Karlsruhe Institute of Technology

Weighted Random Sampling – Alias Tables on the GPU

Master's Thesis
submitted by

Hans-Peter Lehmann

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Peter Sanders
Second Reviewer:	Prof. Dr. Carsten Dachsbacher
Advisors:	Lorenz Hübschle-Schneider
	Emanuel Schrade

July 1, 2020 – December 29, 2020

Abstract

An alias table is a data structure that allows for efficiently drawing weighted random samples in time in $\mathcal{O}(1)$. The PSA algorithm by Hübschle-Schneider and Sanders is able to construct alias tables in parallel on the CPU. In this thesis, we develop a construction algorithm for GPUs that is based on the idea of PSA. Our algorithm specifically fits the architectural properties of GPUs. It achieves a speedup of 34 on a GPU in comparison to the 4-threaded PSA method on a consumer grade CPU. Our implementation of the PSA+ method achieves an additional speedup of up to 1.4 to our PSA implementation. Being faster than the CPU method even when including memory transfers to the GPU and back, our method can be used to offload the construction efficiently. For sampling, we achieve an up to 56 times higher throughput on a consumer GPU than Hübschle-Schneider and Sanders on the CPU.

Acknowledgments

I thank Prof. Dr. Peter Sanders and Prof. Dr. Carsten Dachsbacher for reviewing the thesis. For offering the interesting topic and for suggesting algorithmic ideas to try out, I thank Prof. Dr. Peter Sanders. I also thank my advisors Lorenz Hübschle-Schneider and Emanuel Schrade for their suggestions, detailed reviews and useful feedback. Additionally, I thank Lorenz Hübschle-Schneider for making his CPU code open-source, which provided useful insights. I thank Emanuel Schrade for giving me access to a Nvidia RTX 2080 in the computer graphics pool for performance testing. For running performance and power measurements on powerful servers, I thank Lorenz Hübschle-Schneider.

I thank Linus Schilpp for borrowing me his Nvidia RTX 2070 Super while the pool PCs were unavailable. I thank Miriam Scheer and Liam Wachter for suggesting to experiment with different optimization parameters, some of which led to surprisingly good results. Additionally, I thank Liam Wachter and Niko Wilhelm for proofreading and giving feedback.

Contents

Chapter 1

Introduction

Motivation Randomly drawing items from a set is used in many applications such as simulations or graph generation. With weighted random sampling, each item has a specific probability. The task is to draw items from the input set while honoring their respective probabilities. A data structure that allows for efficiently sampling from a weighted random distribution in $\mathcal{O}(1)$ is the alias table. To sample from an alias table, it is sufficient to generate a uniform random number and to perform one memory access operation. Every discrete random distribution can be represented as an alias table. The first algorithm for parallel alias table construction, PSA,[1] is presented by Hübschle-Schneider and Sanders [27]. GPUs are becoming more important for high performance computing because of their fast memory and high degree of parallelism. Therefore, there is need for an efficient method to construct data structures for drawing weighted random samples on GPUs.

Contribution In contrast to CPUs, GPUs follow a single instruction multiple data (SIMD) scheme. The main contribution of this thesis is to develop algorithms that, based on PSA, can construct and sample from alias tables on GPUs efficiently. In contrast to the original implementation, we modify the memory access patterns so that the operations can be coalesced and therefore are efficient on GPUs. Splitting is executed using a newly developed scheme that we call *partial p-ary search.* Packing still uses linear memory access operations in each thread but the proposed algorithm first copies the input data to the faster shared memory in a coalesced way. For running the PSA+ extension we specifically choose a point in time that allows for reducing memory access operations.

[1]Parallel splitting-based alias table construction.

Structure In the remainder of this chapter we first introduce random numbers and sampling. After that, we continue with details about the GPU architecture that are important for understanding this thesis. In Chapter 2 we introduce other weighted random sampling algorithms and applications of them. In Chapter 3 we introduce our newly designed construction and sampling algorithms. We describe the implementation of those algorithms in Chapter 4. Finally, in Chapter 5, we compare our methods among each other and with the work of Hübschle-Schneider and Sanders [27].

1.1 Random Numbers

Random numbers are a foundation for many scientific fields, such as simulations, cryptography or computer graphics. In order to sample from alias tables, we need sources of random numbers with specific distributions.

1.1.1 Distributions

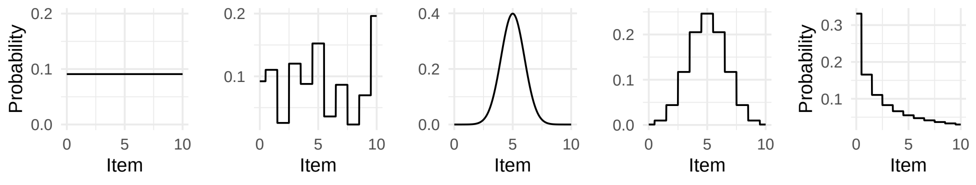
In the following section we introduce different random distributions that are used in this thesis. Figure 1.1 provides a visualization of the distributions.

Uniform Distribution The most well known random number distribution is the uniform distribution. When sampling items from this distribution, each one is drawn with the same probability. The distribution function is a constant function. Even though the distribution is very basic, sampling from alias tables relies heavily on a source of uniform random numbers.

Uniform Random Weights A distribution that cannot be sampled directly is generated by drawing a uniform random number for determining the weight of each item. We heavily use this distribution for evaluating our algorithms.

Normal Distribution The normal distribution (sometimes also called Gaussian distribution) can be observed when a continuous value randomly differs from an expected value. Its density function is described by a typical bell curve.

Binomial Distribution When observing N random experiments with a binary outcome like *heads or tails* and a winning probability p, the binomial distribution $B(N, p, k)$ describes the probability for drawing heads k times. For high values of N, the binomial distribution can be approximated with the normal distribution [37].



(a) Uniform (b) Random (c) Normal (d) Binomial (e) Power law

Figure 1.1: Visualization of different random distributions.

Power Law The power law distribution uses weights $w_i = i^{-\alpha}$. Many processes follow a shuffled power law distribution where these weights are assigned in a random order. Realistic values of α are usually smaller than 2. An example is the word frequency in texts (Zipf's law) [35].

1.1.2 Random Number Generators

Because computers are deterministic machines, generating random numbers is harder than it might seem. There is an extensive amount of literature about random number generators and measuring their quality, such as TestU01 [32]. Describing these quality measures is out of scope for this thesis. Pseudorandom number generators (PRNGs) are the most common class of random number generators. PRNGs are deterministic algorithms that generate seemingly unrelated and hard to predict numbers [64]. In order to understand the basic ideas, as well as advantages and disadvantages in the context of GPUs, we introduce some commonly used algorithms in the following paragraphs.

Multiple Recursive Generator A Multiple Recursive Generator (MRG) [58] generates pseudorandom numbers U_n by multiplying and adding the state vector x_i and parameters a_i in a modulo m group. MRG32k3a is an implementation for GPUs that is available in the `curand` library [4]. A basic idea for the definition of a MRG is given in Equation 1.1.

$$\begin{aligned} x_n &= (a_1 \cdot x_{n-1} + a_2 \cdot x_{n-2} + \ldots + a_k \cdot x_{n-k}) \bmod m \\ U_n &= \frac{x_n}{m} \end{aligned} \tag{1.1}$$

Mersenne Twister Given a state vector, a Mersenne Twister [39] generates the next random number by multiplying parts of its state vector with a tempering matrix. The result of the multiplication is then added bitwise to another part of the

Algorithm 1.1: Xorshift

```
static unsigned long y = 2463534242;
unsigned long xorshift() {
  y^=(y<<13);
  y^=(y>>17);
  return (y^=(y<<5));
}
```

old state vector. Compared with other PRNGs, the state of a Mersenne Twister is rather large, which is a problem on GPUs. Mersenne Twister for Graphic Processors (MTGP) [51] is a variant implemented in the `curand` library [4]. It allows for sharing the state between multiple threads of a group and uses cooperative operations to multiply with the tempering matrix in parallel.

Xorshift Marsaglia presents xorshift [36], a simple and extremely fast PRNG that passes most randomness tests. It is based on a combination of XOR operations (^) and bit shifts (<<, >>). Marsaglia lists 8 variations of the algorithm that only differ in the shift direction and order of variables. They also list all 81 triples of magic numbers that lead to a good 32 bit PRNG and all 275 triples that lead to a good 64 bit PRNG. One of the triples is (13, 17, 5). The resulting C code with that triple is illustrated in Algorithm 1.1. The state y is initialized with an arbitrary seed.

