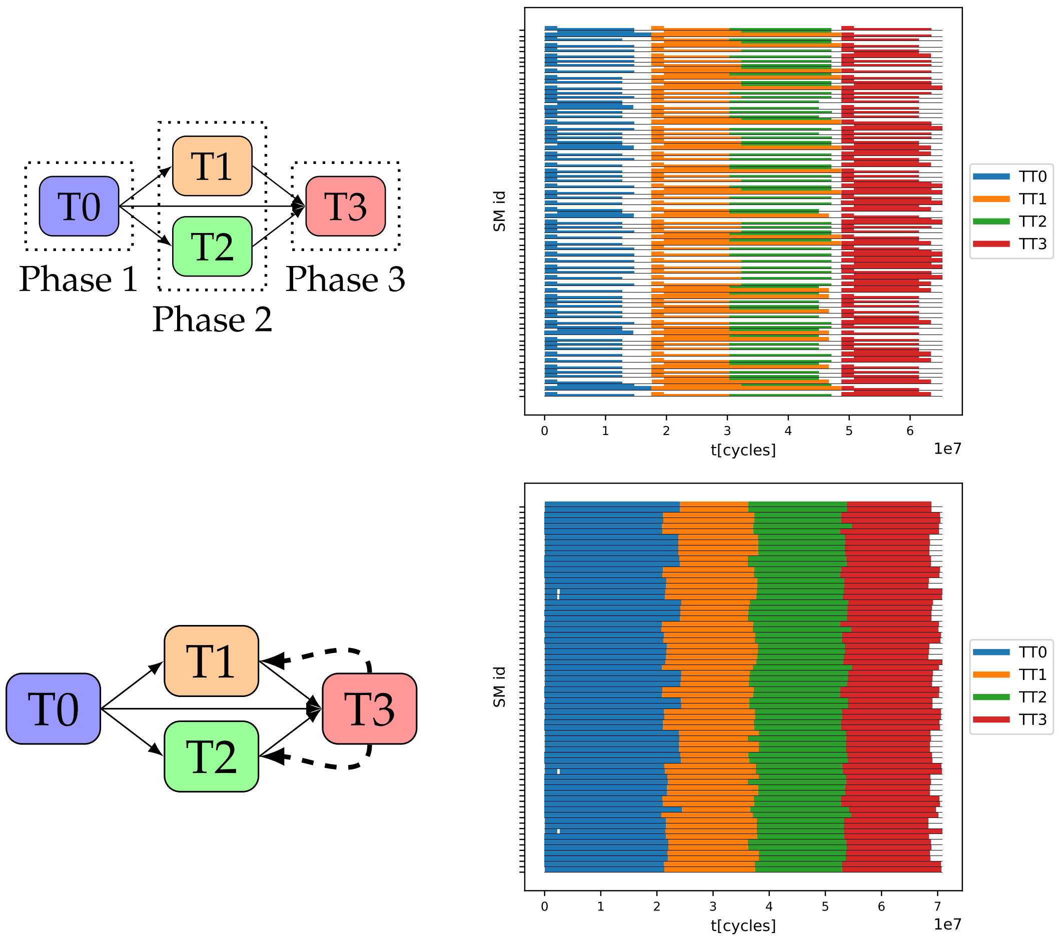

Figure 7.1.: Example configuration of the generic test program with 4 task-types. The first row shows an example with phases; in the bottom row, individual dependencies are used. The left images show the task-types and dependencies, while on the right a scheduling plot is shown.

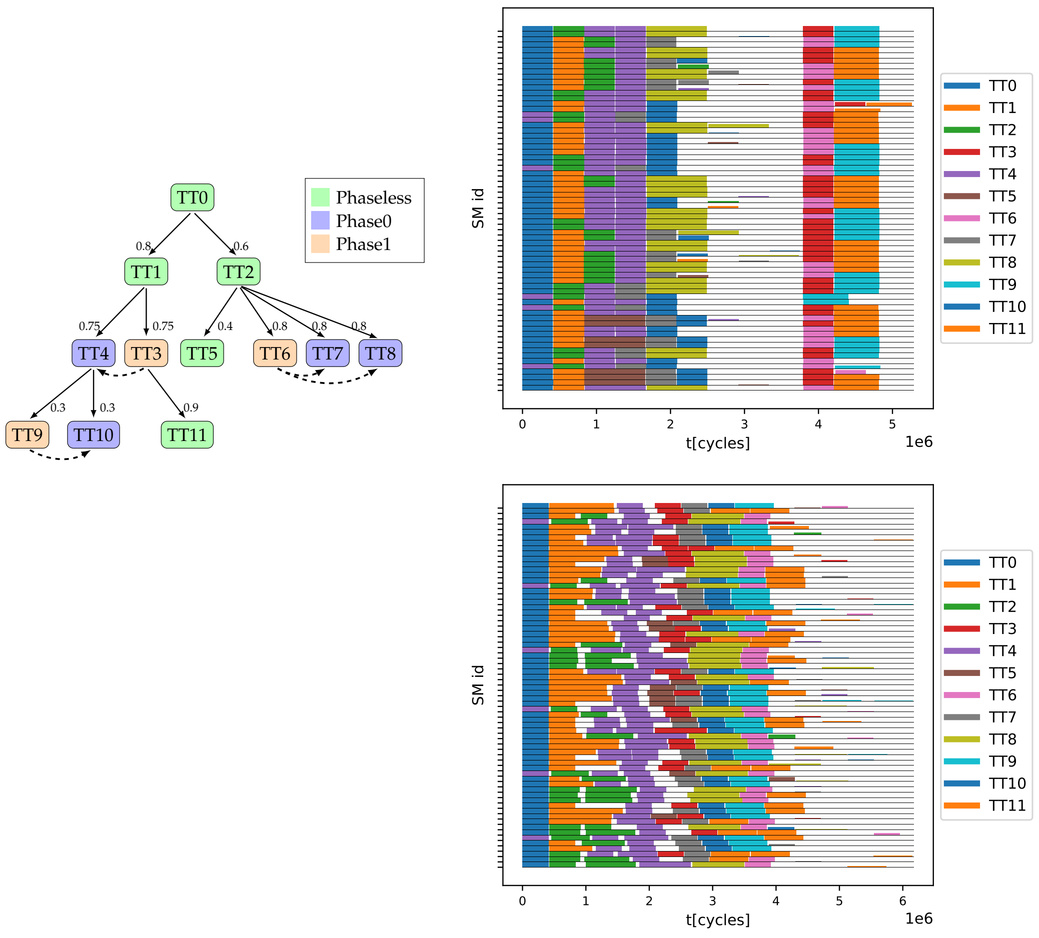

Figure 7.2.: Example configuration of the generic test program with 11 task-types. The first row shows an example with phases; in the bottom row, individual dependencies are used. The left images show the task-types and dependencies, while on the right a scheduling plot is shown.

