Accelerating bioinformatics applications on CUDA-enabled multi-GPU systems

Dissertation
for the attainment of the degree
“Doctor of Natural Sciences”
at the Department of Physics, Mathematics, and Computer Science
of Johannes Gutenberg University
in Mainz

Robin Kobus

born in Wiesbaden-Dotzheim, Germany.
Mainz, January 23, 2023

1. Reviewer	Prof. Dr. Bertil Schmidt
2. Reviewer	Prof. Dr. Andreas Hildebrandt
Date of oral examination	October 16, 2023

D77

Abstract

A wide range of bioinformatics applications have to deal with a continuously growing amount of data generated by high-throughput sequencing techniques. Exclusively CPU-based workstations fail to keep up with the task. Instead of employing dozens of CPU cluster nodes to increase the computational power, massively parallel accelerators like modern CUDA-enabled GPUs can be used to achieve higher throughput and reduce execution times. However, memory capacity of such devices is often limited. Efficient parallelization and data distribution are essential to accelerate performance critical components of bionformatics pipelines like read classification and read mapping.

In this thesis we analyze and optimize tasks common to many GPU-based applications in the context of bioinformatics. We study sequence processing, construction and querying of k-mer-based hash tables, segmented sort as well as multi-GPU communication. With these methods we accelerate suffix array construction and metagenomic read classification on CUDA-enabled GPUs by overcoming the aforementioned challenges. By leveraging multiple GPUs, we extend the limited memory available from a single GPU to allow for the construction of larger indices. Our communication library, called *Gossip*, introduces optimized scatter, gather and all-to-all patterns for multi-GPU systems. Gossip's all-to-all communication pattern is successfully applied to suffix array construction, accelerating it to run in 3.44 s for a full-length human genome on an 8-GPU server, which is faster than previously reported 4.8 seconds achieved by employing 1600 cores on 100 nodes on a CPU-based HPC cluster.

Furthermore, we introduce MetaCache-GPU – an ultra-fast metagenomic short read classifier specifically tailored to fit the characteristics of CUDA-enabled accelerators. Our approach employs a novel hash table variant featuring efficient minhash fingerprinting of reads for locality-sensitive hashing and their rapid insertion using warp-aggregated operations. Our performance evaluation shows that MetaCache-GPU is able to build large reference databases in a matter of seconds, enabling instantaneous operability, while popular CPU-based tools such as Kraken2 require over an hour for index construction on the same data. In the light of an ever-growing number of reference genomes, MetaCache-GPU is the first metagenomic classifier

that makes analysis pipelines with on-demand composition of large-scale reference genome sets practical.

Although many sub-problems in this thesis are optimized in a specific application context, they also apply to other bioinformatics problems like k-mer counting, sequence alignment and assembly, which would benefit from GPU acceleration. In addition to the insights from this work, we make our source code publicly available to allow for easier adaptation of our methods to related problems.

Contents

Introduction

1

1.1 Motivation and Problem Statement

Due to the inexorable progress of *next generation sequencing* (NGS) technologies, the volume of data generated in the life sciences has been steadily increasing and genomics has been considered Big Data for years [104]. Critical to this success was the dramatic cost reduction of high-throughput sequencing technologies. Nowadays, a single (human) genome can be sequenced for low cost[1], which enables large-scale sequencing projects like sequencing of the whole population of Iceland [31], the Earth BioGenome Project [60], metagenomics microbiome sequencing studies [55], and world-wide SARS-CoV-2 sequencing efforts [6].

A wide range of bioinformatics applications rely on data produced by high-throughput sequencers. These machines are able to generate massive amounts of short DNA strings (called reads) in a single run, which can be used in de-novo sequencing, re-sequencing, metagenomics, transcriptomics, and epigenetics pipelines. However, the analysis of these large datasets remains computationally challenging. The initial stages of many NGS pipelines often consist of read classification and read mapping which makes these tasks performance critical to those applications.

Read classification is the task of finding the original organism of each read in a dataset and assigning the corresponding taxonomic label (species, genus, ...), usually by investigating a set of reference genomes and determining the best match. Read mapping additionally requires to find the best mapping location(s) inside one or multiple genomes for each read. Both problems boil down to finding the occurrences of pattern strings from a read in a large body of text consisting of a collection of reference genomes. Although each read may originate from an organism whose genome is included in the reference set, sequencing errors can occur, leading to missing, surplus and/or substituted bases in the read. Additionally, organisms mutate changing their genome, hence we might be interested in finding the most closely related genome in the reference set instead of exactly matching each base of the reads.

[1] Currently around US$1,000 per human genome (http://www.genome.gov/sequencingcosts).

Read datasets may contain many millions of reads which need to be queried against the reference database. For efficient processing it is beneficial to construct an index data structure to accelerate the database queries. Two popular index structures covered in this thesis are suffix arrays and k-mer based hash tables, where k-mers are k-length sub-strings used for the lookup. A suffix array [75] is a list of indices denoting the starting positions of all suffixes of a text, obtained by sorting the suffixes lexicographically. Suffix arrays have been studied intensively in the context of bioinformatics [102] and find their application in a wide range of tasks such as pairwise sequence alignment [47, 113, 120], read mapping [108], read error correction [39, 99], genome assembly [29, 34], k-mer counting [56] and sequence clustering [33]. Alternatively, the reference genomes can be stored and indexed as sets of k-mers [77]. K-mer based hash tables are applied in the contexts of error correction [43], k-mer counting [78, 76] and read classification [81, 117, 116, 87], among others. Distributed k-mer hash tables have also been studied on CPU clusters for long-read to long-read alignment [18], k-mer counting [88] and de-novo assembly [27].

Although continuous progress has been made, runtimes on exclusively CPU-based workstations remain high, while distributed algorithms may require dozens of cluster nodes to reduce the execution time to reasonable levels. Efficient parallelization is key but imposes additional challenges due to variable sequence lengths and query result sizes, as well as the need for large concurrent and distributed data structures [94]. Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs) can sometimes mitigate these obstacles with greater compute capabilities and memory throughput, but speedups on these accelerators are often limited [38, 37]. Algorithmic design and implementations of bioinformatics applications remain challenging tasks and struggle to keep up with the continuously growing amount of data generated by high-throughput sequencing techniques.

In this thesis we analyse and optimize tasks common to many GPU-based applications in the context of bioinformatics. We study sequence processing, construction and querying of k-mer-based hash tables, segmented sort and multi-GPU communication. With these methods we accelerate suffix array construction and metagenomic read classification on CUDA-enabled GPUs by overcoming the aforementioned challenges. By leveraging multiple GPUs we extend the limited memory available from a single GPU to allow for the construction of larger indices. Furthermore, we show that fast index construction permits on-the-fly querying of databases, enabling novel pipelines that are orders of magnitudes faster than established CPU-based applications. Although these sub-problems are optimized in the context of k-mer-based metagenomic read classification, they also apply to other bioinformatics problems like k-mer counting, sequence alignment and assembly, which would benefit from GPU acceleration. Adapting the insights from this work to other problems is left as future work.

1.2 Publications

Our peer-reviewed multi-GPU communication library Gossip [50] addresses the efficient scattering and gathering of data across multiple GPUs as well as all-to-all communication. The library is publicly available at https://github.com/Funatiq/gossip. Gossip's communication patterns are used in suffix array construction [9] to great success.

In *A big data approach to metagenomics for all-food-sequencing* [48] we showed how AFS-MetaCache, an enhanced version of of the metagenomic read classifier MetaCache [81], can be used for whole genome shotgun sequencing-based biosurveillance applications such as food testing. Our GPU-accelerated version of (AFS-)MetaCache was published as MetaCache-GPU [51] and is now included within MetaCache since version 2.0.0. It is inter-operable with the CPU version of MetaCache and allows to query GPU-built databases on the CPU and vice versa. MetaCache and MetaCache-GPU are publicly available at https://github.com/muellan/metacache.

1.3 Thesis Structure

After giving an overview of related similarity search problems in bioinformatics in Chapter 2 and general background information on GPU computing in Chapter 3, the thesis is organized in three parts.

Part I introduces our Gossip library for multi-GPU communication. In Chapter 4 we explain Gossip's design and implementation and evaluate its performance in various experiments. Gossip's application in suffix array construction is showcased in Chapter 5.

Part II revolves around GPU-accelerated metagenomic classification. First, we give an overview of MetaCache and its CPU pipeline in Chapter 6. Chapter 7 shows how MetaCache is applied in All-Food-Sequencing. Then Chapter 8 examines in detail the methods used in our GPU implementation. In Chapter 9 we evaluate the performance of the entire GPU-accelerated MetaCache pipeline. Chapter 10 concludes this part.

Finally, Part III takes a look at future work building on our contributions and concludes the thesis.

2 Related Work

A large number of applications in bioinformatics are tasked with finding similarities between sequences in huge collections of NGS reads and/or reference genomes. For example researchers are interested in pairwise read alignments for error correction and de-novo genome assembly, mapping reads to a specific genome or finding the best matching genome for each read in a sample. Due to genomic variation and sequencing errors exact matches of a complete read to a reference sub-sequence is unlikely. Thus, partial and inexact matches must be considered. However, the corresponding measures can be compute-heavy; e.g., calculating the optimal semi-global alignment score with commonly used dynamic programming algorithms between two sequences exhibits a time complexity proportional to the product of the sequences' lengths. Multiplied by the number of all combinations of sequence pairings, runtimes become prohibitively long. Instead of such a brute-force approach, recent methods try finding shorter sub-string matches via various specialized index structures to identify promising candidates for longer matches. To be able to effectively handle the massive amounts of data in a timely manner, efficient parallelization is essential and concurrent and distributed data structures play an important role. Table 2.1 provides an overview of example applications for various bioinformatics problems which we examine in more detail in the following.

In the NGS context, sub-strings of a fixed length k are usually called k-mers. Accordingly, from a (genomic) sequence of length n we can create a list of $n-k+1$ k-mers, each beginning at a distinct position of the sequence. k-mer counting is often performed as a first step to gain insights on a data set of sequences and is used to filter out erroneous or abundant k-mers in order to improve subsequent processing. Generating a histogram of k-mers relies on index data structures like hash tables for fast insertion and lookup. Earlier approaches construct suffix arrays (like Tallymere [56]) or are sorting based (e.g. Meryl, the k-mer counter included in the Celera assembler [80]) which both are computationally expensive and require huge amounts of memory. More recent programs like Jellyfish [76] employ a multi-threaded, lock-free hash table to efficiently utilize modern multi-core CPUs. Others combine the k-mer index with a Bloom filter [78] or even distribute their hash tables on a CPU cluster [88] to increase performance.

Tab. 2.1.: Example NGS read processing applications.

NGS Problem	Application	Technique[1]	Data Structure[2]
k-mer counting	Tallymere [56]	indexing	enhanced SA
	Meryl [80]	sorting, MT	HT
	Jellyfish [76]	MT, lock-free	HT
	BFCounter [78]	unique k-mers	BF + HT
	Pan et al. [88]	distributed, MT, SIMD	HT
error correction	HiTEC [39]	single thread	SA
	SHREC [99]	MT	SA
	Musket [73]	k-mer spectrum, MT	BF + HT
clustering	KABOOM [33]	k-mer similarity	SA
pairwise sequence alignment	LAST [47]	seed+extend	SA
	RAPsearch [120]	seed+extend	SA
	RAPsearch2 [123]	seed+extend, MT	HT
	BELLA[32]	seed+extend, MT	HT
	diBELLA [18]	distributed BELLA	HT
de-novo genome assembly	Readjoiner [29]	single thread	SA
	Edena [34]	MT	SA
	MetaHipMer [27]	distributed, MT	HT
	SGA [103]	distributed, MT	FM-index
read mapping	Bowtie2 [58]	seed+extend, MT, SIMD	FM-index
	HPG Aligner [108]	seed+extend, MT, SIMD	SA + index
metagenomic classification	CLARK [87]	unique k-mers, MT	HT
	Kraken [117]	LCA, MT	HT
	Kraken 2 [116]	minimizers, LCA, MT	HT
	MetaCache [81]	minhashing, LCA, MT	HT

[1] MT = multi-threaded, SIMD = using vectorization, LCA = lowest common ancestor
[2] SA = suffix array, HT = hash table, BF = Bloom filter

Because genomic reads may contain sequencing errors, error correction is another important pre-processing step. Here again, suffix arrays can be applied [39, 99] to find similar sequences. Alternatively, reads can be corrected using the k-mer spectrum, which is obtained from the the k-mer counts by filtering potentially erroneous k-mers with low occurrence. Musket [73] for example uses this approach. It is optimized for fast multi-threaded execution by combining a Bloom filter and a hash table.

Suffix arrays are also employed for similarity searches among all reads in sequence clustering [33] and pairwise sequence alignment [47], but are often limited to single thread execution. RAPsearch [120] also uses a suffix arrays to find similar reads, but for RAPsearch2 [123] the authors switched to a hash table to reduce the memory footprint and to increase performance. Additionally, this allowed them to utilize multiple threads for even greater speedup. For long-read to long-read alignment BELLA [32] makes use of multi-threaded hash tables, and diBELLA [18] extends this approach to distributed memory systems and achieves good scalabilty by taking efficient parallelism, communication and memory usage into account.

In de-novo genome assembly the goal is to construct the genome of an organism from scratch using only the sequencing reads from a data set. Here, suffix arrays [29, 34] and distributed k-mer hash tables [27] are popular as well to find similar reads which are then combined to form larger sequences. Apart from that, many other tools like SGA (String Graph Assembler) [103] are based on the FM-index derived from the compressed Burrows-Wheeler transform. SGA implements distributed FM-index construction and is optimized for low-end computing cluster due to its low memory requirements and low communication overhead.

In contrast to read-to-read matching, similarity searches between reads and reference genomes are required to generate read mappings or alignments, where the task is to find the region in a genome most similar to each read in a data set. Because these genomes can be millions to billions of base pairs in length, most programs rely on a seed-and-extend approach: first find a short, exactly or inexactly matching sub-sequence between read and genome using an index structure, which is then verified comparing the whole read to a section of the genome. Fortunately, individual read alignments are independent of each other, so the problem can be processed by trivially parallelizing different read alignments of the read data set. Read aligners such as Bowtie2 [58] and HPG Aligner [108] exploit this fact by employing multiple threads and SIMD instructions to accelerate alignments. Bowtie2 utilized an FM-index and a dynamic programming approach, while HPG Aligner relies on an 18-mer index structure for faster lookup into a suffix array, making it especially suitable for longer reads.

In metagenomic read classification one is trying to assign each read to the best matching genome in a large collection of reference genomes. Such a reference database may for example include tens of thousands of bacterial genomes of roughly one million base pairs each. Hence, instead of using full text indices, sophisticated sampling techniques are common in this field. CLARK [87] for example stores only unique k-mers in a lookup table, i.e., such k-mers that occur only in a single species (or other taxonomical level) in the reference set, while Kraken [117] stores the lowest common ancestor for each k-mer instead of mapping k-mers to a specific genome. Kraken 2 [116] combines this approach with using minimizers [96], which are the lexicographically smallest k-mers in a longer sequence. Similar sequences are likely sharing the same minimizer, hence, using minimizers for lookup instead of all k-mers is an efficient sampling technique which reduces the size of the index structure. Another popular technique is called *minhashing* [7], which uses multiple hash functions to select k-mers from a sequence to form a sketch. MetaCache [81] combines minhashing with a windowing approach which divides the reference genomes into windows of roughly the same lengths of the reads, in order to perform sketch comparisons using the Jaccard index. In contrast to CLARK and Kraken(2), this also enables MetaCache to map reads to window positions in the genomes instead of only deciding on which genome matches best. All of these classification methods require to build the index structure before being able to classify the reads, which takes significantly more time than classification and is often not parallalized or does not scale well with multiple threads. Fortunately, the index structure, once constructed, can be reused for additional classification runs, however, it needs to be rebuild each time the set of reference genomes changes or a different reference set is desired. Classification on the other hand is trivially parallelizable because reads can be processed independently.

To achieve high throughput and scalability the aforementioned bioinformatics applications require the use of parallel algorithms to exploit the hardware resources of modern architectures [97]. The applied approaches can be classified in fine-grained and coarse-grained parallelization. The simplest way to achieve parallelization is to identify independent tasks and distribute them to different threads which can each execute a sequential algorithm and run in parallel on multiple CPU cores. To harness even more compute power, it is possible to scale out to multiple compute nodes in a cluster, however, additional considerations like data distribution and network communication are needed to be taken into account. Corresponding High Performance Computing (HPC) implementations often utilize the Message Passing Interface (MPI) + X paradigm [93], where distributed memory parallelization between nodes using MPI is combined with intra-node shared-memory parallelization. Other approaches rely on partitioned global address space (PGAS) extensions like UPC++ [124] or OpenSHMEM [14] which allow for one-sided communication in contrast to the traditional two-sided MPI where matching send and receive calls are required.

Typically, the degree of parallelism achieved by such coarse-grained methods is limited, however, exploiting the capabilities of modern hardware with fine-grained implementations is much harder. This often requires novel algorithm designs with parallelization in mind, instead of running existing sequential code in parallel on multiple execution units. For instance, many tools utilize SIMD vectorization or even bit-parallel algorithms to accelerate sequence alignments. These techniques are for example employed to calculate the shifted hamming distance [118] to filter identified seed as well as the edit distance [15] which optimizes the bit-parallel Myers algorithms [82].

In addition to one or multiple CPUs, workstations and cluster nodes may feature accelerator devices like FPGAs and GPUs, which offer vast compute capabilities although their memory resources are often limited. GPUs especially are a popular choice as they are programmable for general purpose applications (GPGPU) using high level programming languages like CUDA [83] or OpenCL [105] and programs stay compatible across many GPU generations. To some extent source code compatibility is even possible across different vendors, but may require (automatic) code translation and recompilation. Due to the high core count of modern GPUs which allows for massive fine-grained parallelism, they are able to achieve around one order of magnitude higher peak performance than multi-core CPUs. However, extra care has to be taken when designing and implementing algorithms on the GPU. Deep knowledge of the underlying GPU architecture is required to implement optimized algorithms and to unlock their full potential.

NGS algorithms in particular are data intensive, making efficient memory access and concurrent data structures a necessity. Additionally, compute operations in NGS are integer-based most of the time, while the majority of GPU applications focus on floating-point arithmetic, e.g. in graphics pipelines or physics simulations. For these reasons, GPU adoption in sequence-based bioinformatics lags behind other application areas, where algorithms can be converted more naturally.

Nevertheless, some NGS tasks have been successfully translated to the GPU (see Table 2.2). Simple tasks like k-mer histogram generation are well suited to take advantage of the fast GPU memory. For example, k-mer index structures are employed for k-mer counting on one [62, 10] or multiple GPUs [19], and DecGPU [72] even utilizes a distributed k-mer spectrum for error correction implemented with MPI and CUDA.

Other problems like read mapping and metagenomic classification require more complex k-mer indices like multi-value hash tables or suffix arrays which need to be adjusted for efficient GPU usage. Therefore, most other work focuses on GPU acceleration of queries to CPU-built data structures, being far easier than parallel construction on the GPU. Most GPU-based short read aligners are based on accelerating the lookups for popular algorithms such as BWA-MEM or Bowtie2 adopting the FM-index and Burrow-Wheeler-Transform (BWT) [69, 12]. The read mapper Arioc [115] even stores its hash tables in CPU main memory to circumvent the limited amount of GPU memory. For metagenomic read classification cuCLARK

Tab. 2.2.: Example GPU-accelerated NGS read processing applications.

NGS Problem	Application	Technique[1]	Data Structure[2]
k-mer counting	GPU KMC2 [62]	single GPU, MT	HT
	Cadenelli et al. [10]	single GPU, MT	BF
	Gerbil [19]	multiple GPUs, MT	HT
error correction	DecGPU [72]	distributed GPUs, MT	BF
	CARE [43]	GPU build+query, MT	HT
genome assembly	Megahit [61]	GPU BWT build, MT	SdBG
read mapping	CUSHAW2-GPU [69]	single GPU, MT	FM-index
	Chacón et al. [12]	single GPU	FM-index
	Arioc [115]	single GPU, MT	(host) HT
metagenomic classification	cuCLARK [49]	GPU query, MT	HT

[1] MT = multi-threaded on CPU in addition to GPU
[2] HT = hash table, BF = Bloom filter, SdBG = succinct *de Bruijn* graphs [107]

[49] accelerates queries to the k-mer index of CLARK, which needs to be constructed on the CPU beforehand under high time and memory requirements.

There are some exceptions, tackling the task of GPU index construction. Examples include Megahit [61] which adapts the BWT-construction algorithm CX1 [67] for genome assembly, and CARE [43] which builds and queries multi-value hash tables on the GPU to accelerate error correction. Still, parallel construction of concurrent index structures remains a challenging problem. Especially in a multi-GPU distributed memory scenario, often required to increase the amount of available memory, communication and synchronization play an important role and demand additional considerations.

Background

3

In contrast to multi-core CPUs, which employ less than a hundred cores, modern GPUs feature thousands of cores allowing for massively parallel execution. Furthermore, high-end GPUs rely on High Bandwidth Memory with transfer rates of up to 2 TB per second. Compared to CPUs the memory capacity per GPU is limited but access speed is much higher. Often multiple GPUs are incorporated into a single workstation or server which increases the total amount of memory available. Due to their vast compute capabilities and highly competitive compute-to-energy ratio GPUs have been widely adopted for a plethora of use cases. This thesis focuses on CUDA-enabled GPUs [83] produced by NVIDIA, but the presented techniques and algorithms can also be applied to other GPGPU (General-Purpose Computing on Graphics Processing Units) platforms such as AMD GPUs using OpenCL [105] or HIP [@3].

Tab. 3.1.: Comparison between example CPUs and GPUs.

	CPUs		GPUs	
Vendor	AMD	AMD	NVIDIA	NVIDIA
Name	2990WX	5995WX	GV100	A100
Architecture	Zen+	Zen3	Volta	Ampere
Core count	32	64	5120	6912
Peak Performance[1]	~1.5 TFLOPS	~5.5 TFLOPS	14.8 TFLOPS	19.5 TFLOPS
Memory size	up to 2 TB	up to 2 TB	32 GB	80 GB
Memory bandwidth	94 GB/s	205 GB/s	870 GB/s	1,935 GB/s
TDP	250 W	280 W	250 W	300 W
Launch date	08/2018	03/2022	03/2018	06/2021
Data Sheets	[@1]	[@2]	[@13]	[@10]

[1] TFLOPS = Trillion single precision floating point operations per second.

Table 3.1 compares high-end CPUs from 2018 and 2022 to high performance GPUs of similar age. Recent CPUs and GPUs increase in core counts but GPUs are still more than two orders of magnitude ahead. The memory bandwidth of GPUs is also about one order of magnitude higher.

3.1 CUDA Programming Model

GPUs, being accelerator devices, cannot execute complete programs on their own but act as co-processors instructed by the host CPU. They consist of execution units and multiple levels of device memory physically separate from the CPU and are able to run thousands of threads simultaneously. The typical workflow consists of copying data from host to device, executing a GPU *kernel* which manipulates the data in a data-parallel fashion and finally retrieve the results on the host.

In the Compute Unified Device Architecture (CUDA) programming model the programmer has to define kernel launches in host code and supply the number of thread blocks and the number of threads per block which should execute the same kernel function. Thread blocks are scheduled onto the GPU's multithreaded streaming multiprocessors (SMs). Each SM consists of multiple CUDA cores (typically 64 or 128), a certain number of 32-bit registers and has a unified data cache which can be used as L1 cache or manually addressed scratchpad memory (also called *shared memory*). Blocks are divided into sets of 32 consecutive threads forming a *warp* which is scheduled and executed by the SM. The SM is able to hold the execution context of multiple warps on-chip which enables switching between active warps without overhead. This hardware multithreading allows to overlap the program execution of different warps to better utilize the SM's hardware resources.

Since NVIDIA's Volta GPU generation each thread in a warp has its own program counter and call stack, enabling threads to branch and execute independently. However, to reach full efficiency all 32 threads of a warp have to agree on their execution path and perform the same instruction, which can then be executed at the same time. Otherwise, the warp is split into groups of threads which are executed separately. Different warps on the other hand are free to execute divergent code branches independently without penalizing performance. This execution model is often called SIMT (single instruction multiple threads) in reference to SIMD (single instruction multiple data), where in SIMD a single instruction is executed for all of its vector lanes and lanes cannot be programmed individually.

The number of blocks and warps that can reside on a SM at a time depends on the required resources per thread and thread block as well as the available hardware resources per SM. Therefore, it can be beneficial to reduce the amount of registers used per thread or the amount of shared memory used per block to achieve higher SM occupancy and greater performance. Note, that the resources per SM and limitations on the number of resident warps and blocks differ between GPU generations. Table 3.2 shows some of the limits for the Quadro GV100 with compute capability 7.0. For

Tab. 3.2.: GV100 specifications.

Property	Quantity
Streaming Multiprocessors	80
Max Shared Memory	96 KB/SM, 48 KB/Block
Maximum Active Blocks	32/SM
Maximum Active Warps	64/SM
Maximum Active Threads	2048/SM, 1024/Block
Available 32-bit Registers	65536/SM, 65536/Block

a complete overview of the technical specifications per compute capability see the official CUDA programming guide [@11] Chapter K, Table 15.

Kernels launched from the host execute asynchronously on the GPU, meaning the CPU thread which launched the kernel does not have to wait until kernel execution is finished and can continue doing other work. Kernels as well as asynchronous memory copy operations can be inserted into CUDA streams, which execute queued operations sequentially. Different streams can be processed concurrently on the same GPU, allowing to overlap memory operations and kernel execution or to execute multiple kernels at the same time if enough hardware resources are available. Additionally, it is possible to insert synchronization points (so-called events) into the streams to model dependencies between operations in different streams, even across multiple GPUs. These events can also be used to synchronize streams from the host CPU to ensure certain work has finished or to wait for results to arrive in host memory. This enables complex pipelines orchestrated by the host and executed asynchronously on one or multiple GPU(s).

3.2 GPU Memory

Memory management is another important topic regarding GPU computing. Allocating device memory by the host is a synchronizing call to the CUDA runtime. All kernel and copy operations have to finish before memory can be allocated. Therefore, it is best to allocate the memory needed for all steps of an algorithm at the beginning of a program, but also to reuse allocated memory as much as possible to avoid reallocation or wasting space for short lived data structures.

Although bandwidth of global GPU memory is quite high compared to main memory of the CPU (~2 TB/s vs. ~200 GB/s, respectively, compare Table 3.1), high throughput can only be achieved with proper access patterns from threads of a GPU kernel.

Device memory is accessed by 32-, 64-, or 128-byte memory transactions at naturally aligned memory addresses, thus misaligned requests will be padded to achieve aligned access, reducing the throughput. Global memory accesses from threads of a warp can be coalesced into the aforementioned types of memory transactions if the threads access a common memory region. For example, if all 32 threads request consecutive 4-byte words starting at an address aligned to 128 bytes, this can be handled by a single 128-byte memory transaction. However, if the threads access random memory locations in global memory, this can generate up to 32 different 32-byte transactions, massively decreasing the throughput.

Compared to global memory the access to on-chip shared memory is much faster. This memory can be used by thread blocks to store intermediate results instead of writing them to global memory. After synchronizing with the whole block the shared results can then be accessed by all threads of the same block. The registers available to each thread of a block are even faster and can also be used to store results which do not need to be shared within the entire block. However, it is possible to exchange register data with threads of the same warp by using warp shuffle instructions. This enables a warp to solve small problems like a local reduction or prefix scan cooperatively without using auxiliary memory. Even larger problems can benefit from this hierarchy by applying a divide and conquer approach by first calculating local solutions in independent warps, then merging with the whole block and finally combining the partial results in global memory.

In order to accelerate an application on the GPU one might try to handle each individual part with a different kernel (separation of concerns). However, each kernel has to store its results in global memory which can then be reloaded in the next kernel if needed. Due to the latency differences of global memory, shared memory and register access, it is often much more efficient to fuse kernels together (if possible) and store data in shared memory or registers instead to avoid global memory accesses altogether.

3.3 Multi-GPU Systems

In data-parallel tasks it is oftentimes feasible to add additional GPUs to increase the amount of compute available to an application. Independent tasks can be distributed among multiple GPUs instead of processing them one after another on a single one. However, modern data science applications often require a lot of memory and the scarce amount of memory attached to a single GPU becomes a limiting factor. Storing data structures on the host and staging transfers through slow PCIe might be an option in some cases but often hinders performance. To expand the amount of memory available to an application, multiple GPUs have to be combined in an intelligent way. Modern multi-GPU systems feature fast interconnects between GPUs which allow for direct memory access from one GPU to another. Nevertheless, extra care has to be taken on where data should be placed and how to distribute large data structures across multiple GPUs; remote accesses induce higher latencies and don't achieve the same bandwidth as local memory accesses. Hence, efficient communication becomes key to enable fast data movement and keep the GPUs from starving.

To address these issues we examine two different strategies in the following parts. In Part I we focus on multi-GPU communication patterns which are then applied in an iterative algorithm to exchange data between GPUs in each step. In Part II on the other hand we distribute a database across GPUs and use a pipeline approach where partial results are sent from one GPU to the next. Here, we aim to minimize the amount of data which needs to be communicated between GPUs to avoid the communication bottleneck.

Part I

Multi-GPU Communication

4 Gossip Communication Library

This chapter is based on the following peer-reviewed paper:

Gossip: Efficient Communication Primitives for Multi-GPU Systems.
Robin Kobus, Daniel Jünger, Christian Hundt, Bertil Schmidt.
Proceedings of the 48th International Conference on Parallel Processing (ICPP 2019).
https://doi.org/10.1145/3337821.3337889

Abstract – Nowadays, a growing number of servers and workstations feature an increasing number of GPUs. However, slow communication among GPUs can lead to poor application performance. Thus, there is a latent demand for efficient multi-GPU communication primitives on such systems. This work focuses on the gather, scatter and all-to-all collectives, which are important operations for various algorithms including parallel sorting and distributed hashing. We present two distinct communication strategies (ring-based and flow-oriented) to generate transfer plans for their topology-aware implementation on NVLink-connected multi-GPU systems. We achieve a throughput of up to 526 GB/s for all-to-all and 148 GB/s for scatter/gather on a DGX-1 server with only a small memory overhead. Furthermore, we propose a cost-neutral alternative to the DGX-1 Volta topology that provides an expected higher throughput for the all-to-all collective while preserving the throughput in case of scatter/gather. Our Gossip library is freely available at https://github.com/Funatiq/gossip.

4.1 Introduction

Massively parallel accelerators have been widely adopted for number crunching due to their vast compute capability and highly competitive compute-to-energy ratio. The scarce amount of memory attached to a single GPU is often too small to hold the entire data set due to the ever increasing memory requirements of modern data science applications. While streamed workflows can be processed sequentially in batches, traditional data structures such as hash maps and data indexes need to reside in the GPU's memory to avoid expensive data movements from the host to the device. To expand the amount of available memory, a workstation can be equipped with multiple GPUs. However, not every problem can be split into multiple independent parts to allow for data parallelism. Hence, efficient communication primitives are needed to enable fast data movement between compute devices.

Modern CUDA-enabled GPUs such as the P100 and V100 accelerators feature a high-bandwidth interconnect, called NVLink, for fast inter-device communication. The number of physical connections per device is limited. Therefore, there exist NVLink-based multi-GPU topologies where pairs of GPUs are connected by a varying number of links, sometimes even none at all. If a programmer schedules a direct transfer between two devices without an NVLink connection the transfer will be routed via slower PCIe. Usually, multiple GPUs share a PCIe switch and often transfers occur across GPUs connected to different NUMA nodes, which reduces the global bandwidth even further in case of several parallel transfers.

Parallel to the release of the NVLink interconnect, NVIDIA published the NVIDIA Collective Communications Library (NCCL) [@6], which implements a set of common collective operations specifically optimized for multi-GPU topologies. NCCL focuses on implementing NVLink-aware collectives that are commonly used for distributed deep learning models and thus currently supports all-reduce, all-gather, reduce-scatter, reduce and broadcast. However, other widely used collective operations can also benefit from these topologies. To the best of our knowledge, they have not been implemented nor optimized, yet. This work focuses on the gather, scatter and all-to-all collectives. While these communication primitives can be implemented using a ring-based approach similar to collectives included in NCCL, we also show how they can be formulated as flow problems. Those can be fed to a solver for integer linear programming (ILP) problems to obtain optimal transfer schedules which saturate the bandwidth bottlenecks of common topologies.

All-to-all communication is crucial in case data has to be reorganized in GPU memory to avoid expensive host-sided data movement during the partitioning phase.

Single-GPU multisplit approaches as proposed by Ashkiani et al. [4] can naturally be extended to multi-GPU environments by virtue of a subsequent all-to-all communication. The resulting distributed multi-split primitives can for example be used for the efficient querying and construction of multi-GPU hash maps [41]. Furthermore, incremental sorting and merging during GPU-based suffix array construction [114] and multi-dimensional FFTs [17] could be further scaled to multiple GPUs using all-to-all communication. In general, distributed variants of approximate and exact data indices such as bloom filters, quotient filters and histograms are promising candidates for straightforward distribution across multiple GPUs. The 526 GB/s throughput of our proposed all-to-all primitive allows for the efficient interleaving of NVLink-based inter-GPU data partitioning and subsequent memory utilization of participating GPUs. This ranges in the same order-of-magnitude as pure memory access, e.g., 900 GB/s memory bandwidth in case of V100 HBM2 modules.

In this work, we also investigate asymmetric subtopologies and propose alternative NVLink topologies where traditional ring-based schedules can be outperformed. As an example, a minor cost-neutral modification to the DGX-1 Volta topology yields an estimated 33% improvement of all-to-all performance while preserving the triple ring structure used by NCCL.

4.2 Related Work

Common NVLink topologies are composed of distinct rings. NCCL's communication collectives exploit the fact that bandwidth-efficient algorithms for these topologies exist. Patarasuk et al. showed that all-gather [22] (also called all-to-all broadcast) as well as all-reduce [90] can be implemented bandwidth optimally on rings found in topologies of workstation clusters. They further proposed a pipelined broadcast scheme [89] using a contention-free linear tree similar to a ring, which is effective for large message sizes. The same approach is used by NCCL for the broadcast collective.

The all-to-all collective has been extensively studied, but research focuses on specific topologies common to clusters of compute nodes such as meshes [109], hypercubes [100], or tori [112, 35]. Some even assume fully connected networks [8]. None of these are directly applicable to our case. Fraigniaud and Lazard [25] give an overview of several communication methods for various common topologies including the (single) ring. Among other collectives they analyze scatter and all-to-all (here called multiscattering) and give upper and lower bounds. Their ring-based scatter is similar to ours, while their all-to-all scheme relies on a different ordering of the transfers along the ring. To reduce the start-up overhead they aggregate data chunks in every step which we do not pursue in our approach. A survey of Chan et al. [13] explores lower bounds and algorithms for communication collectives, excluding all-to-all, on common topologies and use the results to improve on MPI-based communication. They use a simple model which assumes that nodes can directly address each other and transfers are routed automatically, which is not possible when relying exclusively on NVLink edges.

To increase the performance of communication between nodes in a cluster, efforts have been taken to optimize MPI collectives using knowledge about the network topology. Zahavi et al. [121] try to discover the underlying topology to provide an optimized transfer sequence for all-to-all using tree structures. Kandalla et al. [44] design topology-aware scatter and gather for large-scale clusters exploiting their hierarchical structure. Karonis et al. [45] follow a similar approach and demonstrate its benefits for a multi-level broadcast. Gong et al. [28] develop network-performance-aware collectives for dynamic cloud environments. While these hierarchical topologies differ from the considered single-node multi-GPU networks, they might be of interest for future research regarding interconnected multi-GPU nodes.

4.3 Background

4.3.1 Topologies

Currently there exist two GPU architectures of interest for NVLink-based single-node multi-GPU interconnection networks.

Pascal-based: P100 GPUs support four first generation NVLink connections with 40GB/s bidirectional bandwidth each.

Volta-based: V100 devices support six connections of second generation NVLink with 50GB/s bidirectional bandwidth.

Interconnection networks based on Pascal devices can be connected via two rings spanning all GPUs. In graph theory the rings can be described as disjoint Hamiltonian cycles, i.e., cycles connecting all nodes via disjoint sets of edges. A ring can be characterized by a permutation of the indices of the devices. Take four devices indexed from 0 to 3 for example, one ring could be in ascending order 0-1-2-3, another ring could be 0-2-1-3. Here, two GPUs of successive indices in the permutation as well as the first and the last one are connected through NVLink. Figure 4.1a illustrates this example topology consisting of two rings. Because of NVLink being bidirectional each ring can be traversed in forward or reverse order. Each NVLink connection of a GPU has its own memory controller which allows all links to be used independently at the same time. The two additional connections for Volta-based devices can be used to establish one more ring, for a total of three rings. Current DGX-1 systems feature eight V100 GPUs connected via the rings 0-3-2-1-5-6-7-4 (times two) and 0-1-3-7-5-4-6-2 (see Figure 4.2). Although the topology can be viewed as separate rings, communication does not have to be restricted to rings. In fact, different links between GPUs can be used independently of each other, while multiple connections between two GPUs effectively multiply the bandwidth. Thus, the DXG-1 topology can also be interpreted as a hypercube with additional edges, but in general does not match any common topology for which optimal algorithms are known.

4.3.2 Communication Collectives

In this work we investigate the following three collective communication primitives.

Scatter is a communication collective with a one-to-many relation. One node starts off with all the data in its memory which then has to be distributed equally among all nodes including itself, see Figure 4.3a for a graphical depiction.

Gather is the reverse operation to scatter. Here one node has to collect equal amounts of data from all nodes including itself (see Figure 4.3b).

All-to-all is a many-to-many communication primitive. It can be seen as simultaneous scatter (or gather) operations originating from every node at the same time: Every node sends a data package to every other node including itself. This communication can be interpreted as each node swapping an amount of data with every other node like Figure 4.3c illustrates.

A canonical way to implement the scatter primitive would be to schedule all transfers from the source node in parallel. However, this is only possible if these direct connections exist and even then it might not be the optimal solution because the connections may have different bandwidths. Note, that data which has to reach a certain node can be split into several chunks which can be transferred along different

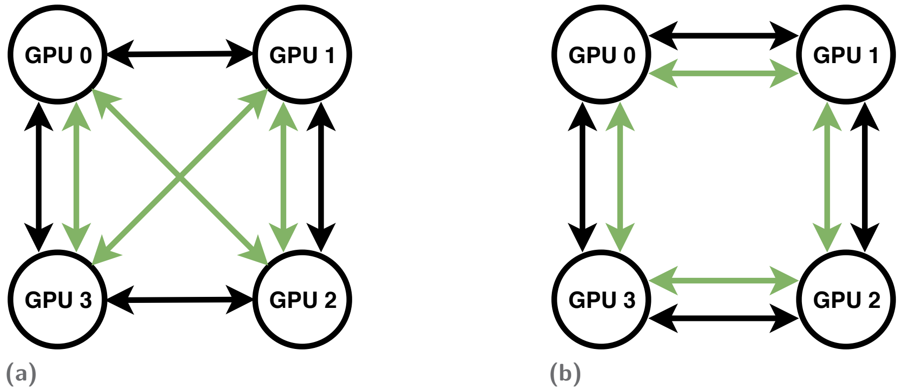


Fig. 4.1.: Two example topologies for four GPUs with four NVLink connections each. The connections can be partitioned in two distinct rings (black and green arrows).

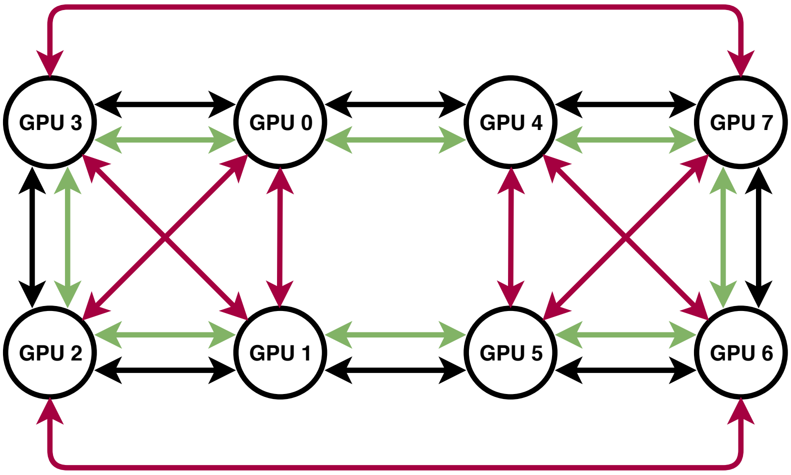
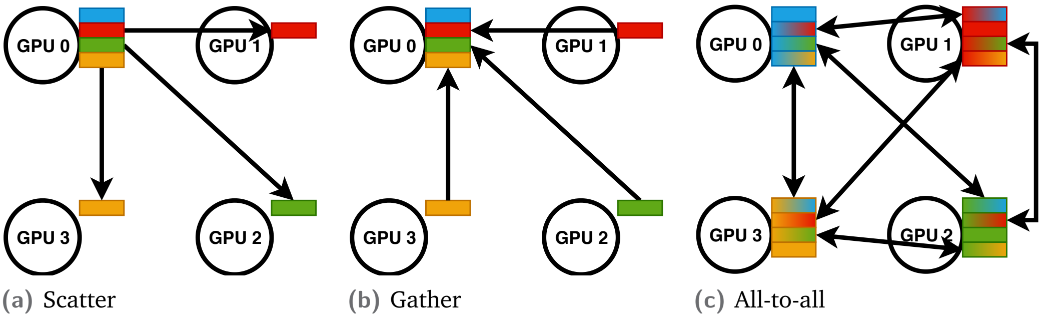


Fig. 4.2.: DGX-1 topology featuring eight V100 GPUs connected via three distinct rings (black, green and red arrows).