Xorwow Xorwow is a variation of xorshift that is introduced by Marsaglia in the same paper as xorshift [36]. It uses 5 arbitrary integers for its state. An additional arbitrary odd number `d` is added to the final random number and increased by an arbitrary odd constant after every generated number. An example for a combination of magic values is given in Algorithm 1.2. The xorwow algorithm is the default PRNG of the `curand` [4] library. It uses a small state and allows for generating independent sub-streams efficiently.

One-PRNG-per-kernel-call-per-thread Scheme A disadvantage of the previously introduced generators is that they need to store state. If multiple threads need to store their state across different kernel calls, this can lead to bottlenecks on the GPU because the data needs to be written to the slow global memory. The one-PRNG-per-kernel-call-per-thread scheme (pK-pT) [48] allows for using independent PRNGs without storing state. Using a cryptographic hash function (such as `MD5`), random numbers with good properties can be generated. The hash func-

Algorithm 1.2: Xorwow

```
static unsigned long x=123456789,y=362436069
    z=521288629,w=88675123,v=5783321,d=6615241;
unsigned long xorwow() {
  unsigned long t;
  t=(x^(x>>2));
  x=y; y=z; z=w; w=v;
  v=(v^(v<<4))^(t^(t<<1));
  return (d+=362437)+v;
}
```

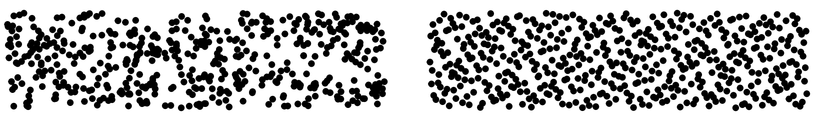

(a) Mersenne Twister PRNG (b) Halton sequence

Figure 1.2: Visualization of pseudorandom and quasi-random numbers.

tions are computationally heavy on GPUs and therefore not suited for drawing large amounts of numbers. pK-pT uses a single hash evaluation to generate the initial state for a PRNG that can then be used without memory access. As an input for the hash function, the authors suggest to use a globally chosen random seed, a time step that is increased for every kernel call and a unique index for each thread. As computational performance of GPUs is growing faster than memory bandwidth, the scheme is becoming more and more relevant.

Quasi-random Numbers Quasi-random numbers are more evenly distributed than actual random numbers. Actual random numbers build clusters, which can cause Monte Carlo methods to converge slowly. Even though quasi-random numbers can be predicted easily, they are important for a large number of scientific fields such as computer graphics [64]. A popular example for a quasi-random number generator is the Halton sequence. A visual comparison between exemplar quasi-random and pseudorandom numbers is given in Figure 1.2.

1.1.3 Stream Partitioning

Partitioning can be used to share a single random number generator between multiple threads. With *parameterization*, each thread uses a different seed or different constants in the algorithm itself. *Cycle splitting* is the method of taking one single stream of random numbers and distributing the numbers to multiple threads. This is what is used in `curand`'s Mersenne Twister implementation. There are different methods to implement cycle splitting. Distributing a fixed block of numbers to each thread is called *blocking*. With the *leapfrog* method each thread receives numbers in a round-robin fashion [58].

1.2 Sampling

Sampling is the operation of selecting an item from a set at random. Picking balls out of bins is taught in schools and is a widely known example of sampling. There are two different variants specifying how to continue after an item is sampled.

With Replacement When sampling with replacement, the sampled items are put back into the set after each turn. Therefore, samples can be drawn multiple times and each sample is independent of other samples. Samples can be drawn from a constant data structure in parallel and without coordination.

Without Replacement When sampling without replacement, the sampled items are taken out of the set and cannot be sampled again. This means that the probability distribution changes after each sample. When drawing multiple samples, each item is returned at most once. Because samples are not independent of each other, some degree of coordination between threads is needed.

1.2.1 Uniform Sampling

Assume having a set of items $\{0, ..., N-1\}$. Uniform sampling is the problem of selecting one of those items at random, with equal probability. The basic problem with replacement is easy to solve because most PRNGs already generate uniformly distributed numbers from the representable range of their data type. The problem becomes interesting when considering sampling without replacement. It is not possible to simply use a PRNG k times because that can lead to duplicates. Various algorithms for uniform sampling are known. Knuth [20] introduces an algorithm that selects or discards items based on the ratio of the number of items left to pick and the unprocessed items that are left. Teuhola and Nevalainen [59] use a hash table to store the already selected samples. Vitter [61] introduces an algorithm that scans over the input items, skipping a random number of items in each iteration.

Algorithm R The parallel algorithm by Sanders et al. [52] is based on a divide and conquer approach. The algorithm uses a recursive *sample* function that repeatedly splits the range of input items in half. It then uses a hypergeometric deviate to determine how many items need to be sampled in each child node of the recursion tree. The algorithm can be parallelized by having each thread only follow one recursion call. To be unbiased, the hypergeometric deviate needs to be the same for all threads that look at the same node of the recursion tree but independent for the children of each node. This can be achieved without any communication by using a PRNG. We apply a similar idea to alias tables in Section 3.4.2.

1.2.2 Weighted Sampling

Assume having a set of items $\{0, ..., N-1\}$ with weights $w_i \in \mathbb{R}$. We denote the total weight with $W = \sum_{0 \leq k < N} w_k$. Weighted sampling is the problem of selecting one of the items i at random, proportionally to its weight. That means that the probability for selecting item i equals $\mathbb{P}(i) = w_i/W$. Algorithms for weighted random sampling are introduced in Chapter 2.

1.3 GPUs

Dedicated graphics processing units (GPUs) are getting more and more powerful, fueled by games, machine learning and crypto-currencies. Central processing units (CPUs) are task parallel, meaning that they are optimized to execute multiple independent tasks at the same time. In contrast, GPUs are data parallel, meaning that the same instruction is executed on different data in parallel. GPUs are accelerators, so while they can speed up calculations, they cannot work on their own. Therefore, the GPU is often called the *device*, while the system that it is connected to is called the *host*.

1.3.1 Architecture

A GPU is highly symmetrical, consisting of multiple streaming multiprocessors (SMs). Each SM has a number of processing blocks as well as some fast memory that is shared between the threads of each block. The Nvidia GeForce RTX 2080, which we use for most of the measurements in this thesis, has 46 SMs and 96 kB of shared memory [1]. Each processing block is responsible for executing multiple threads simultaneously. To do that, processing blocks have multiple arithmetic units for each data type which can be used at the same time. Because the processing blocks are responsible for actually executing the threads, they also

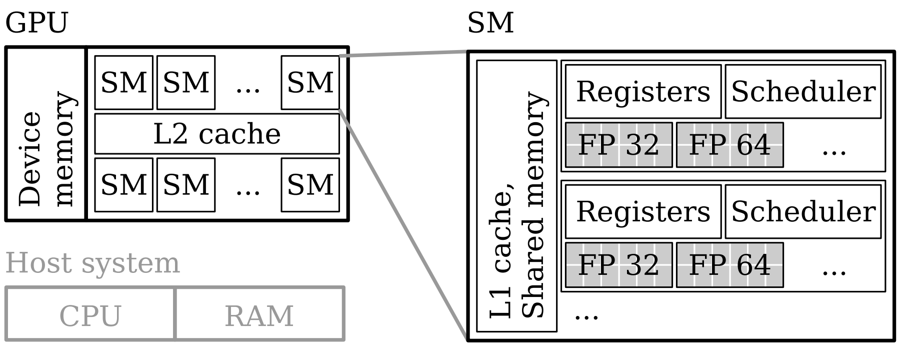


Figure 1.3: Simplified architecture of a GPU. Based on [1].

hold the registers, an instruction cache and a scheduler. Figure 1.3 provides a simplified illustration of the structure [1].

The smallest level of parallelism, 32 threads, is called a warp. All threads in a warp share their instruction pointer. If the threads diverge (for example, on an `if` statement), the inactive threads are masked out and therefore do not touch their data. This means that all threads in a warp take the same time to complete, regardless of the data and individual code paths [1]. The next level of parallelism is called a block. Blocks consist of multiple threads and are guaranteed to be executed on the same SM. They can therefore share data using the memory region located on the SMs. Functions that are executed on the GPU are called *kernels*. When running a kernel, developers can choose the number of blocks and the number of threads per block.