7.2. Compressed sparse row (CSR) format

$$A = \begin{bmatrix} \cdot & 2 & \cdot & \cdot & \cdot & 2 & \cdot & \cdot & 5 & \cdot \\ \cdot & 3 & \cdot & 7 & 1 & 4 & \cdot & \cdot & 8 & 6 \\ \cdot & \cdot & 4 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 9 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \end{bmatrix}$$

$$\begin{aligned} val &= \{2,2,5,3,7,1,4,8,6,4,1,9,1,1\} \\ col\_id &= \{1,5,8,1,3,4,5,8,9,2,6,5,2,8\} \\ row\_offsets &= \{0,3,9,11,12,14\} \end{aligned}$$

Figure 7.3.: A matrix in CSR format with color-coded rows

Since this format is used to store data for the next test programs, it is described shortly here. The CSR format (see fig 7.3) is a commonly used format to store sparse matrices. This format consists of three arrays. The first two store the column indices and values. The third array stores the offset to the first two arrays for each row. So each entry is an index to the start of a row in the first two arrays. The third array has one more entry than the matrix has rows. This last entry holds the number of non zero elements of the matrix and points after the last entry of the first two arrays. The CSR format provides fast access to the rows of the matrix.

7.3. Cuthill-McKee bandwidth reduction

Cuthill-Mckee [CM69] is an algorithm to permute a sparse and symmetric matrix into a band matrix with small bandwidth. The algorithm generates a permutation R. It starts with a pseudo-peripheral node, and then generates sets A_i until all nodes are visited. The set A_i consists of the adjacent nodes of R_i (i-th element of R) that are not in R. Then A_i is sorted by ascending degree (see algorithm 7.1). The permutation R consists of the concatenation of all A_i.

Since the reversed permutation often leads to a better bandwidth reduction, the algorithm is also called *reverse Cuthill-McKee* (RCM) [Geo71]. The output of the RCM algorithm is a permutation vector, which can be applied to the matrix.

Algorithm 7.1 Cuthill-McKee

```
1: $R \leftarrow x$                          ▷ $x$ is a pseudo-peripheral node
2: $i \leftarrow 0$
3: while $|R| < n$ do
4:     $A_i \leftarrow Adj(R_i) \setminus R$
5:     $Sort(A_i)$                            ▷ Sort with ascending degree
6:     $R = R \cup A_i$
7:     $i \leftarrow i + 1$
8: end while
```

The implementation consists of two parts. The first part is the pseudo-peripheral node finding to select the starting node. The second part is the calculation of the permutation.

This implementation uses the CSR format (section 7.2) to store the matrices. Since the matrices are symmetric, the rows and columns can be accessed efficiently using this format.

7.3.1. Pseudo-Peripheral node finding

The selection of a suitable starting node is very important for the RCM algorithm [CM69]. The eccentricity $\epsilon(v)$ is defined as the greatest distance between v and any other node ($\epsilon(v) = \max_{u \in V} d(v,u)$). A node v is pseudo-peripheral, if for each node u with $d(u,v) = \epsilon(v)$ holds $\epsilon(u) = \epsilon(v)$. For the pseudo-peripheral node finding, the heuristic described by Kumfert [Kum00] was implemented. The algorithm implemented here (see algorithm 7.2) is running multiple *breadth first searches* (BFS) to find two nodes with the greatest distance. The start node of the next BFS iteration is one of the nodes with the greatest distance to the start node of the previous BFS execution. Since the number of candidates can be very big, only a small number is selected.

Algorithm 7.2 RCM pseudo-peripheral node finding

```
1: $start \leftarrow 0$
2: $end \leftarrow -1$
3: **repeat**
4:     $start\_dist, candidates \leftarrow \textbf{BFS}(start)$
5:     **for** $Node\ candidate \in candidates$ **do**
6:         $end\_dist, candidates \leftarrow \textbf{BFS}(candidate)$
7:         **if** $end\_dist > start\_dist$ **then**
8:             $start \leftarrow candidate$
9:             $end \leftarrow -1$
10:            **break**
11:        **else**
12:            $end \leftarrow candidate$
13:        **end if**
14:    **end for**
15: **until** $end \neq -1$
```

7.3.2. Permutation

The permutation is created level by level (see Algorithm 7.3). The RCM implementation presented by Rodrigues, Boeres, and Catabriga [RBC17] and Karantasis et al. [Kar+14] also computes the permutation level by level. But they use a prefix sum over the children count to calculate the indices of the parents. These indices are then used to write the children of the individual parents into the permutation array.

At the beginning, the start node is added to the permutation array. Then for all nodes of the current level, the not yet visited children are stored in a temporary array. For these children, a key is generated consisting of the position of the parent in the permutation and the valency of the child node. The children are then sorted using this key. We use bitonic mergesort (Appendix A) to sort the keys. Then, the sorted children are appended to the permutation array. This is repeated until the length of the permutation array is equal to the number of nodes.

Algorithm 7.3 shows a fixed partition of the key, where the upper half of the key is used for the parent position, and the lower half is used for the valency. We used preprocessing to determine the required number of bits for the valency and use the rest for the parent position. The preprocessing goes over all nodes of the graph and finds the largest valency. Then we calculate the most significant set bit of the largest valency to get the number of bits required to store all valencies.

The following task-types were used to implement the RCM algorithm:

- CreateSortData
- CompressSortData
- Delegate
- SortBlock
- SortDelegate
- SortGlobal

The dependencies are defined using phases. Each of the task-types above is in a separate phase, and they are executed in the order they are listed. All tasks are executed with 256 threads.

CreateSortData This task gets an index for the permutation array as input. It reads the parent node from this index and goes over the neighbors of the parent node. For each neighbor that has not been visited, it creates a key. The key consists of the position of the parent in the permutation and the valency of the child node. Then the key is written into the `d_keys` array at the node id of the parent node using an `atomic_min` instruction. This `atomic_min` is used to determine the child node of the parent that occurs first in the permutation array. For all nodes on the same level, these tasks are executed simultaneously.

CompressSortData As described earlier, the keys are stored at specific indices in the `d_keys` array. As a result, this array is sparsely filled, so only a few entries contain valid values. This task goes over the `d_keys` array, and compresses the data. It writes the key into `d_sort_keys` and the array index of the key into `d_sort_values`. Each thread checks 8 entries of `d_keys`, so enough tasks are executed simultaneously to cover the whole array.

Delegate This task is used to enqueue the SortBlock tasks. It has to be executed in a separate phase because the number of required SortBlock tasks is only available after all CompressSortData tasks are finished.