(a) Scatter (b) Gather (c) All-to-all

Fig. 4.3.: Communication collectives.

paths in the topology. If a transfer schedule for the scatter collective is known it can be played back in reverse order with opposite transfer directions to obtain a valid schedule for gather.

Note, that it is not sufficient to use multiple scatter collectives in parallel in order to implement an efficient all-to-all primitive as they may use the same connections of the network topology at the same time leading to edge contention. Thus, all communication operations need to be considered together to find an optimal transfer schedule.

4.4 Ring-based Collectives

A simple approach when creating transfer schedules for NVLink topologies is to look at each ring separately. For some collectives like reduce-scatter, all-gather, and all-reduce, rings can be used for a bandwidth-optimal implementation. Data is split into a number of chunks according to the number of rings, so that all rings can transfer parts of the data independently and in parallel.

4.4.1 Scatter/Gather

A ring-based approach for scatter works as follows. Assume a unidirectional ring of length n with nodes indexed $0, \ldots, n-1$ in ascending order. Without loss of generality let Node 0 be the source node. Let $d_{i,j}$ be the data originating from Node i targeted at Node j. Transfers are scheduled in descending order of their travel distance which corresponds to their respective index. $d_{0,n-1}$ which has to transit from Node 0 to Node $n-1$ will be scheduled first. In each step one chunk of data will be moved to the next node in the ring. $d_{0,n-1}$ will be transferred from Node 0 to Node 1 in the first step, from Node 1 to Node 2 in the second step and so on. $d_{0,n-2}$ follows with a delay of one step, it will be transferred from Node 0 to 1 in the second step. This continues until in the $(n-1)$-th step where all data reach their target destination simultaneously. Figure 4.4 illustrates this scheme for four GPUs. Note, that the rings are actually bidirectional. Thus, we can send another round of data in opposite direction at the same time. This means that for r bidirectional rings the original data is split into $2r$ chunks which will be transferred in parallel.

Another approach is to consider both directions of a bidirectional ring at once in order to send each data chunk on its shortest path to its target node. This effectively cuts the ring in half. Using the same notation and indexing as above, half of the

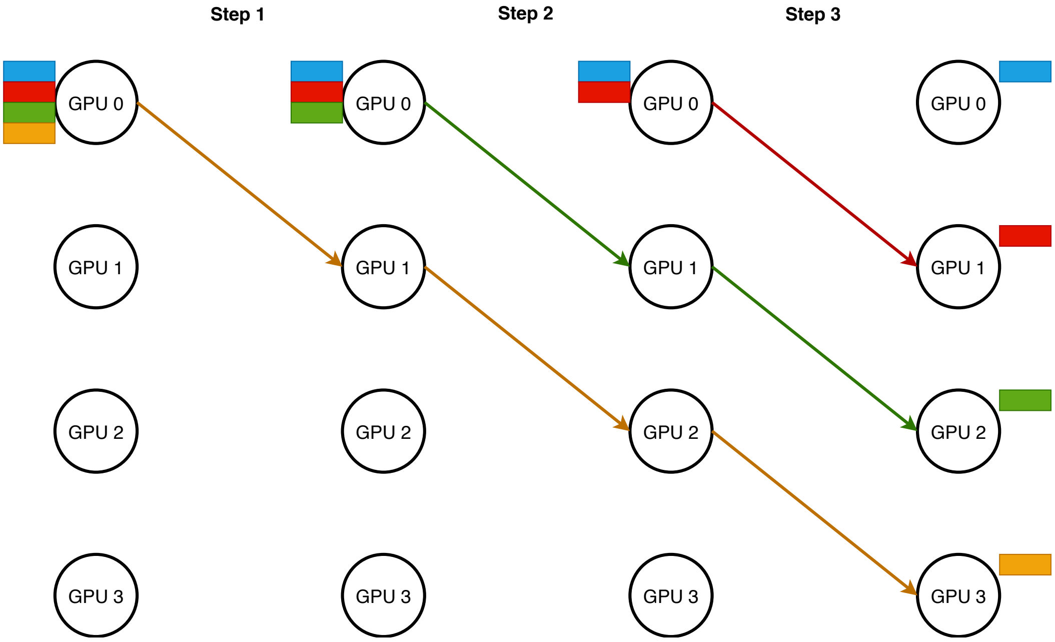


Fig. 4.4.: Ring-based scatter for four GPUs.

data will be sent to the nodes $1, \dots, \lfloor\frac{n}{2}\rfloor$ in forward, the other half to the nodes $n-1, \dots, \lceil\frac{n}{2}\rceil$ in reverse direction. Note, that for an even number of nodes half of the data targeted at Node $\frac{n}{2}$ can be sent in each direction. This approach has some advantages compared to the unidirectional-ring-based one. First, the transfer paths for the data chunks are shorter which leads to less transfer instructions and reduces the necessary synchronization between transfers. Second, the data does not have to be split into a chunk for each direction (apart from $d_{0,\frac{n}{2}}$). Therefore, the number of transfers is reduced further while the transfer sizes increase which decreases the overhead that is produced per transfer. However, regarding the theoretical transfer time excluding any overhead, both approaches are equivalent. The unidirectional scheme needs $n-1$ transfer steps of half size, while the bidirectional one uses $\lfloor\frac{n-1}{2}\rfloor$ full size transfer steps (equal to $\lfloor\frac{n-2}{2}\rfloor$ full size plus another half size step in case of an even number of nodes).

As mentioned before, the gather collective can be implemented as a reverse scatter. This naturally applies to the ring-based schemes.

4.4.2 All-To-All

The idea for our ring-based all-to-all communication scheme is comparable to the bidirectional scatter approach whereby every node acts as a source node at the

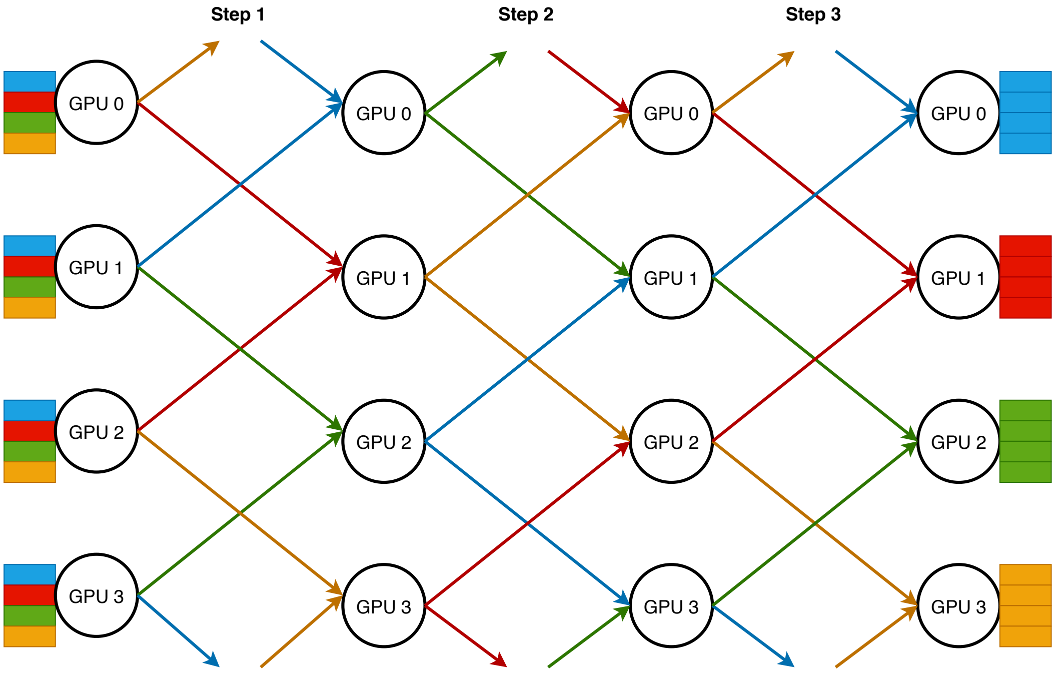


Fig. 4.5.: Ring-based all-to-all for four GPUs.

same time. For each direction, instead of interleaving the transfers of different data chunks of the same origin, these need to be handled one after the other. This allows all nodes to handle transfers in both directions in parallel at each step.

Consider again the bidirectional ring $0, \dots, n-1$. First each Node i sends data $d_{i,i+1}$ to Node $i+1$ (wrapping around), which already occupies all communication paths in forward direction. At the same time Node i sends data $d_{i,i-1}$ in reverse direction and the ring is fully utilized. Subsequently, each node continues with the same scheme for data $d_{i,i+2}$ and $d_{i,i-2}$ which have to be rotated along the ring for 2 steps in their respective direction, followed by $d_{i,i+3}$ and $d_{i,i-3}$ for 3 rotation steps and so on. The last chunks have to be transferred for $\frac{n-1}{2}$ steps if n is odd. For even numbers of n data $d_{i,i+\frac{n}{2}}$ can be split to send half of it in each direction of the ring in $\frac{n}{2}$ half size steps. This leads to a total of

$$\sum_{i=1}^{\frac{n-1}{2}} i = \frac{n^2-1}{8} \qquad \text{and} \qquad \frac{1}{2} \cdot \frac{n}{2} + \sum_{i=1}^{\frac{n-2}{2}} i = \frac{n^2}{8}$$

full size steps for odd n and even n, respectively, where all of the ring's connections are used in parallel. Fig. 4.5 illustrates this scheme for four GPUs. Here, each GPU exchanges data $d_{i,i+1}$ and $d_{i,i-1}$ with its neighbors in Step 1. Step 2 and 3 are used to transfer half of data $d_{i,i+2}$ in each direction across two GPUs.

4.5 Flow Problem Formulation

Transfer schedules in communication networks like a cluster of compute nodes can be modelled as multi-commodity flow problems. The same applies to interconnected GPUs on a single compute node or workstation. A commodity resembles a chunk of data residing at an initial source node or GPU which should be transferred to a specific target node or GPU, respectively. A single communication primitive can consist of many of these commodity transfers which should overlap in time to finish all transfers as fast as possible. The all-to-all primitive for example needs to transfer one chunk of data from each of n nodes to all n other nodes simultaneously, a total of n^2 transfers. Because we want to find a solution which minimizes the amount of time needed to finish all transfers, we use a time-expanded graph to solve our flow problem. This problem is called quickest multi-commodity flow in the literature [23].

Consider an NVLink topology consisting of n nodes. We build a directed graph $G = (V, E)$ of nodes V and edges E. For every discrete point in time $\tau \in \{0, \ldots, T\}$ there exists a node $v_{i,\tau} \in V$ for each GPU i in the topology. Let V_0 and V_T be the set of all nodes at start time $\tau = 0$ and end time $\tau = T$, respectively. If two GPUs i and j are connected by one or multiple bidirectional NVLink connections, we insert two edges $(v_{i,\tau}, v_{j,\tau+t})$ and $(v_{j,\tau}, v_{i,\tau+t})$ with unit capacity into the graph for every time point τ. The considered number of time steps t depends on the number of connections between the GPUs. Multiple NVLink edges between two devices effectively multiply the bandwidth between them and can be viewed as a single, faster link. If we consider a topology restricted to single and double connections, for example, we would use one time step ($t = 1$) for the duration of a transfer along a double connection and two time steps ($t = 2$) for a single connection, because it would take twice as long. Additionally, we insert edges $(v_{i,\tau}, v_{i,\tau+1})$ for every GPU i and time τ, which pass on commodities to the next time step that could not be transferred in step τ due to transfer edges already being used to capacity. Figure 4.6 partially depicts the time-expanded graph for four GPUs connected by the topology from Figure 4.1a.

In our scenario, it is beneficial to keep the number of transfers as low as possible, because every transfer between two devices creates a small overhead. Therefore, we want each commodity flow l which starts at source node $s_l \in V_0$ and ends at target node $t_l \in V_T$ to be transferred on a single path without being split at intermediate nodes. This also reduces the synchronization needed which ensures that a transfer in one time step has finished and the contained data can be transferred further in

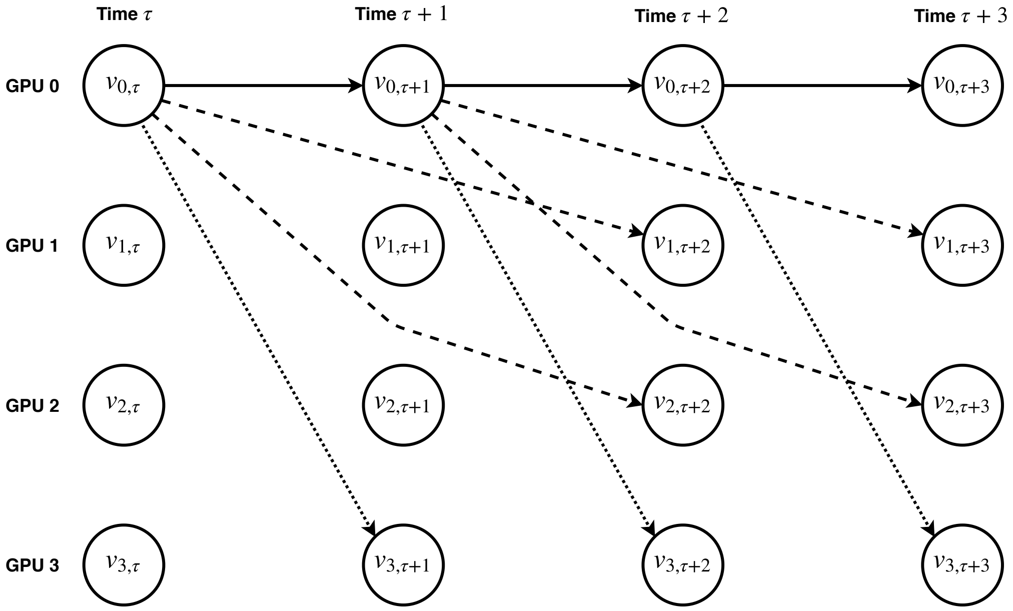


Fig. 4.6.: Partial time-expanded graph for the topology from Fig. 4.1a. Shown are outgoing edges of GPU 0 for three time steps. Transfers along dotted edges need one time step while transfers along dashed edges need two steps. Continuous edges pass on remaining commodities to the next step.

the next time step. Depending on the topology, it can be impossible to saturate the bandwidth at the bottleneck if we limit ourselves to transferring each distinct commodity from source to target as a whole. For this reason we split each commodity into k equally sized chunks, but still guarantee that each chunk will be transferred along a single path. Note, that for the considered primitives this means that all chunks of all commodities have exactly the same size, however, it would be easy to accommodate arbitrary commodity sizes as multiples of a predefined chunk size. Therefore, this model for finding an optimal transfer plan can also be used for the primitives gather-v, scatter-v and all-to-all-v if the sizes are known in advance.

This problem, where multiple commodity flows are each split into exactly k non-zero flows of identical size along paths from source to target, is a special case of the uniform exactly-k-splittable multi-commodity flow [5]. The general case allows flows of different commodities to be of different sizes. We limit every flow to be either one or zero, so that every edge can transport exactly one commodity chunk at a time.

Different from flow problems for a single commodity with potentially multiple sources and/or targets, multiple commodities have to be considered. Let M be the set of distinct commodities. For a commodity $m \in M$ let $S_m^+ \subseteq V_0$ be the set of source nodes and $S_m^- \subseteq V_T$ the set of target nodes, let $S_m = S_m^+ \cup S_m^-$. Each source node $v \in S_m^+$ starts with an integral number of chunks $D_{v,m} \in \mathbb{Z}_+$ and each target node $v \in S_m^-$ has an integral demand $D_{v,m} \in \mathbb{Z}_-$, such that $\sum_{v \in S_m} D_{v,m} = 0$. For every edge $e \in E$ and commodity $m \in M$ we want to find a binary flow value $x_{e,i} \in \{0, 1\}$, which satisfies the *flow conservation constraint* at transit nodes

$$\sum_{e \in \delta^+(v)} x_{e,m} - \sum_{e \in \delta^-(v)} x_{e,m} = 0 \text{ for all } v \in V \setminus S_m \text{ and } m \in M \tag{4.1}$$

as well as source and target nodes

$$\sum_{e \in \delta^+(v)} x_{e,m} - \sum_{e \in \delta^-(v)} x_{e,m} = -D_{v,m} \text{ for all } v \in S_m \text{ and } m \in M \tag{4.2}$$

where $\delta^+(v)$ and $\delta^-(v)$ denote the sets of arriving and leaving transfers, respectively. While the conservation constraints apply for each commodity separately, the edge capacities needs to be respected by all commodities at once. Because transfers can take multiple time steps but the connections should be used exclusively at every time, we need to avoid overlaps of transfers using the same link. Let $E_{i,j,\tau}$ be the set of all edges between GPUs i and j which cover the time step $[\tau, \tau + 1]$, i.e.,

$$E_{i,j,\tau} = \{(v_{i,\tau-u}, v_{j,\tau+t-u}) \in E : t \in \{1, 2\}, u \in [0, t)\}.$$

If the *capacity constraint*

$$\sum_{\substack{m \in M \\ e \in E_{i,j,\tau}}} x_{e,m} \leq 1 \text{ for all } 0 < i, j < n, \ \tau \in [0, T) \tag{4.3}$$

holds, the flow is called *feasible*.

4.5.1 Scatter/Gather Flow Problem

The scatter primitive can be modelled as a multi-commodity flow where one source node starts off with n different commodities, each of which needs to be sent to a different node in the topology. The gather primitive describes the same problem but in reverse. Each node has its own commodity which needs to be transferred to the same target node. Note, that one commodity already resides on the source or target node and does not need to be communicated over the network. For both primitives we identified the bottleneck to be the number of links to the main node where all data originates or converges. If these connections are saturated at all times by outgoing (or incoming) transfers of unique data chunks there cannot be a more efficient transfer schedule. To make this possible the total number of transfers has to be a multiple of the number of connections. For this reason we determine the greatest common divisor of the number of commodities which need to be transferred $(n-1)$ and the number of links L to the main node to calculate the smallest factor k by which all commodities have to be split such that $k \times (n-1)$ is divisible by L.

$$k = \frac{L}{gcd(n-1, L)} \tag{4.4}$$

We call the uniform exactly-k-splittable multi-commodity flow which solves the scatter or gather problem *optimal* if it respects the constraints (4.1), (4.2), (4.3) and saturates the connections of the main node with transfers of unique data chunks at all times. The corresponding transfer schedule is called *optimal*, too.

4.5.2 All-To-All Flow Problem

As mentioned before, the all-to-all primitive requires each node to send a commodity to every other node. To model this problem, we use one distinct commodity per target, so that each node starts with n different commodities like in the scatter case. At the end, each node must have collected all parts of the commodity belonging to itself. For this collective we have determined that the bisection bandwidth is the limiting factor. If we look at an arbitrary bisection of n nodes, we observe that each node partition has to send $\frac{n}{2}$ commodities to each node in the other partition. Therefore, $(\frac{n}{2})^2$ commodities have to cross through the links between the two partitions in each direction. The bisection bandwidth denotes the minimum bandwidth between the two partitions of all possible bisections and is hence the bottleneck of the communication. To saturate the bisection bandwidth it is necessary to split the commodities such that the number of commodity chunks is divisible by the number of links W belonging to the bisection bandwidth. Analogously we use the greatest common divisor of the number of commodities $(\frac{n}{2})^2$ and the number of links W to calculate the splitting factor k.

$$k = \frac{W}{gcd((\frac{n}{2})^2, W)} \tag{4.5}$$

We call the uniform exactly-k-splittable multi-commodity flow which solves the all-to-all problem *optimal* if it respects the constraints (4.1), (4.2), (4.3) and saturates the connections of the bisection bandwidth with transfers of unique data chunks at all times. The corresponding transfer schedule is called *optimal*, too.

4.5.3 Double-Buffered All-To-All

To minimize the memory consumption, we also consider an all-to-all variant using only a double buffer per GPU which holds the input data at the beginning and the results at the end of the communication. In each step the schedule should transfer data from one buffer to the other, then we need to synchronize before the next step can continue. One of these meta steps can include multiple time steps where data is transferred through the same link more than once. For this to work we need to adapt the flow problem in the following way.

Instead of nodes for each point in time we now have nodes for each synchronization point; instead of edges of different lengths we insert edges of different cost between two synchronization points. In the case of single and double connections, we would use unit cost for double edges and twice the cost for single edges. Additionally, we replicate the edges with linear increasing cost to allow for multiple usage in the same step. The same double connection would be inserted as edges of cost $1, 2, 3, \ldots, b$, where b is the buffer size in number of data chunks, single edges with cost $2, 4, 6 \ldots, 2b$, accordingly. When minimizing the cost of the flow this enables the ILP solver to prioritize the cheap edges before using expensive ones while the cost of the most expensive edge in use indicates the total time required for the whole step. Furthermore, we add another constraint to the problem to model the storage limitations of the buffers. Let $E_{i,\sigma}$ be the set of all edges exiting GPU i at synchronization point $\sigma \in [0, \Sigma)$. The *storage constraint*

$$\sum_{\substack{m \in M \\ e \in E_{i,\sigma}}} x_{e,m} \leq b \text{ for all } 0 < i < n, \ \sigma \in [0, \Sigma) \tag{4.6}$$

enforces the amount of exiting flow to be limited by the storage size. Because of the conservation constraints 4.1 and 4.2 the same limit implicitly applies for flow entering each node.

4.6 Implementation Details

4.6.1 Transfer Schedule Generation

The NVLink topology of a system can be queried by the command line utility *nvidia-smi*. We use this information to create the flow problems as formulated in Section 4.5. Alternatively, an adjacency matrix providing the number of links between all

GPUs can be used. To solve the ILP problems we use the CBC solver from Google's OR-Tools suite [@8]. We can calculate the minimum number of time steps needed to get a feasible solution by dividing the total number of data chunks which have to pass through the bottleneck by the number of links at the bottleneck. If no solution can be found we increment the number of time steps of the problems and run the solver again. Because the ring-based schedules (Section 4.4) are a possible solution for the flow problems and we know how many steps are needed for their execution, we always know an upper bound for the number of time steps needed.

After the solver outputs a solution for the flow problem we have to trace the paths of all commodity chunks from the beginning to the end. Subsequently, we generate a complete schedule that determines which data chunks have to be transferred between specific pairs of GPUs in each step. Finally the transfer schedule is saved to disk in human-readable JSON format for later usage.

4.6.2 Mapping Transfer Schedules to Streams

When executing a transfer schedule we have to take care of transfer concurrency and synchronization. Transfers in the same step of the schedule should be carried out in parallel, while consecutive transfers of a transfer path have to wait for their predecessor to finish. To enforce this we use CUDA streams and events. A stream is a pipeline for CUDA memory operations and kernel calls which are executed one after another. Each stream is associated with exactly one GPU while multiple streams can run asynchronously on a single GPU. To synchronize between streams (even between streams of different devices) we can insert events into the pipeline which other streams are able to synchronize with before continuing. This enables us to avoid global synchronization of one or all devices, because only streams which depend on each other have to be synchronized accordingly. Thus, we schedule all transfers originating from the same device in different streams, one for each of its NVLink connections. In further steps the streams which perform followup transfers have to wait for the events in the streams being responsible for the preceding transfers.

4.7 Evaluation

4.7.1 Experimental Setup

Experiments have been conducted on the following systems:

S1: Dual-socket Intel Xeon E5-2680 CPU with 256 GB DDR4 RAM and 4 NVIDIA Tesla P100 GPUs (with NVLink connections as shown in Fig. 4.1a) each with 16 GB HBM2, running CentOS 7, CUDA 9.1, GCC 6.3.0.

S2 (DGX-1): Dual-socket Intel Xeon E5-2698 CPU with 512 GB DDR4 RAM and 8 NVIDIA Tesla V100 GPUs (with NVLink connections as shown in Fig. 4.2) each with 32 GB HBM2, running Ubuntu 18.04, CUDA 10.0, GCC 7.3.0.

S3 (DGX-1 Quad): The same as S2, but using only one half of the topology, i.e. GPUs $\{0, 1, 2, 3\}$. We denote this sub-topology as a *quad* where all four GPUs are connected to the same CPU socket via PCI-e.

For each system we have evaluated the following transfer schedules for scatter[1] and all-to-all communication:

V1 - Direct: Direct transfers from sources to targets all scheduled immediately, ignoring potentially missing edges in the NVLink topology using PCI-e as fallback.

V2 - Rings NVLink: The bidirectional schemes from Section 4.4 executed in parallel for each ring in the topology.

V3 - Optimized unsplit NVLink: The transfer schedule given by the solution of the flow problems without splitting the commodities.

V4 - Optimized k-split NVLink: The transfer schedule given by the solution of the flow problems when splitting the commodities according to Equations (4.4) or (4.5).

V5 - Double-buffered NVLink: The transfer schedule for all-to-all given by the solution of the flow problem presented in Section 4.5.3.

All setups were tested on uniformly randomly distributed data as inputs over varying input sizes in terms of the overall accumulated amount of input data of all participating GPUs. The memory architecture of CUDA-accelerators is optimized for high-throughput data movement of large amounts of data but lacks performance in high-latency scenarios when moving small sizes of data. Thus, we limit our experiments to a minimum input size of 32 KB and 8 KB per GPU for all-to-all and scatter/gather, respectively. We report overall throughput performance which is defined as the total amount of input data of all GPUs over the collective's runtime. This includes those portions of the data that already reside on their respective target GPU

[1] The results for gather are similar to those of scatter because the transfers are the same but in reverse order.

Tab. 4.1.: A flow-based transfer schedule for the scatter collective generated by our library. Each row corresponds to a transfer path expressed by a sequence of device identifiers that describe on which route a chunk of data is transferred. Each column of the paths belongs to the same point in time.

Data	Part	Transfer Path
$d_{0,0}$	0	$0 \to 0 \to 0 \to 0 \to 0 \to 0 \to 0$
$d_{0,0}$	1	$0 \to 0 \to 0 \to 0 \to 0 \to 0 \to 0$
$d_{0,0}$	2	$0 \to 0 \to 0 \to 0 \to 0 \to 0 \to 0$
$d_{0,0}$	3	$0 \to 0 \to 0 \to 0 \to 0 \to 0 \to 0$
$d_{0,1}$	0	$0 \to 0 \to 0 \to 0 \to 0 \to 1 \to 1$
$d_{0,1}$	1	$0 \to 0 \to 0 \to 1 \to 1 \to 1 \to 1$
$d_{0,1}$	2	$0 \to 0 \to 3 \to 3 \to 3 \to 1 \to 1$
$d_{0,1}$	3	$0 \to 1 \to 1 \to 1 \to 1 \to 1 \to 1$
$d_{0,2}$	0	$0 \to 0 \to 0 \to 0 \to 0 \to 2 \to 2$
$d_{0,2}$	1	$0 \to 0 \to 0 \to 2 \to 2 \to 2 \to 2$
$d_{0,2}$	2	$0 \to 2 \to 2 \to 2 \to 2 \to 2 \to 2$
$d_{0,2}$	3	$0 \to 3 \to 3 \to 3 \to 2 \to 2 \to 2$
$d_{0,3}$	0	$0 \to 0 \to 0 \to 0 \to 0 \to 0 \to 3$
$d_{0,3}$	1	$0 \to 0 \to 0 \to 0 \to 0 \to 3 \to 3$
$d_{0,3}$	2	$0 \to 0 \to 0 \to 0 \to 3 \to 3 \to 3$
$d_{0,3}$	3	$0 \to 0 \to 0 \to 3 \to 3 \to 3 \to 3$

and are thus not communicated over the NVLink interconnect but being potentially moved in global memory using fast intra-GPU memcopies.

4.7.2 Experimental Results

Table 4.1 shows a flow-based transfer schedule for the scatter collective generated by our library for S1. Data targeted at a distinct GPU is split into four parts with its own transfer path consisting of a sequence of GPU identifiers. Every time step of the flow problem corresponds to one step in a path. If consecutive IDs remain constant no action has to be taken and the data chunk stays on the corresponding device. Different IDs denote a corresponding transfer that has to be executed in this step. Note, that the faster connection between GPUs 0 and 3 enables us to send one part of data $d_{0,1}$ and $d_{0,2}$ through GPU 3 instead of using the direct links in a naïve fashion.

Figures 4.7 and 4.8 show the total throughput for different input sizes for the scatter and all-to-all collectives executed on S1. As the GPUs are fully connected via the system's topology the schedules generated by the unsplittable flow problem (V3) coincide with the direct transfers between GPUs (V1) and thus are omitted. For

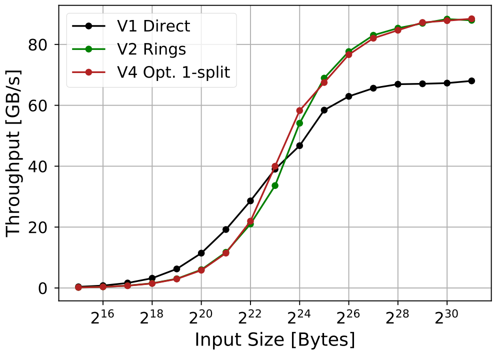


Fig. 4.7.: Scatter/gather throughput on S1.

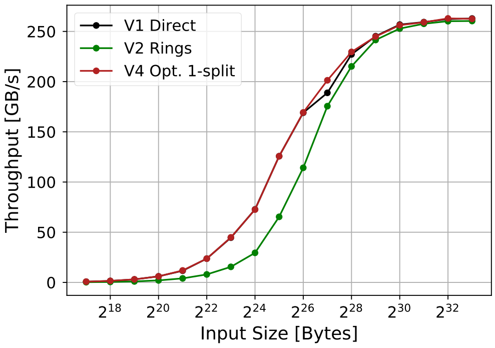


Fig. 4.8.: All-to-all throughput on S1.

all-to-all even the k-split version is the same because the splitting factor equals to 1. The fully-connected topology also renders the double-buffer approach unnecessary since the data can be moved directly to the final locations on the target devices.

In case of scatter the ring-based schedule and the optimized k-split schedule perform equally well on this system. Both versions split data into four parts which fit the number of outgoing edges. Therefore, they are able to saturate the bandwidth bottleneck for large input sizes and reach a total throughput of 88 GB/s. For smaller input sizes the overhead per transfer decreases performance. The schedule using direct transfers performs better than the other methods in these cases although it does not meet the criteria of a (bandwidth-)optimal transfer schedule. It consists of fewer but bigger data transfers which reduces the overhead. Nevertheless it is not able to take advantage of the double NVLink connections between nodes $(0, 3)$ and $(1, 2)$ which limits the maximum throughput. For large input sizes the difference in throughput resembles the proportion of $\frac{3}{4}$ incorporated edges at the source node.

In the all-to-all case the k-split version is able to outperform the ring-based approach. The flow problem is aware of the direct connections and the number of commodities fits the number of links of the bisection bandwidth. Therefore, each data chunk $d_{i,j}$ does not have to be split and can be transferred on the shortest path to its corresponding target. On the other hand, when using rings, data is partitioned and some chunks of it will take a detour to reach their targets through a ring. Thus a higher number of transfers and synchronization is required which causes more overhead. This leads to less performance especially for smaller input sizes.

On the DGX system all transfer schedules differ from each other due to the more complex topology. Some GPU pairs are not connected via NVLink edges and data transfers between the GPUs have to be rerouted to reach good performance. Figures 4.9 and 4.10 present the results for the scatter and all-to-all communication

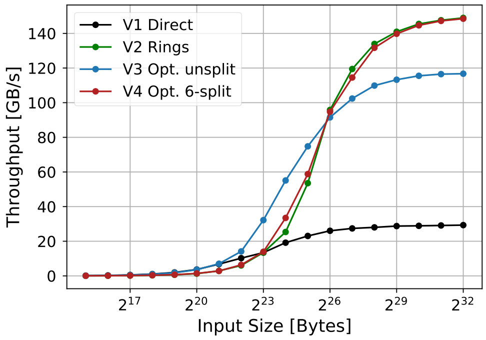


Fig. 4.9.: Scatter/gather throughput on S2 (DGX-1, 8 GPUs).

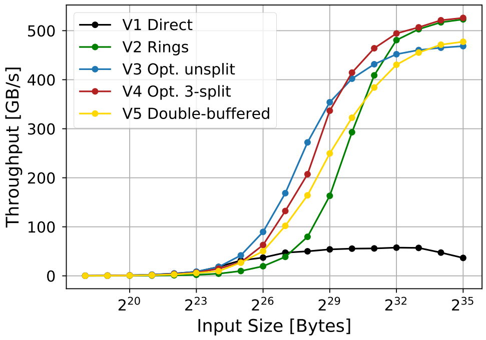


Fig. 4.10.: All-to-all throughput on S2 (DGX-1, 8 GPUs).

benchmarks on S2. The missing NVLink connections cause some transfers of the direct schedule to be routed via slow PCI-e which for both collectives translates to a significantly smaller maximum throughput compared to the other strategies. However, for very small input sizes where transfer speed plays a less important role the direct transfers can compete with other unsplit approaches.

Regarding the scatter primitive running on S2, the ring-based and optimized k-split schedules behave similarly to S1. Here, data is split into six chunks according to the number of rings and connections, respectively. The maximum throughput of 148 GB/s is only limited by the bandwidth bottleneck. The unsplit version is again able to perform better for smaller input sizes, but throughput is limited. The source GPU has to send seven distinct data packages, one to every other GPU, which does not match the six outgoing edges. Still, the solution of the flow problem unveils that double edges can be used frequently to reach $\frac{7}{9}$ of the maximum throughput of the k-split schedule, which matches our experimental results.

For all-to-all the difference between the ring-based and k-split version is analogous to S1. The two approaches achieve the highest throughput of 523 GB/s and 526 GB/s for large input sizes. The k-split version performs better in all cases, especially for medium sizes. The unsplit schedule is closer to the k-split schedule than in the scatter case because the splitting factor is smaller. Because of this it is able to outperform V2 and V4 in more cases than before.

In case of using the DGX-1 Volta topology it is reasonable to use the double-buffered schedule, too. The reduced memory consumption comes at the price of decreased throughput. Here, only a single global synchronization step is needed for this approach, but this already leads to a 10-20% lower performance compared to the asynchronous 3-split version.

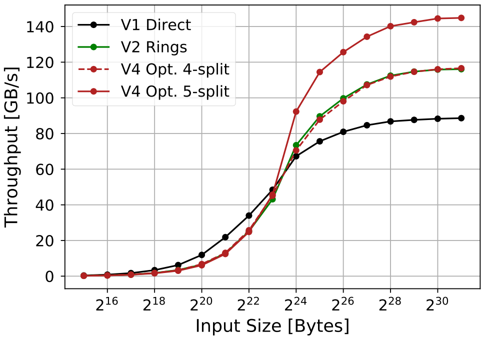


Fig. 4.11.: Scatter/gather throughput on S3 (DGX-1 quad, 4 GPUs).

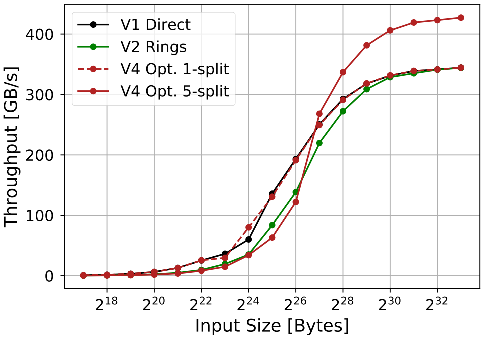


Fig. 4.12.: All-to-all throughput on S3 (DGX-1 quad, 4 GPUs).

To test the different transfer schedules on an asymmetric topology we have used one half of the DGX-1 system. The four GPUs 0,1,3 and 4 are connected with the same topology as S1 plus an additional NVLink edge between GPUs 2 and 3 which cannot be associated with one of the rings. This enables us to use the schedules originally developed for the four GPU system and to generate a specialized version by incorporating the extra edge in the flow problems. Figure 4.11 shows the benefits of this approach for the scatter collective with source GPU 3. For large data sizes the optimized 5-split schedule improves by up to 25% over the original 4-split version since it can make use of five instead of four outgoing NVLink connections. The all-to-all communication can benefit from the additional edge, too, because of the increased bisection bandwidth. To fit the number of connections the splitting factor rises from 1 to 5 which has a negative effect for small input sizes. However, the maximum throughput grows from 344 to 427 GB/s as illustrated in Figure 4.12.

Table 4.2 shows the throughput at the bottlenecks identified in Sections 4.5.1 and 4.5.2 for all three evaluated systems. The theoretically available bandwidth corresponds to the number of links involved at the bottleneck. In S1 those are four connections of 20 GB/s unidirectional (or 40 GB/s bidirectional) bandwidth, while six and five edges of 25 GB/s (50 GB/s bidirectional) bandwidth are concerned in S2 and S3, respectively. The amount of data which has to pass through the bottleneck depends on the number of GPUs. S1 and S3 contain four GPUs and thus $\frac{3}{4}$ of the total input has to leave the source node when executing the scatter collective. In the case of eight GPUs on S2 it amounts to $\frac{7}{8}$ of the input. For all-to-all communication exactly half of the input will be sent through the bottleneck, $\frac{1}{4}$ in each direction. For large input sizes we achieve 82-88% of the total available maximum bandwidth across all configurations.

Tab. 4.2.: Maximum achieved bottleneck bandwidth when running scatter and all-to-all on the tested systems.

	Scatter/Gather		All-To-All	
	Achieved	Available	Achieved	Available
S1	66 GB/s	80 GB/s	131 GB/s	160 GB/s
S2	130 GB/s	150 GB/s	263 GB/s	300 GB/s
S3	108 GB/s	125 GB/s	214 GB/s	250 GB/s

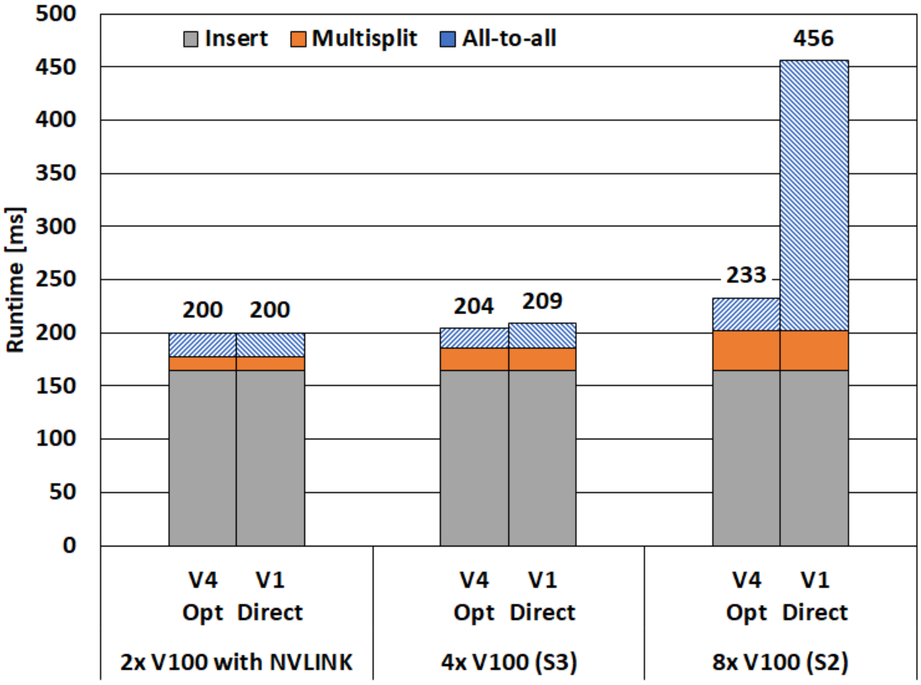


Fig. 4.13.: Weak scalability of our Warpdrive application over 2, 4, and 8 Tesla V100 with 2 GB of input data per GPU.

4.7.3 Case Study: Hash Table Construction

We have analyzed the impact of Gossip on our WarpDrive [41] application for constructing distributed hash tables on multi-GPU systems which uses a two step data distribution scheme: (1) a multisplit partitioning of the input data on each GPU to determine the amount of data to be sent to each other GPU, (2) a subsequent all-to-all communication among all GPUs. Finally, each GPU inserts its received partition into a local hash table. Figure 4.13 shows a weak scalability analysis with runtime breakdowns for the before-mentioned phases. We compare a naive all-to-all communication scheme (V1) to our proposed optimal solution (V4).