A common problem is that memory access is slow compared to the clock speed. When accessing memory in both branches of an `if/else` block and only some threads of a warp enter each block, all threads have to wait for at least two full memory cycles. By storing the memory address and accessing it outside the branches, the code can be executed up to two times faster.

1.3.2 CUDA

CUDA[2] is a platform developed by Nvidia that allows for developing parallel algorithms on the GPU. It extends the C/C++ compiler, which enables writing both kernels and CPU host code in the same source file. Kernels are basic C++ functions with some restrictions. Most importantly, they cannot directly access the host memory and must not be recursive. Kernels are usually launched from the CPU and then run asynchronously. Each kernel is executed on a grid of blocks.

[2]CUDA was formerly called Compute Unified Device Architecture [44].

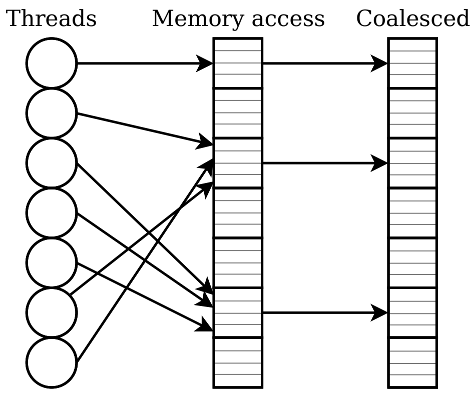


Figure 1.4: Illustration of coalesced memory access. The GPU tries to minimize the total number of 32-byte transactions.

The blocks themselves consist of a grid of threads. Threads from the same block can synchronize and share memory, while threads from different blocks cannot cooperate directly. The blocks are scheduled to streaming multiprocessors by the hardware [50].

1.3.3 Memory

The GPU has a large global memory (also called device memory). Allocations of memory regions are usually performed from the host system. In addition to global memory, each block can allocate shared memory that can be accessed much faster. The memory region is called *shared* because all threads of a block can access it. It cannot be accessed across groups and is not persistent across kernel launches. Shared memory is located in the same hardware area as the L1 cache, directly on the SM. Each thread has its own registers that store variables [3].

Coalescing

Whenever the threads of a warp access global memory, CUDA uses coalesced memory accesses. This means that it collects all memory locations that are requested and then minimizes the number of 32-byte transactions that are needed to fulfill the requests [3]. Coalesced accesses are able to speed up memory operations significantly. If each thread of a warp reads unrelated memory locations though, large parts of the 32-byte transactions' data is not used. Figure 1.4 shows an illustration of coalesced memory access.

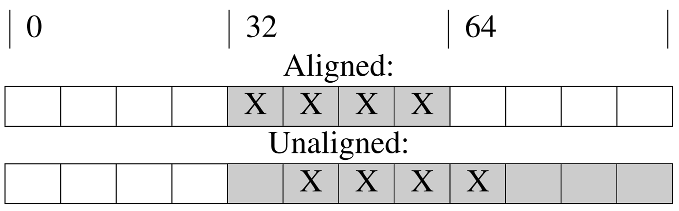


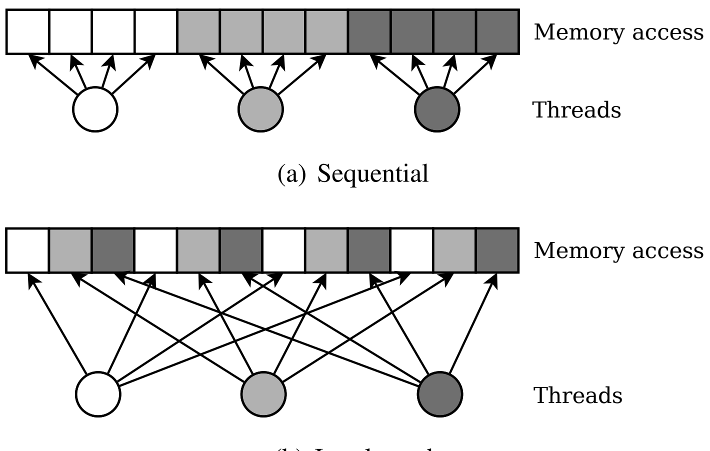
Figure 1.5: Aligned and unaligned memory access. Cells marked with X are actually actively accessed and cells highlighted in gray need to be fetched from memory.

Alignment To ensure that memory accesses can be coalesced, threads should align their memory accesses to 32 bytes. If it is not aligned, the number of required memory transactions may be higher which leads to a noticeable performance penalty [3]. Figure 1.5 illustrates memory regions of 8 bytes each and compares an aligned and an unaligned access pattern.

Interleaved Addressing A common memory access pattern is to loop over input data of size $k \cdot |\mathit{Threads}|$ once. On CPUs, this would be achieved by splitting the memory region into one section for each thread. Each thread t then linearly iterates over its section, accessing memory location $x_{k \cdot t+i}$ in iteration i. This means that the concurrent memory reads of the loop are always k memory locations apart. On GPUs, this access pattern cannot be coalesced and is therefore slow. With interleaved addressing, the indices for the memory accesses are re-arranged, so that each thread instead accesses item $x_{i \cdot |\mathit{Threads}|+t}$. While the pattern is less intuitive when writing the kernel, its performance is significantly better because neighboring threads access neighboring memory locations and therefore enable coalescing. Figure 1.6 shows an illustration of the two access patterns.

Bank Conflicts

The GPU's memory physically consists of multiple memory modules called *banks*. All memory locations are distributed over the memory banks in a round-robin fashion. Each memory bank can only perform one memory transaction at a time. Multiple banks can read their memory in parallel. If k threads access the same memory bank in different rows, a k-way bank conflict occurs. The memory accesses then need to be serialized, resulting in reduced throughput [43]. A simple rule-of-thumb is that threads that are next to each other in CUDA's thread grid should try to also access memory locations that are next to each other. This balances the memory operations between banks. A technique to achieve that is to use interleaved addressing, which is introduced above.



(a) Sequential

(b) Interleaved

Figure 1.6: Memory access patterns. Interleaved addressing can be coalesced by the GPU.

1.3.4 Profiling Tools

In this thesis, we mainly use two profiling tools that are introduced below. The tools provide important information that makes many of our measurements possible in the first place. The performance measurements for charts are executed using GPU timers, which have less impact on the performance than a full profiler.

Nvidia Visual Profiler

The Nvidia Visual Profiler is focused on overall performance optimization of a whole program. It generates a timeline of the program execution for analyzing which kernels are executed at which point in time. It also shows if the kernels are executed directly after each other, in parallel streams, or if they are delayed by synchronization with the CPU.

Nvidia NSight Compute

The Nvidia NSight Compute tool is focused only on one kernel at a time. It benchmarks the kernel and collects detailed information, such as compute utilization of the arithmetic units (e.g. integer, floating point), memory read/write volume and utilization, or disassembly with source counters. The memory's Speed Of Light (SOL) value describes how the actual achieved bandwidth compares to the theoretical maximum.

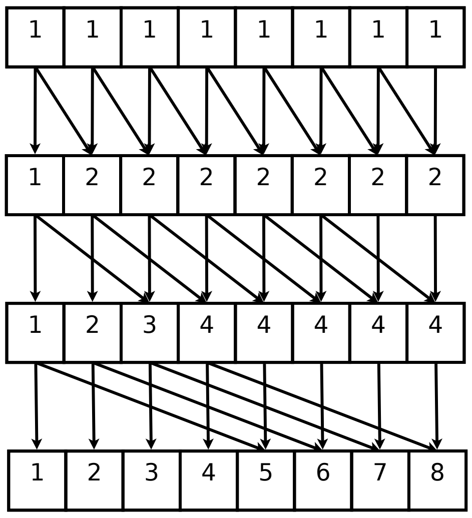


Figure 1.7: Basic parallel prefix sum algorithm. Arrows illustrate memory read and write operations.

1.4 Prefix Sums

Prefix sums are a basic building block for parallel algorithms. An inclusive prefix sum of a given array a is defined as $\textit{prefix}_{\textit{incl}}(a)_k := \sum_{i=0}^{k} a_i$, an exclusive prefix sum $\textit{prefix}_{\textit{excl}}(a)_k := \sum_{i=0}^{k-1} a_i$. In the following section, we introduce two parallel construction algorithms. To switch from an inclusive prefix sum to an exclusive one, the whole array can be shifted one item to the right, setting the first item to 0.