SortBlock Each thread of this task works on 4 elements. This is one of the three tasks responsible to sort the keys and values stored in `d_sort_keys` and `d_sort_values`. Bitonic mergesort is creating bitonic sequences by comparing and swapping elements with increasing stride. Then the bitonic sequences are merged. This task is used when sorting with a stride smaller than $256 \cdot 4$. Otherwise the SortGlobal tasks are used to sort. When the stride is below the block size, we use shuffle instructions and shared memory to exchange elements. The first task of this type enqueues a SortDelegate task.

SortDelegate This task checks whether the stride is below or above the block size of 1024 and enqueues the tasks to continue sorting. SortBlock tasks are enqueued if the stride is below the block size. Otherwise SortGlobal tasks are enqueued. If the stride is above the size of the array then sorting is complete and this task appends the values of `d_sort_values` to the permutation array. Then it enqueues the CreateSortData tasks and SortDelegate task for the next level.

SortGlobal Each thread of this task works on 4 elements. It is used to sort when the stride is larger than the block size. In this case, the elements have to be exchanged using global memory. The first task of this type enqueues a SortDelegate task.

7.3.3. CPU-GPU version

This is a hybrid implementation of the RCM algorithm. At the beginning, the start node is added to the permutation array. Then for all not yet visited children of the current level, a key-value pair is generated. The key consists of the position of the parent in the permutation and the valency of the child node. The value is the node index of the child. Then the key-value pairs

Algorithm 7.3 Our RCM algorithm

```
 1: $permutation[0] \leftarrow start\_node$
 2: $block\_start \leftarrow 0$
 3: $block\_end \leftarrow 1$
 4:
 5: $keys \leftarrow \{\}$
 6: $values \leftarrow \{\}$
 7: while $block\_end < num\_nodes$ do
 8:     for $i \leftarrow block\_start$ to $block\_end$ do in parallel
 9:         $parent \leftarrow permutation[i]$
10:         for $nb : neighbours(parent)$ do in parallel
11:             if $visited[nb] == 0$ then
12:                 $visited[nb] = 1$
13:                 $keys \leftarrow keys \cup (i << 16 + neighbour\_count(nb))$
14:                 $values \leftarrow values \cup nb$
15:                 $count \leftarrow count + 1$
16:             end if
17:         end for
18:     end for
19:     $Sort(keys, values)$
20:     $permutation \leftarrow permutation \cup values$
21:     $block\_start \leftarrow block\_end$
22:     $block\_end \leftarrow block\_end + count$
23:     $count \leftarrow 0$
24: end while
```

are sorted by ascending key. This implementation uses `DeviceRadixSort` [Nvi20c] for sorting. Then the sorted values are appended to the permutation array. This is repeated until the length of the permutation array is equal to the number of nodes. In this implementation, only the sorting is done by the GPU. Everything else is done by the CPU. We use this implementation to compare the performance with our GPU implementation.

7.4. Jacobi method

The Jacobi method is an algorithm to iteratively determine a solution to a system of n linear equations with n unknowns of the form $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A} \in \mathbb{R}^{n \times n}$ and $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$. The matrix $\mathbf{A}$ is sparse in the CSR format (see section 7.2) while $\mathbf{x}$ and $\mathbf{b}$ are dense vectors. The method starts with an initial value $\mathbf{x}^0 = \mathbf{1}$ and is then iterated until it converges. The iterates $\mathbf{x}^{(k)}$ are obtained via

$$\mathbf{x}^{(k)} = \mathbf{D}^{-1}(\mathbf{b} - (\mathbf{A} - \mathbf{D})\mathbf{x}^{(k)}) \tag{7.1}$$

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{i \neq j} a_{ij} x_j^{(k)}\right) \quad \text{for } i = 1, 2, \ldots, n \tag{7.2}$$

Several different versions of this algorithm have been implemented. All versions are using individual dependencies except the *stages* version, which was implemented for phases and individual dependencies. The differences of the individual implementations are described in detail below.

All versions use preprocessing on the host to determine a good distribution of work to individual tasks (see algorithm 7.4). This work distribution ensures a good utilization of the device. The preprocessing goes over all rows of the matrix and accumulates the NNZ of the rows. If this sum exceeds a threshold (in our case $64 \cdot 8$) then the number of summed up rows is appended to the `rowStarts` array. Our implementaion only works for matrices where no single row exceeds the threshold. After that a prefix sum is used on the `rowStarts` array. Then each entry of `rowStarts` contains the start row for a task. The number of tasks needed is equal to the size of the `rowStarts` array.

This preprocessing is done only once per matrix since the work distribution only depends on the matrix and thus does not change during execution. The preprocessing step does not split rows to multiple tasks. So it is not necessary to use atomic operations to write the results into global memory. For matrices with long rows, it would be beneficial to split them to multiple tasks. This would result in a better utilization of the device.

Algorithm 7.4 Preprocessing for the Jacobi method

```
1: $acc \leftarrow 0$
2: $rows \leftarrow 0$
3: $rowStarts \leftarrow \{\}$
4: for $row \leftarrow 0$ to $n$ do
5:     $acc \leftarrow acc + (row\_offsets[row + 1] - row\_offsets[row])$
6:     if $acc < NNZ\_PER\_TASK \wedge rows < ROWS\_PER\_TASK$ then
7:         $rows \leftarrow rows + 1$
8:     else
9:         $rowStarts \leftarrow rowStarts \cup \{rows\}$
10:        $rows \leftarrow 1$
11:        $acc \leftarrow row\_offsets[row + 1] - row\_offsets[row]$
12:    end if
13: end for
14: $rowStarts \leftarrow rowStarts \cup \{rows\}$
15: $rowStarts \leftarrow$ prefixSum$(rowStarts)$
16: $numTasks \leftarrow |rowStarts|$
```

All versions have two exit conditions. The first one is the maximum number of iterations. The second exit condition is the convergence of the vector **x**. The convergence is validated using the residuals $r = ||\mathbf{x}^{(k)} - \mathbf{x}^{(k+1)}||_1$, which are calculated using the norm of the difference of two iterations.

The main difference between the implemented versions is the handling of dependencies. The main part of the algorithm is the same for all versions.