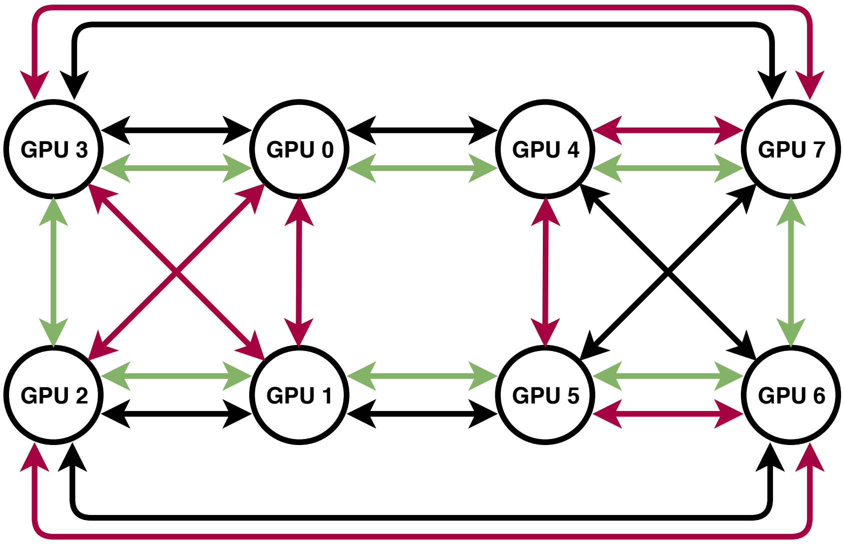


Fig. 4.14.: Alternative DGX topology for eight Volta-based devices connected via three rings (black, green, red arrows).

4.7.4 Alternative DGX topology

Besides the DGX-1 topology being physically available to conduct benchmarks of different approaches, we have investigated alternative NVLink topologies for eight V100 devices. Figure 4.14 depicts a topology which is identical to the DGX-1 with the exception of two connections: Instead of connecting the GPUs $(2, 3)$ and $(6, 7)$ with two links, the ports are used to connect $(2, 6)$ and $(3, 7)$ with an additional link. With this minor modification the degree and total number of edges remains the same, but it allows to increase the bisection bandwidth by 33% while still preserving the availability of three dedicated rings. Therefore the performance of the ring-based all-to-all would be unchanged compared to the results in Section 4.7.2. Yet it would no longer be optimal because some data chunks had to cross the links between the two partitions $\{0, 1, 2, 3\}$ and $\{4, 5, 6, 7\}$ twice which violates the requirement of uniqueness for optimality. Take for example the transfer path 0-4-6-2 for data $d_{0,2}$ on the black ring, where the links 0-4 and 6-2 pass the data between partitions. The transfer schedule resulting from the flow problem however is able to saturate the bisection bandwidth which would lead to a 33% better performance than the ring-based approach. An additional benefit is that commodities do not have to be split here because the splitting factor k from Eq. (4.5) equals 1 in this case.

4.8 Conclusions

Irregular interconnect topologies of current NVLink-based multi-GPU servers impose major challenges for the determination of optimal transfer plans used in common communication collectives such as scatter/gather and all-to-all. Hence, state-of-the-art solutions relying on theoretically proven optimal schedules executed on symmetric subtopologies such as hypercubes or rings cannot saturate the full bandwidth of the whole interconnect in the general case. Asymmetric topologies occurring in case of partial usage of the available accelerator cards render the computation of optimal transfer plans even more complex. In this work, we have investigated the applicability of ring-based and flow-oriented transfer approaches. Our flow-oriented schedules are harvested by means of integer linear programming and have been proven to include solutions equivalent to ring-based ones in case of optimality but may come up with superior solutions in case of irregular topologies.

We achieve a throughput of up to 526 GB/s for all-to-all and 148 GB/s for scatter/-gather on a DGX-1 Volta server with only a small memory overhead which, in both cases, corresponds to 88% of the theoretically achievable bottleneck bandwidth of the underlying NVLink topology. We have further shown that an efficient all-to-all is key for the scalability of distributed hashing on a DGX-1 system.

Moreover, we propose a cost-neutral modification of the DGX-1 Volta topology which is degree-preserving and sustains its triple ring decomposition while providing 33% expected all-to-all bandwidth improvement and identical scatter/gather performance.

The proposed schedules for scatter/gather and all-to-all primitives are optimized for data structure construction algorithms using comparably big chunk sizes. Accordingly, we achieve maximum throughput for increasing data package sizes. However, small data scenarios with low latency demands could also be treated in the context of multi-commodity flow optimization by adding mandatory self-loops in the time-expanded graph with suitably chosen bandwidth constraints. Another direction of future research could be the dynamic adjustment of static transfer plans in case of non-stationary data distributions. This could be achieved by tracking the estimated partition sizes over time based on moving average statistics combined with occasional recomputation of optimal solutions. Determining an optimal solution of the flow problem can be achieved in a few seconds due to the relatively small amount of variables of the integer linear program. We plan to include latency-aware dynamic communication primitives in the future to handle highly unbalanced data distributions into Gossip. Gossip is available at https://github.com/Funatiq/gossip.

Suffix Array Construction 5

This chapter shows how Gossip's all-to-all communication pattern is applied in the multi-GPU suffix array construction approach of the following peer-reviewed paper:

Suffix Array Construction on Multi-GPU Systems.
Florian Büren, Daniel Jünger, Robin Kobus, Christian Hundt, Bertil Schmidt.
Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC 2019)
https://doi.org/10.1145/3307681.3325961

Some text fragments, examples and results are directly taken from said paper.

5.1 Background

A suffix array (SA) contains the starting positions to all suffixes of a string, ordered by lexicographically sorting the suffixes. Let $s = x_0 x_1 \ldots x_{n-1}$ be a string consisting of $n \geq 1$ letters $x_i \in \Sigma^*, 0 \leq i < n$, where Σ is a finite, totally ordered alphabet. For each $i \in \{0, \ldots, n-1\}$ we define the i-th suffix of s as $s_i = x_i x_{i+1} \ldots x_{n-1}$. Because each suffix can be uniquely identified by its index i it is sufficient to store the indices in the SA instead of complete sub-strings s_i. Additionally, we define the *inverse suffix array* (ISA) which maps each suffix s_i to its unique rank $j = rank(i)$ in SA. The lexicographic order of the suffixes can be expressed by each side of the following equivalence

$$SA[j] = i \Leftrightarrow ISA[i] = j, \text{ for } j = rank(i)$$

Commonly, a special character $\$ = x_n$ is appended to the string s to denote its end. \$ lexicographically precedes every other string, therefore including this character in the SA will always result in $SA[0] = n$. Table 5.1 shows the SA and ISA for the input string "banana\$".

Tab. 5.1.: Suffix array (SA) and rank/ISA for the input string “banana$”.

i	Suffix i	rank(i) = ISA[i]	Sorted suffix	SA[i]
0	banana$	4	$	6
1	anana$	3	a$	5
2	nana$	6	ana$	3
3	ana$	2	anana$	1
4	na$	5	banana$	0
5	a$	1	na$	4
6	$	0	nana$	2

5.2 SA Construction: Prefix Doubling

As we are interested in generating the lexicographical order of all suffixes, a naïve approach would be to sort all the suffixes of the input string with a common string sorting algorithm. However, suffixes are obviously not independent strings; their relations can be exploited to develop better algorithms. There exist several different approaches [91] to SA construction which have been studied extensively. Here we focus on *prefix doubling* which was employed as the main algorithm for our multi-GPU SA construction. Figure 5.1 depicts an overview of the complete workflow.

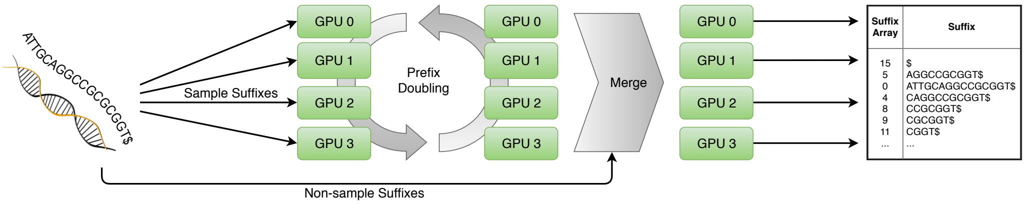


Fig. 5.1.: Overview of the employed suffix array construction algorithm.

Assume that all suffixes are already sorted lexicographically by only considering their prefixes of length h. This so called h-order can be used to infer a $2h$-order of the suffixes because for each suffix s_i we can use the h-order of suffix s_{i+h} which contains the next h letters of s_i. We define $rank_h(i)$ to denote the (non-unique) rank of each suffix s_i according to the h-order. Sorting all suffixes into h-order yields segments of the same h-rank if suffixes share a common prefix of length $\geq h$. The suffixes in each segment can then be sorted independently using $rank_h(i+h)$ to obtain the $2h$-order. This may split each segment into several smaller segments of different $2h$-ranks. Note, that suffixes from different segments will never interfere because their order has already been established in a previous step, their order will only get more accurate with each doubling step. This prefix doubling process is repeated until all suffixes have been assigned a unique rank, leaving no segment of size greater than one, resulting in the SA.

The following example shows how to infer the $2h$-order of the two suffixes s_1 and s_6 from the h-order of s_5 and s_{10}. Prefixes considered so far are underlined.

$$
\begin{array}{l}
\text{Let } s = yabbadabbadoo\$,\ h = 4. \\[6pt]
\left.\begin{array}{rl} s_5 &= \underline{dabb}adoo\$ \\ s_{10} &= \underline{doo\$} \end{array}\right\} \Rightarrow rank_h(5) < rank_h(10) \\[6pt]
\left.\begin{array}{rl} s_1 &= \underline{abba}dabbadoo\$ \\ s_6 &= \underline{abba}doo\$ \end{array}\right\} \Rightarrow rank_h(1) = rank_h(6) \\[6pt]
\left.\begin{array}{r} rank_h(1) = rank_h(6) \\ rank_h(1+4) < rank_h(6+4) \end{array}\right\} \Rightarrow rank_{2h}(1) < rank_{2h}(6) \\[6pt]
\qquad\qquad\qquad\qquad\qquad \Rightarrow \underline{abbadabb}adoo\$ < \underline{abbadoo\$}
\end{array}
$$

5.3 Multi-GPU Prefix Doubling

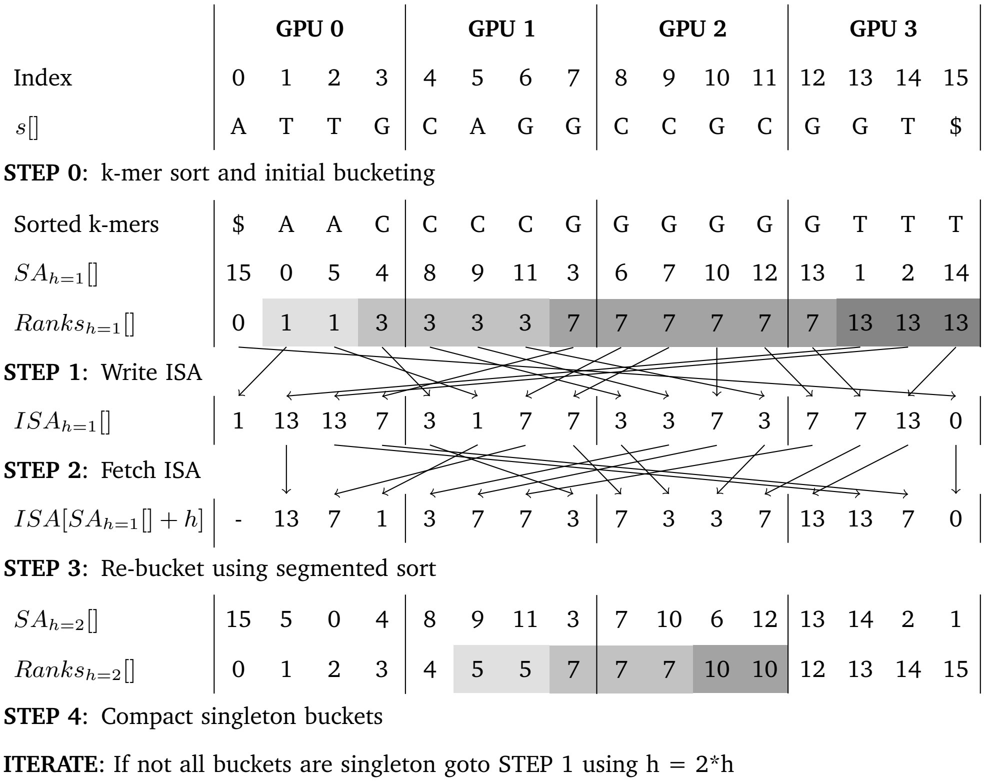
In our multi-GPU prefix doubling implementation, we distribute input and working arrays among all GPUs of a system to take advantage of additional memory capacity and increased compute resources. Our approach to prefix doubling is illustrated in Figure 5.2 and consists of five main steps:

STEP 0: k-mer sort and initial bucketing
STEP 1: Write ISA
STEP 2: Fetch ISA
STEP 3: Re-bucket
STEP 4: Compact

5.3.1 *k*-mer sort and initial bucketing

Initially, the input string s is evenly scattered over p GPUs (see $s[]$ in Fig. 5.2) with an overlap of $k-1$ characters. Each GPU proceeds to generate all k-mers of its share of the input string, i.e., all prefixes of length k of every suffix. For each k-mer, its (global) starting index is written to a separate array, leading to index-k-mer-tuples.

The tuples are sorted locally on each GPU by means of a fast 64-bit-key-value radix sort with k-mers as keys and indices as values. Subsequently, a custom distributed



	GPU 0				GPU 1				GPU 2				GPU 3			
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$s[]$	A	T	T	G	C	A	G	G	C	C	G	C	G	G	T	$

STEP 0: k-mer sort and initial bucketing

Sorted k-mers	$	A	A	C	C	C	C	G	G	G	G	G	G	T	T	T
$SA_{h=1}[]$	15	0	5	4	8	9	11	3	6	7	10	12	13	1	2	14
$Ranks_{h=1}[]$	0	1	1	3	3	3	3	7	7	7	7	7	7	13	13	13

STEP 1: Write ISA

$ISA_{h=1}[]$	1	13	13	7	3	1	7	7	3	3	7	3	7	7	13	0

STEP 2: Fetch ISA

$ISA[SA_{h=1}[] + h]$	-	13	7	1	3	7	7	3	7	3	3	7	13	13	7	0

STEP 3: Re-bucket using segmented sort

$SA_{h=2}[]$	15	5	0	4	8	9	11	3	7	10	6	12	13	14	2	1
$Ranks_{h=2}[]$	0	1	2	3	4	5	5	7	7	7	10	10	12	13	14	15

STEP 4: Compact singleton buckets

ITERATE: If not all buckets are singleton goto STEP 1 using h = 2*h

Fig. 5.2.: Illustration of the main steps of multi-GPU prefix doubling using the input string $s =$ `ATTGCAGGCCGCGGT$`, 4 GPUs, and $k = 1$ for k-mer sort and initial bucketing. All-to-all is performed three times per iteration: once in Step 1 and twice in Step 2.

merge algorithm is applied to (re-)merge the locally sorted portions of the array to a distributed globally sorted array ($SA_{h=1}[]$ in Fig. 5.2).

Afterwards, initial ranking and bucketing takes place. Every bucket is identified by the global index of its starting position. The indices of bucket starts are propagated across all other suffixes in the respective buckets by using local inclusive prefix scans and inter-GPU communication. This yields a globally correct ranking for prefixes up to $h_0 := k$. The initial ranking after this step is shown as $Ranks_{h=1}[]$ in Figure 5.2.

5.3.2 Write ISA

For prefix doubling, the ranks of suffix s_{i+h} (in string order) needs to be available globally, while operating on partially sorted local arrays. Consequently, an distributed ISA is maintained which contains the ranks of suffixes according to the current h-order as shown in $ISA_{h=1}[]$ in Fig. 5.2. The original index (in string order) of any given suffix in partially sorted order might refer to any GPU. Hence, writing the ranks indicating to which bucket each suffix belongs requires **all-to-all communication**. Consider all pairs (i, r) with i being the original starting index of the suffix and r being the rank assigned to it. Each GPU first executes a multisplit operation, permuting both arrays in such a way that elements with the same destination GPU are contiguous. After this step the contiguous ranges are communicated to their specific target GPUs, where the corresponding ranks can be written to the ISA array.

5.3.3 Fetch ISA

For subsequent re-bucketing the new sorting keys $ISA[i + h]$ for every s_i need to be fetched; i.e., given the working SA in partially sorted order, the element $SA[j] + h$ is needed for every index j from the current ISA yielding $ISA[SA_{h=1}[] + h]$ in Fig. 5.2. Since the index $SA[j] + h$ may refer to any GPU, a key-value multisplit is executed on $(j, SA[j] + h)$ according to $GPU(SA[j] + h)$. Using **all-to-all communication**, every GPU sends its query index ($SA[j] + h$) to the GPU where the relevant part of the ISA resides. GPUs look up $ISA[i]$ for every i received and return the ranks queried to the same index in the original array by means of a second call to the **all-to-all primitive**.

5.3.4 Re-bucket

With the appropriate information (original string indices SA_h, sorting keys $ISA[SA_h[]+h]$, segment heads, bucket ranks $Ranks_h$) in place, segmented sort can now be executed. Segments (buckets) spanning more than one GPU are treated as ordinary local segments during segmented sort, but need to be merged after each GPU has completed local sorting. This results in an updated suffix array as shown in $SA_{h=2}[]$ in Fig. 5.2. Subsequently, new ranks ($Ranks_{h=2}[]$ in Fig. 5.2) are assigned reflecting the new buckets formed with respect to the doubled prefix length $(2h)$.

5.3.5 Compact and iterate

Buckets containing a single suffix do not need to be sorted further and can therefore be safely removed from the working arrays. For each SA/Rank pair (i_j, r_j) it is checked whether $r_{j-1} < r_j < r_{j+1}$ applies. If it does, s_i is the only inhabitant of bucket r_j and the responsible GPU compacts i_j and r_j out of its working arrays. The algorithm terminates if all suffixes have been removed from the working list of each GPU. Otherwise, h is doubled and the algorithm continues with Step 1.

5.4 Analysis

In the original paper we have examined the time-consuming steps for fetching and updating the ISA in detail. Both involve accessing the ISA which is ordered in a completely different fashion (i.e., in string order) than the working arrays (partially sorted order). This not only necessitates an all-to-all communication with other GPUs but also leads to irregular memory access patterns through indirect addressing. For the first three prefix doubling iterations, we observed that all-to-all communication takes less time than accessing the memory. For subsequent iterations, the runtime is dominated by the cost for the all-to-all communication.

Throughout the multi-GPU prefix doubling algorithm all-to-all communication occurs three times in every iteration and takes a significant part in the overall runtime, which underlines the importance of an efficient all-to-all primitive.

5.5 Performance Evaluation

Here we show some results from the paper for the complete SA construction. Experiments were performed on the following system:

DGX-1 Volta: Dual-socket Xeon E5-2698 v4 CPU (2x20 cores at 2.20 GHz) with 512 GB DDR4 RAM and 8 Tesla V100 GPUs, each with 16 GB HBM2 memory, CUDA 10.0, GCC 5.4.0.

Similar to Wang et al. [114], we used a collection of datasets selected from the Manzini, Silesia, Large Canterbury and Protein corpora [@9] to evaluate our implementation. Additionally, we investigated the performance for larger real-world datasets like the source code of the Linux kernel version 4.9.99 (697 MB) and the Gutenberg corpus containing a selection of English literature (1211 MB), as well as several mammalian genomes:

- GRCh38 (human reference genome, 3104 MB),
- GCA_000003625.1_OryCun2.0 (rabbit, 2644 MB),
- GCF_000151905.2_gorGor4 (gorilla, 2962 MB),
- GCF_000772875.2_Mmul_8.0.1 (rhesus monkey, 3159 MB),
- GCF_002099425.1_phaCin_unsw_v4.1 (koala, 3083 MB),
- and GCF_002863925.1_EquCab3.0 (horse, 2421 MB).

We compare the performance to two CPU and two GPU implementations: *libdivsufsort* [@9], currently the fastest sequential CPU implementation serves as baseline for the speedup measurements, *parallel divsufsort* [57], a cilk-based shared memory parallelization of libdivsufsort, and two single-GPU CUDA implementation from CUDPP [114] and NVBIO [@5]. Note, that the NVBIO implementation produced incorrect results for six of the test datasets since some of the buckets were not completely sorted.

To reflect the workflow of executing SA construction on the GPU in batches as part of a larger framework, benchmarks exclude the time needed for disk I/O and memory allocation, but include the time for data transfers between CPU and GPU.

Figure 5.3 shows the speedups of our implementation and parallel divsufsort over the baseline libdivsufsort, which is the fastest sequential implementation currently available. Although the parallel divsufsort was executed with 80 CPU threads, it was only able to achieve a maximum speedup of 5.7. In contrast, we achieved speedups between 133 and 354 using 8 GPUs, which is more than two orders-of-magnitudes faster than libdivsufsort and still $30\times$ and $68\times$ faster than parallel divsufsort.

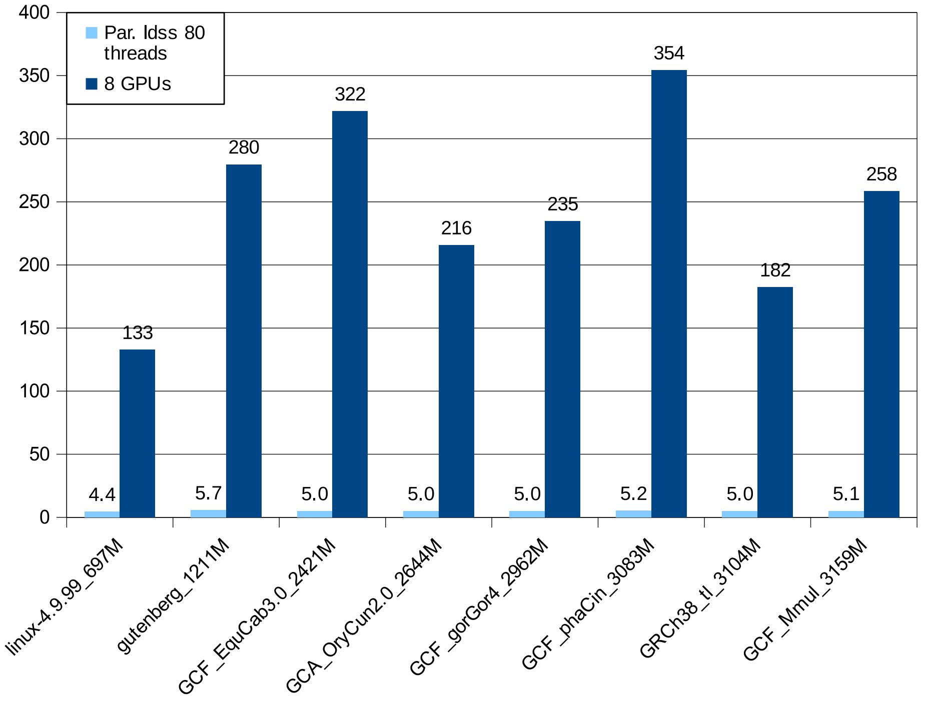


Fig. 5.3.: SA construction speedups over libdivsufsort on a DGX-1 using 8 GPUs for large-scale datasets. The speedups of parallel divsufsort over sequential libdivsufsort using 80 CPU threads are also reported (Par. ldss 80 threads). [9]

For the human genome dataset the state-of-the-art MPI implementation by Flick et al. [24] reported a runtime of 4.8s using 100 nodes with 1600 cores, which we were not able to reproduce. In comparison, we achieved a runtime of 3.44 s on a single DGX-1 server, resulting in a speedup of 1.4.

For the comparison of GPU implementations, we included the first 236 MB of the Gutenberg corpus as an additional dataset, which was the largest input size that could be processed by CUDPP's implementation on a single GPU with 16 GB of memory.

Figures 5.4 and 5.5 show the speedups of our multi-GPU implementation over CUDPP and NVBIO, respectively. On a single GPU, our algorithm is at least $1.9\times$ faster than CUDPP and at least $1.4\times$ faster than NVBIO. With two GPUs it reached $2.6 - 5.8$ and $2.5 - 3.5$ speedup compared to CUDPP and NVBIO, respectively. Using 4 or 8 GPUs, we achieved even greater speedup, but the optimal number of GPUs depended on the dataset. Generally speaking, higher numbers of GPUs required larger datasets to unleash their full potential. Note, that dividing smaller datasets in even smaller parts on each GPU does not efficiently use the compute resources of each GPU, while communication overhead increases with the number of GPUs.

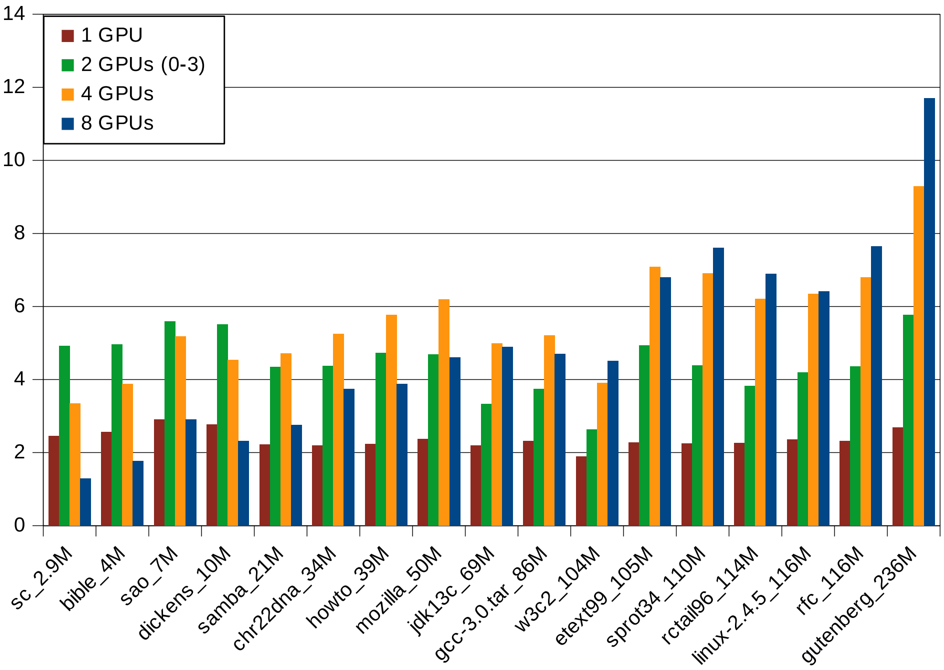


Fig. 5.4.: SA construction speedup over CUDPP's skew implementation (small and medium datasets) on a DGX-1. [9]

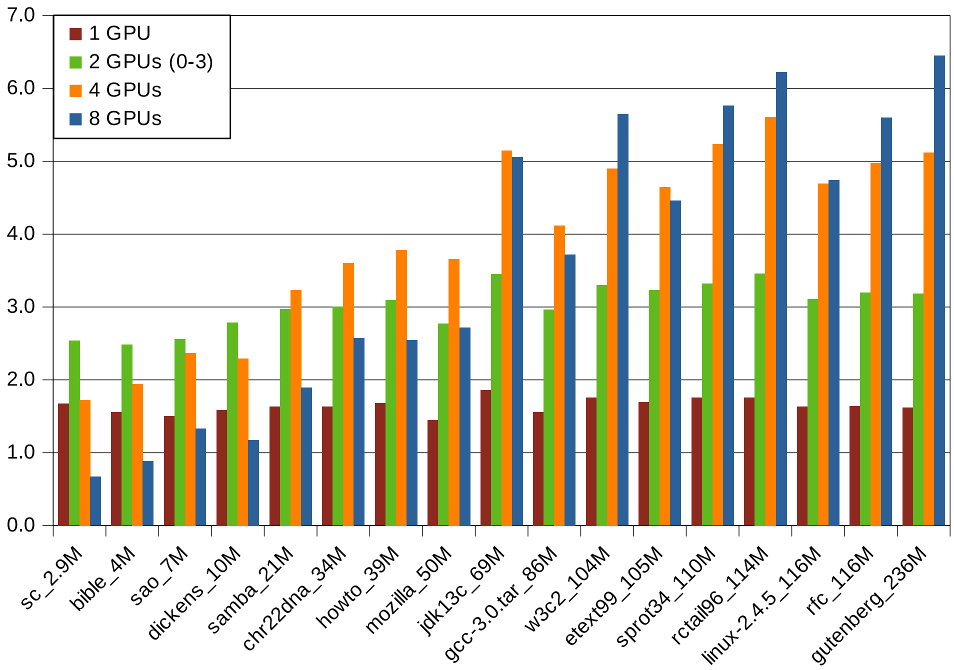


Fig. 5.5.: SA construction speedup over NVBIO's prefix doubling implementation (small and medium datasets) on a DGX-1. Note that NVBIO's results for the following datasets are incorrect: chr22dna_34M, etext99_105M, jdk13c_69M, linux-2.4.5_116M, rfc_116M, w3c2_104M. [9]

For datasets between 20 and 100 MB, using 4 GPUs was the fastest configuration, while larger datasets (except for etext99) were processed the fastest on 8 GPUs. The highest speedups were achieved for the largest dataset: $11.7\times$ faster than CUDPP and $6.5\times$ faster than NVBIO. Larger datasets could result in even greater speedup using 8 GPUs.

5.6 Conclusion

Our SA construction algorithm shows great performance in comparison to state-of-the-art CPU and previous single-GPU solutions. We were able to outperform all tested competitors on datasets larger than a 7M characters. Additionally, by utilizing multiple GPUs the implementation enables larger input sizes than previously possible. Using 8 GPUs the SA construction for a full-length human genome finished in 3.44 s, which is faster than previously reported 4.8 seconds achieved by employing 1600 cores on 100 nodes on a CPU-based HPC cluster.

Because the SA construction algorithm was implemented before Gossip's (Chapter 4) completion, the paper used a handcrafted version of Gossip's optimized unsplit all-to-all primitive for the DGX-1 system. It would have been beneficial for the project to rely on Gossip's library functions, which would allow it to be adapted to different systems by using Gossip's automatic transfer plan generation. Furthermore it would be interesting to see if Gossip's split all-to-all primitive could provide additional speedup. However, these topics are left as future work.

Part II

Metagenomic Classification

MetaCache Overview

6

Chapters 6-10 are based on and extent the following peer-reviewed papers:

A big data approach to metagenomics for all-food-sequencing.
Robin Kobus, José M. Abuín, André Müller, Sören Lukas Hellmann, Juan C. Pichel, Tomás F. Pena, Andreas Hildebrandt, Thomas Hankeln, Bertil Schmidt.
BMC Bioinformatics 21, 102 (2020).
https://doi.org/10.1186/s12859-020-3429-6

MetaCache-GPU: Ultra-Fast Metagenomic Classification.
Robin Kobus, André Müller, Daniel Jünger, Christian Hundt, Bertil Schmidt.
Proceedings of the 50th International Conference on Parallel Processing (ICPP 2021).
https://doi.org/10.1145/3472456.3472460

Abstract – The cost of DNA sequencing has dropped exponentially over the past decade, making genomic data accessible to a growing number of scientists. In bioinformatics, localization of short DNA sequences (reads) within large genomic sequences is commonly facilitated by constructing index data structures which allow for efficient querying of substrings. Recent metagenomic classification pipelines annotate reads with taxonomic labels by analyzing their k-mer histograms with respect to a reference genome database. CPU-based index construction is often performed in a preprocessing phase due to the relatively high cost of building irregular data structures such as hash maps. However, the rapidly growing amount of available reference genomes establishes the need for index construction and querying at interactive speeds. In this paper, we introduce MetaCache-GPU – an ultra-fast metagenomic short read classifier specifically tailored to fit the characteristics of CUDA-enabled accelerators. Our approach employs a novel hash table variant featuring efficient minhash fingerprinting of reads for locality-sensitive hashing and their rapid insertion using warp-aggregated operations. Our performance evaluation shows that MetaCache-GPU is able to build large reference databases in a matter of seconds, enabling instantaneous operability, while popular CPU-based tools such as Kraken2 require over an hour for index construction on the same data. In the context of an ever-growing number of reference genomes, MetaCache-GPU is the first metagenomic classifier that makes analysis pipelines with on-demand composition of large-scale reference genome sets practical.

6.1 Introduction

Recent years have seen a tremendous increase in the volume of data generated in the life sciences, especially propelled by the rapid progress of *next generation sequencing* (NGS) technologies [104]. High-throughput sequencers can produce massive amounts of short DNA strings (called reads) in a single run. This leads to large-scale datasets being processed in a wide range of bioinformatics applications. Furthermore, the cost of these technologies has been decreasing dramatically. The low sequencing cost per genome[1] enables widespread usage and renders population-scale projects feasible. Examples include the Earth BioGenome Project [60], metagenomics microbiome sequencing studies [55], and world-wide SARS-CoV-2 sequencing efforts [6].

However, the analysis of large sequencing datasets poses hard computational challenges. In particular, read mapping and read classification are performance-critical tasks, being initial stages required for manifold types of NGS analysis pipelines. The objective of read mapping is to determine the best mapping location(s) of each read in a given (set of) reference genome(s). A common paradigm to address this issue is to store and index reference genome sequences as sets of k-length substrings (called k-mers) [77]. The constructed index is queried for the k-mers in each read to find exact matches, which are further processed using a seed-and-extend approach.

Nevertheless, associated runtimes on common workstations remain high when processing reads at scale. Parallelization can be employed to reduce execution times but imposes additional challenges due to variable sizes of k-mer matches, large storage requirements of index data structures, and their associated irregular memory access patterns [94]. As a consequence, corresponding speedups on accelerators such as GPUs and FPGAs are often limited [38, 37]. Aforementioned difficulties are amplified in *metagenomic* read classification where a large number of reference genomes is considered; e.g., the recent NCBI RefSeq Release 202 contains 51,326 genomic sequences from 15,461 different species. Metagenomic read classification thus requires both the rapid construction and high throughput querying of large index data structures, since reference genome collections are subject to frequent change and continuous growth.

In summary, algorithmic design and implementations of bioinformatics applications struggle to keep pace with recent high-throughput sequencing techniques and their ever increasing data acquisition rates. In this work, we present MetaCache-GPU – a metagenomic read classification algorithm optimized for CUDA-enabled accelerators.

[1] Currently around US$1,000 per genome: see http://www.genome.gov/sequencingcosts

We demonstrate how to efficiently construct and query a k-mer index for large genome collections employing a novel massively parallel multi-bucket hash table. Furthermore, we leverage multiple GPUs to overcome storage limitations of the scarce video memory of a single GPU. The combination of both approaches leads to up to 69 and 72 times faster database builds and up to 153 and 6 times faster queries compared to the established CPU-based applications MetaCache [81] and Kraken2 [116], respectively. Furthermore, MetaCache-GPU's *on-the-fly* mode avoids saving and reloading the database, enabling nearly instantaneous operability. Thus, it allows for querying the database directly after construction and is up to 410 and 450 times faster than MetaCache and Kraken2, respectively.

The main contributions of this work are as follows.

- MetaCache's extension from the taxonomic labeling of bacterial reads to the detection and quantification of ingredients in food samples.
- Fast sequence processing from FASTA/FASTQ files to feed the MetaCache pipeline.
- A novel multi-value hash table that enables rapid and memory-frugal construction and querying of k-mer indices on GPUs.
- The corresponding GPU-based minhashing scheme and top candidate generation for data-parallel read classification.
- In-memory index construction that allows for on-the-fly classification pipelines avoiding intermediate disk I/O.
- Index distribution across multiple GPUs to support reference genome k-mer indices exceeding single GPU memory.

The rest of Part II is organized as follows. Section 6.2 provides necessary background on the topic of metagenomic classification. Related work is discussed in Section 6.3. The general MetaCache pipeline is described in Section 6.4. Chapter 7 shows how MetaCache can be applied in the detection and quantification of food ingredients (*All-Food-Sequencing*). Chapter 8 explores in detail various methods which we analyzed and optimized for fast execution. Initial sequence processing is examined in Section 8.1. The design of our proposed GPU-based MetaCache pipeline is detailed in Section 8.2. Its components, namely our GPU implementations of min-hashing, the multi-value hash table for k-mer indexing, segmented sort and top candidate generation are further examined in Sections 8.3, 8.4, 8.5 and 8.6, respectively. Chapter 9 evaluates performance of the complete MetaCache-GPU pipeline and Chapter 10 concludes.

6.2 Background

The analysis of the taxonomic composition of a sequenced environmental sample is a fundamental building block in metagenomic pipelines. The corresponding classification problem aims at assigning a suitable taxonomic label (e.g., a species or a genus) to a given NGS read. A traditional approach addressing this problem aligns each read to an annotated database of reference genome sequences.

More concretely, consider a collection of reference (genome) sequences $G = \{G_1, \ldots, G_n\}$ and a collection of short reads $R = \{R_1, \ldots, R_m\}$ with sequence lengths $|G_i| \gg |R_j|, \forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, m\}$. The objective of *metagenomic classification* is to identify the most likely genome in G that each read in R may originate from. Note that exact matches of a complete read to a reference substring is unlikely due to genomic variation and sequencing errors. Thus, partial and inexact matches must be considered. However, the corresponding measures can be compute-heavy; e.g., calculating the optimal semi-global alignment score with commonly used dynamic programming algorithms between a read R_j and a reference G_i exhibits a time complexity proportional to the product of sequence lengths; i.e., $\mathcal{O}(|G_i| \cdot |R_j|)$. Since $n \cdot m$ such alignments must be computed, corresponding runtimes would be prohibitively long.

More recent alignment-free tools can reduce the high complexity based on exact k-mer matching. In this approach, a k-mer index data structure (or database) is constructed in a preprocessing step. The index is usually based on a hash table that contains all distinct substrings of length *k* of each reference in G as keys and their corresponding locations as values. A read R_j is then classified by a look-up procedure, which extracts the set of all k-mers in R_j and subsequently queries it against the precomputed index. If the look-up returns matches, counters for the corresponding reference genomes are incremented. At the end of this procedure R_j can be classified based on high-scoring counters. Kraken [117] is a highly popular metagenomic classification tool following this approach.

However, the number of sequenced genomes is rapidly increasing. Thus, the index data structure storing the k-mers of each reference sequence can become exceedingly large. MetaCache [81] addresses this problem by applying a distinct subsampling technique called *minhashing* [7]. A minhashing filter with sketch size s selects those k-mers within a sequence G for a sketch $S(G)$ whose hash values are among the s smallest. A simple example using $k = 4$ and $s = 2$ looks as follows:

```
sequence:   ACTGACTG
4-mers:     ACTG      hash(ACTG) = 14
```

```
CTGA     hash(CTGA) =  8  <- select
 TGAC    hash(TGAC) =  7  <- select
  GACT   hash(GACT) = 11
   ACTG  hash(ACTG) = 14
```

When comparing two sequences which differ significantly in their length, as it is the case when comparing short reads to reference genomes, it is advantageous to construct a locality sensitive sketch representation which is constrained to a certain region (window) of the reference. Note that the Jaccard index of two sketches approximates the true Jaccard index of the whole sets of k-mers of the corresponding windows and hence can be used as proxy.

6.3 Related Work

Wood and Salzberg [117] were among the first to demonstrate that a k-mer-based exact matching approach can achieve high metagenomic classification accuracy by developing Kraken while being around three orders-of-magnitude faster than the alignment tool MegaBLAST [86]. Recent benchmark studies [66, 101] demonstrated that k-mer based tools such as Kraken [117], Kraken2 [116], CLARK [87], and MetaCache [81] can produce superior read assignment accuracy for selected bacterial metagenomic datasets. While being accurate, the major drawback of the k-mer based approach is high main memory consumption and long database construction times. For medium-sized bacterial reference genome sets the databases used by Kraken and CLARK already consume several hundreds of gigabytes in size. MetaCache and Kraken2 reduce the k-mer index memory consumption by around one order-of-magnitude by using different k-mer subsampling techniques (minhashing resp. minimizers). Nevertheless, the rapidly increasing amount of available bacterial reference genomes and the significantly higher complexities of eukaryotic reference genomes relevant for applications such as the monitoring of food ingredients (see Chapter 7) demand scalable solutions that can build and query k-mer indices at higher speed than current approaches. In this work, we investigate how modern multi-GPU systems can be used to accelerate MetaCache. Note that the GPU-based k-mer index structure introduced in this work can be easily applied to related bioinformatics tasks such as read mapping [94] or long-read-to-long-read alignment [18].