1.4.1 Basic Parallel Prefix Sum

The basic parallel approach is introduced by Horn [24]. The algorithm starts N threads, one for each item in the array, and stores an offset $\delta = 1$. In each iteration of the algorithm, if an index is smaller than δ, it remains unchanged. If it is bigger, the corresponding thread stores the sum of its current item k and the item $k - \delta$. The offset is doubled in each iteration. Figure 1.7 illustrates the algorithm. As described, the method calculates an inclusive prefix sum.

Memory volume The method requires $\log_2(N)$ kernel calls with N threads each. In order to ensure that read and write operations do not interfere, either heavy synchronization or double-buffering has to be used. In total, this approach executes $\sum_{i=1}^{\log_2(N)} N = N \cdot \log_2(N)$ write operations, as well as $\sum_{i=1}^{\log_2(N)} (N + (N - 2^{i-1})) = (2N - 1) \cdot \log_2(N) + 1$ read operations.

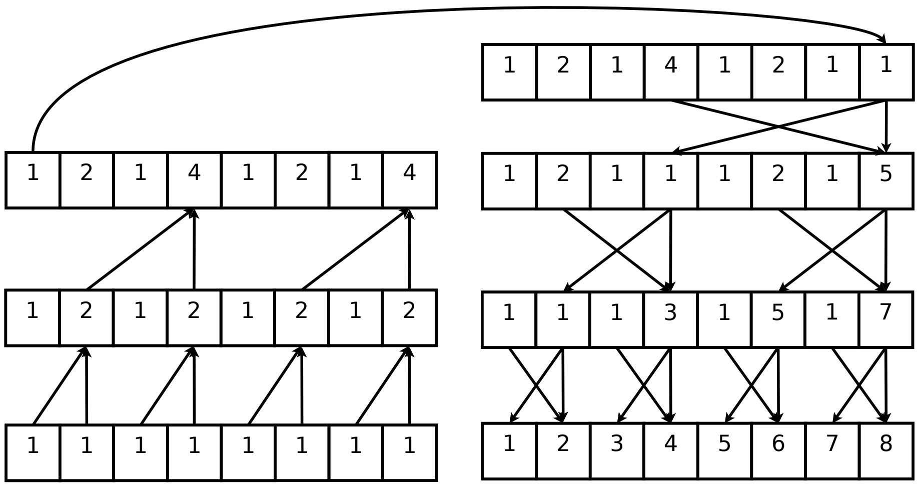


Figure 1.8: Work-efficient prefix sum algorithm. Arrows illustrate read and write operations.

1.4.2 Work-Efficient Prefix Sum

Prefix sum algorithms are memory bound. The basic parallel method with its large number of memory operations is therefore a bottleneck. Blelloch [13] introduces a work-efficient algorithm for prefix sums.

The method operates in two phases, as illustrated in Figure 1.8. In the up-sweep phase, the total sum is calculated by using a reduction in a binary tree. The intermediate results that are calculated during the reduction are stored in the array. In the down-sweep phase, nodes can calculate the total sum of the subtrees that are located to the left of them by adding the sum of items to the left of their parent and the sum of their left sibling. This is propagated back to the leaf nodes to calculate the full prefix sum. When setting the last item to 0 between the two steps, the method calculates an exclusive prefix sum. Setting it to the first item results in an inclusive prefix sum.

Memory volume The algorithm requires $2 \cdot \log_2(N)$ kernel calls with logarithmically decreasing and then increasing number of threads. In total, the approach executes $(\sum_{i=1}^{\log_2(N)-1} 2^i) + 1 + (\sum_{i=1}^{\log_2(N)} 2^i) = 3N - 3$ write operations and $(\sum_{i=2}^{\log_2(N)} 2^i) + 1 + (\sum_{i=2}^{\log_2(N)} 2^i) + (\sum_{i=1}^{\log_2(N)-1} 2^i) = 5N - 9$ read operations. It is therefore asymptotically more efficient than the basic method.

1.5 Concepts

In the following section we introduce some concepts that are touched upon in this thesis. The concepts are widely known but because they provide an important foundation, we reiterate them here.

Speedup Algorithm execution timings are different on each hardware model. To compare the performance of two algorithms, a widely used metric is the *speedup*, which abstracts away the actual machine. If an algorithm A needs time T_a for a task and an algorithm B needs T_b, the speedup of B is $S = T_a/T_b$. Speedup is often used in parallel computing to compare how much faster the parallel algorithm with time T_{par} gets when comparing to the fastest sequential algorithm with time T_{seq}. In this case, the speedup can be calculated by $S = T_{seq}/T_{par}$. The optimal speedup on p processors is p, though some effects like caching can lead to even more improvement, also called *superlinear speedup*. The maximum speedup with unlimited processors is $S = ((1-\alpha)+\alpha/p)^{-1} \to (1-\alpha)^{-1}$ for $p \to \infty$, where α describes the proportion of the program that can be parallelized (Amdahl's Law [8]).

Strong Scaling There are two basic ideas how to scale an algorithm to multiple processors. Strong scaling describes how solving a problem of fixed size gets faster when adding more processors p.

Weak scaling Weak scaling describes how the time for executing an algorithm increases when adding more processors with a fixed problem size per processor. This means that the total problem size increases whenever a new processor is added [3]. For this thesis we only perform strong scaling experiments.

Chapter 2

Related Work

In the following chapter we review weighted random sampling algorithms from the literature. We then continue with describing applications of the algorithms.

2.1 Scan Method

The naive algorithm for weighted random sampling calculates the probability $\mathbb{P}(i) = w_i/W$ of each item, where $W = \sum_i w_i$ is the sum of all weights. It then assigns each item i an interval $I_i \subset [0, 1)$ by calculating the prefix sum of the probabilities $\mathbb{P}(i)$. To sample, the algorithm generates a random number $U \in [0, 1)$ and searches for the interval I_k with $U \in I_k$ [21]. The sampled item is then k. A linear search takes $\mathcal{O}(N)$ for each sample. Using binary search, this can be improved to $\mathcal{O}(\log(N))$. While the method is easy to understand and implement, it is not efficient enough for selecting a large number of samples.

2.2 Cutpoint Method

Fishman and Moore introduce the cutpoint method [21], a modification of the linear scan algorithm. Their algorithm divides the interval $[0, 1)$ into m segments. Each segment stores a reference to the first interval of a linear scan that intersects with the segment. Instead of performing a linear scan looking for U from the beginning, scanning can then be started at the reference of segment $\lfloor U \cdot m \rfloor$. The structure needs $\mathcal{O}(N + m)$ space and provides a trade-off between sampling performance and memory usage. For $m \ll N$, the method's runtime is still dependent on $\log(N)$ and therefore not efficient enough for drawing a large number of samples. In contrast, the methods explained in the following sections are able to sample in time $\mathcal{O}(1)$.

2.3 Lookup Table

A method that can sample in constant time is a lookup table. First, all weights need to be converted to integers by expressing them in the form $w_i = \alpha \cdot d_i$, where $d_i \in \mathbb{N}_0$ is the integer weight of each item and $\alpha \in \mathbb{R}^+$ is a constant chosen small enough to evenly divide all w_i. After that, an array of size $\sum_i d_i$ is allocated. Every item i is written to the array d_i times. To sample, a uniform random element from the array is returned [40]. A big disadvantage of the lookup table is that the space requirements depend not just on the number of items but also on the values of their weights. Even for just two items, the space requirements and therefore the construction time can be arbitrarily high. For example, the weights $w_0 = 10^{-k}$ and $w_1 = 1 - 10^{-k}$ lead to a space requirement of 10^k lookup table entries.

2.4 Alias Table

Alias tables, introduced by Walker [63], are a data structure for sampling from discrete distributions. An alias table T has N rows, where N is the number of items in the input set. Each row represents a bucket that can be sampled. It has two columns, namely a weight $T_i^w \in \mathbb{R}$ and an alias $T_i^a \in \{0, ..., N-1\}$. To sample, a uniform random number $U \in [0, 1)$ is used to select a row $k = \lfloor U \cdot N \rfloor$ from the table. The algorithm then uses the decimal places *frac*$(U \cdot N)$ to either return the item k or its alias T_k^a by checking if $(U \cdot N - k) \cdot W/N < T_k^w$, where W is the sum of all weights. Alias tables allow for sampling an item by generating only one uniform random number and performing one lookup in the data structure. Sampling an item is therefore in $\mathcal{O}(1)$. It is possible to construct an alias table for every discrete distribution. The idea is that heavy items that are more likely to be sampled than a table row ($w_i > W/N$) give excess weight to the buckets of one or more light items. This procedure is illustrated in Figure 2.1.