All versions have a task which calculates the iterations of vector **x** (see Algorithm 7.5). Each of these tasks calculates multiple rows of **A**, so the partial SpMV results ($\sum_{i \neq j} a_{ij} x_j$) are stored in shared memory. The threads sum up multiple entries into a local register to reduce the load of shared memory. If they process the next row, the local result is written into shared

Algorithm 7.5 Jacobi method algorithm

```
$start \leftarrow rowStarts[tid]$
$end \leftarrow rowStarts[tid + 1]$

$prev\_row \leftarrow col\_offsets[row\_offsets[start] + tid * elements\_per\_thread]$
for $i \leftarrow 0$ to $elements\_per\_thread$ do
    $col\_idx \leftarrow row\_offsets[start] + tid * elements\_per\_thread + i$
    if $col\_idx \geq col\_offsets[end]$ then
        break
    end if
    $row \leftarrow col\_offsets[col\_idx]$
    $col \leftarrow col\_ids[col\_idx]$
    if $prev\_row \neq row$ then
        $s\_sum[prev\_row - start] \leftarrow sum$
        $prev\_row \leftarrow row$
        $sum \leftarrow 0$
    end if
    $sum \leftarrow sum + A[idx] \cdot x_{old}[row]$
end for
$s\_sum[prev\_row - start] \leftarrow sum$

for $row \leftarrow start$ to $end$ do
    $x[row] \leftarrow \frac{b[row]-sum[row-start]+diag[row]*x_{old}[row]}{diag[row]}$
end for
```

memory. As Equation 7.2 shows, the diagonal element is not included in the sum. But to reduce thread divergence, it is also summed up and subtracted later. If all entries have been summed up, the sum is subtracted from b and divided by the diagonal element and written to the vector x.

Since the matrix $\mathbf{A}$ is stored in CSR format, it would require an additional condition that must be evaluated in each step. So the diagonal elements are written into an array in the preprocessing step. This enables easy and direct access of diagonal elements.

7.4.1. *Stages* version

This implementation uses two task-types which are executed alternatingly. The first task-type (called *Jacobi*) calculates a new approximation $\mathbf{x}^{(k)}$ and is executed with 64 threads per task.

The second task-type (called *CopyAndCheck*) uses the current and last approximation to check for convergence and switches the pointers for the current vector $\mathbf{x}$ and the last vector $\mathbf{x}_{old}$. This task is executed with 256 threads. Each thread sums up every 256th value, i.e., the first thread sums up values with index $0, 256, 512, \ldots$. The threads sum up their values into a register. If all threads are finished with the sum, the individual sums are combined with warp level primitives. Then, an `atomicAdd` is used to combine the results of the warps. Each *CopyAndCheck* task enqueues itself and *numTasks* *Jacobi* tasks.

The alternating execution of the two task-types is resolved in two different ways. The first one uses phases. Each task-type is in a different phase, so the MHive framework schedules them such that they are executed alternatingly. The second way uses individual dependencies. The *CopyAndCheck* task is enqueued with *numTasks* dependencies. Each *Jacobi* task reduces the dependencies of the *CopyAndCheck* task by one. Since *numTasks* *Jacobi* tasks are executed in each iteration, the *CopyAndCheck* task is executed after all *Jacobi* tasks of the current iteration.

In this version, there is no benefit in using individual dependencies, since the same order of execution is defined with phases. But it allows us to compare the overhead of phases and individual dependencies.

This version only needs to store two versions of the vector $\mathbf{x}$. The one that is currently calculated ($\mathbf{x}^{(k+1)}$) and the previous one ($\mathbf{x}^{(k)}$). After each iteration, the pointers to these vectors are swapped.

7.4.2. *Checkpoints* version

This version defines the dependencies on individual values of the vector $\mathbf{x}$. Every value of the new approximation $\mathbf{x}^{(k)}$ depends only on a small number of values of the last approximation $\mathbf{x}^{(k-1)}$. This is because we calculate $\sum_{i \neq j} a_{ij} x_j$, and when a_{ij} is zero, the value of x_j is irrelevant. Since A is sparse, a lot of entries are zero (see Figure 7.4). There are again two task-types, but they are not executed alternatingly.

As before, the *Jacobi* tasks calculate new approximations. Each Jacobi task is executed with 64 threads and has as many dependencies as the sum of nnz entries in the rows it is working on. After the task has calculated the new value of $x_i^{(k+1)}$, it has to reduce the dependencies of the tasks, that depend on this value. We have to find the NNZ entries of column i to get the indices of these tasks. If the matrix is symmetric, we can equivalently go over the row i. If it is not symmetric, we additionally use the CSC format, which allows easy access of column entries. Every few iterations, the convergence is checked by *Checkpoint* tasks. The *Jacobi* tasks enqueue themself except for the iterations where the *Checkpoint* task runs. In these iterations they reduce the number of dependencies of the *Checkpoint* task.

The *Checkpoint* task is executed with 256 threads. It computes the norm $\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|_1$ similar to the before mentioned *CopyAndCheck* tasks using warp level primitives and shared memory. The *Checkpoint* task enqueues itself and the *Jacobi* if $\mathbf{x}$ has not yet converged.

Because each task could be at a different iteration, a large array to store all iterations of $\mathbf{x}$ is needed. Since the maximum number of iterations is given, this array can be allocated to that size. It would also be possible to make

$$\begin{bmatrix} \cdot & 1 & \cdot & \cdot & 2 & 4 & \cdot \\ \cdot & \cdot & \cdot & 3 & \cdot & 1 & \cdot \\ 2 & \cdot & 6 & \cdot & \cdot & \cdot & \cdot \\ \cdot & 3 & \cdot & \cdot & 8 & \cdot & 5 \\ \cdot & \cdot & \cdot & 4 & 1 & 9 & \cdot \\ 6 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 7 & \cdot & \cdot & \cdot \end{bmatrix} \cdot \begin{pmatrix} 7 \\ 8 \\ 4 \\ 1 \\ 9 \\ 5 \\ 3 \end{pmatrix} = \begin{pmatrix} 46 \\ 8 \\ \vdots \\ \\ \\ \\ \\ \end{pmatrix}$$

Figure 7.4.: Example of a sparse matrix-vector product. Only the blue values are needed to calculate the first result.

the array a ring-buffer and limit the number of iterations that are worked on simultaneously. So a new iteration is only started if the ring-buffer is not full.

7.4.3. *Autonomous* version

This version is similar to the *Checkpoints* version, but the convergence is verified differently. Instead of the *Checkpoint* tasks, this version uses global atomic variables to sum up the residuals. Each task sums up the residuals of the rows it works on using `atomiAdd`. The last task, that adds its residual to the global variable also checks if the value is below the threshold. If it is below, a global flag is set to stop the algorithm (clear queues and stop enqueues). So in this implementation, there are only *Jacobi* tasks that are executed with 64 threads per task.

7.4.4. Kernel-by-kernel

This version is used to compare the performance of MHive with a simple kernel-by-kernel implementation. This version does not use any MHive functions. There are two kernels, which are executed alternatingly. The kernels are called `d_jacobi` and `d_check`.