There has been a limited amount of prior work on using GPUs for metagenomic read classification. cuCLARK [49] accelerates CLARK using CUDA but only achieves speedups between 3.2 and 6.6 while keeping the high memory consumption of CLARK. MetaBinG2 [92] applies a hidden Markov model to estimate the distance of a read to organisms but is over an order-of-magnitude slower compared to CLARK. Usage of k-mer indices on GPUs has also been investigated for the related problem of aligning (or mapping) reads to a single genome. However, most approaches are based on accelerating popular short read aligners such as BWA-MEM or Bowtie2 adopting the FM-index and Burrow-Wheeler-Transform (BWT) [69, 12]. While this data structure can be memory efficient, k-mer querying requires iterative lookups with irregular memory accesses. Thus, reported speedups are relatively low; e.g. [38] only reports a speedup of 2 on a GPU compared to CPU-based BWA-MEM. None of these papers consider the acceleration of k-mer index construction. k-mer histogram generation is another task that relies on such k-mer index data structures and has been studied for GPUs [62, 10, 19]. However, counting k-mer occurrences is a far simpler task compared to the construction of a reference index, i.e. a multi-value key-value store, which is needed for metagenomic classification. For MetaCache-GPU, we introduce a novel multi-value hash table variant optimized for memory-efficient k-mer index construction and querying on multiple GPUs. See Section 8.4 for a detailed discussion.

6.4 MetaCache Pipeline

The workflow of MetaCache can be separated into two distinct phases: *build* and *query*. First, a reference database must be build. Then, reads from metagenomic samples are classified after querying the database. Typical for index based methods, the default MetaCache pipeline is split into separate program calls for build and query, where the database has to be saved to and loaded from disk. Here, querying can be executed in different modes, either a single run processing all supplied input files or an interactive session, which holds the database in memory and allows for performing an arbitrary number of queries in succession. Additionally, we extended MetaCache with a novel on-the-fly mode where queries can be executed directly after building the database before writing it to the file system. To the best of our knowledge, lightweight in-memory queries are unprecedented in the literature since index construction times have been prohibitively long for competing, non-GPU-based solutions. Figure 6.1 shows an overview of the build and query phases which are further described in detail in the following.

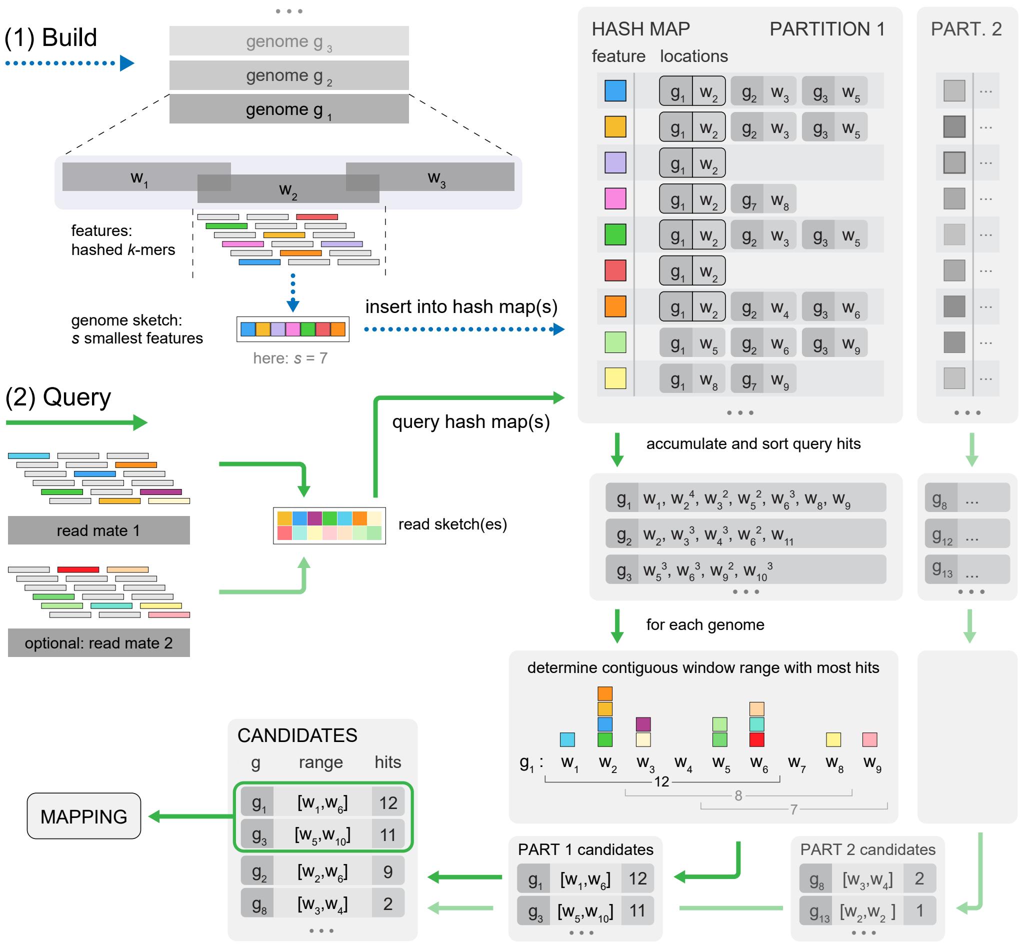


Fig. 6.1.: Workflow: (1) Database construction: Each reference genome g_i is partitioned into slightly overlapping windows w_j. The s smallest k-mer hashes of each window are inserted into the database. (2) Classification: a database is queried with the s smallest k-mer hashes of a read (or read pair). Subsequent counting of hits within each window results in a list of mapping candidates. In the case of several database partitions, the candidates from distinct partition are merged in order to assign a read to the most likely taxon of origin.

6.4.1 Build Phase

Building a reference database starts by generating a taxonomic tree containing the relations of the considered reference genomes. The nodes and structure of the tree are extracted from files provided by NCBI [98]. Next, a hash table is constructed that maps k-mers (keys) to genome locations (values). Here, MetaCache employs a producer-consumer strategy to fill a concurrent queue with batches of reference sequences. Multiple producer threads parse the genome files to split the data into header and sequence strings which are then pushed into the queue. A consumer thread dequeues a batch of sequences and processes one sequence after another. First it obtains the genomic identifier from the header to create a reference target which is then connected to the taxonomic tree structure. Next, it proceeds to process the corresponding sequence data by dividing it into windows of size w overlapping by $k-1$ base-pairs for a k-mer length of k. For each window all $w-k+1$ canonical k-mers are generated and a hash function h_1 is applied to each. The s smallest hash values are selected as features and form the minhashing *sketch*, which is used to represent the window in the database. Sketches are collected in batches and inserted into a different concurrent queue which is consumed by another thread. This thread is responsible for inserting the features of each sketch together with its location information (target and window) into the hash table.

In the CPU version the hash table uses open addressing where each slot maps a feature to a bucket of reference locations. To counteract the biased distribution of sketch values a second hash function h_2 is applied in order to determine the key slot in the hash table. If the computed key slot is empty a new bucket is created for the feature and the value is inserted. If the feature is already present in the slot the value is appended to the corresponding bucket. If the slot is occupied by a different feature a quadratic probing scheme is used to find the next slot where inserting is tried again. Since the distribution of locations per k-mer is usually highly skewed (a large fraction of k-mers occur only once while few occur many times) the buckets can grow dynamically using a geometric growth scheme. Additionally, the maximum number of locations stored per k-mer is limited to a pre-defined value (254 per default).

Although multiple threads are used in this pipeline, MetaCache's default hash table does not support concurrent insertion. It uses a dynamic allocation strategy which allows it to grow if the load factor exceeds a user defined limit. In this case a new hash table of larger size is allocated into which all feature-bucket mappings are re-inserted while the buckets holding the values are preserved. Therefore, the CPU version of MetaCache is limited to a single thread operating the hash table in the build phase. However, the benefit is, that the locations in each bucket remain sorted throughout the whole process due to the ascending target and window IDs assigned by the sketching thread. This allows linear-time merging of multiple location lists into one sorted list, as is required in the query phase.

After database construction has finished, the taxonomic meta information as well as the hash table are written to the file system.

6.4.2 Database Partitioning

Due to the ever increasing size of reference genome collections, the memory of a single workstation might not be sufficient for MetaCache to create a database for all reference sequences at once. Therefore, it is possible to partition the references genomes into separate databases which allows to save memory either while building or while querying or during both phases.

Partitioning of the reference genomes can be done as a preprocessing step followed by successive distinct builds thus minimizing memory usage. Alternatively, if the workstation used for building has enough resources, MetaCache can also build database partitions in parallel.

Partitioned databases can be queried sequentially using independent query runs followed by a merge step to obtain the final classification result. Alternatively, all or a subset of partitions can be loaded at the same time and be queried in parallel. Furthermore, partitioned databases generated by the CPU version of MetaCache can be loaded by the GPU version and vice versa as long as no database part exceeds the memory of a single GPU.

6.4.3 Query Phase

If MetaCache is not executed in on-the-fly query mode following the build phase, the database has to be read from disk before reads can be queried. Here a condensed form of the hash table is used where all buckets of target locations are loaded into one large contiguous array which can be accessed by pointers. The on-the-fly mode uses the hash table from the build phase as is. In addition, an acceleration structure is generated from the taxonomic tree that contains the taxonomic lineage of each target in the database thus allowing to compute the lowest common ancestor of two taxa in constant time during classification.

Similar to the build phase, a concurrent queue is used to distribute work among multiple threads. One thread is responsible for reading sequences from input files and separating header from sequence data. Batches of sequences are inserted into the queue. Multiple threads consume the batches from the queue and process all contained reads. To classify a read, its sequence is first split into windows of the same length as used in the database. From each window all canonical k-mers are generated and minhashing is applied to produce a sketch. All elements of the sketch are then queried against the hash table. The resulting location lists are merged and identical locations are accumulated. This yields a (sparse) histogram of hit counts per window in the reference genomes (*window count statistic*) which indicate the similarity of this region with the read.

To account for single-end or paired-end reads spanning multiple windows the window count statistic is scanned with a sliding window approach to find target regions with the highest aggregated hit counts in a contiguous window range. The top m counts (*top hits*) are then used to classify the read. Usually $2 \leq m \leq 4$ top hits are enough to achieve a reliable classification. If the difference of the highest and second highest count is above a threshold, the read is labeled as belonging to the taxon of the genome corresponding to the maximum count. Otherwise, all targets with counts close to the maximum are considered, the lowest common ancestor of the corresponding taxa is calculated and used to label the read.

6.4.4 Coverage Filter

False positive read assignments can be caused by shared regions of DNA among multiple reference genomes [16]. MetaCache is able to use coverage information to detect some of these cases as follows.

Before assigning reads to classification targets we can filter the list of candidate genomes identified during the read assignment phase by checking the coverage per genome as follows. We analyze which windows of a target genome are covered by reads from the dataset. If the percentage of covered windows of a genome is much lower compared to other genomes, it is likely to be a false positive and will be deleted from the list of possible target genomes. In fact we delete a quantile (e.g. 10%) of the target genomes with the lowest coverage. The reads are then classified with respect to the remaining genomes.

Note that this strategy is only applicable if the number of reads is large enough to cover significant parts of the genomes. In our experience it proofed especially efficient in case of bacterial genomes which are orders of magnitudes smaller than animal or plant genomes.

6.4.5 Quantification

In addition to the per-read classification we are able to estimate the abundances of organisms contained in a dataset at a specific taxonomical rank. For each taxon which occurs in the dataset we count the number of reads assigned to it. We then build a taxonomic tree containing all found taxa.

Taxa on lower levels than the requested taxonomic rank are pruned and their read counts are added to their respective parents, while reads from taxa on higher levels are distributed among their children in proportion to the weights of the sub-trees rooted at each child. After the redistribution the estimated number of reads and abundance percentages are returned as outputs.

7 MetaCache in All-Food-Sequencing

Abstract – All-Food-Sequencing (AFS) is an untargeted metagenomic sequencing method that allows for the detection and quantification of food ingredients including animals, plants, and microbiota. While this approach avoids some of the shortcomings of targeted PCR-based methods, it requires the comparison of sequence reads to large collections of reference genomes. The steadily increasing amount of available reference genomes establishes the need for efficient big data approaches.

We show how MetaCache's alignment-free method can be applied for detection and quantification of species composition in food and other complex biological matters. It is orders-of-magnitude faster than the previous alignment-based AFS pipeline. In comparison to the established tools CLARK, Kraken2, and Kraken2+Bracken it is superior in terms of false-positive rate and quantification accuracy. Furthermore, the usage of an efficient database partitioning scheme allows for the processing of massive collections of eukaryotic and bacterial reference genomes with reduced memory requirements. In summary, MetaCache is a suitable tool for broad-scale metagenomic screening applications. Information on how to adjust MetaCache's settings for the AFS context is available at `https://muellan.github.io/metacache/afs.html`.

7.1 Background

Monitoring of food ingredients is becoming an increasingly important task. Relevant issues include correct labeling, fraud detection, and assessment of health risks [20]. This motivates the need for analytical methods that allow for accurate determination and quantification of food ingredients ideally spanning all kingdoms of life including animals, plants, bacteria, fungi, and possibly even viruses.

Quantitative real-time polymerase chain reaction (qPCR) [53] and droplet digital PCR (ddPCR) [52] are DNA-based technologies for food control that are widely used in practice. Unfortunately, these methods are limited by the number of target species within a single assay and thus are not suitable for broad-scale species screening.

Similar restrictions apply to approaches based on sequencing of species-specific DNA bar codes [110].

High-throughput sequencing of total metagenomic DNA from biological samples provides the possibility to screen for a wide range of species as it does not require any prior definition of possible target species. However, subsequent bioinformatic analysis of large amounts of sequence-reads is required to identify and quantify actual food components. The All-Food-Seq (AFS) pipeline [95, 70] maps each sequenced read to a number of reference genomes and then determines species composition and relative quantities based on a read counting procedure. Evaluation based on simulated as well as real data has demonstrated that AFS can detect anticipated species in food products and achieve quantification accuracy comparable to qPCR. However, the AFS pipeline relies on applying a read alignment tool (such as BWA [65, 64, 63], Bowtie2 [58], or CUSHAW [71]) for each considered reference genome. Thus, runtime scales linearly with the number of considered genomes. For example, the quantification of a typical short read dataset consisting of a few million reads using ten mammalian and avian reference genomes with the BWA-based AFS pipeline already requires several hours on a standard workstation (not including the time for index construction). For broader scale screening of many species a much larger amount of reference genomes would be required, making this approach unfeasible.

Recent benchmark studies [66, 101] demonstrated that k-mer based tools such as Kraken [117], Kraken2+Bracken [74], CLARK [87], and MetaCache [81] can produce superior read assignment accuracy compared to several other tools including MetaPhlAn [111], mOTU [106], QIIME [11], and Kaiju [79] for selected bacterial metagenomic datasets. While being accurate, the major drawback of the k-mer based approach is high main memory consumption and long database construction times. For typical bacterial reference genome sets the databases used by Kraken and CLARK already consume several hundreds of gigabytes in size. The significantly higher complexities of eukaryotic reference genomes relevant for monitoring food ingredients therefore make an extension of this method to food-monitoring challenging.

Here, we apply MetaCache for broad-scale detection and quantification of species composition in food and other complex biological matters. MetaCache proves to be a superior substitute for the alignment tools previously employed in the AFS pipeline. Our experimental results using a number of sequenced calibrator sausages of known species composition show that MetaCache runs orders-of-magnitude faster than the alignment-based AFS pipeline while yielding similar results. Furthermore, MetaCache yields lower false-positive rates and higher quantification accuracy compared to Kraken2, Kraken2+Bracken, and CLARK. It also provides faster database construction times and competitive query speeds. Our database partitioning scheme allows the reduction of peak main memory consumption on a single workstation or a cluster node significantly and therefore enables scalability to growing genome collections.

7.2 Evaluation

7.2.1 Datasets

In order to measure performance and accuracy of our approach in comparison to other metagenomic tools, we have created databases of varying size containing different organisms. Food-related genomes (selection of main ingredients) used for database construction are listed in Table 7.2 while the considered bacteria, viruses, and archaea from NCBI RefSeq (Release 90) are summarized in Table 7.1. The created databases with their included reference genomes are described in Table 7.3.

We use ten short read datasets sequenced from calibrator sausage samples containing admixtures of a set of food relevant ingredients (chicken, turkey, pork, beef, horse, sheep) on an Illumina HiSeq machine (downloaded from ENA project ID PRJNA271645 (Kal_D and KAL_D) and PRJEB34001 (all other data)). Table 7.4 shows the read datasets together with the corresponding percentage of meat components used during preparation. The samples comprise meat proportions ranging from 0.5% to 80% and can be subdivided into two categories: Kal A-E consist only of mammalian meat, while KLyo A-D represent Lyoner-like sausages containing poultry in addition to mammals [54, 21]. The dataset KAL_D is identical to Kal_D but sequenced with higher coverage.

Tab. 7.1.: Reference genomes from NCBI RefSeq (Release 90) used for database construction.

Organism	Number of references	Size on disk
bacteria	10838	41.0 GB
viral	7857	269 MB
archaea	269	656 MB
Total	**18964**	**41.9GB**

Tab. 7.2.: Food-related reference genomes used for database construction.

Item	Name	ID	File size
1	Sus scrofa (pig)	GCF_000003025.6	2.4GB
2	Equus caballus (horse)	GCF_002863925.1	2.4GB
3	Meleagris gallopavo (turkey)	GCF_000146605.2	1.2GB
4	Mus musculus (house mouse)	GCF_000001635.26	2.7GB
5	Gallus gallus (chicken)	GCF_000002315.5	1.1GB
6	Ovis aries (sheep)	GCF_000298735.2	2.5GB
7	Rattus norvegicus (Norway rat)	GCF_000001895.5	2.8GB
8	Bos taurus (cattle)	GCF_002263795.1	2.6GB
9	Bubalus bubalis (water buffalo)	GCF_003121395.1	2.6GB
10	Cervus elaphus hippelaphus (red deer)	GCA_002197005.1	3.3GB
11	Capreolus capreolus (Western roe deer)	GCA_000751575.1	3.0GB
12	Struthio camelus australis (African ostrich)	GCA_000698965.1	1.2GB
13	Anas platyrhynchos (mallard)	GCF_003850225.1	1.1GB
14	Capra hircus (goat)	GCF_001704415.1	2.8GB
15	Oryctolagus cuniculus (rabbit)	GCF_000003625.3	2.6GB
16	Cavia aperea (Brazilian guinea pig)	GCA_000688575.1	2.6GB
17	Camelus ferus (Wild Bactrian camel)	GCF_000311805.1	1.9GB
18	Canis lupus familiaris (dog)	GCF_000002285.3	2.3GB
19	Felis catus (domestic cat)	GCF_000181335.3	2.4GB
20	Homo sapiens (human)	GCF_000001405.38	3.1GB
21	Equus asinus (ass)	GCA_001305755.1	2.3GB
22	Rangifer tarandus (reindeer)	GCA_004026565.1	2.9GB
23	Phasianus colchicus (Ring-necked pheasant)	GCA_004143745.1	987MB
24	Glycine max (soybean)	GCF_000004515.5	946MB
25	Zea mays (maize)	GCF_000005005.2	2.1GB
26	Triticum aestivum (bread wheat)	GCA_900519105.1	14.0GB
27	Secale cereale (rye)	GCA_900079665.1	1.8GB
28	Hordeum vulgare (barley)	GCA_004114815.1	3.8GB
29	Oryza sativa Japonica Group (Japanese rice)	GCF_001433935.1	362MB
30	Arachis hypogaea (peanut)	GCF_003086295.1	2.4GB
31	Saccharomyces cerevisiae S288C (baker's yeast)	GCA_000146045.2	12MB
Total			**74GB**

Tab. 7.3.: Data sets used for database construction.

Name	Number of species	Size on disk
AFS10	Food genomes 1 to 10	22.3GB
AFS20	Food genomes 1 to 20	45.8GB
AFS20RS90	Food genomes 1 to 20 plus NCBI RefSeq (Release 90)	87.5GB
AFS31	Food genomes 1 to 31	76.8GB
AFS31RS90	Food genomes 1 to 31 plus NCBI RefSeq (Release 90)	118.5GB

Tab. 7.4.: Calibrator sausage datasets and their meat composition.

Name	#Reads (paired-end)	Cattle	Sheep	Pig	Horse	Chicken	Turkey
KLyo_A	401K	14.0%	0.0%	80.0%	0.0%	0.5%	5.5%
KLyo_B	302K	36.0%	0.0%	58.0%	0.0%	2.0%	4.0%
KLyo_C	507K	58.0%	0.0%	36.0%	0.0%	4.0%	2.0%
KLyo_D	417K	80.0%	0.0%	14.0%	0.0%	5.5%	0.5%
Kal_A	830K	1.0%	9.0%	35.0%	55.0%	0.0%	0.0%
Kal_B	977K	9.0%	1.0%	55.0%	35.0%	0.0%	0.0%
Kal_C	404K	25.0%	25.0%	25.0%	25.0%	0.0%	0.0%
Kal_D	403K	35.0%	55.0%	9.0%	1.0%	0.0%	0.0%
Kal_E	289K	55.0%	35.0%	1.0%	9.0%	0.0%	0.0%
KAL_D	26,114K	35.0%	55.0%	9.0%	1.0%	0.0%	0.0%

7.2.2 Quantification Accuracy

Tables 7.5 and 7.6 show the quantification results returned by the tested tools (MetaCache (v.0.5.3), CLARK (v.1.2.6), Kraken2 (v.2.0.7-beta), and Kraken2 with subsequent abundance estimation by Bracken v.2.0.0) using AFS20 as reference database. Besides showing the quantification for each included meat component, we also show the (false positive) results for water buffalo (closely related to cattle) and goat (closely related to sheep). In addition, we provide the sum of all false positive (Σ FP) read classifications over all of the detected reference genomes that were not included in the sample. In addition, the sum of the deviations of the measured proportions to the real sausage composition (Σ Dev) as well as the averages over all tested datasets are shown.

In terms of sensitivity, all methods are able to detect the included meat components. In addition, several tools detect false positive signals; e.g., Kraken2+Bracken detects over 1% of water buffalo in KLyo_C and KLyo_D and over 3% of goat in Kal_C, Kal_D, and Kal_E. False positive quantities in these cases correlate with the amount of beef and the amount of sheep present in the respective sample. Overall, MetaCache achieves the lowest FP-rates for each tested dataset with an average FP-sum per sample of only 0.67% for the Klyo samples and 1.12% for the Kal samples. This is much lower compared to CLARK (1.34% for Klyo, 3.59% for Kal), Kraken2 (1.86% for Klyo, 3.87% for Kal), and Kraken2+Bracken (2.14% for Klyo, 4.41% for Kal). The relative differences become even more significant when looking at some of the individual FP signals. In the Klyo samples (Table 7.5) MetaCache only detects negligible amounts of goat (0.05% on average) and water buffalo (0.14%), while the amounts detected by CLARK, Kraken2, and Kraken2+Bracken are higher by

Tab. 7.5.: Quantification results for the Klyo samples using the reference dataset AFS20 and the average result for AFS31RS90. MC: MetaCache, K2+Brack: Kraken2 with subsequent Bracken, W.Buf: Water Buffalo, Σ FP: Sum of all false positive read classifications, Σ Dev: Sum of absolute deviations to the given meat composition (best results for each dataset in bold).

Dataset	Classifier	Cattle	Pig	W.Buf.	Goat	Chicken	Turkey	Σ FP	Σ Dev
KLyo_A	Expected	14.0%	80.0%	0.00%	0.00%	0.50%	5.50%		
	MC	16.6%	71.5%	0.04%	0.02%	0.60%	4.64%	**0.28**%	**12.39**%
	CLARK	16.4%	70.4%	0.20%	0.09%	0.62%	4.61%	0.51%	13.55%
	Kraken2	15.9%	70.0%	0.27%	0.11%	0.65%	4.59%	0.87%	13.82%
	K2+Brack	17.6%	70.3%	0.30%	0.14%	0.66%	4.63%	0.97%	15.33%
KLyo_B	Expected	36.0%	58.0%	0.00%	0.00%	2.00%	4.00%		
	MC	37.6%	51.0%	0.12%	0.04%	2.05%	2.99%	**0.50**%	10.16%
	CLARK	35.9%	50.4%	0.47%	0.19%	2.10%	3.01%	1.03%	**9.84**%
	Kraken2	34.5%	49.9%	0.68%	0.24%	2.12%	2.99%	1.57%	12.11%
	K2+Brack	39.1%	50.2%	0.32%	0.78%	2.15%	3.02%	1.84%	13.93%
KLyo_C	Expected	58.0%	36.0%	0.00%	0.00%	4.00%	2.00%		
	MC	57.7%	27.1%	0.16%	0.06%	3.56%	1.16%	**0.95**%	**11.47**%
	CLARK	54.1%	25.9%	0.69%	0.29%	3.58%	1.16%	1.88%	17.11%
	Kraken2	52.2%	25.7%	0.95%	0.36%	3.57%	1.17%	2.58%	19.94%
	K2+Brack	58.6%	25.8%	1.07%	0.46%	3.60%	1.18%	2.89%	14.90%
KLyo_D	Expected	80.0%	14.0%	0.00%	0.00%	5.50%	0.50%		
	MC	74.7%	10.9%	0.23%	0.08%	4.66%	0.33%	**0.93**%	10.27%
	CLARK	70.8%	10.8%	0.94%	0.39%	4.73%	0.35%	1.94%	15.27%
	Kraken2	68.0%	10.7%	1.26%	0.48%	4.70%	0.36%	2.42%	18.62%
	K2+Brack	77.6%	10.8%	1.45%	0.62%	4.76%	0.36%	2.87%	**9.35**%
Average	MC			**0.14**%	**0.05**%			**0.67**%	**11.07**%
	CLARK			0.58%	0.24%			1.34%	13.94%
	Kraken2			0.79%	0.30%			1.86%	16.12%
	K2+Brack			0.71%	0.50%			2.14%	13.38%
AFS31RS90 Average	MC							**0.58**%	**13.97**%

factors of 4.2 and 4.8, 5.6 and 6.0, and 5.1 and 10.0, respectively. Similar results can be observed for the Kal samples (Table 7.6): MetaCache only detects 0.07% of water buffalo meat on average and 0.80% of goat meat on average, while the amounts detected by CLARK, Kraken2, and Kraken2+Bracken are higher by factors of 7.3 and 3.3, 8.3 and 3.5, and 9.4 and 4.0, respectively.

In terms of deviation from the expected foodstuff ingredients, MetaCache shows the lowest average of the sums of absolute differences for both Klyo (11.07%) samples and Kal samples (10.56%). Kraken2+Bracken (13.38% and 12.74%) has smaller deviations on average than Kraken2 alone (16.12% and 16.77%), showing that quantification after read assignment is beneficial.

When scanning the calibrator sausage read datasets with MetaCache using the bigger AFS31 and AFS31RS90 databases, we can make the following observations:

Tab. 7.6.: Quantification results for the Kal samples using the reference dataset AFS20 and the average result for AFS31RS90. MC: MetaCache, K2+Brack: Kraken2 with subsequent Bracken, W.Buf: Water Buffalo, Σ FP: Sum of all false positive read classifications, Σ Dev: Sum of absolute deviations to the given meat composition (best results for each dataset in bold).

Dataset	Classifier	Cattle	Sheep	Pig	Horse	W.Buf.	Goat	Σ FP	Σ Dev
Kal_A	Expected	1.00%	9.0%	35.0%	55.0%	0.00%	0.00%		
	MC	1.25%	11.0%	30.5%	54.1%	0.01%	0.29%	**0.42**%	8.13%
	CLARK	1.29%	9.1%	31.1%	54.0%	0.09%	0.89%	1.15%	**6.43**%
	Kraken2	1.23%	8.7%	30.9%	53.9%	0.08%	0.96%	1.31%	6.99%
	K2+Brack	1.43%	10.3%	31.0%	54.0%	0.10%	1.12%	1.53%	8.24%
Kal_B	Expected	9.0%	1.00%	55.0%	35.0%	0.00%	0.00%		
	MC	10.5%	1.42%	49.3%	35.6%	0.03%	0.06%	**0.27**%	8.43%
	CLARK	10.3%	1.26%	50.0%	35.8%	0.17%	0.18%	0.56%	**7.85**%
	Kraken2	10.0%	1.21%	49.6%	35.7%	0.20%	0.20%	1.03%	8.40%
	K2+Brack	11.0%	1.40%	35.8%	49.7%	0.22%	0.23%	1.09%	9.60%
Kal_C	Expected	25.0%	25.0%	25.0%	25.0%	0.00%	0.00%		
	MC	23.3%	29.6%	19.2%	23.0%	0.06%	0.73%	**1.08**%	15.28%
	CLARK	23.4%	25.6%	19.4%	23.2%	0.45%	2.56%	3.38%	**12.98**%
	Kraken2	22.7%	24.7%	19.4%	23.1%	0.49%	2.69%	3.48%	13.65%
	K2+Brack	24.8%	27.8%	19.4%	23.2%	0.54%	3.02%	3.89%	14.35%
Kal_D	Expected	35.0%	55.0%	9.00%	1.00%	0.00%	0.00%		
	MC	32.9%	51.5%	7.14%	1.14%	0.09%	1.50%	**2.07**%	**9.62**%
	CLARK	32.8%	43.1%	7.31%	1.16%	0.72%	4.40%	5.69%	21.61%
	Kraken2	31.6%	41.3%	7.26%	1.16%	0.79%	4.62%	5.77%	24.75%
	K2+Brack	35.8%	48.4%	7.28%	1.16%	0.89%	5.40%	6.70%	15.96%
Kal_E	Expected	55.0%	35.0%	1.00%	9.00%	0.00%	0.00%		
	MC	50.4%	33.7%	0.99%	7.80%	0.12%	0.96%	**1.52**%	**8.55**%
	CLARK	50.7%	28.7%	1.02%	7.81%	0.84%	3.07%	4.43%	16.26%
	Kraken2	49.2%	27.6%	1.00%	7.80%	0.99%	3.28%	4.58%	18.96%
	K2+Brack	54.1%	31.4%	1.00%	7.81%	1.10%	3.71%	5.15%	10.86%
KAL_D	Expected	35.0%	55.0%	9.00%	1.00%	0.00%	0.00%		
	MC	30.3%	49.6%	7.27%	1.16%	0.08%	1.25%	1.38%	**13.36**%
	CLARK	30.8%	43.3%	7.51%	1.20%	0.86%	4.57%	6.30%	23.85%
	Kraken2	29.6%	41.3%	7.47%	1.19%	0.95%	4.98%	7.03%	27.86%
	K2+Brack	33.5%	48.7%	7.58%	1.19%	1.08%	5.84%	8.07%	17.44%
Average	MC					**0.07**%	**0.80**%	**1.12**%	**10.56**%
	CLARK					0.51%	2.61%	3.59%	14.83%
	Kraken2					0.58%	2.79%	3.87%	16.77%
	K2+Brack					0.66%	3.22%	4.41%	12.74%
AFS31RS90 Average	MC							**1.84**%	**13.38**%

(1) More k-mers are removed from the hash table due to overflowing target lists. Therefore, the number of classified reads is reduced and total deviation increases slightly. (2) Additional false positive targets are introduced, but the total number of false positives is reduced for the Klyo datasets (excluding bacteria).

A benefit of screening for microbiota and eukaryotic foodstuff species at the same time is a lower false positive rate. Usually reads of a dataset are queried against either one or the other and only the remaining unclassified reads are investigated further. This can lead to false assumptions about the data. In our experiments some reads are falsely classified as Triticum aestivum (bread wheat) when using the AFS31 database. With the AFS31RS90 database, however, those reads are identified as bacterial or unspecific (classified as the lowest common ancestor of bread wheat and bacteria).

7.2.3 Runtime and Memory Consumption

Non-Partitioned Databases

Runtime and memory consumption where the whole database can fit into the available main memory are measured on a system with a dual Xeon E5-2630v4 (2.2 GHz, 2×10 cores) CPU with 512 GB of DDR4 RAM. We have compared the speed and the peak memory consumption during database construction and classification of the default versions of MetaCache (v.0.5.3), CLARK (v.1.2.6), Kraken2 (v2.0.7-beta), and Kraken2 with subsequent abundance estimation by Bracken v.2.0.0 (Kraken2+Bracken) using 40 threads. Table 7.7 shows the results for the reference genome datasets listed in Table 7.3 and the KAL_D read dataset (26 million paired-end reads of length 101 bp) for classification. Note, that the time to load the databases is excluded when measuring query speed for all programs to make the results independent of dataset size.

MetaCache is fastest for database construction for all tested data sets. Furthermore, it requires least memory for constructing the database for AFS20 and AFS31, but requires slightly more memory than Kraken2 for AFS20RS90 and AFS31RS90.

Kraken2 is fastest in terms of query (classification) speed. If Kraken2 is executed with subsequent quantification by Bracken, corresponding runtimes increase. Even though query speeds of MetaCache-AFS are slowest, corresponding execution times are still competitive (only around three minutes for the largest data set (KAL_D)).

Tab. 7.7.: Runtimes and peak memory consumption for non-partitioned database construction (build) and querying for different data sets on a workstation with 512 GB RAM. Query speeds are measured for the KAL_D dataset in terms of million reads per minute (MR/m). For the cases with "-" the corresponding program exceeds the main memory capacity of 512 GB. Fastest runtimes and lowest memory consumption for each dataset are indicated in bold.

Data set		MetaCache	CLARK	Kraken2	Kraken2+Bracken
AFS20	Build time	**1h 11m**	15h 37m	1h 27m	5h 32m
	Build memory	**64 GB**	428 GB	69 GB	147 GB
	Query time	136 s	93 s	**37 s**	111 s
	Query speed	11.5 MR/m	16.9 MR/m	**43.2 MR/m**	14.2 MR/m
	Query memory	**50 GB**	152 GB	54 GB	54 GB
AFS31	Build time	**1h 47m**	-	3h 19min	11h 41min
	Build memory	**91 GB**	-	107 GB	296 GB
	Query time	175 s	-	**44 s**	58 s
	Query speed	8.9 MR/m	-	**35.9 MR/m**	27.0 MR/m
	Query memory	78 GB	-	**72 GB**	**72 GB**
AFS20RS90	Build time	**1h 42m**	-	2h 58m	8h 53m
	Build memory	110 GB	-	**94 GB**	168 GB
	Query time	180 s	-	**43 s**	117 s
	Query speed	8.7 MR/m	-	**37.0 MR/m**	13.5 MR/m
	Query memory	94 GB	-	**79 GB**	**79 GB**
AFS31RS90	Build time	**3h 10m**	-	5h 55min	17h 44min
	Build memory	135 GB	-	**134 GB**	329 GB
	Query time	217 s	-	**49 s**	61 s
	Query speed	7.2 MR/m	-	**32.1 MR/m**	25.7 MR/m
	Query memory	117 GB	-	**97 GB**	**97 GB**

For common data set sizes in food control applications runtimes for database construction (a few hours) are typically much higher than for the classification stage (a few minutes). Since the amount of relevant reference genomes is increasing rapidly corresponding databases have to be constructed or extended frequently. Thus, fast built times are of high importance. Besides having the fastest database construction time, MetaCache is also the only tool that supports the functionality of extending an existing database.

Partitioned Databases

In this subsection we evaluate the ability of MetaCache to reduce the consumed main memory by partitioning the database into smaller chunks. MetaCache is again evaluated on a workstation with a dual Xeon E5-2630v4 CPU and 512 GB of DDR4 RAM.

Table 7.8 shows the speed and memory consumption of MetaCache for partitioned database construction and querying using the AFS31RS90 reference genome dataset

Tab. 7.8.: Partitioned build time and query speed for AFS31RS90 database. Query speed measured for dataset KAL_D in million reads per minute (MR/m).

Tool	Build time	Max. RAM	Query Speed	Max. RAM
MetaCache (1 part.)	3h 10min	135 GB	7.2 MR/m	117 GB
MetaCache (2 part.)	3h 04min	82 GB	3.1 MR/m	70 GB
MetaCache (4 part.)	3h 45min	52 GB	2.5 MR/m	39 GB

Tab. 7.9.: Runtimes and peak memory consumption for database construction (build) and querying for AFS10. Query speeds are measured for the KAL_D dataset in terms of million reads per minute (MR/m).

Data set		MetaCache	AFS-previous
AFS10	Build time	**47m**	7h 0m
	Build memory	35 GB	**5GB**
	Query speed	**17.1 MR/m**	0.04 MR/m
	Query memory	30 GB	**6GB**

and the KAL_D dataset. Using four partitions, MetaCache can reduce the main memory consumption from 135 GB to only 52 GB while the construction time only slightly increases from 3h 10m to 3h 45m. In addition, memory consumption for classification is reduced from 117 GB to 39 GB. However, the corresponding query speed decreases from 7.2 MR/m to 2.5 MR/m since the partitions have to be queried by all reads in succession and the individual results need to be merged.

7.2.4 Comparison to Previous AFS Pipeline

To compare MetaCache to the previous alignment-based AFS pipeline the same dual-socket workstation as before is used. Runtimes and memory consumption of both approaches are shown in Table 7.9. For the small genome dataset AFS10 the previous AFS pipeline already takes several hours to construct the index. Querying of the KAL_D dataset takes even more than 10 hours. For bigger numbers of reference genomes this approach becomes unfeasible because the runtime scales linearly with the number reference genomes. On the other hand, MetaCache takes less than an hour for database construction of AFS10 while the query speed improves by more than two orders of magnitude. As shown before even larger databases like AFS31 can be built by MetaCache in just a few hours and query speed drops by less than a factor of two.

The average quantification results for the Klyo and Kal samples produced by MetaCache and the previous AFS pipeline are shown in Table 7.10. The k-mer based

Tab. 7.10.: Average quantification results for the Klyo and Kal samples using the reference dataset AFS10. AFS-prev: previous AFS pipeline, Σ FP: Sum of all false positive read classifications, Σ Dev: Sum of absolute deviations to the given meat composition.

Dataset	Classifier	Σ FP	Σ Dev
KLyo Average	MetaCache	**0.37%**	**10.71%**
	AFS-prev	**0.37%**	10.80%
Kal & KAL_D Average	MetaCache	**0.19%**	8.43%
	AFS-prev	0.33%	**6.65%**

Tab. 7.11.: Detected bacteria in dataset KLyo_C using reference dataset AFS31RS90. Genera with less 500 than hits ($< 0.1\%$ of the dataset) are omitted.

Genus	MetaCache	Kraken2	Kraken2+Bracken
Brochothrix	1.94%	1.94%	1.98%
Pseudomonas	1.23%	1.73%	1.92%
Psychrobacter	0.59%	1.43%	1.45%

MetaCache is able to match quantification accuracy of the previous alignment-based pipeline for the KLyo datasets. The average deviation to the meat components is even lower for MetaCache. For the Kal datasets MetaCache reduces the false positive rate while the average deviation increases slightly. However, it is still possible to identify the correct components with the benefit of less false positives.

7.2.5 Detection of Microbiota

A major strength of next generation sequencing when applied to foodstuffs, is its theoretically infinite range of species that can be detected. We therefore analyzed the microbiota detected by MetaCache in more detail. A visualization of the MetaCache results using Krona [85] for the dataset KLyo_C using the AFS31RS90 reference data set is shown in Figure 7.1. The results of Kraken2 and Bracken agree on the most prominent bacteria as shown in Table 7.11. The detected bacterial genera Brochothrix, Pseudomonas, and Psychrobacter are well known representatives in foodstuffs. In some sausages a very high amount of the species Brochothrix thermosphacta and even the corresponding Brochothrix phage BL3 could be found, possibly indicating meat spoilage. Furthermore, in several cases a significant amount of Actinoalloteichus was initially detected which has no known relation to foodstuff. However, after application of the coverage filter these matches could be detected as false positives and were removed.

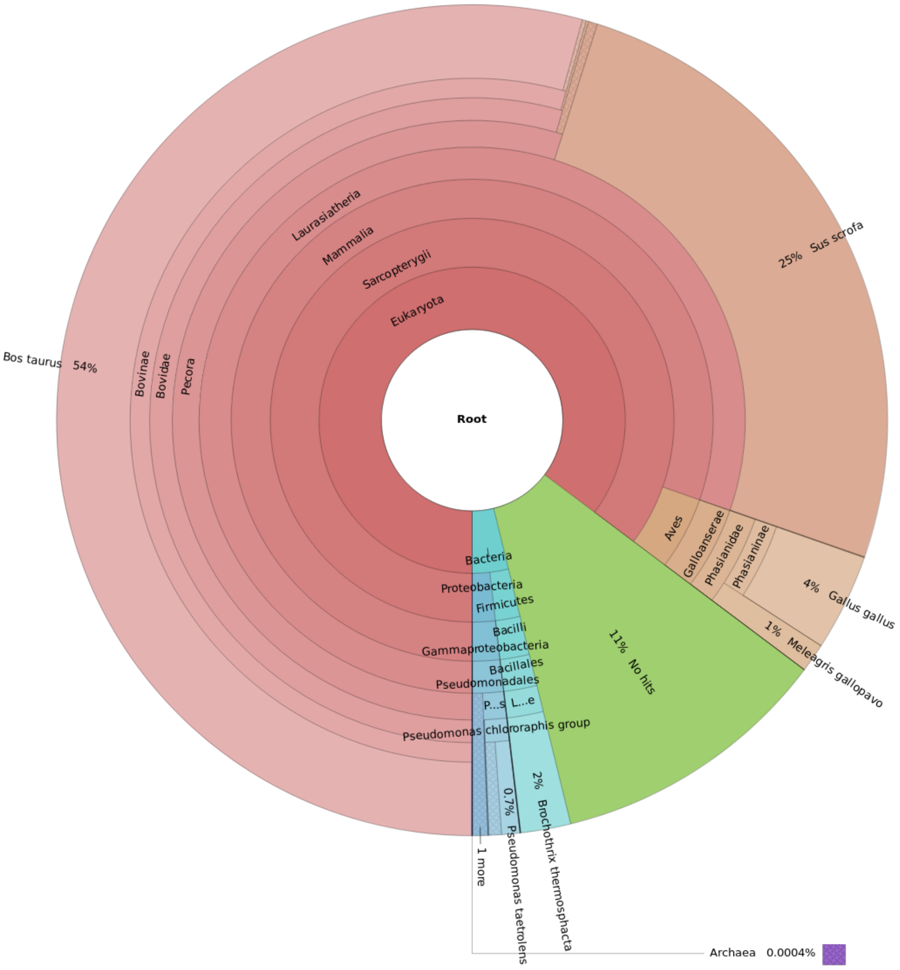


Fig. 7.1.: Visualization of the AFS-MetaCache results using Krona [85] for the dataset KLyo_C using the AFS31RS90 reference data set.