2.4.1 Walker's Construction

The original alias table construction algorithm by Walker [63] subtracts W/N from each item's weight and stores the result in B_i, describing the left-over space in each bucket that may be negative. The algorithm then loops over the items and searches for the bucket k with maximum remaining weight and bucket l with minimum remaining weight. It then assigns $T_l^a := k$. The weight of the light item's bucket is assigned the share of W/N that it fills: $T_l^w := B_l/(W/N) + 1$. The remaining weights compensate each other using $B_k := B_k + B_l$ and $B_l := 0$. This continues until all items have remaining weight $B_i = 0$. The construction therefore requires time $\mathcal{O}(N^2)$ and does not lend itself to parallelization.

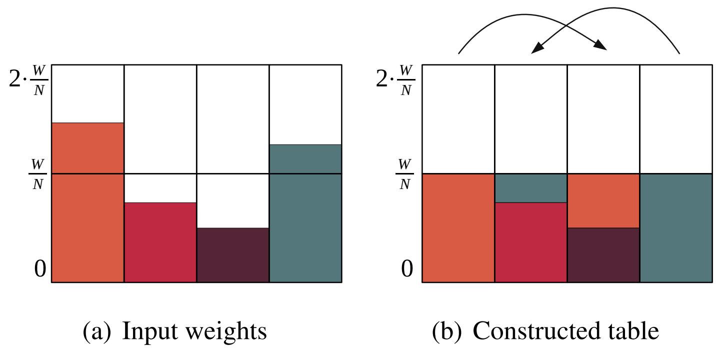


Figure 2.1: Illustration of alias table construction. Buckets of items with weight smaller than the average are filled with excess weight of heavy items.

2.4.2 Vose's Construction

Vose [62] develops a variation of the alias table construction algorithm with the main idea of explicitly maintaining lists of *light* and *heavy* items. Items with weight $w_i \leq W/N$ are considered *light* and are added to the list `l`. The other items are considered *heavy* and are added to the list `h`. While there are items available, the algorithm takes a *heavy* item $j \in$ `h`. It then distributes the excess weight of that item by taking *light* items and filling their buckets. The item j is stored as the alias of those *light* items and the excess weight of j is decreased accordingly. When the remaining weight of j drops below W/N, it is considered *light* and added to `l`. In each iteration, at least one of the *light* items' buckets is filled, ensuring that the running time is in $\mathcal{O}(N)$.

2.4.3 Sweeping Alias Table Construction

Hübschle-Schneider and Sanders [27] introduce an alternative alias table construction algorithm that does not require a secondary array or preprocessing. It is therefore more space efficient than Vose's [62] approach. The algorithm uses two indices, i and j, that always point to the next *light* and *heavy* item. In each iteration, either the bucket of a light item is filled from a heavy item or the remaining weight of a heavy item drops below W/N and its bucket is filled from the next heavy item. Therefore, one of the pointers can be advanced to point to the next item. Because i and j are monotonic, the algorithm has linear time complexity. An advantage on CPUs is that the algorithm accesses the memory linearly and does not need preprocessing for classifying the items.

2.4.4 Parallel Alias Table Construction

The algorithms described in the previous sections are all sequential. For large data sets, sequential construction of the alias table can become a performance bottleneck. Hübschle-Schneider and Sanders [27] extend the idea of the sweeping alias table construction to develop the parallel splitting-based alias table construction algorithm (PSA). The method uses a two-step approach, as described in the following paragraphs.

Splitting During the first step, splitting, the algorithm precomputes the state of the sweeping construction (see Section 2.4.3) at s positions. These splits define sections that can later be worked on in parallel. The algorithm selects a number of *light* and *heavy* items in a way that the number of items in each section is N/s and the weights are balanced. This is done by executing a binary search on the prefix sums of light and heavy items. To deal with the weight that does not fully fit into an section, the algorithm stores the remaining weight as *spill*. The result of the splitting step is a list of section boundaries and their respective spill values.

Packing The second step then writes the actual alias table. In parallel, each processor takes one of the sections. If there is weight spill from the previous section, the algorithm initializes the weight that needs to be distributed accordingly. It then iterates over the items of the section and distributes weight from buckets of *heavy* items to buckets of *light* items like in Vose's algorithm (see Section 2.4.3).

2.4.5 PSA+

Hübschle-Schneider and Sanders [27] develop a semi-greedy variant of their PSA algorithm, called PSA+. The idea is that instead of calculating prefix sums and splits for all items, the algorithm greedily builds the alias table in fixed-size sections until each section runs out of light or heavy items. PSA+ then only performs the PSA construction with the remaining items. When assigning uniform random weights to each item, it is possible to pack a majority of the items greedily.

2.4.6 Sampling on GPUs

Constructing alias tables on the GPU is not covered in the literature yet. Mohanty et al. [42] implement the sampling step of alias tables for Monte Carlo simulations. They use two independent tables in order to sample 2-dimensional values. Their implementation shows a significant increase in sampling throughput when comparing to the CPU. In their work, the tables only need to be constructed once. They therefore construct the table on the CPU and copy it afterwards.

2.4.7 NP Completeness

Smith and Jacobson [55] analyze the performance of alias table sampling. They note that table rows without an alias ($T_i^w = W/N$) can be sampled more efficiently than rows with an alias. By reducing the NP complete problem *Numerical 3-Dimensional Matching* to the problem of building an alias table with minimal items having an alias, they show that the problem is NP complete. An input that can lead to a non-optimal alias table is to take $w_0 = 0.5$, $w_n = 1.5$ and for all other items $i \notin \{0, N\}$ take $w_i = 1.0$. This results in a bucket size of $W/N = 1$. A greedy algorithm might assign the alias of all items to the subsequent item instead of moving the excess weight of the very last heavy item to the front. As this is a rather theoretical result, we do not minimize the number of items without an alias.

2.5 Brown's Method

Brown's Method [15, 53] is a random sampling algorithm that is similar to alias tables. It uses one single array D that contains items and their aliases alternately. To sample an item, the algorithm generates a uniform random number $U \in [0, N) \subseteq \mathbb{R}$. The item to be sampled is then determined by accessing $D_{\lfloor U \rfloor + U - w_{\lfloor U \rfloor}}$. A problem that the author notes is that in order for the algorithm to be unbiased, the lower digits of U (namely *frac*(U)) need to be uniformly distributed. This can become a problem when dealing with large floating point numbers. The problem can be solved by drawing another random number $U_2 \in (0, 1]$ and returning $D_{2 \cdot \lfloor U \rfloor + U_2 - w_{\lfloor U \rfloor}}$ instead. For sampling, the method needs two memory access operations to unrelated locations. We therefore assume that it is slower than standard alias tables.

2.6 Sarno's Method

Sarno et al. [53] propose a branchless sampling method using a lookup table of only N items. Their algorithm generates a new set of items with weights $w_i^{sarno} := i \cdot w_i \cdot N$. It then uniformly samples an item from the new set. While the method actually samples different random variables than the ones given as input, it preserves mean and variance. The modified distribution is still useful for Monte Carlo simulations. This thesis focuses only on exact solutions.

2.7 Binder's Method

Binder and Keller [11] introduce a monotonic sampling algorithm for GPUs that can sample in $\mathcal{O}(1)$ average case and in $\mathcal{O}(\log(N))$ worst case running time. They use a modification of the cutpoint algorithm (see Section 2.2). In order to find the actual samples within the cut regions, they use a radix tree forest. They introduce a parallel algorithm for GPUs to construct the forest.

The authors criticize the fact that the alias method is not monotonic. When using a monotonic sampling algorithm, a higher random number U also generates a higher sample. Other examples for monotonic sampling algorithms are scanning (see Section 2.1) or the cutpoint method (see Section 2.2). When sampling with a quasi-random number generator (see Section 1.1), monotonic methods are able to preserve the low discrepancy. In this thesis, we do not consider the additional requirement of monotonicity and are rather interested in a better throughput.

2.8 Space-Efficient Rejection Sampling

Bringmann and Larsen introduce an algorithm for weighted sampling that is based on rejection sampling [14]. The basic idea is that instead of filling the buckets of items with an alias, they just keep them empty and reject samples that would draw the empty bucket space. For each item i, they calculate $\mathit{Occ}_i := \lfloor N \cdot w_i / W \rfloor + 1$ and then generate an array A that contains it $\mathit{Occ}(i)$ times.