The `d_jacobi` kernel is responsible for calculating one iteration of the jacobi method. The block size of the kernel is 128 threads. The calculation is done similar as described above. Each block works on multiple rows. The preprocessing described above is used to assign the rows to the blocks. The threads sum up multiple values into a local register. If the next row is processed, the local result is written to shared memory. If all values have been summed up, the sum is subtracted from b, divided by the diagonal element and written to the vector x.

The `d_check` kernel calculates the residuals and returns whether they are below the threshold. The block size of this kernel is 256 threads. The residuals are calculated similar to the *Checkpoint* and *CopyAndCheck* tasks using warp level primitives and shared memory.

The host runs a loop which launches the kernels. This loop runs until the maximum number of iterations is reached or until the residuals are below the threshold.

8. Evaluation

In this section, the results of the implemented examples (Cuthill-McKee and Jacobi) are shown. The test system used to run the experiments consisted of an Intel i7-7700K@4.20GHz CPU, 32 GB of host memory and a Nvidia RTX2080Ti GPU. The test system was running Windows 10 and used the Cuda Toolkit 11 as well as MSVC 19.4 to compile the examples.

8.1. Cuthill-McKee bandwidth reduction

In this section, the quality of the reorder as well as the performance of the implemented Cuthill-McKee algorithm is discussed. The reordering quality is measured by the bandwidth of the matrix. The bandwidth b is calculated with

$$b = \max(\mathbf{r} - \mathbf{c}) - \min(\mathbf{r} - \mathbf{c}) + 1 \tag{8.1}$$

where **r** and **c** are the row and column indices of the nnz entries of the matrix. The results were also verified visually by plotting the reordered matrices using the *spy* method of the python *matplotlib* package.

A set of square and symmetric matrices were selected from the University of Florida Sparse Matrix Collection [DH11]. The tested matrices cover different types of problems. The properties of the tested matrices are shown in Table 8.1. The algorithms were performed multiple times for each matrix to calculate the average execution time. To the best of our knowledge, there is no GPU implementation of the RCM algorithm available, so we compared the results of this implementation to the cuSolver RCM implementation [Nvi20b].

Matrix	Dimension	non-zeros	avg. NNZ / row
bcsstk32	44,609	2,014,701	45
bmw3_2	227,362	11,288,630	50
d_pretok	182,730	1,641,672	9
F1	343,791	26,837,113	78
filter3D	106,437	2,707,179	25
gsm_106857	589,446	21,758,924	37
inline_1	503,712	36,816,342	73
mario001	38,434	206,156	5
msdoor	415,863	20,240,935	49
offshore	259,789	4,242,673	16
SiO2	155,331	11,283,503	73
CurlCurl_4	2,380,515	26,515,867	11
dielFilterV2real	1,157,456	48,538,952	42
dielFilterV3real	1,102,824	89,306,020	81
Dubcova3	146,689	3,636,649	25
G3_circuit	1,585,478	7,660,826	5
nlpkkt80	1,062,400	28,704,672	27
com-Youtube	1,134,890	5,975,248	5
thermomech_TC	102,158	711,558	7

Table 8.1.: Input matrices for the RCM algorithm

Matrix		Final Bandwidth		
Name	Bandwidth	cuSolver	CPU-GPU	MHive
bcsstk32	86,061	4,811	5,001	5,003
bmw3_2	438,221	10,907	10,683	10,672
d_pretok	259,835	5,151	5,399	5,900
F1	687,509	20,075	21,011	21,215
filter3D	16,553	7,633	6,525	6,525
gsm_106857	1,177,489	36,995	36,767	36,510
inline_1	1,004,807	12,005	12,005	12,005
mario001	75,371	735	923	726
msdoor	582,227	16,841	18,000	18,000
offshore	475,477	49,715	50,445	50,501
SiO2	110,137	41,587	40,419	40,477
CurlCurl_4	87,363	67,959	64,171	64,171
dielFilterV2real	1,896,065	39,191	36,091	36,013
dielFilterV3real	2,072,951	51,125	50,836	51,006
Dubcova3	292,713	4,561	4,556	4,556
G3_circuit	1,894,257	10,193	10,199	10,241
nlpkkt80	1,100,963	77,433	77,433	77,433
com-Youtube	2,266,265	1,210,385	1,211,865	1,212,122
thermomech_TC	204,277	525	507	504

Table 8.2.: Bandwidth comparison after reordering

8.1.1. Reorder quality

The bandwidth of the tested matrices before and after applying the permutation are shown in Table 8.2. As the table shows, there are small differences between our bandwidth and the values of cuSolver. The reason for this is that nodes with the same parent and the same degree may be interchanged.

The sparsity pattern of the matrices before and after the reordering is visualized in Figure 8.1. The first row of each group shows the sparsity of the input matrices. The rows below show the sparsity after application of the permutation derived by our RCM implementation and the cuSolver RCM implementation. The plots in Figure 8.1 confirm that the output of RCM algorithm is a band matrix.

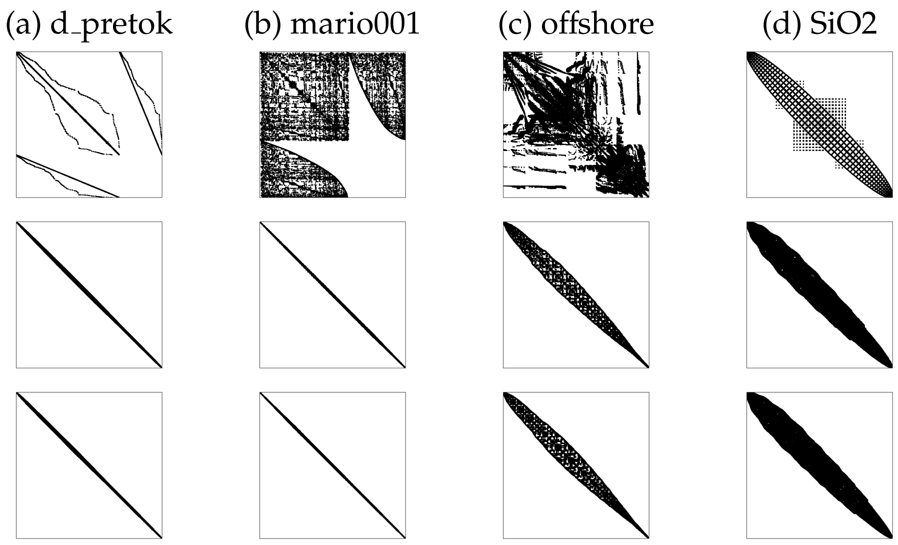

Figure 8.1.: Sparsity pattern of the matrices yielded by the RCM algorithms. From top to bottom: unordered, MHive RCM, cuSolver RCM.