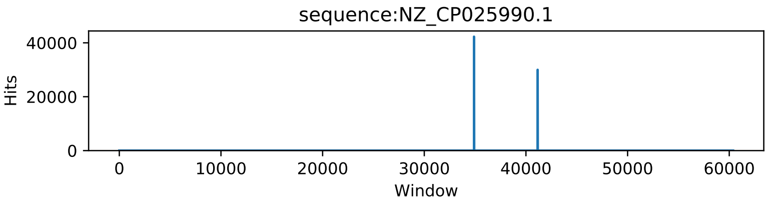


Fig. 7.2.: Genome coverage of Actinoalloteichus for the dataset KLyo_C. The sparse coverage is an indicator for false positives.

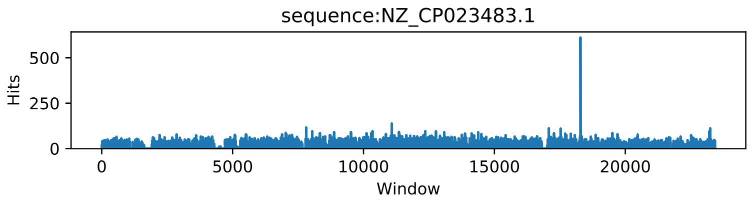


Fig. 7.3.: Genome coverage of Brochothrix thermosphacta for the dataset KLyo_C. The even coverage is an indicator for true positives.

Figures 7.2 and 7.3 show the corresponding genome coverage diagrams for Actinoalloteichus and Brochothrix thermosphacta for the KLyo_C read dataset. The highly uneven genome coverage of Actinoalloteichus is taken as an indicator by MetaCache for a false-positive species identification. The Brochothrix genome is evenly covered by reads and is thus classified as a true positive.

7.3 Discussion

The determination and quantification of food ingredients is an important issue in official food control [20]. Furthermore, microbiological contamination or the presence of non-declared allergenic food components establishes the need for a broad-scale screening method that allows for precise determination and quantification of ingredients ideally spanning all kingdoms of life including plants, animals, fungi, and bacteria. DNA-based methods like quantitative real-time PCR are established technologies for analyzing foodstuff. However, they have the drawback of being limited to a set of target species within a single assay that need to be defined beforehand. The usage of next-generation sequencing of total genomic DNA from biological samples followed by bioinformatics analyses based on comparisons to available reference genomes can overcome this limitation. The previous alignment-based AFS-pipeline was found suitable to screen for species in processed food samples [95, 70]. However, the utilized algorithms put limitations on the number species to be screened and on the computational throughput.

Here, we have presented how MetaCache can be employed for the efficient detection and quantification of species composition in food samples from sequencing reads. With MetaCache being based on an alignment-free exact k-mer matching approach, we gain significant speed compared to the previous alignment-based AFS method at the expense of a higher memory consumption for constructing and querying reference genome databases. We apply an intelligent subsampling technique based on minhashing within local windows to reduce the database size. Further reductions of peak memory consumption can be achieved by the introduced partitioning schemes at the expense of query speed. Applications of the previous alignment-based AFS pipeline have been limited to around ten complex genomes. With MetaCache we are able to significantly extend this limit, which is of high importance since the amount of available reference genomes continues to grow rapidly [104, 97]. Thus, our results are particularly encouraging since MetaCache is fastest in terms of database construction times. Corresponding peak memory consumption is competitive and can be even further reduced by the partitioned version of MetaCache.

Within this study we have applied our approach on a broad set of reference samples, containing admixtures of a set of food relevant ingredients (chicken, turkey, pork, beef, horse, sheep). The results demonstrate that our approach is able to reliably detect the components even at the 0.5% level. The comparison to the established metagenomics tools Kraken2, CLARK, and Kraken2+Bracken shows that MetaCache is superior in terms of false positive (FP) rates. In particular for pairs of closely related genomes MetaCache can achieve almost an order-of-magnitude lower FP-rates. These results demonstrate that our classification approach based on counting k-mer matches within small windows is effective compared to simply counting k-mer matches over an entire genome (as used by CLARK and Kraken) and to an alignment-based approach (as used by the our previous AFS pipeline). Our results also show that MetaCache achieves the lowest sum of absolute deviations to the included food ingredients. As different types of tissue can contain different concentrations of DNA (matrix effect), deviations could possibly be further reduced by a subsequent normalization procedure that takes tissue ratios into account.

7.4 Conclusion

We have presented a fast screening and quantification method together with a corresponding publicly available implementation for whole genome shotgun sequencing-based biosurveillance applications such as food testing. By relying on a big data approach, MetaCache can scale efficiently towards large-scale collections of complex eukaryotic and bacterial reference genomes making it suitable for broad-scale metagenomic screening applications.

MetaCache Methods 8

This chapter explores in detail various methods in the MetaCache (GPU-)pipeline. We analyze the corresponding problems and showcase our design and implementation choices. Our optimized implementations are evaluated on their own before we assess the performance of the complete pipeline in Chapter 9.

8.1 Genomic Sequence Processing

To achieve high throughput multi-threaded CPU programs and especially many-threaded GPU programs need fast input methods which provide them with data. The fastest algorithm implementations are of no use if they are bottlenecked by slow file I/O. Therefore it is important to optimize the input and output of a program to unlock its full potential. Here we want to analyze input methods for genomic sequence data by comparing MetaCache v1 to MetaCache v2 and its main competitor Kraken 2 [116].

8.1.1 Background

Many problems in bioinformatics like read alignment or read classification involve processing of sequencing data which ultimately boils down to text processing. The four nucleotide bases found in DNA are encoded into letters: **C** for cytosine, **G** for guanine, **A** for adenine and **T** for thymine. The same goes for RNA with the exception of **T** which is replaced by **U** for uracil. Additionally, it is common practice to use the letter **N** to encode ambiguous or unknown bases.

Strings of sequencing data are usually stored in the FASTA or the FASTQ file formats. Figure 8.1 shows example files with two sequences each.

FASTA files contain single header lines starting with the character '>' and holding the identification and description of a sequence, followed by one or multiple lines for the sequence string. Typically, a line of the sequence string is limited to 70-80 base characters.

```
>A_hydrophila_HiSeq.922
AGGCCCACTGGAAGTTGTAGCCACCGAGCCAGCCGGTCACGTCCACCACCTCGCCGATGAAGTAGAGACCGGCTA
CCTTGCGCGCCTCCATGGTCTTGGAG
>A_hydrophila_HiSeq.1263
TGACTTGACGTCATCCCCACCTTCCTCCGGTTTATCACCGGCAGTCTCCCTTGAGTTCCCACCATTACGTGCTGG
CAACAAAGGACAGGGGTTGCGCTCGT
```

```
@HWI-ST558:60:C00B3ACXX:2:1101:1560:1991_1:N:0:GGCTACA
TCAGGAACAATTTCTGTGATTAGAAATTATCATCATAGTTTATATAGGGGGCTGGCTTGAGTCGACCACCAGCCACCTGATGCATGAGTG
+
@CCFFFFDHHHHHGIJEHHGJIIGAGIEAHIGGJHJBHEGHEIJJIGHJJGGIJJIAG/.B7=E=BE<ACCD(55(9,,5(:@:>>AC4>
@HWI-ST558:60:C00B3ACXX:2:1101:1934:1970_1:N:0:GGCTACA
NTGTCAGAAGCTTTCCCTATCCCTTTTCTTAGTTTAATAAAACTTTATTACACAAAAGCTCTGAGCGATCAAGCCTCATCTCTGGCCCCA
+
#1=DDDFFHHHHHJJJJJIJJJJJJJIJJIJIJIJGIJJJJJIJJJJIJJJIJAGHIIJJJIEIIJJJJJEJHG?EFFFDFFCECEDDD
```

Fig. 8.1.: Examples of FASTA (top) and FASTQ (bottom) file formats with two sequences each.

FASTQ files consist of exactly four lines per sequence:

- The first line is initiated with character '@' and holds the identifier and description.
- The second line contains the complete sequence string.
- The third line begins with character '+' and may include additional comments.
- The last line provides quality scores for each base of the sequence. The quality is encoded in ASCII characters with hexadecimal values between 0x21 and 0x7e. This also includes the characters '+' (0x2b) and '@' (0x40), which may complicate parsing.

These sequencing files may incorporate millions to billions of sequencing reads generated from high-throughput sequencers. Aside from large amounts of independent reads (called single read or single-end), sequencing machines can produce paired-end reads which are two reads originating from the same fragment of DNA/RNA, providing more information than two separate reads. Both strings of paired-end reads (also called *mates*) are stored either in direct succession in a single FASTA or FASTQ file, or the mates of all pairs are separated and stored in two files, where each file contains all strings from one mate position in the same read order.

8.1.2 Sequence File Processing

Reading and parsing these kinds of genomic sequence files is a required step to be able to perform any kind of analysis on the data. Ideally, this has to be performed fast enough to not stall the sequence processing in multiple CPU threads or on the GPU. Because of the nature of the file formats shown above, reading a file has to happen sequentially to ensure that we read complete sequences with their header and comment lines as there does not exist an indexing structure listing the start and end points of sequences.

Parsing and processing of the data can happen in parallel once the sequence data has been read. However, parsing read files in parallel becomes challenging because of inconsistencies in the paired-end file formats. Two reads of the same read pair may have differently formatted headers or different sequence lengths. A thread starting to read data from an arbitrary starting point in a file or a pair of files may not be able to find the corresponding paired reads without causing significant overhead. Additionally, it may be required to enumerate all reads for post-processing or evaluation of classification results. MetaCache's merge mode for example necessitates a consistent enumeration to merge results from multiple classifications resulting from different database queries.

To overcome these challenges MetaCache employs a producer-consumer scheme, where only one thread parses the input files as fast as possible and stores batches of reads in a concurrent queue. The single reader thread keeps track of the read IDs and matches consecutive paired-end reads or read pairs from two files. This linear access is also beneficial when accessing files from hard disks, where opening multiple file handles and jumping between file positions can cause additional overhead.

Concurrent to the reader thread multiple threads can be used to process the sequence data independently in parallel. Each CPU consumer thread dequeues batches of reads and either processes the sequences on or copies the sequence data to the GPU for massively parallel processing. After finishing a batch the allocated memory for the batch is preserved by inserting it into another concurrent queue. The reader thread recycles the memory by dequeing batches from this storage and filling it with newly read sequences. By preallocating a number of batches which are used throughout the lifetime of the program we avoid costly memory allocation for each new sequencing read.

8.1.3 Related Work

MetaCache's original sequence reader was based on Kraken's [117] and featured two different C++ classes to handle FASTA and FASTQ files separately. MetaCache v1 already incorporated improvements over Kraken like reusing allocated memory and reducing the amount of temporary buffers and intermediate data copies.

MetaCache v2's sequence reader and parser is inspired by kseq taken from klib[1], a popular standalone and lightweight C library. kseq features a single parser for handling FASTA and FASTQ files. We reimplemented kseq in C++ and applied additional performance tuning. Like kseq we use a custom character string and buffered input stream which also supports compressed sequencing files by incorporating zlib[2].

Compared to MetaCache Kraken 2 uses a different approach in its multi-threaded classification. Here each thread reads a batch of sequences from the input file(s) into a buffer and then continues to process the whole batch itself. Reading is guarded by a critical section where only one thread at a time is permitted. Parsing the data, i.e. separating the characters into sequence header, data, comment and quality scores, is done outside the critical section.

Kraken 2 chooses one of two strategies for reading a batch depending on whether the input consists of single-end or paired-end sequences. For single-end data they read a block of three megabyte of characters from the input files without parsing the data (*block strategy*). Then they check the following data and search for the start of a new sequence. Data belonging to the last sequence of the block is appended to the buffer to ensure that blocks always contain complete sequences. This strategy postpones parsing a large number of characters by storing them into the buffer and keeping the critical section small, but it cannot determine the number of sequences in the batch until later in the pipeline. Therefore it cannot be used to create a continuous enumeration of the reads.

The second strategy is used when reading paired-end input data. Here Kraken 2 creates batches of 10000 paired sequences by reading the input files line by line to count the number of sequences (*batch strategy*). All read lines are stored into the buffer. After the critical section this buffer has to be separated into lines again and parsed to process the sequences.

8.1.4 Performance Evaluation

In this section we compare the sequence readers from Kraken 2 (v2.1.2), MetaCache v1 (v1.1.1), MetaCache v2 (v2.1.1) and klib. To evaluate the performance of the sequence file readers we utilized three metagenomic read datasets. HiSeq and MiSeq use the FASTA file format which only contains header and sequence lines, while the third dataset Kal_D consists of paired-end reads separated into two FASTQ

[1] https://github.com/attractivechaos/klib

[2] https://www.zlib.net/

Tab. 8.1.: Metagenomic read datasets with variable minimum, maximum and average sequence lengths.

Dataset	Format	Sequences	Min	Max	Average	File Size
HiSeq	FASTA single	10,000,000	19	101	92.3	1.3 GB
MiSeq	FASTA single	10,000,000	19	251	156.8	1.9 GB
KAL_D*	FASTQ paired	10,000,000	101	101	101	2x 2.5 GB

files, which also include comment lines and quality scores. Each file holds 10 million reads. Table 8.1 shows the exact number of sequences as well as the minimum, maximum and average sequence lengths for each datasets. More details on the composition of the datasets can be found in Section 9.

Experiments were conducted on the following system:

System 1 AMD Ryzen Threadripper 3990X (64 cores at 2.9 GHz) with 256 GB DDR4 RAM, Ubuntu 20.04, GCC 9.3.0.

Initially, a dry run is performed to read the input files into RAM in order to circumvent performance penalties induced by slow hard disk access. Thereafter, each sequence reader was executed in a single thread five times for each benchmark. We report the average runtime of the five runs. Kraken 2's reader was run one time only creating batches of sequence data, and one additional time with subsequent parsing of the batch buffer. Parsing is needed for processing the sequences but normally happens outside Kraken 2's critical section. MetaCache and kseq already parse the sequences while reading the data from files.

Single-end Benchmark

For the single-end benchmark we evaluated each sequence reader on the HiSeq, MiSeq datasets and one file from Kal_D. Although Kraken 2 normally uses the block strategy for single-end data, we also tested how the batch strategy performs in this case.

Figure 8.2 reveals the results for this benchmark. Kraken 2's block strategy achieves highest speed for all three datasets. Reading chunks of raw data is as fast as it gets. After reading such a chunk Kraken 2 only needs to find the next beginning of a sequence to create a valid block of sequence data, which can be found fast if sequences are short. For longer sequences in MiSeq we can see that the speed drops by 30%, and another 27% for Kal_D in FASTQ format, because more lines need to be checked to find the next sequence. As we can also see from Kraken 2's batch

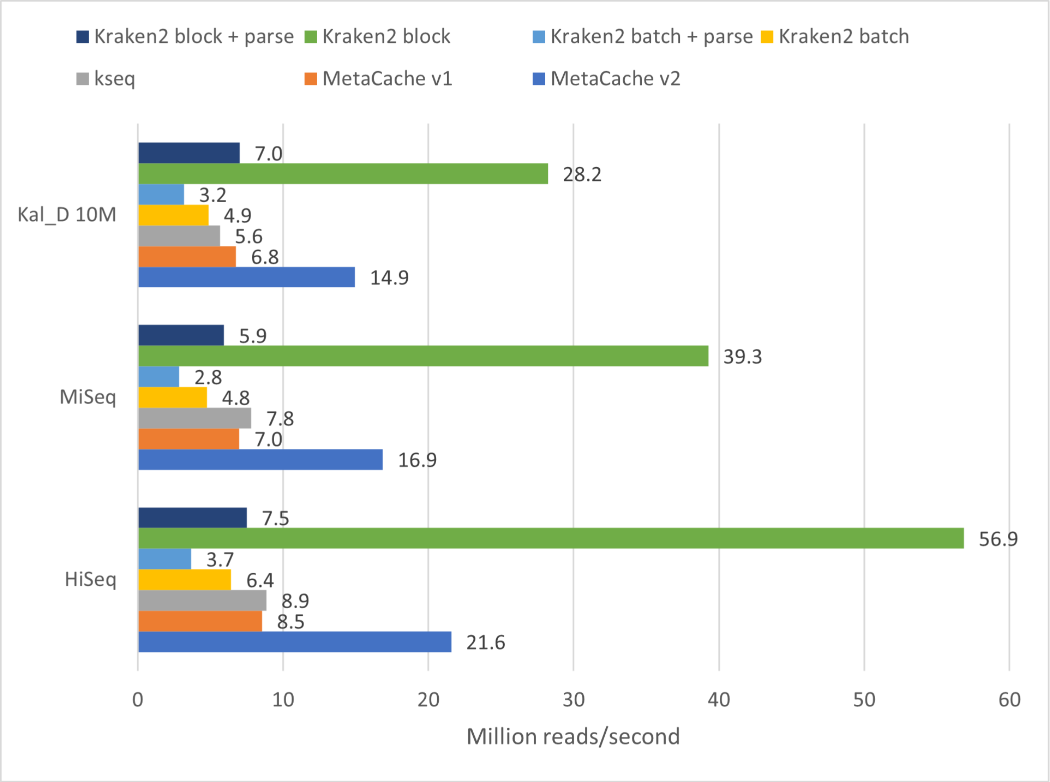


Fig. 8.2.: Single-end Sequence File Processing Benchmark.

strategy, it is not as efficient when reading the input line by line. Is is slower than kseq and both MetaCache versions even without parsing the data.

When looking at Kraken 2's runtimes including the parsing we see that it is quite inefficient. Even when combined with the fast block strategy, Kraken 2's parser is slower than the other sequence readers, except for the Kal_D dataset where it is slightly faster than kseq and MetaCache v1.

The default kseq is faster than Kraken 2 with parsing and MetaCache v1 for the FASTA datasets, but slower for the FASTQ file. This may be caused by kseq being the only tool that parses the comment line from FASTQ, all other tools skip the comment lines when parsing.

MetaCache v2's sequence reader outperforms all others when reading and parsing the sequence files. It is more than twice as fast as the best contender in each case. Kraken 2 has the only sequence reader able to achieve a higher performance but only when reading the data without parsing.

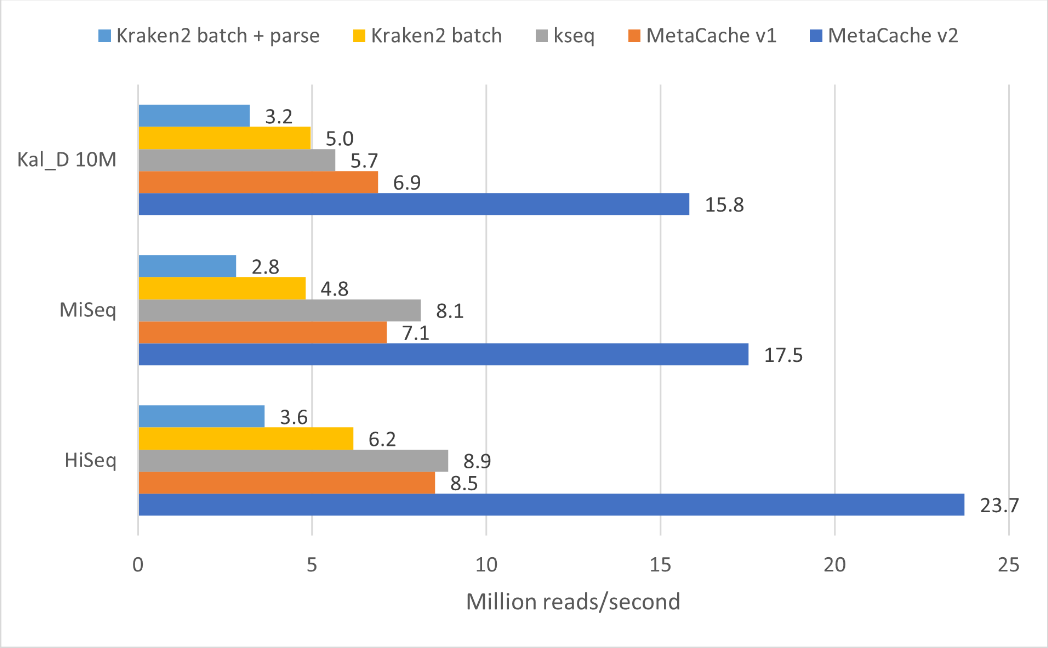


Fig. 8.3.: Paired-end Sequence File Processing Benchmark.

Paired-end Benchmark

For the paired-end benchmark we tested each sequence reader in paired-end mode with two separate files per dataset. For HiSeq and MiSeq we created a copy of each file to emulate a dataset of 10 million read pairs, for Kal_D we used both original files. In the case of paired-end data Kraken 2 is not able to use the block strategy and thus always uses the slower batch strategy.

Figure 8.3 shows the results for this benchmark. The results are very similar to the single-end benchmark, but here we read twice the amount of sequences – two per read. We can see that most of the time the sequence readers perform a little better than in the single-end benchmark. Nevertheless, MetaCache v2's performance stays more than two times higher than all other readers.

8.1.5 Conclusion

File I/O is a non-negligible part of genomic data processing pipelines. Fast sequence readers for the FASTA or FASTQ formats are needed to achieve high throughput. MetaCache v2's sequence reader and parser is able to achieve twice the performance of competing tools, while providing a consistent enumeration of processed reads, which is required for post-processing methods like MetaCache's own merge mode.

While Kraken 2's own parser is not able to achieve the same level of performance, Kraken 2's block strategy combined with a kseq-style parser like in MetaCache v2 has the potential to be a viable alternative for single-end datasets in cases where the enumeration is not needed. The implementation of such an approach remains as future work.

8.2 MetaCache GPU Pipeline

The overall structure of MetaCache is the same for the CPU and GPU versions. However, many data structures and sub-routines of the program had to be adapted to run efficiently on the GPU. We employ a novel, specialized hash table on the GPU to accelerate database building, querying, and classification. To overcome the memory limitations of a single GPU we extend MetaCache to work with hash tables distributed across multiple GPUs. Figure 8.4 shows an overview of the GPU workflow.

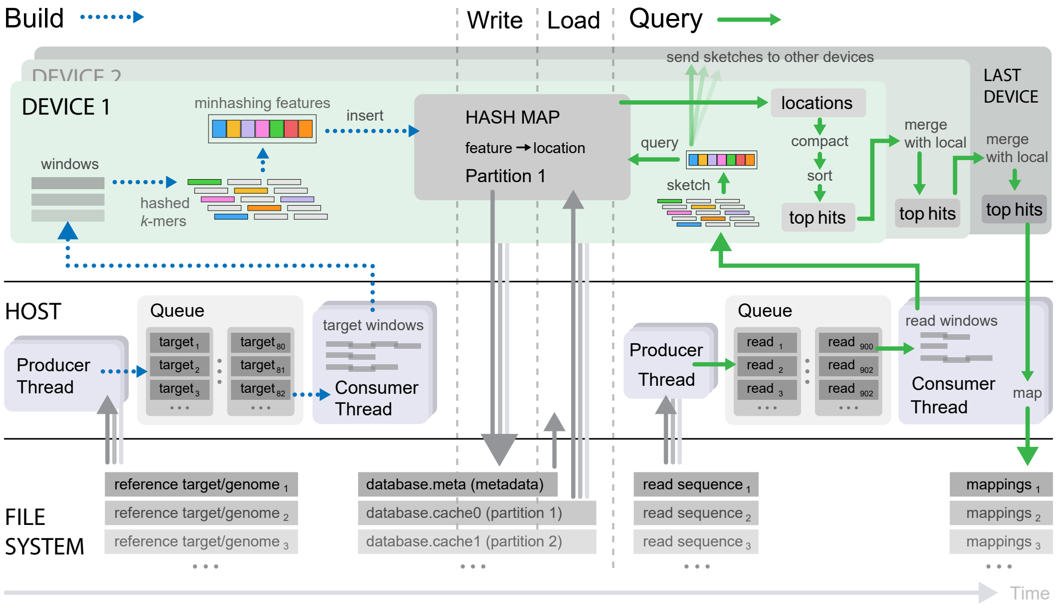


Fig. 8.4.: GPU Workflow: (1) Build: producer threads enqueue genomic sequences. Consumer threads split them into windows and copy these to the GPUs. GPUs generate minhashing sketches and insert them into their local hash maps. (2) Query: batches of read windows are copied to the first GPU for sketch generation. Each GPU queries its local hash map and sends the sketches to the next GPU. Resulting location lists are compacted and sorted. Top locations are selected and sent to the next GPU for merging. The last GPU obtains the final top list and sends it to the host for read mapping and output.

8.2.1 Build Phase

Looking at the build pipeline, the first stage of the producer-consumer scheme remains the same as in the CPU version. Here we use multiple producer threads to read the reference genome files in order to keep the faster consumers busy. Unlike the CPU pipeline, a consumer thread does not process the sequence data itself anymore.

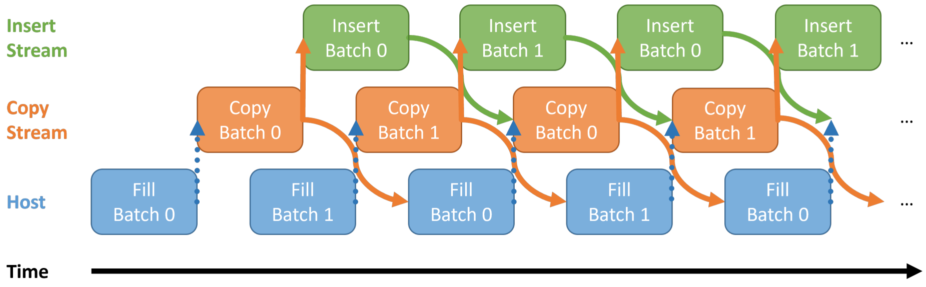


Fig. 8.5.: GPU Build Phase: Using two alternating buffers on host and device, fill, copy and insert can be executed in parallel for different batches. Dashed arrows show implicit synchronization, solid arrows show explicit synchronization using CUDA events. Operation lengths and gaps are not to scale.

Instead it collects the sequence strings of one or more reference genomes in a linear array of fixed size which is sent to the GPU together with the corresponding genome location information. The GPU is now responsible for extracting k-mers, generating minhashing sketches and insertion into the hash table.

In a multi-GPU environment we spawn as many consumer threads as there are GPUs, each thread scheduling work on a distinct GPU. All calls to the GPU are executed asynchronously which means that the host threads do not have to wait until copies to device memory and insertion of a batch are completed on the GPU. It can already continue to collect the next batch in a second buffer. Figure 8.5 depicts this double buffer strategy. We employ two CUDA streams and CUDA events to synchronize between fill, copy and insert operations to ensure that buffers can be safely reused when the previous operation has finished. Host work, copying and insert kernel execution can all happen at the same time for different batches. In this parallel pipeline the insert kernel takes the most time and keeps the GPUs busy throughout the whole build phase while the time for filling and copying batches can be completely hidden (except for the first batch).

If a hash table exceeds the chosen maximum load factor, the GPU will still finish inserting batches scheduled by the corresponding host thread. However, the thread will stop filling the buffer array and return unprocessed sequences to the queue, leaving the remaining work to other consumer threads. Note, that a single reference sequence will never be distributed across multiple GPUs. However, the same k-mer might be present in multiple hash tables because it can appear in several genomes. This partitioning is deliberate and helps to reduce data communication in the classification phase.

After database construction finishes, the hash tables are retrieved one-by-one in batches of keys and associated values. We run a segmented sort on the whole batch to sort the list of locations for each k-mer in order to stay consistent with the ordering of the CPU version. Each hash table is stored in a separate file and can be used for querying by both versions of MetaCache.

8.2.2 Query Phase

The query phase can either start directly after the database build has finished (on-the-fly mode) using the hash tables that are already in GPU memory or alternatively it can be performed in a separate run.

In case the query is performed using an already existing database, MetaCache has to first load the data from files into the GPUs. Similar to the CPU version, we use a condensed layout where all buckets are stored in one contiguous array in GPU memory. To associate keys and buckets we employ a single-value hash table to map the keys to pointers into the array. Note that if the query phase is started directly after a build, the hash table is used as-is and will not be compacted. Additionally, we copy the acceleration structure containing the taxonomic lineages to each GPU.

After preparing the hash tables on all GPUs, we begin to process the input files using the same produce-consumer scheme as in the CPU version. However, consumer threads do not process the sequence data themselves. Instead they split reads into windows and send batches of these sequence windows to the first GPU. Because the database may be distributed across multiple GPUs, each device has to be queried and results are combined until they are finally copied to the host, which will output the classification results.

In order to enable multiple host threads to provide work while limiting memory occupancy on the devices, we use a pipeline approach, allocating memory for all steps needed for processing a single batch of sequences on each GPU beforehand. This also avoids expensive memory allocations while executing the queries, which would stall the entire pipeline. CUDA events are used to orchestrate the pipeline, signaling when a stream has to wait for data or can continue work using the same memory resources as the previous batch. This allows to overlap memory copies and kernel executions of different batches and across devices. See Figure 8.6 for an overview of the query pipeline.

The query pipeline on the GPU consists of the following steps:

1. Encode all sequence characters of a window into 3-bit representations of the nucleotide bases (A,C,G,T,N).

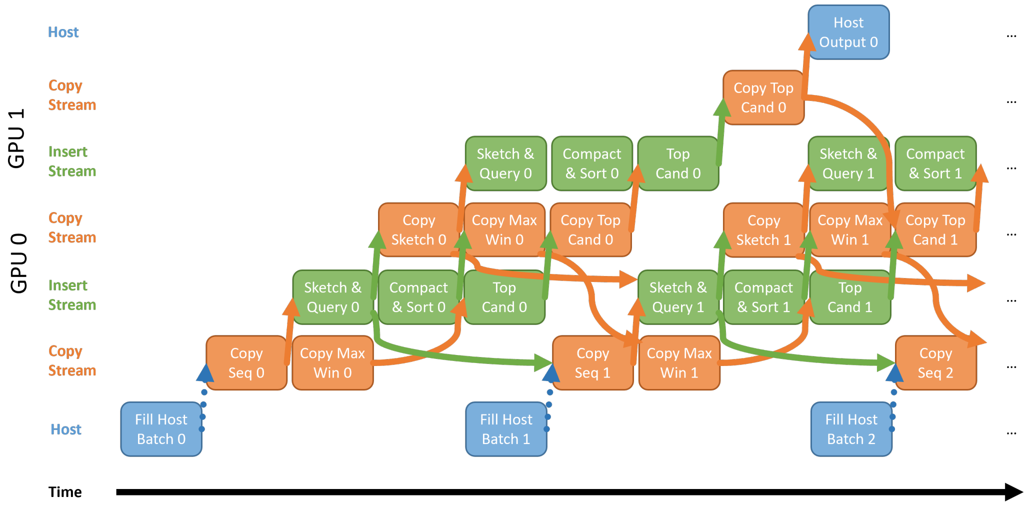


Fig. 8.6.: GPU Query Phase: Batch data moves from host through all GPUs. Results are aggregated and final result is copied back to host. Dashed arrows show implicit synchronization, solid arrows show explicit synchronization using CUDA events. Operation lengths and gaps are not to scale.

2. Generate all valid k-mers and hash them using hash function h_1.
3. Sort the hashes to get the minhashing sketch.
4. The sketch is queried against the hash table and resulting locations are written to memory.
5. Compact the location lists of all processed windows.
6. Execute a segmented sort to sort the locations for each read.
7. Obtain the window count statistic by accumulating identical locations.
8. Find the top hits using a sliding windows scan.

After generating sketches on the first GPU, they are send to the other GPUs which allows them to skip the first three steps. Each GPU generates its own top hits for each query, which are then send to the next GPU and merged with its local top hits. We finally obtain the top hits of the whole database on the last GPU. This result is copied back to the host, which assigns the final classification for each read.

Our GPU kernels combine multiple steps of the pipeline and have been optimized for efficient work sharing between many CUDA threads. We employ groups of 32 threads (so-called *warps*) to tackle the same problem. We enforce additional constraints on MetaCache's sub-sampling parameters to improve data layout and memory access patterns. Especially, we require the offset between window beginnings (*window stride*) to be a multiple of 4 to enable aligned access to 4 characters per thread. Kernel implementation details are presented in the following. Section 8.3 goes into detail of steps (1)–(4) in the GPU pipeline, which are executed by the first kernel. Section 8.5 examines steps (5) and (6). Steps (7) (8) are combined in the last kernel and discussed in Section 8.6.

8.3 Minhashing and Querying

In MetaCache(-GPU) we employ a subsampling technique called minhashing to reduce the amount of sequence data that has to be stored in the database. Sequences are first split into windows of fixed size. Then for each window a sketch is generated to represent the window sequence in the database. The first kernel in the GPU pipeline combines these steps with hash table operations to insert or query the generated sketches.

8.3.1 k-merization and Minhashing

Because of the small alphabet of the sequence strings, it is beneficial to encode the letters into fewer bits instead of storing each letter as a separate character (usually the size of one byte). Using two bits to encode the four bases (or three bits if one needs to include ambiguous N bases) and storing multiple bases per byte significantly reduces the amount of memory needed and enables to use fast bit arithmetic and hash functions on integer types.

On the CPU a thread encodes one character of a sequence window after another into two bits and stores multiple characters into a 32-bit or 64-bit integer, allowing for k-mers of size up to 16 or 32 base pairs, respectively. In each step the integer is shifted by two bits to make room for the next base. When the number of bases in the integer reaches the desired k the complete k-mer is extracted before continuing to add the next base.

Due to the unknown orientation (*strandedness*) of the sequences, for each extracted k-mer its canonical k-mer is calculated, which is the lexicographically smaller sequence of the k-mer and its reverse complement. The reverse complement represents the string of base pairs opposite the original string in the double stranded DNA molecule. The canonical k-mers are then hashed using a hash function h_1 and inserted into a sorted array of maximum length s, which holds the minhashing sketch. After processing all k-mers of a window the sketch is finished and can be queried against the database.

8.3.2 Naive GPU Kernel

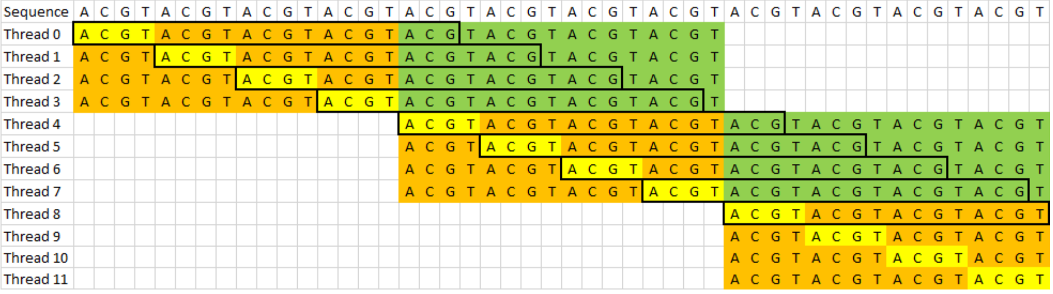
On the GPU we apply more fine grained parallelism where one warp of 32 threads is responsible for each window. In a naive approach we divide the window among the threads of the warp and let each thread generate a number of canonical k-mers and their hashes. We first load all characters of the window from global into faster shared memory, 32 characters (one character per thread) at a time. For a maximum window size of 128 bases each thread has to generate up to four k-mers in order to process the whole window. Each thread i starts with an offset of four characters to its predecessor (i.e. at position $4i$) in the window, reading characters one by one from shared memory to generate k-mers using code similar to the CPU version.

To produce the minhashing sketch we have to find the s smallest hash values in the warp. After all threads calculated their canonical k-mers and the corresponding hashes, we sort all hash values of the window using a thread block based sort primitive from the CUB library [@14], which also requires some shared memory. Next we remove duplicates and store the s smallest unique values in memory.

8.3.3 Improved GPU Kernel

The improved kernel avoids using shared memory by storing all sequence information in registers and using fast shuffle instruction to communicate data between threads of a warp.

Similar to the naive kernel each warp processes the sequence characters of a single window of maximum size 128 at a time. However, here each thread in the warp loads 4 consecutive characters from global memory into register using a single 4-byte load operation. Next, it encodes the characters using a 2-bit representation of the regular nucleotide bases A,C,G,T and combines them in a single 32-bit integer. Ambiguous base characters N are noted as single bits in an auxiliary integer. 4 adjacent threads form a sub-warp and combine their integers using XOR shuffle operations so that every thread holds the information of $4 \cdot 4 = 16$ consecutive characters. Then each



Sequence	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T
Thread 0	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T																
Thread 1	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T																
Thread 2	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T																
Thread 3	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T																
Thread 4																	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T
Thread 5																	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T
Thread 6																	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T
Thread 7																	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T
Thread 8																																	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T
Thread 9																																	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T
Thread 10																																	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T
Thread 11																																	A	C	G	T	A	C	G	T	A	C	G	T	A	C	G	T

Fig. 8.7.: Example of k-mer generation using a warp of size 12: (1) Load 4 characters per thread (yellow). (2) Get characters of sub-warp (orange). (3) Get characters from subsequent sub-warp (green). (4) Generate four 16-mers from framed characters.

sub-warp gets the information from the subsequent sub-warp by means of another shuffle operation. Now each thread contains the data of 32 characters, overlapping by 16 characters with the next sub-warp. From these characters thread i is able to generate four k-mers starting at positions $4i, \ldots, 4i + 3$ of the window, which are then hashed using hash function h_1. The data sharing between threads of a warp is visualized in Figure 8.7. Note, that the last few threads (number depends on window and k-mer length) do not generate any k-mers because they would exceed the window boundary.

Instead of using CUB to sort the hash values, we reorder them using a bitonic sort implementation similar to our approach in Section 8.5 which operates only on registers with the help of warp shuffles. Again the first unique s hashes from the sorted result form the minhashing sketch.

8.3.4 Hash table operations

It would be perfectly fine to store the minhashing sketches in global memory and execute the hash table operations in a separate kernel or library call. However, this would incur expensive accesses to global memory to store data in one kernel and read the same data in another kernel. Fortunately, WarpCore provides device-sided hash table operations which can be called from inside the same kernel. It uses a cooperative probing scheme which means that multiple threads work together as a group to insert or find a key. Therefore we split the warp which generated a sketch from a window into thread groups to query the values of its sketch. The queries retrieve the bucket pointers to the locations associated with the queried keys. After replacing the sketch with these pointers in shared memory, all threads in the warp work in unison again to retrieve the target locations from each bucket. The locations are then stored in global memory.

In MetaCache-GPU the number of employed warps for the minhashing kernel matches the batch size in the GPU pipeline, so that each warp processes a single window from a batch. Therefore, the memory requirements scale linearly with the number of warps. For each window we require the input sequence in global memory and have to reserve enough memory to accommodate the hash table results for the whole sketch. By design the number of results retrieved by each hash table query is limited, so the memory can be allocated for the worst case beforehand. However, the batch size and number of warps should be kept small to reduce the memory requirements per batch and leave more space for the hash table itself.

8.3.5 Performance Evaluation

In order to evaluate the performance of the GPU approaches, we measured the time the kernels took to generate sketches for 2^{20} windows of 127 random characters using MetaCache's default parameters. The default parameters include a k-mer length of $k = 16$ characters, a sketch size of $s = 16$, a window length of $w = 127$ characters and a window overlap of $k - 1$ which results in a window stride of $127 - 16 + 1 = 112$. Note that the improved kernel requires that the beginning of each window is aligned to 4 bytes to allow for aligned access to 4 characters per thread. Benchmarks were conducted on an NVIDIA Quadro GV100 which employs 80 streaming multiprocessors.