For sampling, they choose a uniform random index k in A. If A_k is the first occurrence of the item ($k = 1$ or $A_{k-1} \neq A_k$), they use the remaining weight ($\mathit{frac}(N \cdot w_{A_k} / W)$) to decide if they accept the sample. If the sample is rejected, they just sample again. The probability for accepting a sample is > 0.5, so the expected time for sampling is in $\mathcal{O}(1)$.

Because A contains a linear, ordered list of numbers, it is enough to just store the fact that a new item starts at position i. This makes it possible to store A implicitly in a bit array. The actual item that is stored in position i can then be calculated using the rank (number of bits before position i that are 1).

2.9 Applications

The following section introduces applications of weighted random sampling that can be found in the literature. Some methods are specifically focused on alias tables or are especially interesting on GPUs.

2.9.1 Graph Generation

Graphs have many applications in computer science. The World Wide Web, maps or social networks are only a few examples of graphs. Graphs in real world applications are getting larger and larger, so research is done to develop algorithms that can process large graphs. In order to test the algorithms, researchers need data sets. Unfortunately, real world data sets are often not publicly available. R-MAT (**R**ecursive **Mat**rix) [17] is a model to generate large, realistic network graphs that can be weighted, bipartite and directed while having only a few input parameters to tweak. The algorithm recursively generates the adjacency matrix by splitting the matrix into 4 quadrants. Edges are "dropped" onto the graph one-by-one depending on probability parameters for each of the quadrants. In each recursion layer, the parameters are modified with noise for better fluctuation. Hübschle-Schneider and Sanders [26] suggest an algorithm that precomputes paths of the recursion process and calculates their probability. To insert an edge, they sample from the paths using an alias table. In general, many random graph generation algorithms benefit from fast sampling techniques, as noted in Ref. [46].

2.9.2 Text Generation

Hmedeh et al. [23] develop a recommender that suggests news articles from multiple RSS feeds. For their evaluation, they use alias tables to generate large numbers of synthetic subscriptions with specific word distributions.

2.9.3 Graph Databases

GraphJet [54] is a graph database developed at Twitter that is used to give recommendations in real-time. The database stores a bipartite graph where one set contains users and the other set tweets. Labeled edges between users and tweets represent interactions, such as marking a tweet as favorite. The graph is represented in adjacency lists that are organized in multiple segments. Only the most recent segment is written to. After a segment is full, it is optimized for reading speed. In order to generate recommendations for the timeline, GraphJet performs a random walk on the bipartite graph. It starts with a user and follows edges to tweets and back to users repeatedly. An outgoing edge within a segment is sampled by generating a uniform random number between 0 and the degree of the vertex. In order to sample edges from different segments, the algorithm keeps track of the total degrees in each segment and then uses an alias table to choose the segment that the edge should be sampled from.

2.9.4 Nuclear Medicine

In simulations, random numbers with specific distributions are often required. One example is the dose planning method used in nuclear medicine [65]. Nuclear medicine can be used to analyze physiological processes in patients. To do that, the patients consume a nuclear emitter. Attaching the emitter to specific substances allows for observing how those substances travel through a living body. In order to minimize the radiation exposure on the patient, the process is simulated with the patient's specific anatomy beforehand. Wilderman and Dewaraja [65] simulate how the nuclear energy travels through the body by using path tracing. In order to sample source positions for simulated particles, they use an alias table initialized with the measured intensity of the nuclear emitter.

2.9.5 Stochastic Flow Networks

Flow networks are graphs with weighted edges that can be compared to water pipes. They have a specific source node and a sink node. Each edge can transport a specific capacity, comparable to the diameter of the pipe. A flow is an assignment of numbers to edges where each number is from the interval $[0, \text{capacity}]$. The total incoming flow at each node must match the outgoing flow, except for source and sink. Stochastic flow networks modify the definition by assigning varying capacities to each edge using a probability distribution. Alexopoulos and Fishman [7] introduce a Monte Carlo method for determining the maximum flow. To sample the capacity of edges, they use alias tables.

2.9.6 Noise Function Generation

In computer graphics, noise functions are an important building block of procedural texturing and modeling. One of the most famous examples is the Perlin noise [47], a superposition of multiple frequencies. Another noise function that is extremely flexible is the Sparse Gabor Convolution [31]. It is determined by using a convolution of white noise with the gabor kernel. Because convolutions are simply a multiplication in the frequency domain, the Sparse Gabor Convolution noise allows for precise control over the appearance by modifying the kernel in the frequency domain. Galerne et al. [22] present an optimized version they call bandwidth-quantized Sparse Gabor Convolution and a way to automatically estimate the parameters to look like exemplar Gaussian textures. To determine the kernel parameters, they use weighted random sampling from alias tables. For their interactive noise editor or for dynamic noise changes, they explicitly highlight that it is important to be able to quickly construct the alias tables.

2.9.7 Image-based Lighting

In photorealistic image rendering, Monte Carlo methods are used to estimate the amount of incoming light on a surface. Light could originate from any direction, so all directions can contribute to the final image. In order to speed up convergence, it is advisable to use importance sampling, a technique where directions that likely contribute more light are sampled with higher probability. A method to achieve photorealistic lighting in a scene is to use real images such as environment maps as a light source. Burke et al. [16] introduce *bidirectional importance sampling*, a way to combine both reflectivity and distribution of lights. They account for the fact that the reflectance function is different for each point on the surface by estimating the light distribution at runtime. They then use importance sampling from an alias table that is built from the light distribution in the scene. In contrast, multiple importance sampling by Veach [60] mixes samples from different distributions instead of sampling the target distribution directly.

2.9.8 Machine Learning

Latent Dirichlet Allocation (LDA) [12, 57] is a method for topic modeling. Given a set of documents (usually text), it can determine and assign relevant topics. The method assumes that each topic has a specific distribution of words that characterize it. It can then, for each document, provide a distribution of underlying topics. The algorithm first assigns random topics to each word. It then iteratively scans the documents, modifying their classification, until the process converges. During classification, weighted random sampling from different distributions is needed.

Steele and Tristan [56] optimize LDA modeling on GPUs by choosing the loop variables in a way that achieves coherent memory access (see Section 1.3.3). Instead of assigning a document to each thread, the threads of a warp work together to first read data into the local memory. To sample, the authors use the binary search method (see Section 2.1). Li et al. [33] review different sampling algorithms for LDA modeling. While alias tables are fast to construct and fast to sample, the authors use a slower approach. The reason is that at the time of writing their paper, no parallel algorithm for constructing alias tables was known.

Chapter 3

Algorithm Design

The following chapter introduces our newly developed split, pack and sampling methods. For preliminary measurements, we use a Nvidia GeForce RTX 2080 GPU. The full hardware setup and extensive performance evaluations can be found in Section 5.

3.1 Split Method

In the following section we introduce our optimized split methods for parallel alias table construction. We denote the number of splits with the variable s.

3.1.1 Baseline Method

As a baseline, we transfer the original split algorithm by Hübschle-Schneider and Sanders [27] (see Section 2.4.4) to the GPU.

3.1.2 Partial p-ary Split

The original split algorithm of Ref. [27] uses binary search to find the split positions. On GPUs, this works well because threads of the same group search for neighboring splits. Therefore, the first search iterations take the same branches and therefore access the same array elements. The GPU can coalesce these memory operations. The binary search method does not utilize the fact that GPUs have multiple memory banks that can be accessed in parallel. In each iteration with binary search, a group only reads from a single memory bank. We hereby introduce a new search operation that we call *partial p-ary search* that utilizes multiple memory banks.

p-ary Search For finding an item in a sorted list, Kaldewey et al. [30] evaluate p-ary search on GPUs. With p-ary search, each iteration splits the search range into p groups by looking at equally distributed memory locations in parallel. Instead of halving the search range in each iteration, p-ary search can therefore reduce the search range by $1/p$. The GPU has 32 memory banks that can be accessed in parallel, though. Even when $p > 32$ or there are bank conflicts, the GPU can schedule memory operations efficiently. After each memory access the threads synchronize and limit the search range to one of the sections. For GPUs, thread synchronization is cheaper than memory access. When using the p-ary search algorithm as described in the literature, all threads cooperate to search for one single item. Still, Kaldewey et al. demonstrate a speedup for large numbers of queries.