8.1.2. Runtime

The preprocessing times are less than 0.1 *ms* and therefore not directly shown in the tables. The execution times of the implementations are shown in Table 8.3. As the results show, the PDC model performs better than the TSK model. This is because PDC runs the scheduler on the device, so no additional overhead is produced by copying data between the host and the device. It is also clear that our algorithm performs better than cuSolver since cuSolver runs on the host only. The MK model performs far worse than TSK and PDC because this implementation uses too many registers to provide good utilization. The register usage of this implementation only allows one resident block per SM.

Comparing our implementation results to cuSolver we see a speedup of up to 15×. For very small matrices, cuSolver performs better, because there is not enough parallelism to draw an advantage from the GPU.

The peak memory consumption (in bytes) can be calculated with the following equation, which depends on the number of nodes D and the NNZ entries of the matrix.

$$peak\_memory = 48\ MB + 8 \cdot NNZ + 28 \cdot D\ B \tag{8.2}$$

The CPU-GPU version performs quite similar to PDC. The sorting of the keys is a demanding step of the implementation. The performance difference grows larger for matrices where we have to sort very often. This means that our sorting algorithm is not as well optimized as `RadixSort` from cub. The execution times of the different tasks are visualized in Figure 8.2. These figures show that sorting is a bottleneck. Especially if there are only a few hundred elements to sort, then the sorting is executed by only a few SMs. This does not yield a good utilization of the device.

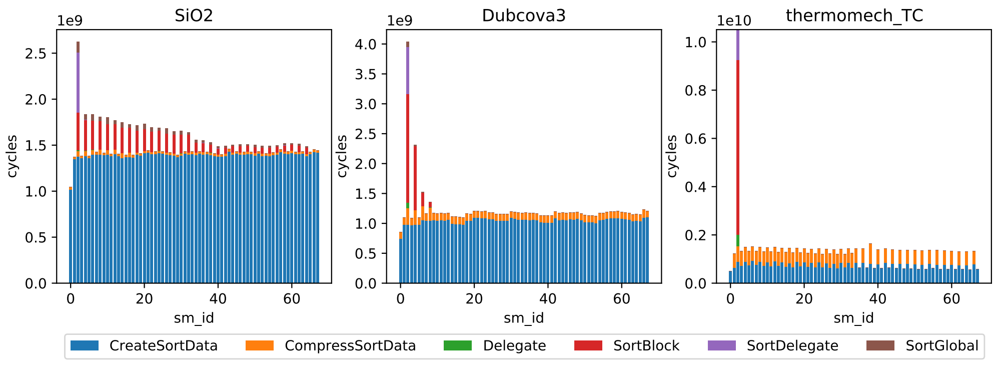

Figure 8.2.: Execution times of the different tasks of the RCM algorithm for three matrices.

Matrix	cuSolver	CPU-GPU		TSK		PDC	
		RCM	total	RCM	total	RCM	total
bcsstk32	140.02	12.00	19.01	87.12	94.13	16.44	23.45
bmw3_2	978.20	72.30	117.36	252.90	297.96	43.11	88.17
d_pretok	133.30	49.40	68.14	251.16	269.90	44.46	63.20
F1	2,330.20	89.00	188.48	329.71	429.19	50.80	150.28
filter3D	206.10	33.30	59.97	136.66	163.33	23.15	49.82
gsm_106857	2,202.70	161.20	402.87	430.07	671.74	70.17	311.84
inline_1	2,943.60	150.30	293.87	632.84	776.41	97.95	241.52
mario001	15.80	21.30	23.84	240.35	242.89	39.10	41.64
msdoor	1,742.20	115.10	240.02	581.05	705.97	72.30	197.22
offshore	515.00	55.50	147.50	212.53	304.53	29.86	121.86
SiO2	1,060.80	41.00	114.72	126.22	199.94	17.25	90.97
CurlCurl_4	2,223.80	322.80	703.05	1,298.84	1,679.09	234.69	614.94
dielFilterV2real	3,843.50	171.80	414.90	815.09	1,058.19	126.16	369.26
dielFilterV3real	7,108.30	188.70	524.12	619.48	954.90	107.00	442.42
Dubcova3	297.60	30.10	49.16	194.67	213.73	35.58	54.64
G3_circuit	864.40	347.20	480.80	1,820.98	1,954.58	308.30	441.90
nlpkkt80	2,534.60	193.40	385.34	627.47	819.41	98.72	290.66
com-Youtube	856.90	95.10	423.87	140.82	469.59	40.08	368.85
thermomech_TC	73.90	98.20	110.47	460.17	472.44	104.24	116.51

Table 8.3.: Execution time of different execution models using phases. All timings in *ms*. For our implementations, the RCM times as well as total (RCM + pseudo-peripheral node finding) times are shown. The cuSolver implementation does not allow to measure those times separately.

8.2. Jacobi method

This section presents the performance numbers of the implemented Jacobi method (which solves the system $\mathbf{Ax} = \mathbf{b}$). A set of square and symmetric matrices were selected from the University of Florida Sparse Matrix Collection [DH11]. The Matrices $A3$ to $A6$ are diagonally dominant matrices generated from random values. The tested matrices cover different types of problems. The properties of the tested matrices are shown in table 8.4. The corresponding vectors $\mathbf{b}$ were generated from random values such that we have solvable linear systems.

The following criteria was used to check, whether the Jacobi method converges for a given matrix:

$$\rho(\mathbf{D}^{-1}(\mathbf{A} - \mathbf{D})) < 1$$

where $\rho(\cdot)$ is the spectral radius and $\mathbf{D}$ is a matrix containing the diagonal elements of $\mathbf{A}$.

As described in section 7.4, different versions of the algorithm were implemented. All versions were tested using the different execution models. The result of the implementations were compared to a reference implementation in python using *SciPy*. The results were considered correct if the norm of the difference was less than 10^{-5}:

$$|\mathbf{x} - \mathbf{x}_{ref}|_1 < 10^{-5}$$

The preprocessing is done on the host and is thus quite slow. The preprocessing takes about 180 *ms* for all tested matrices.

The execution times for the different versions and execution models can be found in tables 8.5 to 8.7. As these tables show, the best performance is achieved by the MK and PDC execution models. Smaller matrices run better with the MK model, while larger matrices run better with the PDC model. The handoff threshold is at an nnz value of approximately $500,000$. The reason for this is that there are no kernel launches during the runtime when the MK execution model is used. The worst performance is achieved by TSK, which has the additional overhead of copying the queue fill-levels to the host and launching the kernels from the host.