To investigate the performance impact of the character processing as well as the different sort implementations we benchmarked four different kernel versions. The first kernel uses the naive approach described in Section 8.3.2 (*shared chars* + *cub sort*), the second kernel combines the improved character processing with CUB's sort (*shuffle chars* + *cub sort*), the third kernel uses the naive character processing with bitonic sort (*shared chars* + *bitonic sort*), the last kernel features the improved implementation from Section 8.3.3 (*shuffle chars* + *bitonic sort*). Figure 8.8 shows the results for all four kernels on a GV100 GPU using different numbers of thread blocks. Each thread block contained one warp of 32 threads and processed one window at a time, looping until all 2^{20} windows have been processed.

A small number of 256 blocks is not enough to efficiently utilize the 80 SMs of the GV100 GPU. Because of the low occupancy it is not possible to hide the latency of memory accesses, hence the improved access pattern of the *shuffle chars* kernels has more impact on the runtime here. For larger numbers of blocks the improvement is less pronounced because the SMs can switch between more resident warps which are waiting for data. In general a certain number of blocks is needed to achieve a high occupancy with results in higher throughput. With 4096 blocks we can see that both improvements, character processing and sorting, each reduce the runtime by about 25% while combining them yields 63% total improvement. For larger block numbers all kernels still see improved runtimes but to a smaller and smaller degree. The *shuffle chars* + *bitonic sort* kernel achieves the highest speedup compared to the naive kernel even higher than the individual improvements would suggest. Using the superior memory access scheme and the faster sort implementation results in a more efficient execution.

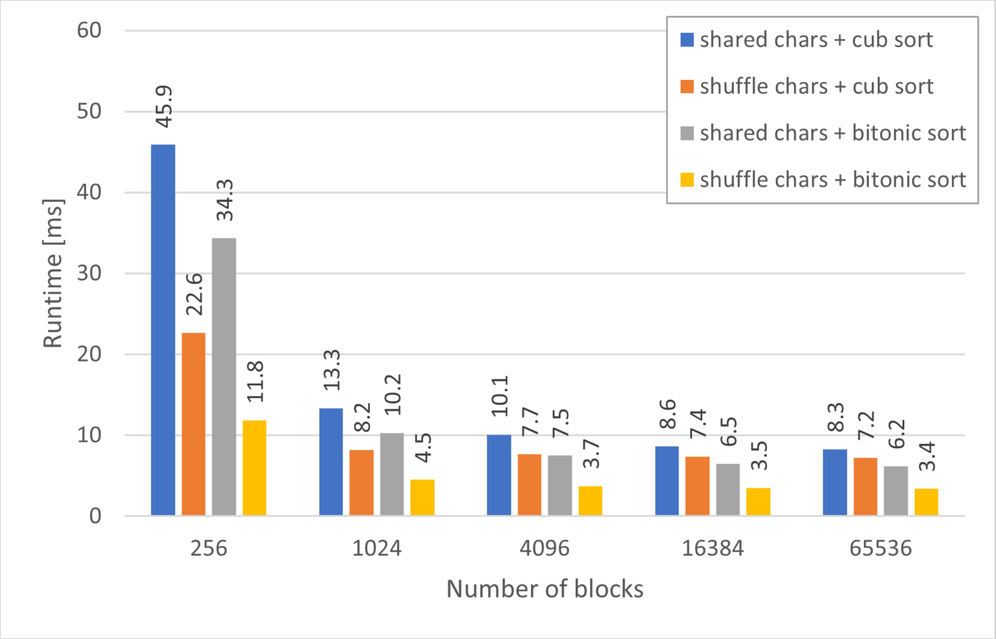


Fig. 8.8.: Kernel benchmarks for k-merization and minhashing of 2^{20} windows of length 127 on a GV100 GPU.

8.3.6 Conclusion

We designed and implemented an efficient CUDA kernel for k-merization and minhashing by improving the memory access pattern and choosing an optimized sort algorithm which both avoids unnecessary shared memory usage. Our improved kernel yields a significant speedup compared to a more naive implementation. Our benchmark showed that the number of blocks is an important parameter which can have a large impact on the runtime. Modern GPUs feature a large number of SMs which need to be fed with a sufficient number of threads to achieve high occupancy which is needed for efficient resource utilization. Therefore MetaCache-GPU uses a batch size of 8192 windows for the minhashing kernel to enable high throughput while limiting the amount of required memory for the pipeline to a tolerable level.

8.4 Multi-Value Hash Tables

MetaCache-GPU stores reference genome information in a hash table which maps hashed k-mers resulting from minhashing to genome locations. Because the same k-mer can occur several times in different genomes or even at different locations in the same genome, the database may map each k-mer to a list of associated locations. Corresponding hash tables that implement this one-to-many mapping are called multi-value hash tables (or multi-value hash maps). When building a database with GPUs, MetaCache-GPU allocates the hash tables in GPU memory, while meta information like the taxonomic tree remain in host memory.

Database (also called k-mer index) construction performance is predominantly governed by the throughput of the underlying hash table implementation. To alleviate this bottleneck, we leverage the fast memory interface of modern CUDA accelerators. Our aim is to build and query the hash table completely in GPU memory to achieve high processing speed. Using a static allocation strategy avoids costly resizing of the data structure and subsequent rehashing which would stall the parallelized insertion process. To be able to fit as much genome data on the GPU as possible memory overhead has to be kept small. Therefore, we employ a batching strategy which leaves most of the GPU memory available for the database. Furthermore, the hash table should be able to manage various key-value distributions efficiently. Depending on the user-supplied genomes the database may consist of either many different k-mers (keys) with a small number of location (values) or fewer keys with higher value multiplicity. For MetaCache-GPU we introduce a novel multi-value hash table variant optimized for memory-efficient k-mer index construction and querying on multiple GPUs.

8.4.1 Related Work

High-throughput GPU hash tables have been studied extensively [59]. However, most existing implementation show limitations which make them unsuitable for our use case. Among the first implementations, Alcantara et al. proposed two cuckoo hashing variants [2, 1] which were both incorporated in the cuDPP library. However, cuDPP does not support dynamic table builds, i.e., building the table in multiple batches in case the input data exceeds the available GPU memory space. Additionally, both versions only support 32-bit wide key types which would limit the minhash subsampling approach to 16-mers when using 2-bit encoding for base pairs. CoherentHash [26] employs a Robin Hood hashing scheme which promises a lower on-average probing length but requires additional memory in the form of a 4-bit age indicator per table slot, thereby reducing the overall memory utilization. StadiumHash [46] introduces an open addressing hash table where the table itself may either reside in the global memory space of the GPU or out-of-core, i.e., inside host memory. An auxiliary ticket-board, which persists in video memory is used to track slot occupation within the table. In the case that the hash table has to be stored out-of-core due to the limited amount of available video memory, the performance drops drastically due to the imposed PCIe bottleneck. SlabHash [3] proposes a dynamic GPU hash table based on separate chaining. The table consists of an array of linked lists, each of which represents a chain of equally sized memory units, so-called slabs, that store colliding keys during insertion. HashGraph [30] uses a table construction method that is highly similar to a compressed sparse row matrix layout. HashGraph only supports static table builds, which again implies a lack of support for batched workflows. Furthermore, their approach has high memory overhead since it requires $3n$ temporary memory during table construction with n input key-value pairs.

In particular none of the implementations feature out-of-the-box multi-GPU support which is key for metagenomic classification since many real world databases exceed the memory space of a single GPU. As a more recent publication, WarpCore [42], successor of WarpDrive [40], proposes a framework that allows for the design of purpose-built GPU hash tables that can be tailored towards optimal performance for a given use case and can outperform previous approaches such as cuDPP and SlabHash. Their cooperative probing scheme uses sub-warp tiles, i.e., CUDA cooperative groups, over a hybrid two-stage probing scheme, where an outer double hashing strategy is used to suppress table clustering effects, while an inner group-parallel linear probing scheme ensures coalesced memory access. Additionally, the authors propose a multi-GPU extension based on an efficient all-to-all communication pattern over dense NVLink topologies [50]. In this work we extend the aforementioned WarpCore framework by a novel hash table layout which better suits typical k-mer distributions in terms of performance as well as storage density.

8.4.2 Background

WarpCore's *Multi Value Hash Table* is based on open-addressing scheme, where arrays for keys and values reside in contiguous GPU memory, either in Array of Structures (AoS) or Structure of Arrays (SoA) layout. Multiple values belonging to the same key are stored as separate key-value pairs, thus there exist the same number of key and value slots in the data structure. In AoS keys-value pairs are packed together as composites and stored in a single array, while SoA uses two separate arrays for keys and corresponding values. The capacity denotes the maximum number of available key and value slots in those arrays. To initialize the hash table the arrays are allocated in GPU memory and each key slot is filled with empty-indicator k_e in order to identify free slots during the probing.

WarpCore employs a hybrid probing scheme called COPS (Cooperative Probing Scheme) which combines double hashing with linear probing. The *outer* probing scheme uses double hashing to provide starting indices for the *inner* linear scheme, which is executed cooperatively by a group of CUDA threads. This yields improved data locality and memory access for threads of the same group while avoiding primary clustering effects common to linear probing.

For an example group size of 4 threads the generated probing sequence looks like $\{\big(h(k, \lfloor \frac{i}{4} \rfloor) + 0\big) \bmod c, \ldots, \big(h(k, \lfloor \frac{i}{4} \rfloor) + 3\big) \bmod c\}$, where h is the outer probing scheme acting on a key k. Figure 8.9 shows the insertion of a key-value pair (k, v) into a hash table in seven steps:

(1) The starting index for the whole group of threads is determined by the outer probing scheme h.

(2) The group loads consecutive keys from the hash table according to the group size.

(3) Each thread checks if its assigned slot is free, then all threads use a group voting function to communicate the results to the whole group.

(4) Select the thread associated with the lowest candidate slot index as leader.

(5) If the group found no free slot at the current indices (left column in Figure 8.9), the outer probing scheme determines the next index where steps 1 to 4 are repeated.

(6) The selected thread tries to insert the key using an atomic CAS. If the slot was occupied by another key in the meantime, the CAS fails and steps 4 and 6 are repeated as long as the group has remaining candidate slots. Otherwise the probing continues from step 5.

(7) In case of a successful key insertion, the associated value can be stored with an relaxed write operation.

In order to retrieve values from the hash table the same probing scheme is applied. In WarpCore's Multi Value Hash Table a key may occur in multiple slots and we do not know the exact number of occurrences for each key. Therefore, the retrieval is executed in two phases. The first phase counts the number of values per queried key

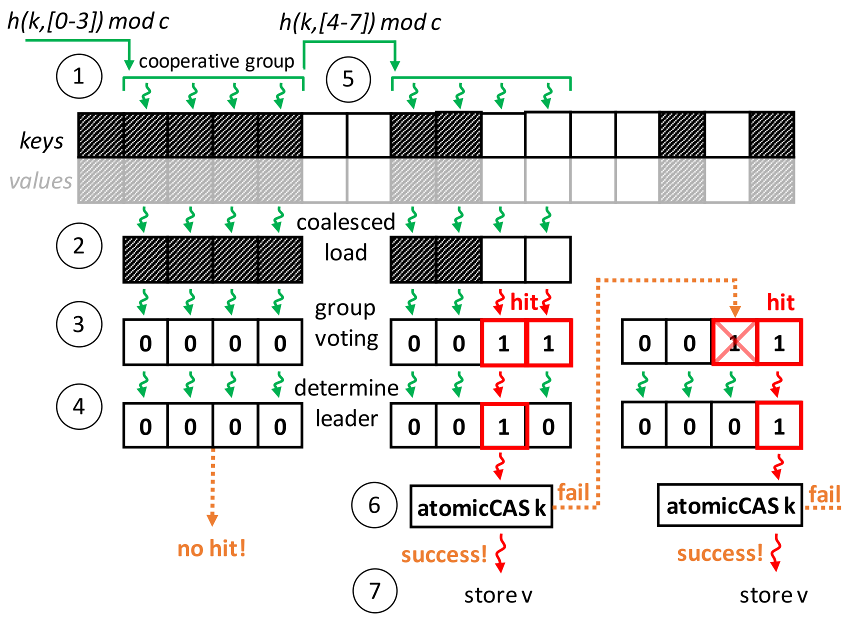
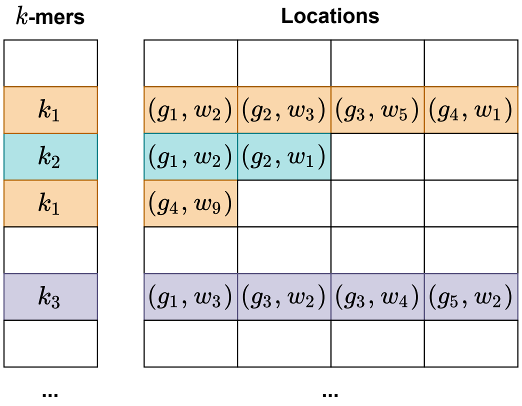


Fig. 8.9.: Insertion of a key-value pair (k, v) into a hash table with capacity c using COPS with an outer probing scheme h, an inner probing window size of 4 and a cooperative group size of 4. [42]

to determine the amount of memory required and calculate the output positions for the values, while the second phase actually retrieves them. In each phase the threads of a group compare the probed slots to a queried key to find all corresponding slots and either increase the value counter or copy the value to the output array. As soon as an empty slot is found in the probing sequence the retrieval phase can be stopped, because the insert process fills the slots in the same order and never skips empty slots.

8.4.3 Multi Bucket Hash Table

Due to the structure of WarpCore's Multi Value Hash Table key-value pairs with an identical key each occupy a separate slot in the hash table. This causes a memory overhead because the same key is stored repeatedly. Other multi-value, like WarpCores's *Bucket List Hash Table*, don't insert identical keys multiple times but instead map each key to a dynamically sized (linked) list, where all associated values are stored. However, this approach lacks the flexibility to accommodate various key-value distributions. When allocating memory for this kind of hash table the user has to decide on fixed capacities for keys and values separately which requires prior knowledge about the key-value distribution for optimal memory efficiency. Allotting less memory for keys would allow to store more values in case of high value multiplicity, but would also preclude storing many different keys when average value multiplicity is small.



k-mers	Locations			
k_1	(g_1, w_2)	(g_2, w_3)	(g_3, w_5)	(g_4, w_1)
k_2	(g_1, w_2)	(g_2, w_1)		
k_1	(g_4, w_9)			
k_3	(g_1, w_3)	(g_3, w_2)	(g_3, w_4)	(g_5, w_2)
...	...			

Fig. 8.10.: Multi Bucket Hash Table Layout. Each slot maps a k-mer to a fixed number of locations (4 in this example).

In order to overcome these limitations we utilized WarpCore's modular design to create a novel multi-value hash table variant. Our *Multi Bucket Hash Table*, instead of storing a single key-value pair per slot, stores a key and fixed number (*bucket size*) of associated values in each slot. This layout is illustrated in Figure 8.10. We extended WarpCore's probing scheme to respect the new layout as following. In addition to initializing each key slot with k_e, all value slots are filled with an empty-indicator v_e. When inserting key-values pairs into the hash table, a thread group first checks for existing keys matching the to be inserted key and subsequently check the associated bucket of values slots for v_e. Figure 8.11 shows the extended insertion process of a key-value pair (k, v) by a cooperative group of 4 threads. The steps are explained in the following:

(1) The starting index for the whole group of threads is determined by the outer probing scheme h.

(2) The group loads consecutive keys from the hash table according to the group size.

(3a) Each thread checks if its assigned slot is equal to k, then all threads use a group voting function to communicate the results to the whole group.

(4a) Select the highest candidate index for insertion. If the key wasn't found continue with step 3b.

(5a) The group loads associated values from the hash table according to the group size.

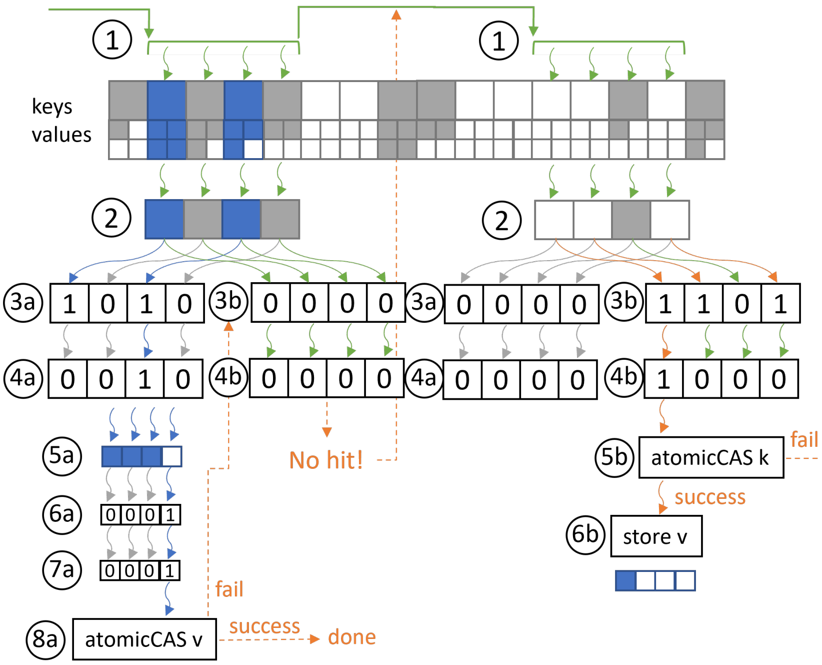


Fig. 8.11.: Insertion of a key-value pair (k, v) into our Multi Bucket Hash Table using 4 values per key slot.

(6a) Each thread checks if its assigned value slot is free, then all threads use a group voting function to communicate the results to the whole group.

(7a) Select the thread associated with the lowest candidate slot index as leader.

(8a) The selected thread tries to insert the value using an atomic CAS. If the slot was occupied by another value in the meantime, the CAS fails and steps 7a and 8a are repeated as long as the group has remaining candidate slots. Otherwise the probing continues in step 3b.

(3b) Each thread checks if its assigned key slot is free, then all threads use a group voting function to communicate the results to the whole group.

(4b) Select the thread associated with the lowest candidate slot index as leader. If the group found no free slot at the current indices the outer probing scheme determines the next index where the process repeats from step 1.

(5b) The selected thread tries to insert the key using an atomic CAS. If the slot was occupied by another key in the meantime, the CAS fails and steps 4b and 5b are repeated as long as the group has remaining candidate slots. Otherwise the probing continues from step 1.

(6b) In case of a successful key insertion, the associated value can be stored with an relaxed write operation in the first value slot.

Tab. 8.2.: Reference genome set used as database.

Database	Genomes	Size on disk
RefSeq 202 Bacteria 1/6 (RS202 B1/6)	3,286	12.6 GB

Tab. 8.3.: Properties of used metagenomic read datasets.

Dataset	Sequences	Min length	Max length	Average length
HiSeq	10,000,000	19	101	92.3
MiSeq	10,000,000	19	251	156.8

8.4.4 Performance Evaluation

We compared the memory requirements as well as build and query speeds of different Multi Bucket Hash Table configurations using either AoS or SoA layout and bucket sizes of 1, 2, 4 or 8 values per key slot. A bucket size of one is equivalent to WarpCore's Multi Value Hash Table. For a fair comparison of the Multi Bucket Hash Table configurations in the MetaCache context we chose a subset of the NCBI RefSeq Release 202 which was small enough to fit entirely into each resulting hash table on a single GV 100 GPU with 32 GB of memory. The subset consists of one sixth of the bacterial genomes and accumulates to about 12.6 GB (see Table 8.2). Based on MetaCache v2.2.2 we compiled a binary for each hash table variant. We executed MetaCache-GPU's on-the-fly mode to build a database from the set of reference genomes and run the query immediately afterwards. For querying we used the metagenomic datasets HiSeq and MiSeq, each containing 10 million single-end reads, with an average length of 92 and 156 base pairs, respectively. See Table 8.3 and Section 9 for more details on the datasets. We chose to suppress the per-read output in the query phase to reduce the impact of disk I/O on our experiments. After an initial warm-up run to load all data into RAM, runtimes measurements for build and query phase were performed five times and we report the average throughput of each phase. All experiments were performed on the following System:

System 2 2x Intel Xeon Gold 6238 (22 cores @ 2.1-3.7 GHz) with 192 GB DDR4 RAM, 2x NVIDIA Quadro GV100, each with 32 GB HBM2 memory, Ubuntu 20.04, GCC 9.3.0, CUDA 11.5.

Tab. 8.4.: Key-Value Distribution of RS202 B1/6. Max, Mean, Stddev and Skewness refer to values per key. Stddev = standard deviation.

Database	Unique keys	Total values	Max	Mean	Stddev	Skewness
RS202 B1/6	246,329,568	1,843,224,995	254	7.48	19.94	7.564

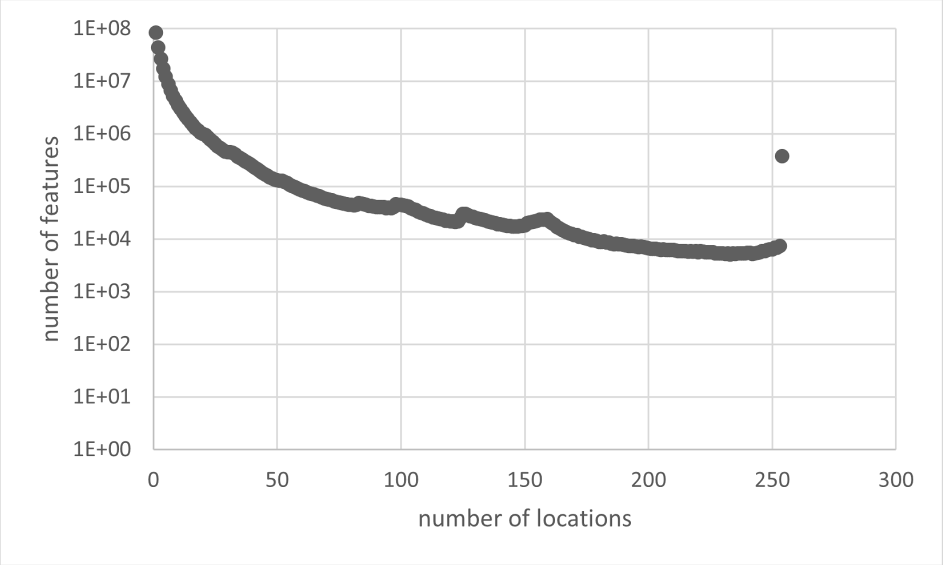


Fig. 8.12.: Histogram showing the key-value distribution of RS202 B1/6.

Memory Efficiency

Our hash tables are statically allocated at the beginning of MetaCache-GPU's build process in order to avoid costly reallocation and rehashing in case the current size was not sufficient. Hence, we allocate most of the GPU's memory for the hash table (about 29.4 of the 32 GB of a GV100) and leave 2.6 GB for buffers required in the build and query phases.

Performing MetaCache's minhashing algorithm on the RS202 B1/6 genomes resulted in 246,329,568 unique features and a total of 1,843,224,995 locations that have to be stored in the database. Figure 8.12 and Table 8.4 show the distribution of locations (values) per feature (key) for this dataset. In the histogram we can see an exponential distribution where a large number of features only appear at a few locations, in fact more than 80 million keys map to a single value. On the opposite end of the spectrum several thousand keys map to lists of 200 values or more. Note, that MetaCache limits the number of locations per features to 254, thus all keys which would map to more than that are collected in the last bin of the histogram.

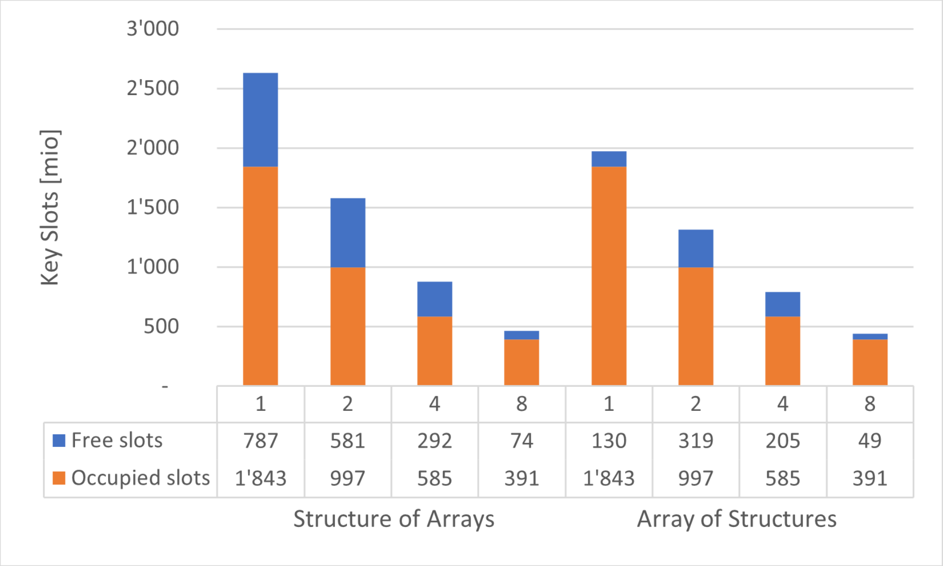
Tab. 8.5.: Multi Bucket Hash Table occupancies for different layouts and bucket sizes. Number of slots in millions.

Layout	Bucket size	Total slots Key	Total slots Value	Occupancy Key	Occupancy Value	Occupancy Memory
SoA	1	2’630	2’630	70%	70%	70%
SoA	2	1’578	3’157	63%	58%	59%
SoA	4	877	3’507	67%	53%	54%
SoA	8	465	3’717	84%	50%	52%
AoS	1	1’974	1’974	93%	93%	93%
AoS	2	1’316	2’632	76%	70%	72%
AoS	4	790	3’159	74%	58%	61%
AoS	8	439	3’515	89%	52%	56%

For 32-bit features and 64-bit locations (32-bit genome id, 32-bit window id) the total memory requirement accumulates to 14.7 GB of data. Due to the structure of our Multi Bucket Hash Table many keys are stored multiple times in the hash table, resulting in higher memory requirements. The actual number of times a key needs to be stored depends on the bucket size: the bigger the buckets are the more values can be stored per key slot and therefore less duplicate keys need to be stored. Compared to SoA the AoS layout entails an additional overhead because keys are also padded to 64-bit, doubling the memory consumption of the key slots.

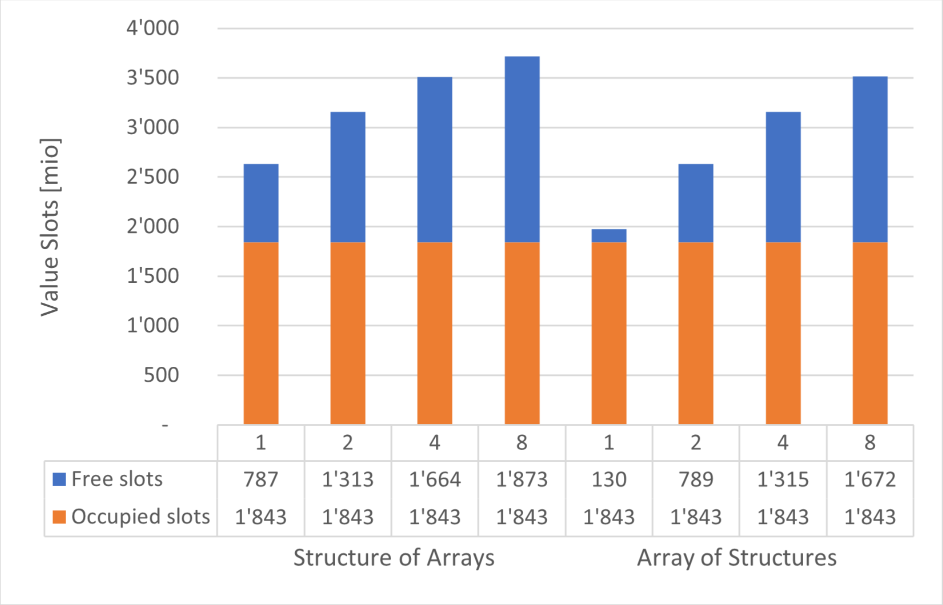
Table 8.5 reveals the total number of created slots for keys and values when using different layouts and bucket sizes as well as the occupancy of key slots, values slots and memory. We can see that the number of key slots reduces dramatically if we increase the bucket size, while the number of available value slots increases. Ideally, this would allow us to store less duplicate keys while being able to store more total values. Looking at the memory occupancy we see that this strategy works as intended, with greater bucket sizes the amount of memory required to store all keys and values is reduced.

Figures 8.13 and 8.14 visualize the number of free and occupied slots for keys and values, respectively, using different hash table layouts and bucket sizes. The sum of free and occupied slots equals the total number of available slots. While the number of occupied key slots drops with increasing bucket size, the percentage of occupied slots rises again for a bucket size greater than two. For a bucket size of eight we reach key occupancies of 84% and 89% for SoA and AoS, respectively. WarpCore’s performance results showed that high occupancy may lead to diminished insertion and deletion speeds. We verify these results in the following sections. While the



	Structure of Arrays				Array of Structures			
	1	2	4	8	1	2	4	8
🟦 Free slots	787	581	292	74	130	319	205	49
🟧 Occupied slots	1'843	997	585	391	1'843	997	585	391

Fig. 8.13.: Multi Bucket Hash Table key occupancies for SoA and AoS layouts and bucket sizes 1, 2, 4 and 8. Number of slots in millions.



	Structure of Arrays				Array of Structures			
	1	2	4	8	1	2	4	8
🟦 Free slots	787	1'313	1'664	1'873	130	789	1'315	1'672
🟧 Occupied slots	1'843	1'843	1'843	1'843	1'843	1'843	1'843	1'843

Fig. 8.14.: Multi Bucket Hash Table value occupancies for SoA and AoS layouts and bucket sizes 1, 2, 4 and 8. Number of slots in millions.

number of inserted values stays constant we find that bigger buckets allow for more values to be stored and thus the value occupancy decreases accordingly.

The AoS layout requires twice as much memory for keys as the SoA layout. Hence, the number of key and value slots is greatly decreased when keeping the total hash table size fixed. From a memory standpoint the AoS layout is not suitable if we want to store as many key-values pairs as possible.

Build Performance

Depending on the chosen layout and bucket size MetaCache-GPU's build phase took 9.5 to 17.2 seconds to process the reference genomes and construct the database on the GPU. The throughput in gigabytes of input data per second is shown in Figure 8.15 for SoA and AoS layout using different bucket sizes. For the same bucket size the hash table performs better with SoA in all cases compared to AoS layout. However, the difference for buckets of size one is most pronounced which can be linked to the big difference in key occupancy for this bucket size. A lower occupancy enables faster probing because candidate slots can be found more frequently. For the same reason the throughput increases for larger bucket sizes. The best build performance is achieved using buckets of size eight and SoA layout which is 40% faster than using buckets of size one which resembles WarpCore's Multi Value Hash Table. While for buckets of size four we still reach 96% of the maximum throughput, the highest throughput achieved for the AoS layout reaches 91% of the maximum also using a bucket size of four.

Query Performance

As we can see in Figure 8.16 the hash table configuration has a huge impact on query performance. For both SoA and AoS layout we achieve twice the throughput with a bucket size of four compared to buckets of size one. Here again the SoA layout leads to higher performance than AoS. The best performance is achieved in SoA layout for bucket sizes four and eight which show similar performance independent of the queried dataset.

Compared to HiSeq the longer reads in MiSeq result in two minhashing windows per read and thus twice the amount of hash table queries. This leads to more retrieved reference locations which need to be analyzed and therefore reduces the number of reads processed per minute. Apart from the lower throughput numbers

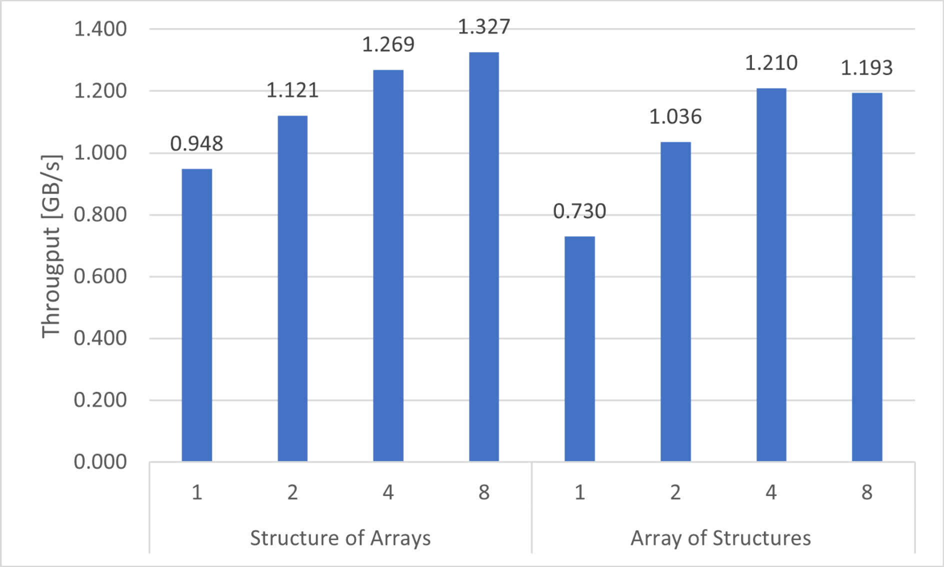


Fig. 8.15.: Multi Bucket Hash Table build performance for SoA and AoS layouts and bucket sizes 1, 2, 4 and 8.

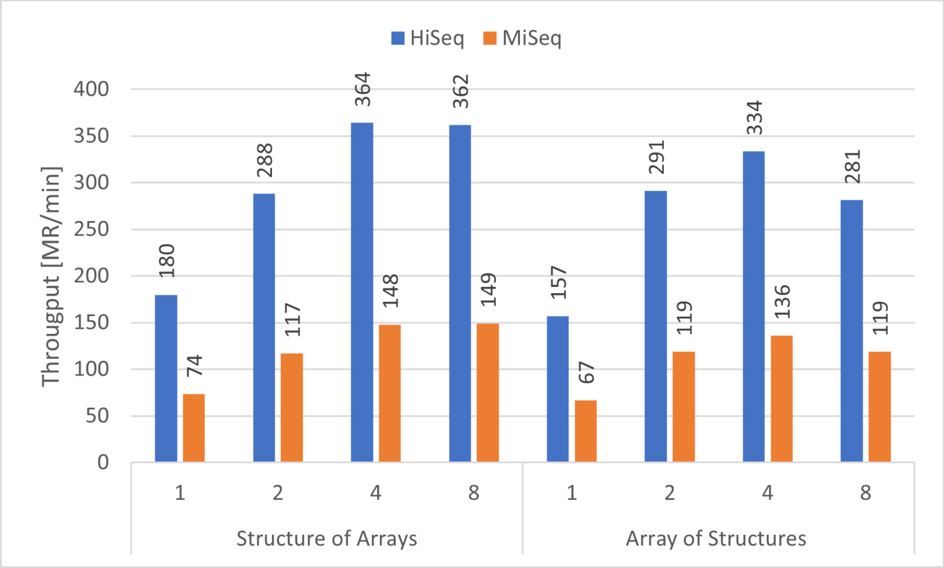


Fig. 8.16.: Multi Bucket Hash Table query performance for SoA and AoS layouts and bucket sizes 1, 2, 4 and 8. Throughput in million reads per minute.

MetaCache-GPU's query performance for MiSeq similarly depends on the hash table layout and bucket size.

8.4.5 Conclusion

Using WarpCore we designed and implemented a new open addressing hash table variant, where each slot consists of a key mapped to a small, fixed number (a *bucket*) of values. Analogous to WarpCore's Multi Value Hash Table a key can occur in multiple slots which allows it to be associated to an arbitrary number of values. Compared to WarpCore's hash table variants this new variant is a better fit to the various key-value distributions that we encountered in our experiments. Our Multi Bucket Hash Table consumes less memory, which conversely allows for more data to be stored per GPU. For a fixed set of reference genomes we achieve reduced hash table occupancy which leads to quicker probing, resulting in up to 40% faster builds and twice the query performance.

In our experiments the SoA layout showed superior to AoS, therefore we set the former as our default layout in MetaCache-GPU. Peak throughput was achieved for bucket sizes of four or eight values with only a negligible difference between the two. Due to the lower key occupancy when using four values per bucket instead of eight we decided on this size as our default value, allowing to insert additional keys if necessary.

8.5 Segmented Sort

Sorting the lists of target locations resulting from database queries is a time-consuming step in our GPU pipeline (Section 8.2.2 Step 6). For each batch of queries a segmented sort algorithm is employed on the GPUs to efficiently sort multiple location lists in parallel.

Segmented sort is the problem of sorting multiple independent lists (*segments*) of keys or key-value-pairs of arbitrary sizes. In a sequential approach a single thread could simply sort one segment after the other. However, in a parallel implementation load-balancing problems may emerge from the varying number of elements per segment when distributing work among multiple threads. There exist several different approaches to solve these problems on GPUs.

8.5.1 Related Work

One simple solution for segmented sort is to transform the problem into a global sort of a single list. In order to achieve this, the input data has to be augmented by adding segment IDs to each element, and then a global sort primitive can be called which respects the IDs as well as the original keys [@7, 24]. This not only adds memory overhead but also increases computational complexity. A similar approach is used by other GPU programs [68, 122] which reformulate their problems to be able to call global sort from support libraries.

Another strategy, employed by ModernGPU [@4], is to use a merge sort algorithm which respects the segment boundaries. They first assign a fixed number of elements to each thread which are rearranged using a sorting network without crossing segments. Subsequently, neighboring blocks of elements are merged as long as they contain elements from a common segment. The merge steps continue until even the largest segments are completely sorted.

The CUB library [@14] which is included in the CUDA toolkit uses a radix sort algorithm for their global and segmented sort primitives. For the segmented sort they spawn as many thread blocks as there are segments. Each block is responsible for sorting a designated segment regardless of segment size. This may waste resources on small segments while larger segments could benefit from the use of more parallel processing power than a single block can provide. CUB's documentation states that this strategy was suited for larger segment sizes ("tens of thousands of items and more"[3]).

Since version 1.15 CUB provides an alternative segmented sort algorithm which improves runtimes for smaller or imbalanced segment sizes. They partition the segments according to their size into different groups which are processed by different sorting strategies. Large segments are still sorted using radix sort while smaller segments are sorted in a separate kernel using a merge sort implementation.

Hou et al. [36] also try to take advantage of the data distribution by treating segments of different size separately. Their segmented sort consists of multiple kernels, each tailored to a specific range of segment sizes. In each CUDA kernel, threads operate conjointly sorting the elements of a segment using bitonic sort. The corresponding sorting networks can be implemented efficiently exploiting fast register accesses and warp shuffles. For larger segments with more than 2048 elements they first sort chunks of elements with bitonic sort and subsequently merge the partial results until the whole segments are sorted. Their implementation has been shown to outperform other libraries like CUB (radix sort version) and ModernGPU , however, they only provide primitives for key-value sort and require additional memory allocations.

8.5.2 Improvements

We adapted the approach by Hou et al. to allow for key-only sorting by refactoring all involved sorting kernels. Furthermore, we rewrote the interface to accept begin and end offsets for all segments instead of expecting that all segments are stored consecutively in memory. In addition, our implementation allows to pre-allocate temporary memory for the algorithm which avoids synchronization overhead caused by intermediate memory allocations and allows to reuse the memory for following invocations. Finally, we bundle the sorting kernels in a CUDA graph which speeds up the scheduling and removes gaps between kernel executions.

[3] https://nvlabs.github.io/cub/struct_device_segmented_sort.html

8.5.3 Performance Evaluation

To evaluate the different segmented sort approaches, we extracted batches of segment sizes from MetaCache's query pipeline to generate benchmark cases representing real-world classification runs. Experiments were conducted on the following system:

System 2 2x Intel Xeon Gold 6238 (22 cores @ 2.1-3.7 GHz) with 192 GB DDR4 RAM, 2x NVIDIA Quadro GV100, each with 32 GB HBM2 memory, Ubuntu 20.04, GCC 9.3.0, CUDA 11.5.

We used all complete bacterial genomes from NCBI RefSeq [84] Release 202 for database building. Due to the GPU memory limitations on this system, we split the set of genomes into two equally sized parts (see Table 8.6) and constructed a database for each using both GV100 GPUs. We ran MetaCache's query mode for the metagenomic datasets HiSeq and MiSeq containing 10 million reads each, with an average length of 92 and 156 base pairs, respectively. See Table 8.7 and Section 9 for more details on the datasets.

Tab. 8.6.: Reference genome sets used for databases.

Database	Genomes	Size on disk
RefSeq 202 Bacteria 1 (RS202 B1)	9,442	37 GB
RefSeq 202 Bacteria 2 (RS202 B2)	9,391	37 GB

Tab. 8.7.: Properties of used metagenomic read datasets.