Partial p-ary Search For searching multiple items in a sorted list, we introduce the new *partial* p-ary search algorithm. The method, to our knowledge, has not previously been described in the literature. Our method can be used if each thread group looks for multiple elements that are close together in memory. Our partial p-ary search algorithm works in two phases. The first phase executes p-ary search for all items of the group at once. In each iteration, instead of continuing the search on one section, partial p-ary search reduces the search range to the smallest and largest section that contain at least one of the searched elements. Therefore, in contrast to p-ary search, a range of more than one section might be used for the next iteration. The algorithm continues with that until it is no longer able to reduce the search range. In the second phase, each thread looks for its own item using ordinary binary search. The binary search operation only needs to search the range determined by the p-ary search operation. We call the method *partial* p-ary search because only the first iterations of searching are executed in p-ary fashion before falling back to standard binary search. Algorithm 3.1 illustrates the idea.

Application to Splitting When looking for splits during parallel alias table construction, threads have to search for the number of heavy items to include. For our new partial p-ary split algorithm, we use partial p-ary search instead of binary search.

3.1.3 Uncompetitive Methods

The following paragraphs introduce additional approaches to splitting. Preliminary experiments show that the methods are uncompetitive, so we only describe them briefly.

Algorithm 3.1: Partial p-ary search

```
function binarySearch($\langle l_1, ..., l_N \rangle$: List to search in,
            x: Item to search, (a, b): Initial search range)
    while a–b > 1 do
        s := (a + b) / 2
        if $l_s$ > x then b := s else a := s + 1
    return a

function partialParySearch($\langle l_1, ..., l_N \rangle$: List to search in,
            $\langle x_1, ..., x_p \rangle$: Ordered items to search, t: Thread index)
    (a, b) := (0, N)
    $\langle s_1, ..., s_p \rangle$: Pivots of all threads (shared)
    $\langle r_1, ..., r_p \rangle$: State of all threads (shared)

    while range can be reduced do
        $s_t$ := a + t · (b–a)/(p–1)
        if $x_0 > l_{s_t}$ then
            $r_t$ = smaller
        else if $x_p < l_{s_t}$ then
            $r_t$ = larger
        else
            $r_t$ = within
        a := $s_m$ where $m$ is the maximum number with $r_m$ == smaller
        b := $s_n$ where $n$ is the minimum number with $r_n$ == larger
    return binarySearch(l, $x_t$, a, b)
```

Inverse split For each of the s threads, the baseline split method performs a binary search for the number j of heavy items from `h` to include. To make use of interleaved addressing, we present the inverse split algorithm. It starts one thread for each item in `h` and checks them all in parallel, performing a simple loop over all s splits. The idea is that having the number of heavy elements j fixed throughout the kernel call allows for reducing memory operations. The method is 60 times slower than the baseline method and makes the transferred memory amount more than 500 times larger.

Parallel inverse split The parallel inverse split method extends on the idea of inverse splitting. Using a 2-dimensional grid for the execution, each loop iteration of k can be done in its own thread in parallel. Kernels therefore no longer need any loops. The method is 70 times slower than the baseline method and increases the transferred memory amount by a factor of 1500.

3.2 Pack Method

The following section introduces our new algorithms for packing an alias table on the GPU. The pack step is similar to the sequential alias table construction but starts at a specific position that is determined by the split.

3.2.1 Baseline Method

As a baseline, we transfer the original pack algorithm by Hübschle-Schneider and Sanders [27] (see Section 2.4.4) to the GPU.

3.2.2 `l` and `h` in Shared Memory

The baseline pack method accesses the `l` and `h` arrays in a way that cannot be coalesced. For the shared memory pack method, we first copy the interesting array sections to shared memory. Copying can take place in an efficient interleaved fashion. The threads that then execute the actual packing read `l` and `h` from shared memory instead of the global memory. Because the shared memory region is much faster, the inefficient access pattern has a significantly lower performance penalty. Packing itself is, except for the accessed memory locations, identical to the baseline method.

3.2.3 Weight in `l` and `h` Arrays

Using the original pack method by Hübschle-Schneider and Sanders [27], the `l` and `h` arrays only store the index of the light and heavy items. The pack method reads items from the arrays and then loads the corresponding weight from the input array. Using the shared memory method above, access to the `l` and `h` arrays is cheap but access to the weights array is still expensive and not properly coalesced. Instead of only storing the item index in `l` and `h`, we now also store the weight of the items. When partitioning the input items into the `l` and `h` arrays, the weights array needs to be read anyway. We can therefore directly write the weights in a coalesced way without additional read operations.

3.2.4 Chunked Loading

A disadvantage of the shared memory method is that it needs a large number of splits because the shared memory is rather small. The idea of the chunked pack method is to reduce the number of splits required by loading the `l` and `h` arrays to shared memory in chunks as needed. As soon as all threads of a block have no light or no heavy items left, the threads cooperate to load new data from the global `l` and `h` arrays to shared memory in an interleaved way (see Algorithm 3.2). Packing itself (`packUntilChunkEnd`) is similar to the baseline pack method, differing only in the initialization of `i`, `j` and `w`, and an additional check for the chunk size.

3.2.5 Uncompetitive Methods

The following paragraphs introduce additional approaches for packing alias tables. Because preliminary experiments show that they are uncompetitive, they are only described briefly.

Alias Table in Shared Memory When writing the finished alias table, the global memory is accessed inefficiently. We experiment with a method that writes the finished alias table to the shared memory instead of the global memory. It then copies the result back to the global memory in an interleaved fashion. A problem of the method is that different threads of multiple groups can write to overlapping sections of the alias table. Additionally, depending on the distribution of weights, the memory range that is written to can be too large for the shared memory. Even with optimal distributions that do not cause overlap of the alias table, preliminary experiments show a worse performance than with the baseline method.

Algorithm 3.2: Chunked pack method

```
function chunkedPack()
    while not all threads are finished do
        copyChunks()
        if current thread is not finished then
            packUntilChunkEnd()

function copyChunks()
    foreach worker thread do
        if light items are mostly used up then
            Copy light items to shared memory
        if heavy items are mostly used up then
            Copy heavy items to shared memory

function packUntilChunkEnd()
    i, j, w: State like in the PSA method
    Restore state of i, j, w
    while true do
        if light or heavy array ran out of items then
            Store state of i, j, w
            return

        // Normal packing loop
        if w $\leq W/N$ then
            ...
        else
            ...
    Mark thread as finished
```

Pre-filled Alias Table Hübschle-Schneider and Sanders [27] do not use the weights array in the pack step. Instead, they initialize the alias table with the weights. That way, the pack method can use memory access operations that are less scattered. Preliminary experiments show that the pre-filled pack method is roughly 15% slower than the baseline method and reads 40% more data from global memory. This is caused by the fact that the plain weights array is packed more densely than the alias table entries. In addition to the pack method itself being slower, it also requires preprocessing to copy the weights.

Sweeping Pack Method The primary bottleneck of the pack method is its memory access. Both reading the `l` and `h` arrays and writing to the alias table cannot be coalesced directly. For the sweeping pack method, instead of using the `l` and `h` arrays to access the next light/heavy item, we use the original sweeping alias table construction algorithm that reads the weights array linearly to search for the next item (see Section 2.4.3). Preliminary experiments show that the method is more than 3.7 times slower than the baseline method.

Reordering `l` and `h` Arrays Each thread reads a section of the `l` and `h` arrays to fill the alias table. Because the arrays are sorted by item index, write operations to the alias table are done in a way that cannot be coalesced. In order to make concurrent memory access of the threads more local, we reorder the `l` and `h` arrays before executing the prefix sum and the split kernel. Preliminary experiments show that the method is up to 2.2 times slower than the baseline method.

Precomputed `weight[l[i]]` The pack method accesses the weights array indirectly using `weight[l[i]]`. To speed up this memory access, we develop a method that precomputes an array for both `l` and `h` that contains the weights directly. Preliminary experiments show that the pack step of the method is roughly 10% slower than the baseline method. While the method is able to decrease the total memory read volume by 10%, the L1 cache hits are reduced by 20%. Additionally, the method needs more time for preprocessing.

3.3 PSA+

Hübschle-Schneider's and Sanders' algorithm [27] executes PSA+ before partitioning into the `l` and `h` arrays. For that, it uses the sweeping pack method, which is not efficient on GPUs. Packing greedily before partitioning is therefore infeasible. Packing greedily after partitioning would reduce the total time no more than the time saved because of the prefix sums with fewer items. The idea of our PSA+

implementation is to perform greedy packing while partitioning. During partitioning, the arrays are already available in the fast shared memory. With this method, we are able to reduce both the time of the prefix sum and the memory reads and writes to the `l` and `h` arrays. Our PSA+ implementation does not perform any additional access to global memory that would not have been done with PSA.