Matrix	Dimension	non-zeros	avg. NNZ / row
Chem97ZtZ	2,541	7,361	3
A3	10,000	109,976	11
A4	10,000	209,876	21
A6	32,000	431,942	14
A5	20,000	539,824	27
finan512	74,752	596,992	8
G2_circuit	150,102	726,674	5
Andrews	60,000	760,154	13
parabolic_fem	525,825	3,674,625	7
G3_circuit	1,585,478	7,660,826	5
thermal2	1,228,045	8,580,313	7

Table 8.4.: Input matrices for the Jacobi method.

Due to the fine-grained dependencies, the checkpoints and autonomous versions are able to utilize more parallelism. This is especially noticeable for matrices with irregular workloads. The performance difference between the versions with individual dependencies and phases is larger for matrices with irregular row lengths. Each row is only worked on by one task, since we do not split up rows to several tasks. This results in a bad load balancing for matrices with irregular row lengths.

The peak memory consumption (in bytes) can be calculated with the following equation, which depends on the number of nodes D and the NNZ entries of the matrix.

$$peak\_memory = 64\ MB + 8 \cdot NNZ + 1040 \cdot D\ B \tag{8.3}$$

Matrix	stages (p)	stages (id)	cp(id)	aut(id)	kbk(id)
Chem97ZtZ	218.50	94.85	55.57	52.99	29.51
A3	218.98	94.11	55.97	51.14	35.75
A4	219.91	93.35	57.39	53.82	38.19
A6	232.87	106.07	76.29	72.87	54.96
A5	236.98	106.90	80.50	80.62	57.02
finan512	248.81	118.82	86.17	86.97	86.71
G2_circuit	306.45	177.30	121.10	120.39	117.11
Andrews	248.06	120.05	95.70	92.66	83.63
parabolic_fem	695.28	559.85	361.29	364.23	451.44
G3_circuit	1,691.58	1,555.78	923.12	908.72	1,220.22
thermal2	1,367.44	1,227.62	818.56	775.18	979.47

Table 8.5.: Execution time of different versions of the Jacobi method for the TSK execution model. All timings in *ms*. p=phases, id = individual dependencies, cp=checkpoints, aut=autonomous, kbk = kernel-by-kernel

Matrix	stages (id)	cp(id)	aut(id)	kbk(id)
Chem97ZtZ	8.13	9.79	15.80	29.51
A3	12.66	14.72	19.34	35.75
A4	16.09	19.63	24.78	38.19
A6	32.75	32.89	33.47	54.96
A5	32.19	36.47	36.66	57.02
finan512	59.98	58.84	50.72	86.71
G2_circuit	122.77	109.45	75.48	117.11
Andrews	63.03	56.83	49.62	83.63
parabolic_fem	496.88	445.17	316.30	451.44
G3_circuit	1,469.36	1,255.75	879.23	1,220.22
thermal2	1,210.80	1,039.89	753.86	979.47

Table 8.6.: Execution time of different versions of the Jacobi method for the MK execution model. All timings in *ms*. p=phases, id = individual dependencies, cp=checkpoints, aut=autonomous, kbk = kernel-by-kernel

Matrix	stages (p)	stages (id)	cp(id)	aut(id)	kbk(id)
Chem97ZtZ	39.59	48.53	33.41	31.80	29.51
A3	43.80	52.27	38.45	36.74	35.75
A4	35.80	41.82	41.22	41.03	38.19
A6	55.27	61.04	57.07	56.64	54.96
A5	61.73	66.48	61.97	61.55	57.02
finan512	80.57	84.63	47.50	44.98	86.71
G2_circuit	148.46	157.64	72.18	72.02	117.11
Andrews	89.12	93.95	76.40	76.25	83.63
parabolic_fem	518.24	502.92	293.25	299.47	451.44
G3_circuit	1,451.40	1,379.06	792.84	818.09	1,220.22
thermal2	1,179.87	1,106.22	692.55	717.00	979.47

Table 8.7.: Execution time of different versions of the Jacobi method for the PDC execution model. All timings in *ms*. p=phases, id = individual dependencies, cp=checkpoints, aut=autonomous, kbk = kernel-by-kernel

9. Conclusion

In this thesis we presented a dependency resolution for a task scheduling framework. This dependency resolution allows for two different approaches to define dependencies. Phases is used to define coarse dependencies on the task-types. These phases are then executed in a pre-defined order. Individual dependencies allows us to define fine-grained dependencies on individual tasks.

We have shown that both approaches to define dependencies are necessary, because depending on the algorithm, only one of them may be usable. Some algorithms require phases to be implemented, while others only work with individual dependencies.

We have also implemented two algorithms with our framework. The first algorithm is RCM, which allows to reorder a matrix into a band matrix. The second algorithm is the Jacobi method, which is used to determine the solution to a system of linear equations. These algorithms require different dependencies, so it would not have been possible to implement both of them if we would not have both ways to define dependencies.

For irregular workloads, better performance is achieved when using individual dependencies. This was confirmed in our Jacobi implementation, where matrices with irregular row length show better performance when using individual dependencies compared to phases.

10. Future Work

Our implementation consist of two ways to define dependencies. These two ways already cover a wide spectrum of use-cases. But there is an extension to individual dependencies, which could increase performance and usability. Instead of having one dependency for one task, we could use one dependency for a group of tasks. These so-called grouped dependencies are coarser than individual dependencies but still more fine-grained than phases. By using only a single atomic counter for several tasks, this would reduce the overhead.

The MK scheduler uses a lot of registers, which results in bad utilization. With this scheduler, there are usually only one or two resident blocks per SM. As a future improvement, the number of required registers for the MK scheduler could be reduced.

Appendix

Appendix A.

Bitonic Mergesort

Bitonic mergesort [Bat68] is a parallel sorting algorithm. The version implemented here is sorting *keys* with additional *values*. The input data consists of arrays of size *SIZE* where each thread has its own *keys* and *values* array. Warp level intrinsics are used to communicate between threads, so the algorithm has to be called with a complete warp (i.e. 32 threads).

A bitonic sequence is a combination of two monotonic sequences, one ascending and the other one descending. Bitonic mergesort creates bitonic sequences of increasing size. Since the size of the bitonic sequence is doubled after each step, only k steps are required to sort 2^k numbers. The bitonic sequences are created by alternatingly sorting ascending and descending with different strides (as shown in Figure A.1).

The function swap (Algorithm A.1) is swapping the given keys and values depending on the sort direction and the values itself. It gets the reference to the values and keys as parameters so that it can change them. For the complete algorithm see A.2. A graphical representation of the algorithm is given in Figure A.1.