Dataset	Sequences	Min length	Max length	Average length
HiSeq	10,000,000	19	101	92.3
MiSeq	10,000,000	19	251	156.8

For MetaCache's default batch size of 8192 and window size of 127 base pairs this resulted in 2442 and 4798 batches of windows, respectively, which are queried against the databases and involve a segmented sort of candidate locations. We extracted the segment sizes for all batches and stored them on disk in order to enable benchmarks decoupled from the complete MetaCache pipeline. Table 8.8 shows some statistical properties of the distribution of segments sizes generated by querying HiSeq and MiSeq against the two databases from Table 8.6. In total all batches of HiSeq encompassed more than 11 billion items, while all batches of MiSeq add up more than 27 billion items due to the longer read lengths. We can see that some segments are empty because for some sequence windows none of the selected

Tab. 8.8.: Statistical properties of the segment sizes generated by different dataset queries. Stddev = standard deviation.

Dataset	Database	Min	Max	Mean	Stddev	Skewness	Total items
HiSeq	RS202 B1	0	4064	591.5	588.3	-1.739	11.829 bn
HiSeq	RS202 B2	0	4064	576.5	568.5	-1.584	11,529 bn
MiSeq	RS202 B1	0	9846	1411.5	1133.7	-4.812	28.230 bn
MiSeq	RS202 B2	0	9866	1386.1	1110.9	-4.788	27.721 bn

k-mers were found in the databases, while other segments reach the maximum size possible. For HiSeq, where each read fits into a single window, MetaCache generates a minhashing sketch of size 16 which results in at most $16 \cdot 254 = 4064$ candidate locations for a maximum location list size of 254 for each k-mer in the database. Using the default window stride of 112 base pairs the longest reads in MiSeq are split into three windows where the last window covers only a few bases resulting in less than three times the maximum segment size of HiSeq.

The following results show the throughput achieved by the different segmented sort approaches when performing the segmented sort operation for all batches of a dataset executed in a row.

Initially, MetaCache-GPU included a compaction step prior to the segmented sort step because the location lists resulting from the hash table query were stored at separate memory locations. This allowed using the original interface of the implementation by Hou et al. as well as ModernGPU's segmented sort, which expect that all segments are stored in consecutive memory locations. In contrast CUB and our improved implementation are able to handle separated segments by accepting lists of begin and end offsets. This eliminates the need for the compaction step reducing code complexity and execution time. Nevertheless, we investigated both the compacted case (*dense segments*) as well as the un-compacted case (*sparse segments*) in our benchmarks to be able to include ModernGPU and to study potential performance differences. Figures 8.17 and 8.18 show the results for the dense and sparse benchmark, respectively.

Comparing the results of the two databases RS202 B1 and B2 we find only negligible differences in throughput of less than 2% resulting from the differences in segment sizes. Our segmented sort sort gets slightly faster when processing more smaller segments while the other methods show slightly worse result. We see the same effect but much more pronounced comparing the results for HiSeq to MiSeq using a common database. Due to the larger segments produced by MiSeq, CUB's radix sort achieves almost twice the throughput compared to sorting segments for HiSeq.

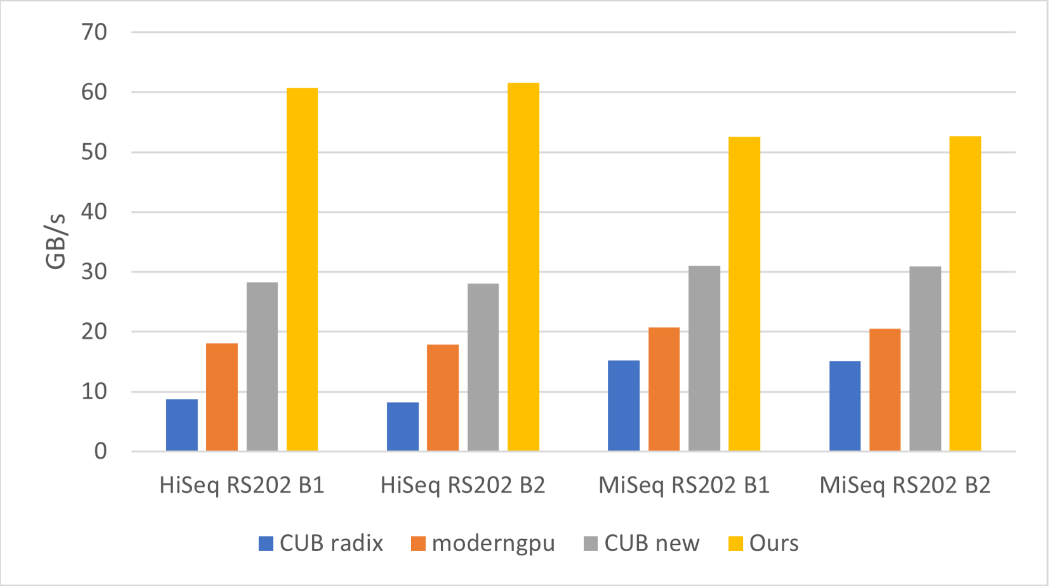


Fig. 8.17.: Throughput for dense segmented sort.

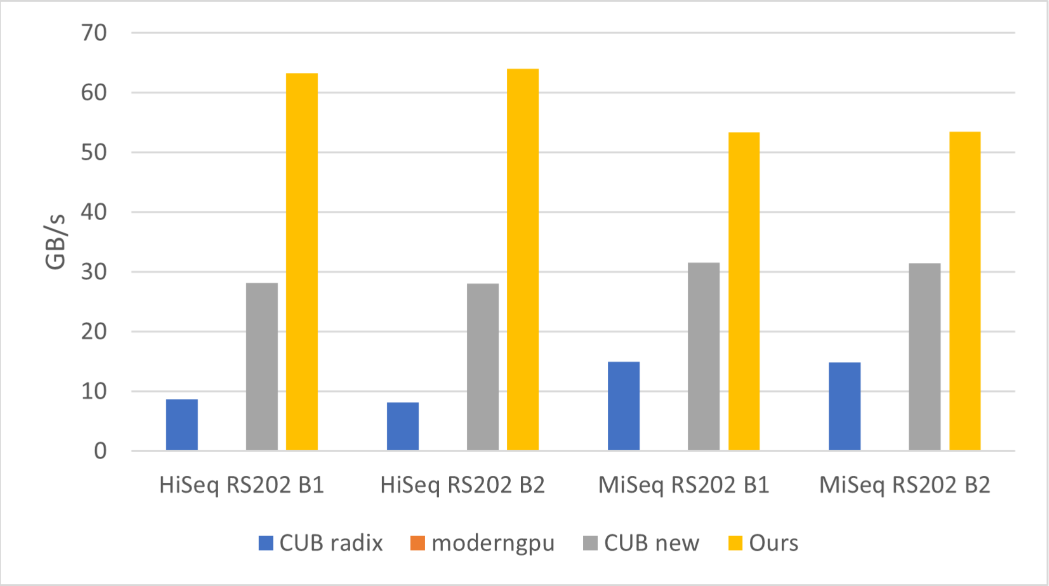


Fig. 8.18.: Throughput for sparse segmented sort.

However, CUB's segmented radix sort remains the slowest approach. ModernGPU and and CUB's new method are also benefiting slightly from larger segment sizes, increasing throughput by 15% and 10%, respectively.

In contrast, our approach uses specialized kernels for smaller segment sizes which results in 15-17% higher throughput for HiSeq than for MiSeq. This approach is very well suited for this use case. Our implementation outperforms all other tested methods by a significant amount resulting in speedups of 3.5-7.2, 2.5-3.4 and 1.7-2.2 compared to CUB's radix sort, ModernGPU and CUB's new segmented sort, respectively.

Looking at the sparse benchmark in comparison we only see small differences in performance. The largest variation is found for HiSeq, where our approach is faster by another 4%, which may result from improved memory access because individual segments are aligned to 128 byte memory addresses here. As mentioned before ModernGPU is not able to handle segments with gaps in between, so it was excluded from this benchmark.

8.5.4 Conclusion

Segmented sort is an important step in our GPU query pipeline. Although popular CUDA libraries like ModernGPU and CUB provide segmented sort primitives out of the box they are not necessarily the fastest solution. We showed that our modified implementation of the approach by Hou et al. is able to outperform CUB and ModernGPU by a large amount using bitonic sorting kernels optimized for small segment sizes. Our implementation allows to pre-allocate temporary memory avoiding synchronization overheads. Furthermore, its interface accepts begin and end offsets for segments instead of expecting that all segments are stored consecutively in memory, enabling us to skip the costly compaction step in the GPU pipeline.

8.6 Top Candidate Generation

The final kernel in MetaCache's GPU query pipeline combines steps (7) and (8) of Section 8.2.2 to generate the top candidates from the sorted lists of reference locations. Locations consist of a target ID g_i and a window ID w_j, denoting the j-th window in the i-th reference genome. Each warp of 32 threads running the kernel is used to find the best matching reference region for one read at a time by employing a sliding window approach. The sliding window size sws is determined by the length of the respective read and defines the maximum number of contiguous locations that can be accumulated to form a top candidate. The goal is to find the candidate(s) with the highest score, i.e., the highest number of matched locations in its sliding window range.

First, threads load the sorted locations and perform a segmented reduction to accumulate identical values (same target and window ID). This is repeated until at least $32 + sws - 1$ unique locations are collected in shared memory, so that every thread is able to calculate its own sliding window. Next, each thread has to inspect up to sws locations to determine if they belong to the same region by comparing their target and window IDs. Starting with its first locations each thread either accumulates the scores of consecutive locations or discards all following locations if their IDs are out of range. The resulting location ranges and their scores are potential top candidates.

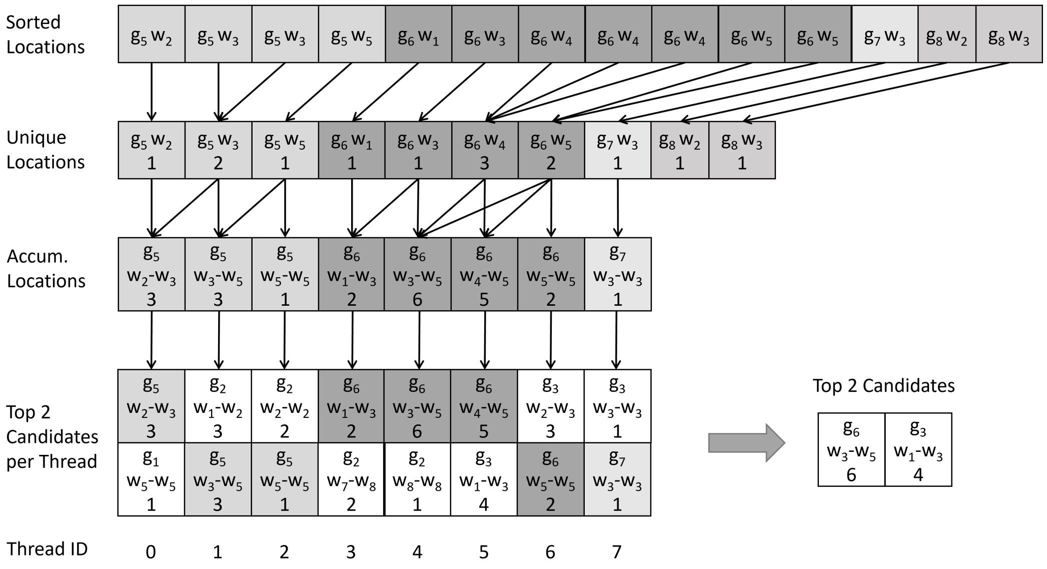

Sorted Locations
$g_5\,w_2$ | $g_5\,w_3$ | $g_5\,w_3$ | $g_5\,w_5$ | $g_6\,w_1$ | $g_6\,w_3$ | $g_6\,w_4$ | $g_6\,w_4$ | $g_6\,w_4$ | $g_6\,w_5$ | $g_6\,w_5$ | $g_7\,w_3$ | $g_8\,w_2$ | $g_8\,w_3$
Unique Locations
$g_5\,w_2$ 1 | $g_5\,w_3$ 2 | $g_5\,w_5$ 1 | $g_6\,w_1$ 1 | $g_6\,w_3$ 1 | $g_6\,w_4$ 3 | $g_6\,w_5$ 2 | $g_7\,w_3$ 1 | $g_8\,w_2$ 1 | $g_8\,w_3$ 1
Accum. Locations
g_5 w_2-w_3 3 | g_5 w_3-w_5 3 | g_5 w_5-w_5 1 | g_6 w_1-w_3 2 | g_6 w_3-w_5 6 | g_6 w_4-w_5 5 | g_6 w_5-w_5 2 | g_7 w_3-w_3 1
Top 2 Candidates per Thread
g_5 w_2-w_3 3 | g_2 w_1-w_2 3 | g_2 w_2-w_2 2 | g_6 w_1-w_3 2 | g_6 w_3-w_5 6 | g_6 w_4-w_5 5 | g_3 w_2-w_3 3 | g_3 w_3-w_3 1
g_1 w_5-w_5 1 | g_5 w_3-w_5 3 | g_5 w_5-w_5 1 | g_2 w_7-w_8 2 | g_2 w_8-w_8 1 | g_3 w_1-w_3 4 | g_6 w_5-w_5 2 | g_7 w_3-w_3 1
Thread ID
0 1 2 3 4 5 6 7
Top 2 Candidates
g_6 w_3-w_5 6 | g_3 w_1-w_3 4

Fig. 8.19.: Top Candidates Generation Kernel for an example warp size of 8 threads and an sliding window size of 3. The top 2 per thread already contain candidates from a previous iteration. g_i and w_j denote target and window IDs, respectively.

Because the number m of top candidates required to classify a read is small, each thread is able to maintain its own list of m ranges with highest hit count in registers. After each iteration of the candidate generation the local top lists are updated individually. Finally, after all locations for a read have been processed the whole warp generates the combined top hits list by using warp shuffles to find the highest scores. Figure 8.19 shows an example workflow for the kernel.

In the multi-GPU scenario different reference sequences are stored in hash tables of different GPUs. Consequently, all locations of the same reference are contained on the same GPU and top candidates can be determined independently on each GPU. This reduces the required communication to a minimum because only the top candidates from different GPUs need to be merged instead of globally gathering all locations from all GPUs. In our GPU pipeline the top m candidates from the preceding GPU are merged with the local results, and the combined top m candidates are sent to the subsequent GPU. The last GPU in the pipeline obtains the top m candidates of the whole database and copies the result back to the host. This concludes the query pipeline for a batch of reads on the GPU.

9 MetaCache-GPU Performance Evaluation

In order to evaluate the performance of the complete MetaCache-GPU pipeline we investigated the build and query phases for two different databases of reference genomes. The first set consists of genomes from NCBI RefSeq [84]. We included all complete archaeal, bacterial, fungal and viral genomes from RefSeq Release 202. For the second set we combined the RefSeq set with 31 food-related genomes from the All-Food-Sequencing (AFS) pipeline [48]. In contrast to the first set these animal and plant genomes consist of much longer sequences. Furthermore, some of the genomes are only available at scaffold level which results in hundreds of thousands of different target sequences per genome. Table 9.1 shows the number of included species for each database as well as their total sizes.

For the query phase we chose to test three metagenomic datasets with at least 10 million reads each. HiSeq and MiSeq contain single-end reads in FASTA format and were introduced by Wood and Salzberg [117], KAL_D is taken from AFS [48] and contains paired-end reads in the FASTQ fromat. Table 9.2 shows the total number of reads per dataset together with their minimum, maximum and average lengths. The datasets HiSeq and MiSeq represent bacterial mock communities consisting of reads from ten different species each, produced by Illumina sequencers of the same names. Finally, KAL_D is a real world dataset obtained by sequencing material from a sausage made from beef, mutton, pork and horsemeat.

Experiments were conducted on the following system:

DGX-1 Volta: Dual-socket Xeon E5-2698 v4 CPU (2x20 cores at 2.20 GHz) with 512 GB DDR4 RAM and 8 Tesla V100 GPUs, each with 32 GB HBM2 memory, CUDA 11.0, GCC 9.3.0.

Reference genomes, taxonomy files and datasets were loaded into a virtual RAM drive before executing a build or query to minimize the influence of slow I/O from the file system. Kraken2 [116] and MetaCache (CPU version) were executed using a maximum of 80 threads incorporating simultaneous multithreading on the 40 cores of the system. MetaCache-GPU was evaluated using a 4 GPU and an 8 GPU configuration. Additionally, we compare querying of the original paper version

Tab. 9.1.: Reference genome sets used for databases.

Database	Species	Size on disk
RefSeq 202	15,461	74 GB
All-Food-Seq 31	31	77 GB
AFS 31 + RefSeq 202	15,492	151 GB

Tab. 9.2.: Metagenomic read datasets.

Dataset	Format	Sequences	Min	Max	Average
HiSeq	FASTA single	10,000,000	19	101	92.3
MiSeq	FASTA single	10,000,000	19	251	156.8
KAL_D	FASTQ paired	26,114,376	101	101	101

(MetaCache v2.0) with an updated version (MetaCache v2.3) which among other improvements utilizes the sparse segmented sort from Section 8.5 to eliminate the compaction step in the query pipeline.

In the 4 GPU configuration our Multi Bucket Hash Table introduced in Section 8.4 needed 10% and 11% less memory than WarpCore's Multi Value and Bucket List Hash Table, respectively. It was the only hash table that could fit RefSeq202 on 4 GPUs without further restricting the number of locations per k-mer. The larger AFS31+RefSeq202 database did not fit in the memory of 4 V100 GPUs and therefore always uses 8 GPUs.

9.1 Build Performance

Table 9.3 presents the build performance of Kraken2 and MetaCache's CPU and GPU versions. As mentioned in Section 6.4.1 the CPU version of MetaCache is based on a two stage producer-consumer scheme which uses only three threads in total. Nevertheless, it is faster than Kraken2 which uses 80 threads. While CPU-based MetaCache and Kraken2 take more than an hour for building the RefSeq202 database and more than 3 or more than 4 hours for AFS31+RefSeq202, respectively, MetaCache-GPU is able to create the index structures in seconds to minutes. The speedup when using 8 GPUs is 64x and 61x for building and storing RefSeq202 compared to Kraken2 and MetaCache-CPU, respectively. For AFS31+RefSeq202 the speedups are 72x and 51x, respectively. Looking at the build time without writing the databases to the file system, the GPU version is 414 times and 272 times faster than the CPU version of MetaCache for RefSeq202 and AFS31+RefSeq202, Tab. 9.3.: Build performance for different databases. Total time includes build time and time for writing DBs to files.

Method	Build time	Total time	DB size	RAM
RefSeq 202 database:				
Kraken2	-	72 min	40 GB	46 GB
MC CPU	67 min	69 min	51 GB	71 GB
MC 4 GPUs	10.4 s	59.6 s	88 GB	1 GB
MC 8 GPUs	9.7 s	67.0 s	97 GB	1 GB
AFS 31 + RefSeq 202 database:				
Kraken2	-	256 min	110 GB	160 GB
MC CPU	194 min	201 min	127 GB	194 GB
MC 8 GPUs	42.7 s	3 min 31 s	176 GB	30 GB

respectively. Building the RefSeq202 database using 4 GPUs is a little slower than when using 8 GPUs because of less parallelization. But the overall database size is smaller and less data needs to be written to files, resulting in a slightly faster total runtime compared to the 8 GPU version. The speedups are 72x and 69x compared to Kraken2 and MetaCache-CPU, respectively.

9.2 Query Performance

Figures 9.1 and 9.2 reveal the query performance of Kraken2 and MetaCache for the databases RefSeq202 and AFS31+RefSeq202, respectively. MetaCache's speed depends on the number of reference locations found per read, which need to be merged and analyzed to get the final classification results. While the HiSeq dataset only contains reads which are smaller than MetaCache's window size, longer reads from MiSeq are split into two windows, resulting in more database queries and likely more retrieved locations. The KAL_D dataset on the other hand contains mostly reads from meat components and does not register many hits against the RefSeq202 database, resulting in the fastest queries for MetaCache. Kraken2 reaches higher speeds than MetaCache's CPU version when querying RefSeq202 with MiSeq and HiSeq, but needs more time for KAL_D. Since Kraken2 relies on mapping minimizers directly to taxa and does not need to process locations lists for hits in the database the speed in not affected much by the database size, Kraken2 is even able to increase the speed for the bigger AFS31+RefSeq202 database, while MetaCache's query speed takes a hit and is 6 to 14 times lower.

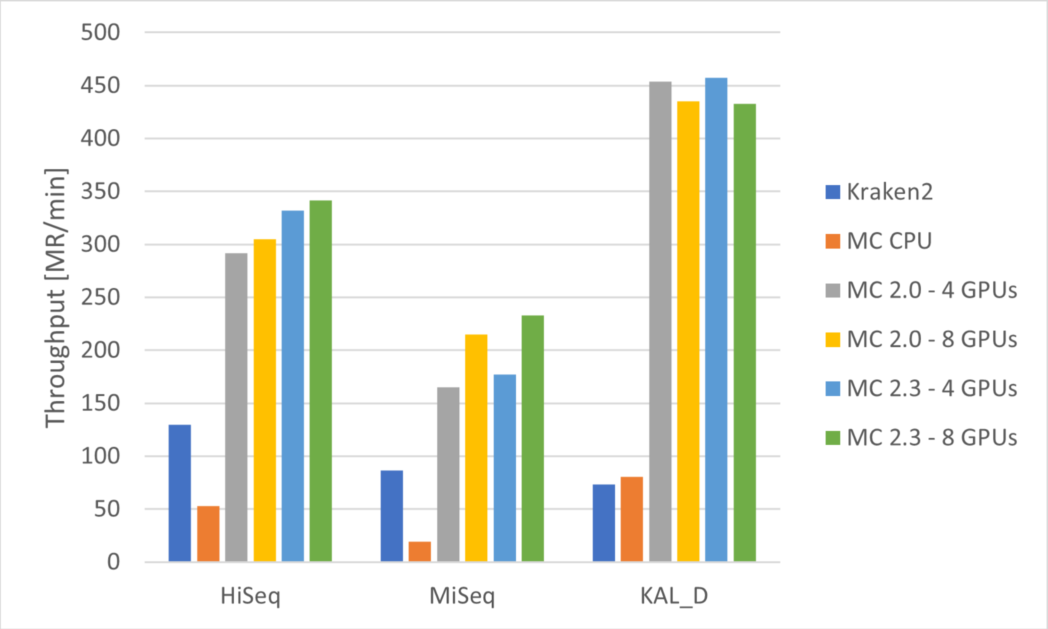


Fig. 9.1.: Query performance for querying different datasets against RefSeq202 database. Query speed in million reads per minute.

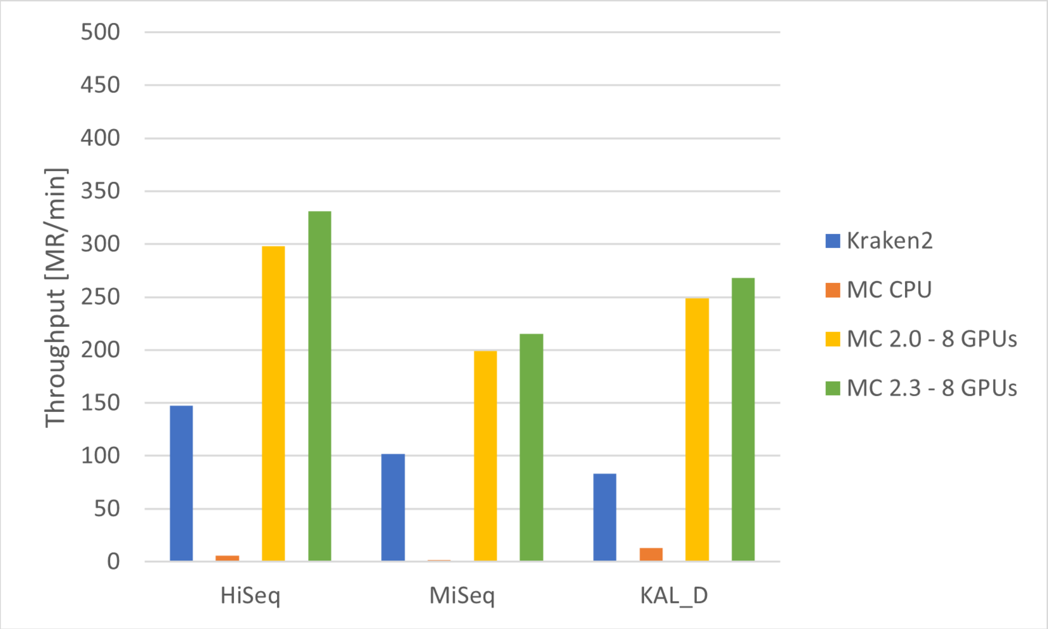


Fig. 9.2.: Query performance for querying different datasets against AFS31+RefSeq202 database. Query speed in million reads per minute.

Tab. 9.4.: MetaCache-GPU query performance comparison. Query speed in million reads per minute.

Dataset	Method	Output	RefSeq 202		AFS31+RefSeq202	
			4 GPUs	8 GPUs	4 GPUs	8 GPUs
HiSeq	MC 2.0	per Read	292	305	–[1]	298
	MC 2.3	per Read	331	341	–[1]	331
	MC 2.0	Summary	362	405	–[1]	405
	MC 2.3	Summary	405	460	–[1]	436
MiSeq	MC 2.0	per Read	165	215	–[1]	199
	MC 2.3	per Read	177	232	–[1]	215
	MC 2.0	Summary	166	228	–[1]	209
	MC 2.3	Summary	177	245	–[1]	225
KAL_D	MC 2.0	per Read	454	435	–[1]	249
	MC 2.3	per Read	457	432	–[1]	268
	MC 2.0	Summary	463	437	–[1]	249
	MC 2.3	Summary	461	437	–[1]	268

[1] 4 V100 GPUs do not provide enough memory for AFS31+RefSeq202.

Note that Kraken2 can only map reads to candidate taxa while MetaCache is able to map reads to the most likely locations of origin within reference sequences and thus produce candidate regions for further downstream analysis like, e.g., alignments.

MetaCache's GPU version however is not so much affected by the database size and achieves high speeds for both RefSeq202 and AFS31+RefSeq202. It is able to outperform Kraken2 and MetaCache's CPU version on all datasets. MetaCache version 2.3 is even able to improve the performance by up to an additional 14%. Compared to Kraken2 MetaCache-GPU (version 2.3) is 2.0-6.2 times faster for RefSeq202 and 2.1-3.3 times faster for AFS31+RefSeq202, compared to MetaCache-CPU it is 2.9-12.1 and 20.3-165 times faster, respectively.

Table 9.4 compares the query performance of MetaCache 2.0 and 2.3 for different numbers of GPUs and different output options. For HiSeq we can see that suppressing the per-read output has a big impact on the performance, while MiSeq and KAL_D stay mostly unaffected. HiSeq contains single-end reads which are small enough to fit into a single window in MetaCaches algorithm, while most reads from MiSeq will be processed in two windows. Both reads in a read pair from KAL_D are also processed as two separate windows. More windows result in more minhash sketches which have to be queried which in term results in greater processing time. However, processing and output of different batches are overlapped. For MiSeq and KAL_D the processing time is large enough to hide the result output completely, while many batches from HiSeq are processed too fast for the output to keep pace. Nevertheless, we see runtime improvements for version 2.3 with and without per-read output.

Comparing MetaCache 2.0 and 2.3 we see no significant difference in performance for KAL_D when querying the RefSeq202 database because this dataset mainly consists of reads from animal genomes which are not included in this database. Therefore, only few results need to be processed and the improvements in version 2.3 are less relevant. For HiSeq and MiSeq, however, version 2.3 achieves 7-14% increased performance querying RefSeq202. For queries against AFS31+RefSeq202 we observe improvements of 8-11% for all datasets.

9.3 Performance Breakdown

Figures 9.3 and 9.4 illustrate the average runtimes of the components of the query pipeline explained in Section 8.2 when querying a batch of reads from different datasets against the RefSeq202 and AFS31+RefSeq202 database, respectively. In MetaCache version 2.0 creating the minhashing sketch from the reads and querying the database takes 19-27% of the time while the rest is spent on processing the retrieved location lists. In contrast, in version 2.3 this first kernel takes 25-33% of the total time because its absolute runtime is increased and the compaction step is missing from the total. The increased runtime derives from atomically summing the location list sizes, which was done in the compaction step in version 2.0, and from code changes to allow for longer reads which resulted in increased number of registers required per thread leading to lower GPU occupancy. In a future version the code path for long reads should ideally execute a separate kernel to avoid performance penalties for short reads. Nevertheless, the time needed until the segmented sort can begin is reduced by 8-27%. Segmented sort takes the biggest share of the pipeline and is responsible for about halve of the total runtime. The top candidate generation takes 16-20% in version 2.0 and 17-21% in version 2.3.

For HiSeq and MiSeq the execution time on the GPU is smaller when using 8 GPUs with the AFS31+RefSeq202 database compared to 4 GPUs with the RefSeq202 database because the relevant bacterial genomes are distributed among all GPUs which means less results to be process per GPU. The additional 31 food-related genomes only cause a small overhead for these datasets. For KAL_D on the other hand we see the opposite effect. Due to the small number of hits in the RefSeq202 database the query processing is much faster compared to the AFS31+RefSeq202

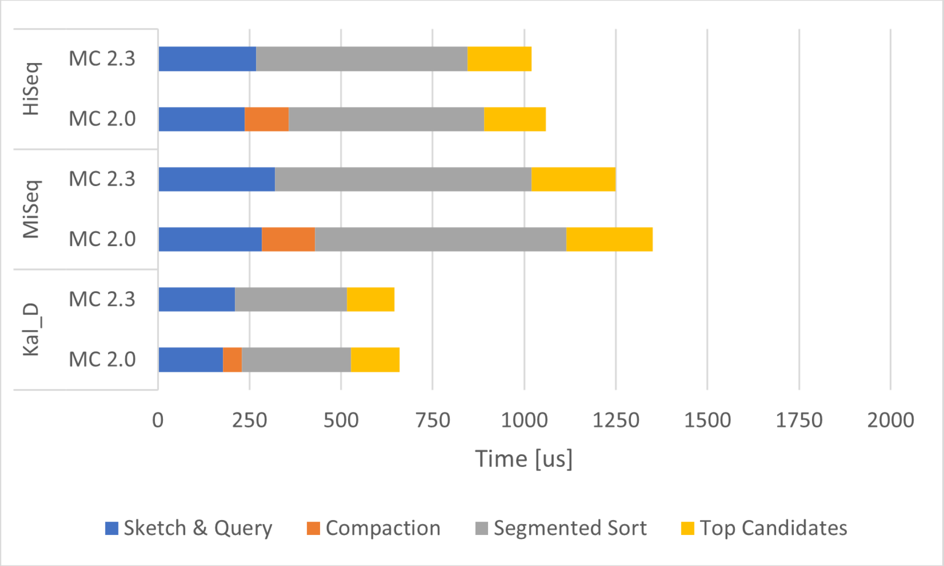


Fig. 9.3.: Performance breakdown for queries against RefSeq202 database using 4 GPUs.

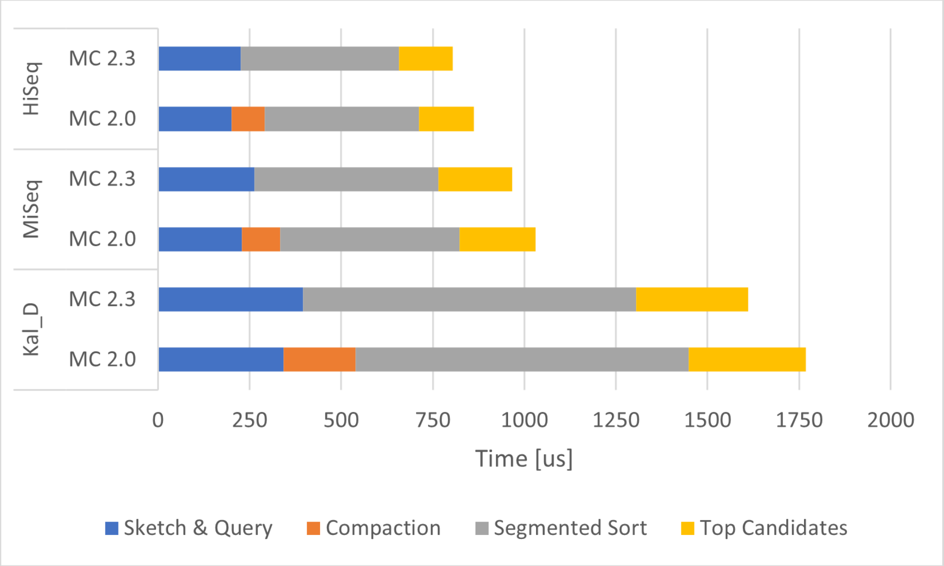


Fig. 9.4.: Performance breakdown for queries against AFS31+RefSeq202 database using 8 GPUs.

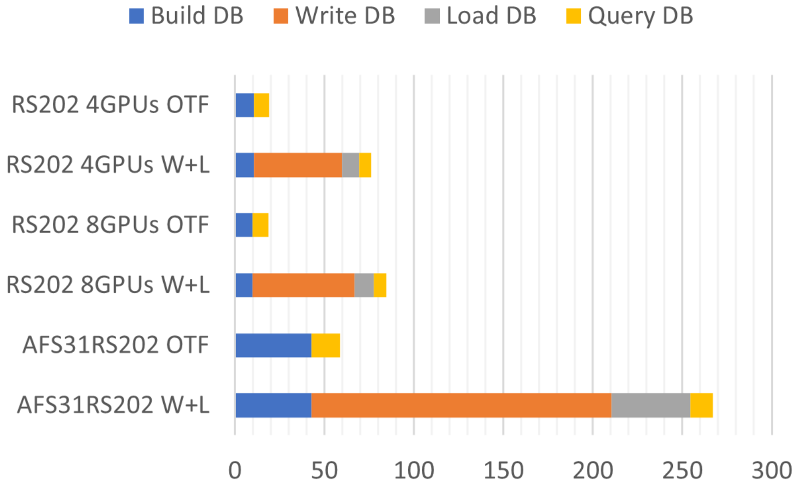


Fig. 9.5.: Runtime comparison of our on-the-fly (OTF) mode to separate build and query execution (W+L) for different databases, querying KAL_D dataset.

database. Most of the reads in KAL_D belong to meat components whose genomes are only contained in the larger database.

9.4 On-The-Fly Mode

Figure 9.5 compares the runtimes of separate build and query to the on-the-fly mode mentioned in Section 6.4 using multiple GPUs. Because of the high GPU build speed, most of the time in the build phase is actually spent writing the database to the file system. Loading the database takes almost the same time as building it from scratch. When using the on-the-fly mode the database can be queried without having to write and reload the database. Table 9.5 shows that the time needed until a query can be executed is greatly reduced when using MetaCache-GPU in on-the-fly mode. MetaCache's GPU databases are ready for use in under a minute which translates to a speedup of 360-450 compared to Kraken2. Note, that the query speed of the GPU hash table used for building is lower than that of the one used in separate query execution resulting in about 20% less performance. Nevertheless, the on-the-fly query mode can be beneficial in situations where a database is not queried more than once and does not need to persist on disk.

Tab. 9.5.: Comparison of time needed until a query can be executed when using MetaCache's on-the-fly (OTF) mode. TTQ is Time-to-Query.

Method	Build	Load	TTQ	Speedup
RefSeq 202 database:				
Kraken2	72 min	23 s	73 min	1.0
MC CPU OTF	67 min	-	67 min	1.1
MC 4 GPUs OTF	10.4 s	-	10.4 s	420
MC 8 GPUs OTF	9.7 s	-	9.7 s	450
AFS 31 + RefSeq 202 database:				
Kraken2	256 min	63 s	257 min	1.0
MC CPU OTF	201 min	-	201 min	1.3
MC 8 GPUs OTF	42.7 s	-	42.7 s	360

9.5 Query Accuracy

To analyze the query accuracy we compared classification results for HiSeq and MiSeq from MetaCache's versions and Kraken2 to the ground truth. Table 9.6 reveals the precision and accuracy using the RefSeq202 database with different methods. On the genus level Kraken2 offers greater sensitivity for the HiSeq datasets and similar sensitivity for MiSeq. The genus-level precision over 99% is comparable for Kraken2 and MetaCache with a small advantage for Kraken2 regarding MiSeq and MetaCache for the other two datasets. MetaCache is able to surpass Kraken2's accuracy on the species level for HiSeq and MiSeq, yielding 5% and 12% more sensitivity, respectively. MetaCache's precision is also higher for HiSeq, but lower for MiSeq.

Compared to MetaCache's CPU version the GPU variants are able to improve the accuracy. The reason for this is that using multiple database parts allows to store more locations for each k-mer, which are lost in the CPU version due to the enforced bucket limit. The additional location information leads to a better sensitivity and precision in most cases, only species-level results for HiSeq are slightly worse. This effect increases when using more GPUs.

Note, that we only tested Kraken2 and MetaCache with default parameters. Both algorithms allow the user to choose a different hit threshold which defines how many database hits are necessary to classify a read. Lowering the threshold typically trades precision for sensitivity while an increased threshold may improve precision at the cost of sensitivity.

Tab. 9.6.: Classification accuracy using RefSeq 202 database.

Dataset	Method	Species		Genus	
		Prec.	Sens.	Prec.	Sens.
HiSeq	Kraken2	82.52%	58.39%	99.09%	88.46%
	MC CPU	**89.41**%	**63.68**%	99.20%	81.36%
	MC 4 GPUs	88.70%	62.61%	**99.36**%	82.32%
	MC 8 GPUs	88.81%	62.63%	**99.36**%	**82.40**%
MiSeq	Kraken2	**77.91**%	48.53%	**99.38**%	93.25%
	MC CPU	72.28%	60.67%	99.21%	93.23%
	MC 4 GPUs	73.07%	61.55%	99.37%	93.82%
	MC 8 GPUs	73.53%	**61.99**%	99.37%	**93.92**%

For the KAL_D dataset there is no true per-read mapping available, only the ratio of meat components is known. To examine this dataset we queried the AFS31+RefSeq202 database which includes the corresponding genomes. Using MetaCache's abundance estimation functionality achieved quantification results close to the true ratios with a accumulated deviation of 6.5% and 2.5% false positives for the GPU version and a deviation of 16.0% and 2.0% false positives for the CPU version. In contrast, comparing the species results from Kraken2's sample report to the truth yielded a deviation of 21.4% and 7.5% false positives.

MetaCache Conclusion 10

The steadily increasing amount of available reference genomes and NGS data establishes the need for efficient and highly optimized processing approaches. In this work we have presented MetaCache-GPU – an alignment-free method for metagenomic read classification on CUDA-enabled GPUs based on massively parallel construction and querying of a novel hash table structure for k-mers. MetaCache-GPU's on-the-fly mode enables classification pipelines that can be rapidly updated to make use of the latest reference genomes or use custom reference genome sets on demand achieving over two orders-of-magnitude speedup compared to Kraken2 and the CPU version of MetaCache while still being memory-efficient. This is particular important for the analysis of complex biological matters such as food stuff which often requires custom reference databases where the size of individual genomes can exceed several gigabytes (e.g., plant genomes).

As part of MetaCache-GPU we investigated several components of the pipeline like sequence I/O, hash tables for k-mer index construction and querying as well as segmented sort. All these are common concept in bioinformatics. Thus, the introduced methods could easily be adapted to related NGS tasks such as read mapping or long-read-to-long-read alignment.

MetaCache-GPU is publicly available at https://github.com/muellan/metacache.

Part III

Future Work and Conclusion

11 Future Work

Although GPU programs written in CUDA stay compatible to future NVIDIA GPU architectures, the CUDA ecosystem keeps steadily evolving. It is to be expected that new and coming GPUs will further increase the number of compute cores and the amount of available memory. Additionally, recent NVLink-based multi-GPU servers rely on NVSwitch technology which fully connects all GPUs within the same node and can even be extended across nodes to create a high-bandwidth, multi-node GPU cluster.

Our GPU Communication library Gossip focuses on direct connections between pairs of GPUs in a single node environment, but could be extended to incorporate switches between GPUs as well as hierarchical topologies with multiple levels of interconnects. To tackle these problems the multi-commodity flow formulation would have to be augmented with additional nodes to accommodate switches and would have to include transfer edges between GPUs and those switches. In a similar manner, the time-expanded graph could be enhanced to model transfer latencies in order to create optimal transfer plans for small data scenarios, while Gossip currently achieves maximum throughput only for larger data package sizes. Furthermore, it would be interesting to see if Gossip's static transfer plans could be dynamically adjusted over the runtime of a program, in case the data distribution and transfer sizes changed over time. This could be achieved by tracking the estimated partition sizes based on moving average statistics combined with occasional recomputation of optimal solutions.