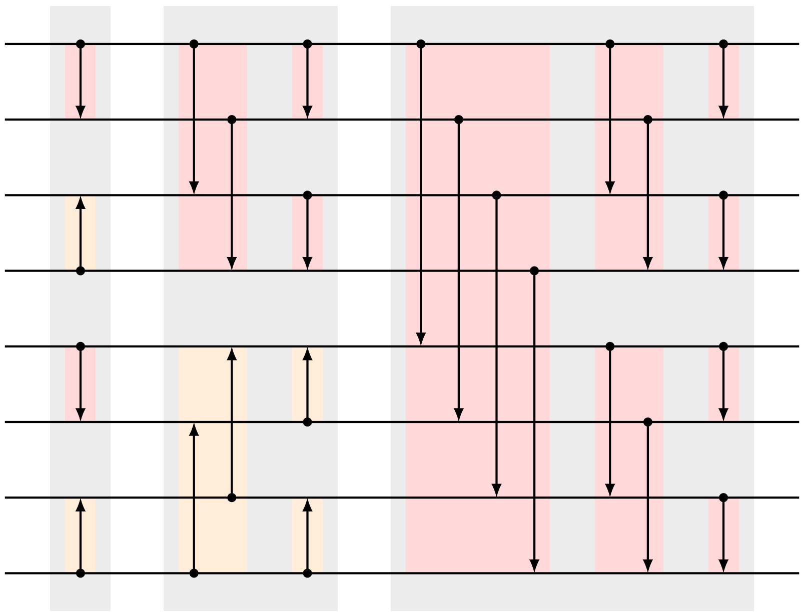

Figure A.1.: Graphical representation of Bitonic mergesort on an input of length 8. The direction of the arrows specifies where the larger element should be after the swap.

Algorithm A.1 Bitonic swap

```
 1: function SWAP($up, key1, key2, value1, value2$)
 2:     if $(key1 > key2 \wedge up) \vee (key1 < key2 \wedge \neg up)$ then
 3:         $temp \leftarrow value2$
 4:         $value2 \leftarrow value1$
 5:         $value1 \leftarrow temp$
 6:
 7:         $temp \leftarrow key2$
 8:         $key2 \leftarrow key1$
 9:         $key1 \leftarrow temp$
10:     end if
11: end function
```

Algorithm A.2 Bitonic merge sort

```
1: $i \leftarrow 2$
2: while $i \leq N$ do
3:     $k \leftarrow \frac{i}{2}$
4:     while $k \geq 1$ do
5:         if $k \geq SIZE$ then                                    ▷ Swap between threads
6:             for $idx \leftarrow 0$ to $SIZE$ do
7:                 $data\_id \leftarrow tid \cdot SIZE + idx$
8:                 $other\_key \leftarrow$ __shfl_xor$(flag, keys[idx], \frac{k}{SIZE})$
9:                 $other\_value \leftarrow$ __shfl_xor$(flag, values[idx], \frac{k}{SIZE})$
10:                $up \leftarrow \neg((data\_id\&i) \oplus (data\_id\&k))$
11:                swap$(up, keys[idx], other\_key, values[idx], other\_value)$
12:            end for
13:        else                                                     ▷ Swap within threads
14:            for $idx \leftarrow 0$ to $SIZE$ do
15:                if $idx\&k = 0$ then
16:                    $data\_id \leftarrow tid \cdot SIZE + idx$
17:                    $up \leftarrow \neg((data\_id\&i) \oplus (data\_id\&k))$
18:                    swap$(up, keys[idx], keys[idx + k], values[idx], values[idx + k])$
19:                end if
20:            end for
21:        end if
22:        $k \leftarrow \frac{k}{2}$
23:    end while
24:    $i \leftarrow 2i$
25: end while
```

Bibliography

[Bat68] K. E. Batcher. “Sorting networks and their applications.” In: Association for Computing Machinery (ACM), 1968, p. 307 (cit. on p. 54).

[Bel+14] Mehmet E Belviranli et al. “A paradigm shift in GP-GPU computing: Task based execution of applications with dynamic data dependencies.” In: *DIDC 2014 - Proceedings of the 2014 ACM International Workshop on Data Intensive Distributed Computing, Co-located with HPDC 2014*. 2014, pp. 29–33 (cit. on pp. 7, 8).

[Cha+13] Sanjay Chatterjee et al. “Dynamic task parallelism with a GPU work-stealing runtime system.” In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 7146 LNCS. 2013, pp. 203–217 (cit. on p. 6).

[CM69] E. Cuthill and J. McKee. “Reducing the bandwidth of sparse symmetric matrices.” In: *Proceedings of the 1969 24th National Conference, ACM 1969*. 1969 (cit. on pp. 28, 29).

[CT08] Daniel Cederman and Philippas Tsigas. “On dynamic load balancing on graphics processors.” In: *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*. 2008, pp. 57–64 (cit. on p. 6).

[DH11] Timothy A. Davis and Yifan Hu. “The University of Florida Sparse Matrix Collection.” In: *ACM Transactions on Mathematical Software* 38.1 (Nov. 2011) (cit. on pp. 41, 47).

[Geo71] J. A. George. “Computer implementation of the finite element method.” In: 1971 (cit. on p. 29).

[Gra19] Alan Gray. *Getting Started with CUDA Graphs*. Sept. 2019. URL: https://devblogs.nvidia.com/cuda-graphs/ (visited on 03/26/2020) (cit. on p. 1).

[Kar+14] Konstantinos I. Karantasis et al. "Parallelization of Reordering Algorithms for Bandwidth and Wavefront Reduction." In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. 2014 (cit. on p. 30).

[Kum00] Gary Karl Kumfert. "An Object-Oriented Algorithmic Laboratory for Ordering Sparse Matrices." In: *Computer Science Theses & Dissertations* (Apr. 2000) (cit. on p. 29).

[Nvi20a] Nvidia. *CUDA Toolkit Documentation*. 2020. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 03/26/2020) (cit. on pp. 3–5).

[Nvi20b] Nvidia. *cuSolver Documentation*. 2020. URL: https://docs.nvidia.com/cuda/cusolver (cit. on p. 41).

[Nvi20c] Nvidia. *Nvidia CUB library*. 2020. URL: https://nvlabs.github.io/cub/ (visited on 03/26/2020) (cit. on p. 34).

[RBC17] Thiago Nascimento Rodrigues, Maria Claudia Silva Boeres, and Lucia Catabriga. "A non-speculative parallelization of reverse cuthill-McKee algorithm for sparse matrices reordering." In: *Proceedings of the 2017 Federated Conference on Computer Science and Information Systems, FedCSIS 2017*. Institute of Electrical and Electronics Engineers Inc., Nov. 2017, pp. 527–536 (cit. on p. 30).

[TLO12] Stanley Tzeng, Brandon Lloyd, and John D. Owens. "A GPU task-parallel model with dependency resolution." In: *Computer* 45.8 (2012), pp. 34–41 (cit. on p. 8).