As we showed in Sections 5.3 and 9.3 (segmented) sort plays an important role in the context of suffix array construction as well as our metagenomic classifier MetaCache. We see potential to improve the employed segmented sorting primitives by adapting individual kernels to different data types and exploiting increased shared memory sizes of recent GPUs. The number of registers usable by each streaming multiprocessor of the GPU is limited and register usage depends on the data types of keys (and values) when sorting. To achieve high GPU occupancy and throughput, sorting kernels should be optimized according to the utilized data types. Furthermore, larger shared memory sizes enable local sorting of larger sized segments inside of thread blocks before reverting to global memory. The recent

Hopper H100 GPU features 227 KB of shared memory per thread block instead of the 96 KB available on Volta GPUs like the V100, enabling local sorting of segments more than twice as large. Additionally, the Hopper generation implements yet another layer in the thread hierarchy, which allows thread blocks to be grouped in clusters and grants access to shared memory of blocks in the same cluster (*distributed shared memory*). Thus, multiple thread blocks in a clusters could work together to sort large segments without storing intermediate results in global memory.

In order to minimize the influence of slow I/O from the file system we performed benchmarks for MetaCache-GPU using a virtual RAM drive. However, our implementation is sometimes still hindered by slow input and output operations of the host system. Here, we see two promising paths for advancements. The first approach follows the performance improvements achieved by recent quality control tool RabbitQCPlus [119]. The authors implement parallel compressed file I/O and also reduce memory copies by employing a pointer-based data structure instead of splitting sequence data into separate character arrays. Both techniques could be applied in MetaCache as well. The second approach would be to move file I/O to the GPU by means of GPUDirect Storage [@12]. This would remove the CPU as a bottleneck when performing input and output operations, however, specialized code would have to be developed to implement the required operations on the GPU. Both approaches are orthogonal and could potentially be combined.

Furthermore, some procedures like the coverage filter and quantification, exerted in the AFS pipeline, are currently executed on the CPU and could be accelerated on the GPU. Our current implementation takes advantage of multiple GPUs within the same node to process large-scale metagenomic databases in memory. It would be interesting to investigate an extension of our method to use even more GPUs within cluster systems. This would allow to include an even greater variety of reference genomes for broad-scale screening of metagenomic samples.

Another path to processing larger reference databases would involve a hierarchical approach, akin to the taxonomic tree. First, coarse-grained classification could be achieved by utilizing a database consisting of a limited number of genomes from the reference set, e.g., one genome per genus. Then, for the genera that have been identified to likely be included within the sample, genomes from all related organisms can be selected and a new database is created to allow a more fine-grained classification on species or even strain level. Due to the fast on-the-fly database construction of MetaCache-GPU, such an online approach becomes feasible without incurring large runtimes. Such a hierarchy could also be expanded to not only consist of two but multiple levels, so that each level includes a different sub-set of the taxonomic tree and classifications become more precise at each level. In addition to confining the sets of reference genomes, it would also be possible to increase the minhashing sub-sampling factor for the coarser databases and rely on a smaller sub-sampling factor only for the finer-grained classifications, which would further reduce memory requirements.

In contrast to other metagenomic classifiers, MetaCache is able to map reads to the most likely locations of origin within reference sequences instead of only identifying those sequences. This location information could be used for further downstream analysis like, e.g., alignments, variant calling and methylation detection.

Furthermore, the introduced methods could be adapted to related NGS such as long-read-to-long-read alignment. Long reads generated by third-generation sequencing instruments can offer enormous advantages for biological analysis and insight, however, they usually have high error rates and therefore often require different algorithmic approaches from processing short Illumina reads. Thus, overlapping and aligning long reads is a crucial step for subsequent error correction or de-novo assembly. Based on MetaCache, long reads could be partitioned into windows and sketches of the k-mers of each window can be computed using minhashing. These sketches are then inserted into a hash table and can later be used for querying in order to find reads with significant similarity. Candidate read pairings then need to be verified by computing their semi-global alignments in order to identify significant overlaps. The time-consuming alignment computation can also be accelerated using the identified matching k-mers as seeds.

12 Conclusion

As the cost for sequencing technologies continues to decrease, the amount of generated data will certainly grow further in the foreseeable future. To meet the increasing storage and processing demands, NGS pipelines will have to scale accordingly. Efficient parallel and distributed algorithms help to tackle unreasonably large runtimes, but solely CPU-based programs have a hard time keeping up to the task. GPUs provide greater compute capabilities and memory throughput, however, the limited amount of main memory available on a single GPU has posed a challenge to the GPU acceleration of many NGS applications. In order to achieve satisfactory results important algorithms such as metagenomic read classification, error correction and sequence alignment require large index structures which do not fit on a single GPU. In this work we presented methods to overcome memory restrictions and enable GPU acceleration of suffix array construction and metagenomic classification.

In order to alleviate single-GPU memory restrictions we have developed Gossip – a library of highly efficient communication primitives between multiple GPUs within the same node. By formulating an integer linear program, we are able to automatically generate transfer plans adjusted to the underlying NVLink topology for high throughput scatter, gather and all-to-all communication. We showed that with the usage of Gossip we are able to build and query distributed index data structures across several GPUs within modern DGX servers. This in turn allows for the usage of significantly larger index data structures through the combined main memory of multiple GPUs. We also showed that this allows for extremely fast suffix array construction; e.g., the suffix array of a full human reference genome can be constructed in only 3.4 seconds on a single DGX-1 server.

Furthermore, we introduced MetaCache-GPU – an ultra-fast metagenomic short read classifier specifically tailored to fit the characteristics of CUDA-enabled accelerators. In this context we optimized several methods to harness the vast compute capabilities of modern GPUs. In order to overcome the memory limitations of a single GPU we extend MetaCache to work with hash tables distributed across multiple GPUs.

File I/O is a non-negligible part of many genomic data processing pipelines and has to keep up with the immense (GPU) data processing speed. We showed that the improved sequence reader and parser we integrated into MetaCache(-GPU) is able to achieve twice the performance of competing tools, while providing a consistent enumeration of processed reads, which is required for post-processing steps like MetaCache's own merge mode.

Hash tables are key data structures in the GPU implementation of MetaCache dictating the performance of k-mer index construction and querying as well as the memory consumption. Off-the-shelf solutions represented a bottleneck in large-scale sequence analysis applications. To alleviate this bottleneck, we implemented a novel multi-value hash table variant featuring efficient minhash fingerprinting of reads for locality-sensitive hashing and their rapid insertion using warp-aggregated operations. This new variant is a better fit for the various key-value distributions encountered in genomic data sets and consumes less memory than previous implementations. The improved memory efficiency allows for more data to be stored per GPU or alternatively accelerates build and query times for a fixed amount of data. We suspect that many other applications relying on k-mer index structures could also benefit from our proposed hash table format.

Another time-consuming step in our GPU pipeline is sorting the lists of target locations resulting from database queries. For each batch of queries a segmented sort algorithm is employed on the GPUs to efficiently sort multiple location lists in parallel. We showed that our modification of the implementation by Hou et al. is able to outperform popular CUDA libraries like ModernGPU and CUB by a large amount using bitonic sorting kernels optimized for different segment sizes. Of course the performance of segmented sort algorithms is data dependent; the number of segments and their sizes are important factors and different algorithms might optimize for different use cases. However, we see potential for further improvement, demanding further investigation. For a generalized approach it would be useful to develop heuristics which choose the optimal sorting strategy based on the data distribution.

Our performance evaluation of MetaCache-GPU shows that the program is able to build large reference databases in a matter of seconds, enabling instantaneous operability, while popular CPU-based tools such as Kraken2 require over an hour for index construction on the same data. In the context of an ever-growing number of reference genomes, MetaCache-GPU is the first metagenomic classifier that makes analysis pipelines with on-demand composition of large-scale reference genome sets practical. While CPU-based MetaCache and Kraken2 take more than an hour for building the RefSeq202 database and more than 3 or more than 4 hours for AFS31+RefSeq202, respectively, MetaCache-GPU is able to create the index structures in seconds to minutes. Looking at the build time without writing the databases to the file system, the GPU version is 414 times and 272 times faster than the CPU version of MetaCache for RefSeq202 and AFS31+RefSeq202, respectively.

Our results clearly demonstrate the enormous potential of using GPUs for accelerating data-intensive NGS pipelines. Thus, we expect that, using our GPU-based approaches as enabling technologies, significantly faster methods can be designed for related problems in the field of bioinformatics. Although our implementations exhibit high processing throughput and outperform competitors, we still see possibilities for improvement by incorporating recent advancements in the CUDA hardware and software platform. Extending our solutions from single multi-GPU servers to distributed clusters while maintaining scalability will be one of the interesting research challenges in this area.

Bibliography

[1]D.A.F. Alcantara. “Efficient Hash Tables on the GPU”. PhD thesis. Davis, CA, USA: University of California at Davis, 2011 (cit. on p. 99).

[2]DA Alcantara, A Sharf, F Abbasinejad, et al. “Real-time Parallel Hashing on the GPU”. In: *ACM SIGGRAPH Asia 2009*. Yokohama, Japan: ACM, 2009, 154:1–154:9 (cit. on p. 99).

[3]S Ashkiani, M Farach-Colton, and JD Owens. “A Dynamic Hash Table for the GPU”. In: *IPDPS 2018*. IEEE. 2018, pp. 419–429 (cit. on p. 100).

[4]Saman Ashkiani, Andrew Davidson, Ulrich Meyer, and John D. Owens. “GPU Multisplit”. In: *Proc. of the 21st ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. PPoPP ’16. Barcelona, Spain: ACM, 2016, 12:1–12:13 (cit. on p. 21).

[5]Georg Baier, Ekkehard Köhler, and Martin Skutella. “The k-Splittable Flow Problem”. In: *Algorithmica* 42.3-4 (July 2005), pp. 231–248 (cit. on p. 29).

[6]DC Bauer, AP Tay, L Wilson, et al. “Supporting pandemic response using genomics and bioinformatics: a case study on the emergent SARS-CoV-2 outbreak”. In: *Transboundary and Emerging Diseases* (2020) (cit. on pp. 1, 56).

[7]AZ Broder. “Identifying and Filtering Near-Duplicate Documents”. In: *Proc. 11th Annual Symposium on Combinatorial Pattern Matching*. COM ’00. 2000, pp. 1–10 (cit. on pp. 8, 58).

[8]J. Bruck, Ching-Tien Ho, S. Kipnis, E. Upfal, and D. Weathersby. “Efficient algorithms for all-to-all communications in multiport message-passing systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 8.11 (Nov. 1997), pp. 1143–1156 (cit. on p. 21).

[9]Florian Büren, Daniel Jünger, Robin Kobus, Christian Hundt, and Bertil Schmidt. “Suffix Array Construction on Multi-GPU Systems”. In: *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC ’19. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 183–194 (cit. on pp. 3, 50, 51).

[10]N Cadenelli, J Polo, and D Carrera. “Accelerating K-mer frequency counting with GPU and non-volatile memory”. In: *IEEE HPCC 2017; IEEE SmartCity 2017; IEEE DSS 2017*. IEEE. 2017, pp. 434–441 (cit. on pp. 9, 10, 60).

[11]J Gregory Caporaso, Justin Kuczynski, Jesse Stombaugh, Kyle Bittinger, et al. “QIIME allows analysis of high-throughput community sequencing data”. In: *Nature methods* 7.5 (2010), pp. 335–336 (cit. on p. 68).

[12]A Chacón, S Marco-Sola, A Espinosa, P Ribeca, and JC Moure. “Boosting the FM-index on the GPU: Effective techniques to mitigate random memory access”. In: *IEEE/ACM Trans. on Computational Biology and Bioinformatics* 12.5 (2014), pp. 1048–1059 (cit. on pp. 9, 10, 60).

[13]Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. “Collective communication: theory, practice, and experience”. In: *Concurrency and Computation: Practice and Experience* 19.13 (2007), pp. 1749–1783 (cit. on p. 22).

[14]Barbara Chapman, Tony Curtis, Swaroop Pophale, et al. “Introducing OpenSHMEM: SHMEM for the PGAS community”. In: *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. 2010, pp. 1–3 (cit. on p. 8).

[15]Haoyu Cheng, Huaipan Jiang, Jiaoyun Yang, Yun Xu, and Yi Shang. “BitMapper: an efficient all-mapper based on bit-vector computing”. In: *BMC bioinformatics* 16.1 (2015), pp. 1–16 (cit. on p. 9).

[16]Temesgen Hailemariam Dadi, Bernhard Y Renard, Lothar H Wieler, Torsten Semmler, and Knut Reinert. “SLIMM: species level identification of microorganisms from metagenomes”. In: *PeerJ* 5 (2017), e3138 (cit. on p. 64).

[17]Jun Doi and Yasushi Negishi. “Overlapping methods of all-to-all communication and FFT algorithms for torus-connected massively parallel supercomputers”. In: *SC’10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2010, pp. 1–9 (cit. on p. 21).

[18]M Ellis, G Guidi, A Buluç, L Oliker, and K Yelick. “diBELLA: Distributed Long Read to Long Read Alignment”. In: *Proc. of the 48th Int. Conference on Parallel Processing*. 2019, pp. 1–11 (cit. on pp. 2, 6, 7, 59).

[19]M Erbert, S Rechner, and M Müller-Hannemann. “Gerbil: a fast and memory-efficient k-mer counter with GPU-support”. In: *Algorithms for Molecular Biology* 12.1 (2017), pp. 1–12 (cit. on pp. 9, 10, 60).

[20]M Esteki, J Regueiro, and J Simal-Gándara. “Tackling Fraudsters with Global Strategies to Expose Fraud in the Food Chain”. In: *Comprehensive Reviews in Food Science and Food Safety* 18.2 (2019), pp. 425–440 (cit. on pp. 67, 79).

[21]Albert Eugster, Jürg Ruf, Jürg Rentsch, and René Köppel. “Quantification of beef, pork, chicken and turkey proportions in sausages: use of matrix-adapted standards and comparison of single versus multiplex PCR in an interlaboratory trial”. In: *European Food Research and Technology* 230.1 (2009), p. 55 (cit. on p. 69).

[22]A. Faraj, P. Patarasuk, and X. Yuan. “Bandwidth Efficient All-to-All Broadcast on Switched Clusters”. In: *2005 IEEE Int. Conference on Cluster Computing*. Sept. 2005, pp. 1–10 (cit. on p. 21).

[23]L. Fleischer and M. Skutella. “Quickest Flows Over Time”. In: *SIAM Journal on Computing* 36.6 (2007), pp. 1600–1630 (cit. on p. 28).

[24]Patrick Flick and Srinivas Aluru. “Parallel distributed memory construction of suffix and longest common prefix arrays”. In: *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA*. 2015, 16:1–16:10 (cit. on pp. 50, 112).

[25]Pierre Fraigniaud and Emmanuel Lazard. “Methods and problems of communication in usual networks”. In: *Discr. Applied Math.* 53.1 (1994), pp. 79–133 (cit. on p. 21).

[26]Ismael García, Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. “Coherent Parallel Hashing”. In: *ACM SIGGRAPH Asia 2011*. SA ’11. Hong Kong, China: ACM, 2011, 161:1–161:8 (cit. on p. 99).

[27]E Georganas, R Egan, S Hofmeyr, et al. “Extreme scale de novo metagenome assembly”. In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 122–134 (cit. on pp. 2, 6, 7).

[28]Y. Gong, B. He, and J. Zhong. “Network Performance Aware MPI Collective Communication Operations in the Cloud”. In: *IEEE Trans. on Par. and Distr. Sys.* 26.11 (Nov. 2015), pp. 3079–3089 (cit. on p. 22).

[29]Giorgio Gonnella and Stefan Kurtz. “Readjoiner: a fast and memory efficient string graph-based sequence assembler”. In: *BMC bioinformatics* 13.1 (2012), pp. 1–19 (cit. on pp. 2, 6, 7).

[30]Oded Green. “HashGraph – Scalable hash tables using a sparse graph data structure”. In: *ACM Transactions on Parallel Computing (TOPC)* 8.2 (2021), pp. 1–17 (cit. on p. 100).

[31]DF Gudbjartsson, H Helgason, SA Gudjonsson, et al. “Large-scale whole-genome sequencing of the Icelandic population”. In: *Nature Genetics* 47.5 (2015), pp. 435–444 (cit. on p. 1).

[32]Giulia Guidi, Marquita Ellis, Daniel Rokhsar, Katherine Yelick, and Aydın Buluç. “BELLA: Berkeley efficient long-read to long-read aligner and overlapper”. In: *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*. SIAM. 2021, pp. 123–134 (cit. on pp. 6, 7).

[33]Scott Hazelhurst and Zsuzsanna Lipták. “KABOOM! A new suffix array based algorithm for clustering expression data”. In: *Bioinformatics* 27.24 (2011), pp. 3348–3355 (cit. on pp. 2, 6).

[34]David Hernandez, Patrice François, Laurent Farinelli, Magne Østerås, and Jacques Schrenzel. “De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer”. In: *Genome research* 18.5 (2008), pp. 802–809 (cit. on pp. 2, 6, 7).

[35]S. Hinrichs, C. Kosak, D.R. O’Hallaron, T.M. Stricker, and R. Take. “An Architecture for Optimal All-to-all Personalized Communication”. In: *Proc. of the Sixth Annual ACM Symp. on Parallel Alg. and Arch.* SPAA ’94. Cape May, New Jersey, USA: ACM, 1994, pp. 310–319 (cit. on p. 21).

[36]K Hou, W Liu, H Wang, and W Feng. “Fast Segmented Sort on GPUs”. In: *31th International Conference on Supercomputing (ICS)*. Chicago, USA, June 2017 (cit. on p. 113).

[37]EJ Houtgast, V Sima, K Bertels, and Z Al-Ars. “An FPGA-based systolic array to accelerate the BWA-MEM genomic mapping algorithm”. In: *SAMOS 2015*. IEEE. 2015, pp. 221–227 (cit. on pp. 2, 56).

[38]EJ Houtgast, V Sima, K Bertels, and Z Al-Ars. “Hardware acceleration of BWA-MEM genomic short read mapping for longer read lengths”. In: *Computational Biology and Chemistry* 75 (2018), pp. 54–64 (cit. on pp. 2, 56, 60).

[39]Lucian Ilie, Farideh Fazayeli, and Silvana Ilie. “HiTEC: accurate error correction in high-throughput sequencing data”. In: *Bioinformatics* 27.3 (2011), pp. 295–302 (cit. on pp. 2, 6).

[40]D Jünger, C Hundt, and B Schmidt. “WarpDrive: Massively Parallel Hashing on Multi-GPU Nodes”. In: *IPDPS 2018*. IEEE. 2018, pp. 441–450 (cit. on p. 100).

[41]D. Jünger, C. Hundt, and B. Schmidt. “WarpDrive: Massively Parallel Hashing on Multi-GPU Nodes”. In: *2018 IEEE Int. Par. and Distr. Proc. Symp. (IPDPS)*. May 2018, pp. 441–450 (cit. on pp. 21, 39).

[42]Daniel Jünger, Robin Kobus, André Müller, et al. “WarpCore: A Library for fast Hash Tables on GPUs”. In: *HiPC 2020*. IEEE, 2020, pp. 11–20 (cit. on pp. 100, 102).

[43]Felix Kallenborn, Andreas Hildebrandt, and Bertil Schmidt. “CARE: context-aware sequencing read error correction”. In: *Bioinformatics* 37.7 (2021), pp. 889–895 (cit. on pp. 2, 10).

[44]Krishna Chaitanya Kandalla, Hari Subramoni, Abhinav Vishnu, and D.K. Panda. “Designing topology-aware collective communication algorithms for large scale InfiniBand clusters: Case studies with Scatter and Gather”. In: *2010 IEEE Int. Symp. on Par. & Distr. Proc., Workshops and Phd Forum (IPDPSW)* (2010), pp. 1–8 (cit. on p. 22).

[45]N. T. Karonis, B. R. de Supinski, I. Foster, et al. “Exploiting hierarchy in parallel computer networks to optimize collective operation performance”. In: *Proc. 14th Int. Par. and Distr. Proc. Symp. (IPDPS)*. May 2000, pp. 377–384 (cit. on p. 22).

[46]F Khorasani, ME Belviranli, R Gupta, and LN Bhuyan. “Stadium Hashing: Scalable and Flexible Hashing on GPUs”. In: *PACT 2015*. IEEE, 2015, pp. 63–74 (cit. on p. 100).

[47]Szymon M Kiełbasa, Raymond Wan, Kengo Sato, Paul Horton, and Martin C Frith. “Adaptive seeds tame genomic sequence comparison”. In: *Genome research* 21.3 (2011), pp. 487–493 (cit. on pp. 2, 6).

[48]R Kobus, JM Abuín, A Müller, et al. “A big data approach to metagenomics for all-food-sequencing”. In: *BMC Bioinformatics* 21.1 (2020), pp. 1–15 (cit. on pp. 3, 121).

[49]R Kobus, C Hundt, A Müller, and B Schmidt. “Accelerating metagenomic read classification on CUDA-enabled GPUs”. In: *BMC Bioinformatics* 18.1 (2017), pp. 1–10 (cit. on pp. 10, 59).

[50] R Kobus, D Jünger, C Hundt, and B Schmidt. "Gossip: Efficient Communication Primitives for Multi-GPU Systems". In: *48th Int. Conference on Parallel Processing (ICPP '19)*. 2019, pp. 1–10 (cit. on pp. 3, 100).

[51] Robin Kobus, André Müller, Daniel Jünger, Christian Hundt, and Bertil Schmidt. "MetaCache-GPU: Ultra-Fast Metagenomic Classification". In: *50th International Conference on Parallel Processing*. 2021, pp. 1–11 (cit. on p. 3).

[52] René Köppel, Arthika Ganeshan, Franziska van Velsen, et al. "Digital duplex versus real-time PCR for the determination of meat proportions from sausages containing pork and beef". In: *European Food Research and Technology* 245.1 (2019), pp. 151–157 (cit. on p. 67).

[53] René Köppel, Jürg Ruf, and Jürg Rentsch. "Multiplex real-time PCR for the detection and quantification of DNA from beef, pork, horse and sheep". In: *European Food Research and Technology* 232.1 (2011), pp. 151–155 (cit. on p. 67).

[54] René Köppel, Jürg Ruf, and Jürg Rentsch. "Multiplex real-time PCR for the detection and quantification of DNA from beef, pork, horse and sheep". In: *European Food Research and Technology* 232.1 (2011), pp. 151–155 (cit. on p. 69).

[55] K Korpela, A Salonen, LJ Virta, et al. "Intestinal microbiome is related to lifetime antibiotic use in Finnish pre-school children". In: *Nature Communications* 7 (2016), p. 10410 (cit. on pp. 1, 56).

[56] Stefan Kurtz, Apurva Narechania, Joshua C Stein, and Doreen Ware. "A new method to compute K-mer frequencies and its application to annotate large repetitive plant genomes". In: *BMC genomics* 9.1 (2008), pp. 1–18 (cit. on pp. 2, 5, 6).

[57] Julian Labeit, Julian Shun, and Guy E. Blelloch. "Parallel lightweight wavelet tree, suffix array and FM-index construction". In: *J. Discrete Alg.* 43 (2017), pp. 2–17 (cit. on p. 49).

[58] Ben Langmead and Steven L Salzberg. "Fast gapped-read alignment with Bowtie 2". In: *Nature methods* 9.4 (2012), p. 357 (cit. on pp. 6, 7, 68).

[59] B Lessley and H Childs. "Data-Parallel Hashing Techniques for GPU Architectures". In: *IEEE Transactions on Parallel and Distributed Systems* 31.1 (2019), pp. 237–250 (cit. on p. 99).

[60] HA Lewin, GE Robinson, WJ Kress, et al. "Earth BioGenome Project: Sequencing life for the future of life". In: *Proceedings of the National Academy of Sciences* 115.17 (2018), pp. 4325–4333 (cit. on pp. 1, 56).

[61] Dinghua Li, Chi-Man Liu, Ruibang Luo, Kunihiko Sadakane, and Tak-Wah Lam. "MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph". In: *Bioinformatics* 31.10 (2015), pp. 1674–1676 (cit. on p. 10).

[62] H Li, A Ramachandran, and D Chen. "GPU acceleration of advanced k-mer counting for computational genomics". In: *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2018, pp. 1–4 (cit. on pp. 9, 10, 60).

[63] Heng Li. “Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM”. In: *arXiv:1303.3997v2* (2013) (cit. on p. 68).

[64] Heng Li and Richard Durbin. “Fast and Accurate Long-Read Alignment with Burrows-Wheeler Transform”. In: *Bioinformatics* 26.5 (2010), pp. 589–595 (cit. on p. 68).

[65] Heng Li and Richard Durbin. “Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform”. In: *Bioinformatics* 25.14 (2009), pp. 1754–1760 (cit. on p. 68).

[66] S Lindgreen, K L Adair, and P Gardner. “An evaluation of the accuracy and speed of metagenome analysis tools”. In: *Scientific Reports* 6.19233 (2016) (cit. on pp. 59, 68).

[67] Chi-Man Liu, Ruibang Luo, and Tak-Wah Lam. “GPU-accelerated BWT construction for large collection of short reads”. In: *arXiv preprint arXiv:1401.7457* (2014) (cit. on p. 10).

[68] Weifeng Liu and Brian Vinter. “A Framework for General Sparse Matrix-Matrix Multiplication on GPUs and Heterogeneous Processors”. In: *J. Parallel Distrib. Comput.* 85.C (Nov. 2015), pp. 47–61 (cit. on p. 112).

[69] Y Liu and B Schmidt. “CUSHAW2-GPU: empowering faster gapped short-read alignment using GPU computing”. In: *IEEE Design & Test* 31.1 (2013), pp. 31–39 (cit. on pp. 9, 10, 60).

[70] Yongchao Liu, Fabian Ripp, Rene Koeppel, et al. “AFS: identification and quantification of species composition by metagenomic sequencing”. In: *Bioinformatics* (2017), btw822 (cit. on pp. 68, 79).

[71] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. “CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows–Wheeler transform”. In: *Bioinformatics* 28.14 (2012), pp. 1830–1837 (cit. on p. 68).

[72] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. “DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI”. In: *BMC bioinformatics* 12.1 (2011), pp. 1–13 (cit. on pp. 9, 10).

[73] Yongchao Liu, Jan Schröder, and Bertil Schmidt. “Musket: a multistage k-mer spectrum-based error corrector for Illumina sequence data”. In: *Bioinformatics* 29.3 (2013), pp. 308–315 (cit. on p. 6).

[74] Jennifer Lu, Florian P Breitwieser, Peter Thielen, and Steven L Salzberg. “Bracken: estimating species abundance in metagenomics data”. In: *PeerJ Computer Science* 3 (2017), e104 (cit. on p. 68).

[75] Udi Manber and Gene Myers. “Suffix Arrays: A New Method for On-Line String Searches”. In: *Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, USA.* 1990, pp. 319–327 (cit. on p. 2).

[76] Guillaume Marçais and Carl Kingsford. “A fast, lock-free approach for efficient parallel counting of occurrences of k-mers”. In: *Bioinformatics* 27.6 (2011), pp. 764–770 (cit. on pp. 2, 5, 6).

[77] Camille Marchet, Christina Boucher, Simon J Puglisi, et al. “Data structures based on k-mers for querying large collections of sequencing data sets”. In: *Genome Research* 31.1 (2021), pp. 1–12 (cit. on pp. 2, 56).

[78] Pall Melsted and Jonathan K Pritchard. “Efficient counting of k-mers in DNA sequences using a bloom filter”. In: *BMC bioinformatics* 12.1 (2011), pp. 1–7 (cit. on pp. 2, 5, 6).

[79] Peter Menzel, Kim Lee Ng, and Anders Krogh. “Fast and sensitive taxonomic classification for metagenomics with Kaiju”. In: *Nature communications* 7 (2016), p. 11257 (cit. on p. 68).

[80] Jason R Miller, Arthur L Delcher, Sergey Koren, et al. “Aggressive assembly of pyrosequencing reads with mates”. In: *Bioinformatics* 24.24 (2008), pp. 2818–2824 (cit. on pp. 5, 6).

[81] A Müller, C Hundt, A Hildebrandt, T Hankeln, and B Schmidt. “MetaCache: context-aware classification of metagenomic reads using minhashing”. In: *Bioinformatics* 33.23 (2017), pp. 3740–3748 (cit. on pp. 2, 3, 6, 8, 57–59, 68).

[82] Gene Myers. “A fast bit-vector algorithm for approximate string matching based on dynamic programming”. In: *Journal of the ACM (JACM)* 46.3 (1999), pp. 395–415 (cit. on p. 9).

[83] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. “Scalable Parallel Programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for?” In: *Queue* 6.2 (2008), pp. 40–53 (cit. on pp. 9, 11).

[84] NA O’Leary, MW Wright, JR Brister, et al. “Reference sequence (RefSeq) database at NCBI: current status, taxonomic expansion, and functional annotation”. In: *Nucleic Acids Research* 44.D1 (2016), pp. D733–D745 (cit. on pp. 114, 121).

[85] Brian D Ondov, Nicholas H Bergman, and Adam M Phillippy. “Interactive metagenomic visualization in a Web browser”. In: *BMC bioinformatics* 12.1 (2011), p. 385 (cit. on pp. 77, 78).

[86] A Morgulis others. “Database indexing for production MegaBLAST searches”. In: *Bioinformatics* 24.16 (2008), pp. 1757–1764 (cit. on p. 59).

[87] R Ounit, S Wanamaker, TJ Close, et al. “CLARK: fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers”. In: *BMC Genomics* 16.1 (2015), pp. 1–13 (cit. on pp. 2, 6, 7, 59, 68).

[88] TC Pan, S Misra, and S Aluru. “Optimizing high performance distributed memory parallel hash tables for DNA k-mer counting”. In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 135–147 (cit. on pp. 2, 5, 6).

[89] P Patarasuk, A Faraj, and Xin Yuan. “Pipelined broadcast on Ethernet switched clusters”. In: vol. 2006. May 2006 (cit. on p. 21).

[90] Pitch Patarasuk and Xin Yuan. “Bandwidth optimal all-reduce algorithms for clusters of workstations”. In: *J. of Parallel and Distr. Comp.* 69.2 (2009), pp. 117–124 (cit. on p. 21).

[91] Simon J. Puglisi, William F. Smyth, and Andrew Turpin. “A taxonomy of suffix array construction algorithms”. In: *ACM Comput. Surv.* 39.2 (2007), p. 4 (cit. on p. 44).

[92] Y Qiao, B Jia, Z Hu, et al. “MetaBinG2: a fast and accurate metagenomic sequence classification system for samples with many unknown organisms”. In: *Biology Direct* 13.1 (2018), pp. 1–21 (cit. on p. 60).

[93] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. “Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes”. In: *2009 17th Euromicro international conference on parallel, distributed and network-based processing*. IEEE. 2009, pp. 427–436 (cit. on p. 8).

[94] K Reinert, B Langmead, D Weese, and D J Evers. “Alignment of Next-Generation Sequencing Reads”. In: *Annual Review of Genomics and Human Genetics* 16 (2015), pp. 133–151 (cit. on pp. 2, 56, 59).

[95] F Ripp, C F Krombholz, Y Liu, et al. “All-Food-Seq (AFS): a quantifiable screen for species in biological samples by deep DNA sequencing”. In: *BMC Genomics* 15:639 (2014) (cit. on pp. 68, 79).

[96] Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. “Reducing storage requirements for biological sequence comparison”. In: *Bioinformatics* 20.18 (2004), pp. 3363–3369 (cit. on p. 7).

[97] Bertil Schmidt and Andreas Hildebrandt. “Next-generation sequencing: big data meets high performance computing”. In: *Drug discovery today* 22.4 (2017), pp. 712–717 (cit. on pp. 8, 79).

[98] C Schoch. *NCBI Taxonomy Help*. National Center for Biotechnology Information (US), 2020 (cit. on p. 62).

[99] Jan Schröder, Heiko Schröder, Simon J Puglisi, Ranjan Sinha, and Bertil Schmidt. “SHREC: a short-read error correction method”. In: *Bioinformatics* 25.17 (2009), pp. 2157–2163 (cit. on pp. 2, 6).

[100] D. S. Scott. “Efficient All-to-All Communication Patterns in Hypercube and Mesh Topologies”. In: *6th Distr. Memory Computing Conf., 1991. Proc.* Apr. 1991, pp. 398–403 (cit. on p. 21).

[101] M Seppey, M Manni, and E Zdobnov. “LEMMI: A continuous benchmarking platform for metagenomics classifiers”. In: *Genome Research* 30 (July 2020), gr.260398.119 (cit. on pp. 59, 68).

[102] Anish Man Singh Shrestha, Martin C. Frith, and Paul Horton. “A bioinformatician’s guide to the forefront of suffix array construction algorithms”. In: *Briefings in Bioinformatics* 15.2 (Jan. 2014), pp. 138–154 (cit. on p. 2).

[103] Jared T Simpson and Richard Durbin. “Efficient de novo assembly of large genomes using compressed data structures”. In: *Genome research* 22.3 (2012), pp. 549–556 (cit. on pp. 6, 7).

[104] ZD Stephens, SY Lee, F Faghri, et al. “Big data: astronomical or genomical?” In: *PLoS Biology* 13.7 (2015), e1002195 (cit. on pp. 1, 56, 79).

[105] John E Stone, David Gohara, and Guochun Shi. “OpenCL: A parallel programming standard for heterogeneous computing systems”. In: *Computing in science & engineering* 12.3 (2010), p. 66 (cit. on pp. 9, 11).

[106] Shinichi Sunagawa, Daniel R Mende, Georg Zeller, et al. “Metagenomic species profiling using universal phylogenetic marker genes”. In: *Nature methods* 10.12 (2013), p. 1196 (cit. on p. 68).

[107] Wing-Kin Sung. *Algorithms in bioinformatics: A practical introduction*. Chapman and Hall/CRC, 2009 (cit. on p. 10).

[108] Joaquín Tárraga, Vicente Arnau, Héctor Martínez, et al. “Acceleration of short and long DNA read mapping without loss of accuracy using suffix array”. In: *Bioinformatics* 30.23 (2014), pp. 3396–3398 (cit. on pp. 2, 6, 7).

[109] R. Thakur and A. Choudhary. “All-to-all communication on meshes with wormhole routing”. In: *Proc. of 8th Int. Parallel Processing Symposium*. Apr. 1994, pp. 561–565 (cit. on p. 21).

[110] Andreas O Tillmar, Barbara Dell’Amico, Jenny Welander, and Gunilla Holmlund. “A universal method for species identification of mammals utilizing next generation sequencing for the analysis of DNA mixtures”. In: *PloS one* 8.12 (2013), e83761 (cit. on p. 68).

[111] D T Truong, Eric A. Franzosa, Timothy L. Tickle, et al. “MetaPhlAn2 for enhanced metagenomic taxonomic profiling”. In: *Nat Meth* 12.10 (2015), pp. 902–903 (cit. on p. 68).

[112] Yu-Chee Tseng and S. K. S. Gupta. “All-to-all personalized communication in a wormhole-routed torus”. In: *IEEE Trans. on Par. and Distr. Sys.* 7.5 (1996), pp. 498–505 (cit. on p. 21).

[113] Michaël Vyverman, Bernard De Baets, Veerle Fack, and Peter Dawyndt. “essaMEM: finding maximal exact matches using enhanced sparse suffix arrays”. In: *Bioinformatics* 29.6 (2013), pp. 802–804 (cit. on p. 2).

[114] L. Wang, S. Baxter, and J.D. Owens. “Fast parallel skew and prefix-doubling suffix array construction on the GPU”. In: *CCPE* 28.12 (2016), pp. 3466–3484 (cit. on pp. 21, 49).

[115] Richard Wilton, Tamas Budavari, Ben Langmead, et al. “Arioc: high-throughput read alignment with GPU-accelerated exploration of the seed-and-extend search space”. In: *PeerJ* 3 (2015), e808 (cit. on pp. 9, 10).

[116] D E Wood, J Lu, and B Langmead. “Improved metagenomic analysis with Kraken 2”. In: *Genome biology* 20.1 (2019), p. 257 (cit. on pp. 2, 6, 7, 57, 59, 81, 121).

[117] D E Wood and S L Salzberg. “Kraken: ultrafast metagenomic sequence classification using exact alignments”. In: *Genome Biology* 15:R46 (2014) (cit. on pp. 2, 6, 7, 58, 59, 68, 83, 121).

[118]Hongyi Xin, John Greth, John Emmons, et al. "Shifted Hamming distance: a fast and accurate SIMD-friendly filter to accelerate alignment verification in read mapping". In: *Bioinformatics* 31.10 (2015), pp. 1553–1560 (cit. on p. 9).

[119]Lifeng Yan, Zekun Yin, Hao Zhang, et al. "RabbitQCPlus: More Efficient Quality Control for Sequencing Data". In: *2022 International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE. 2022 (cit. on p. 136).

[120]Yuzhen Ye, Jeong-Hyeon Choi, and Haixu Tang. "RAPSearch: a fast protein similarity search tool for short reads". In: *BMC bioinformatics* 12.1 (2011), pp. 1–10 (cit. on pp. 2, 6).

[121]E. Zahavi, G. Johnson, D.J. Kerbyson, and M. Lang. "Optimized InfiniBandTM fat-tree routing for shift all-to-all communication patterns". In: *CCPE* 22.2 (2010), pp. 217–231 (cit. on p. 22).

[122]Jing Zhang, Hao Wang, and Wu-chun Feng. "cuBLASTP: Fine-Grained Parallelization of Protein Sequence Search on CPU+GPU". In: *IEEE/ACM transactions on computational biology and bioinformatics* 14.4 (2015), pp. 830–843 (cit. on p. 112).

[123]Yongan Zhao, Haixu Tang, and Yuzhen Ye. "RAPSearch2: a fast and memory-efficient protein similarity search tool for next-generation sequencing data". In: *Bioinformatics* 28.1 (2012), pp. 125–126 (cit. on p. 6).

[124]Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. "UPC++: a PGAS extension for C++". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE. 2014, pp. 1105–1114 (cit. on p. 8).

Webpages

[@1]AMD. *2990WX Specifications*. 2022. URL: https://www.amd.com/en/product/7921 (visited on Dec. 12, 2022) (cit. on p. 11).

[@2]AMD. *5995WX Specifications*. 2022. URL: https://www.amd.com/en/product/11786 (visited on Dec. 12, 2022) (cit. on p. 11).

[@3]AMD. *HIP Programming Guide*. 2022. URL: https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html (visited on Dec. 12, 2022) (cit. on p. 11).

[@4]S. Baxter. *ModernGPU: Patterns and behaviors for GPU computing*. 2016. URL: https://github.com/moderngpu/moderngpu (visited on May 23, 2022) (cit. on p. 112).

[@5]NVIDIA Corporation. *NVBIO*. 2015. URL: http://nvlabs.github.io/nvbio/ (visited on Nov. 23, 2018) (cit. on p. 49).

[@6]NVIDIA Corporation. *NVIDIA Collective Communications Library (NCCL)*. Jan. 2019. URL: https://developer.nvidia.com/nccl (visited on Jan. 30, 2019) (cit. on p. 20).

[@7]S. Dalton, N. Bell, L. Olson, and M. Garland. *CUSP: A C++ Templated Sparse Matrix Library*. 2015. URL: http://cusplibrary.github.io/ (visited on May 23, 2022) (cit. on p. 112).

[@8]Google. *OR-Tools*. Jan. 2019. URL: https://developers.google.com/optimization/ (visited on Jan. 30, 2019) (cit. on p. 33).

[@9]Yuta Mori. *libdivsufsort 2.0.2-1*. 2016. URL: https://github.com/y-256/libdivsufsort (visited on Nov. 5, 2018) (cit. on p. 49).

[@10]NVIDIA. *A100 Data Sheet*. 2022. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-nvidia-us-2188504-web.pdf (visited on Dec. 12, 2022) (cit. on p. 11).

[@11]NVIDIA. *CUDA C++ Programming Guide*. Apr. 2022. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (visited on Apr. 15, 2022) (cit. on p. 13).

[@12]NVIDIA. *GPUDirect Storage*. Apr. 2022. URL: htthttps://docs.nvidia.com/cuda/pdf/GDS.pdf (visited on Apr. 15, 2022) (cit. on p. 136).

[@13]NVIDIA. *GV100 Data Sheet*. 2022. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/productspage/quadro/quadro-desktop/quadro-volta-gv100-data-sheet-us-nvidia-704619-r3-web.pdf (visited on Dec. 12, 2022) (cit. on p. 11).

[@14]NVIDIA Research. *CUB: Cooperative primitives for CUDA C++*. 2021. URL: https://nvlabs.github.io/cub/ (visited on May 23, 2022) (cit. on pp. 95, 112).

Acronyms

AoS Array of Structures. 100, 105, 107–111

CAS Compare and Swap. 101, 104

CG cooperative group. 102

CUDA Compute Unified Device Architecture. 2, 9, 11–13, 20, 33, 34, 90–92, 98–101, 105, 112–114, 117

FPGA Field-Programmable Gate Array. 2, 9

GPGPU General-Purpose Computing on Graphics Processing Units. 9, 11

GPU Graphics Processing Unit. 2

HPC High Performance Computing. 8

MPI Message Passing Interface. 8

NGS *next generation sequencing*. 1, 5, 9, 137, 139, 141

PGAS partitioned global address space. 8

SIMD Single Instruction Multiple Data. 6–8

SM streaming multiprocessor. 12, 13, 135

SoA Structure of Arrays. 100, 101, 105, 107–111

List of Figures

List of Tables