3.4 Sampling

In the following section we introduce our new GPU algorithms for drawing samples from an alias table.

3.4.1 Baseline Method

The baseline sampling method directly follows the algorithm by Walker [63] (see Section 2.4). Preliminary experiments with constant table size N indicate that the throughput scales with the number of samples. The reason is that items that are sampled a second time might still be in the cache. The larger the fraction of the table that fits into the cache, the higher the maximum sampling throughput.

3.4.2 Cached Sectioned Sampling

To increase the number of cache hits, we use a similar idea as in Algorithm R (see Section 1.2.1). For uniform sampling, Algorithm R splits the items to be sampled into multiple sections. Because each thread then accesses more local memory areas, the method is more cache efficient. Like in Algorithm R, it is possible to determine the sections without communication by using a PRNG. Splitting an alias table is easier than splitting a generic weighted distribution because its rows are sampled uniformly. Our new cached sectioned sampling algorithm calculates a section and a number of samples to draw for each group. It then simply draws samples from that section, relying on the cache to improve sampling throughput. The size of the sections serves as a tuning parameter between the number of sections to calculate and the cache hit probability.

3.4.3 Cached Limited Sectioned Sampling

Even if the whole section would theoretically fit into the cache, the cached sectioned sampling method only achieves a small increase in throughput. This is due to multiple groups being scheduled to each SM and therefore evicting each other's cache entries. Our new cached *limited* sectioned method allocates (but does not use) so much shared memory that only one single group can be executed on each

SM. Like the cached sectioned method, the limited sectioned method allows for using the section size as a tuning parameter.

3.4.4 Shared Memory Sectioned Sampling

Our new shared memory sampling algorithm explicitly copies each group's section to the fast shared memory in an interleaved fashion. Each group then only uses the shared memory for actual sampling. The section size is limited by the size of the shared memory, so it cannot be used as a tuning parameter. The method can achieve higher peak speeds because it can sample with multiple groups per SM but it has a higher startup overhead.

Chapter 4

Implementation

The following chapter explains how we implement the alias table construction and sampling algorithms that are introduced in Chapter 3.

4.1 Class Structure

To provide a broad overview of the code structure, in the following section we introduce the class structure.

Construction Our base class `SamplingAlgorithm` handles weight generation and provides a common interface. This allows for developing other sampling algorithms. To have a comparison with a completely different method than alias tables, we implement the rejection method of Bringmann and Larsen [14]. The corresponding section of the class diagram is displayed in Figure 4.1.

Alias Tables The `AliasTable` class is parent to all alias table construction methods on CPU and GPU. The class defines the table itself, as well as its validation. Because the different CPU methods differ significantly, they all are modeled as individual classes (see Figure 4.2). The GPU implementations, which are the core contribution of this thesis, are based on the PSA method [27] and therefore all share a similar code structure. They can be modeled as a single base class `AliasTableSplitGpu` that calls the different split and pack methods.

Split Methods Storing a kernel in a class is not supported by the `nvcc` compiler, so we model each split method as a C++ namespace. The class diagram still visualizes the namespaces as classes because they are conceptually used as such. The basic split method can be used from the CPU but all other methods are so

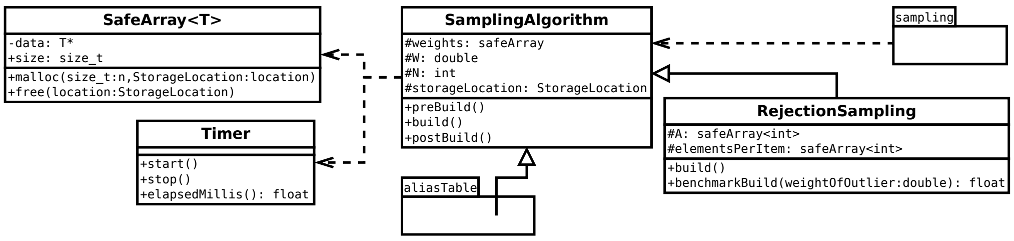


Figure 4.1: Diagram of general classes related to construction of sampling data structures.

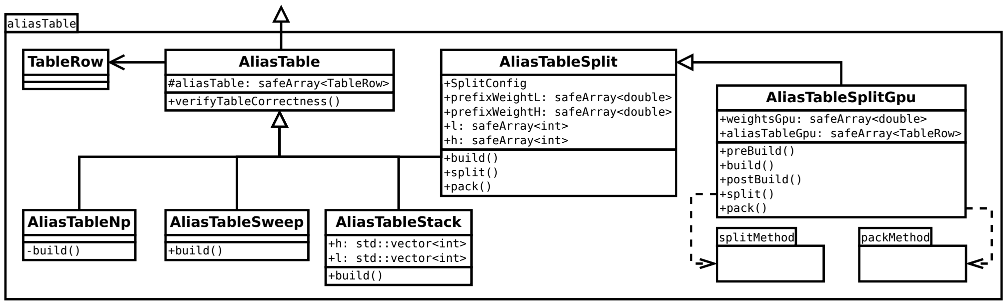


Figure 4.2: Diagram of alias table construction classes. The CPU based methods are modeled as individual classes.

closely tied to the GPU architecture that they only contain optimized GPU code. The different split methods are displayed in Figure 4.3.

Pack Methods Similarly to the split methods, we develop different pack methods, each in its own namespace. Each pack method offers a GPU kernel and internal helper functions. Our methods provide a `pack` method that is easy to read for understanding the basic concept. The `packOptimized` method is optimized for performance and therefore harder to read. The class structure of the pack methods is displayed in Figure 4.4.

Sampling We develop and optimize different techniques to perform the actual sampling, all modeled as individual classes. The special method `SamplerExpected` returns the multiplication of each item with its probability. It can be used to compare the actual random samplers with their expected values. The classes are displayed in Figure 4.5.

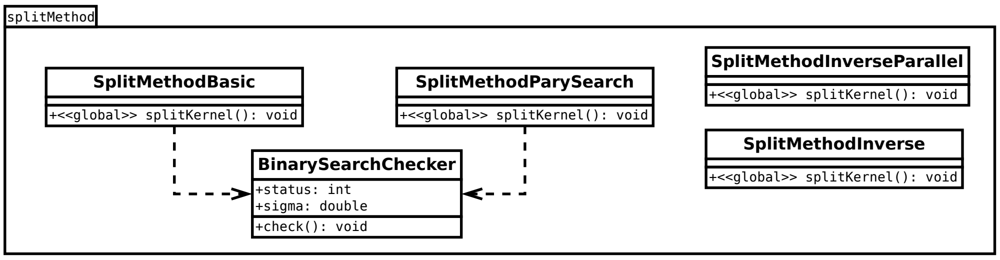

splitMethod
SplitMethodBasic
+<<global>> splitKernel(): void
SplitMethodParySearch
+<<global>> splitKernel(): void
SplitMethodInverseParallel
+<<global>> splitKernel(): void
SplitMethodInverse
+<<global>> splitKernel(): void
BinarySearchChecker
+status: int
+sigma: double
+check(): void

Figure 4.3: Diagram of different split methods for alias table construction.

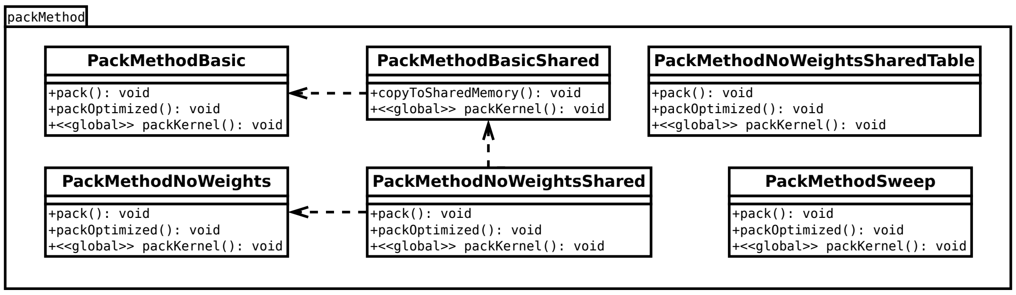

packMethod
PackMethodBasic
+pack(): void
+packOptimized(): void
+<<global>> packKernel(): void
PackMethodBasicShared
+copyToSharedMemory(): void
+<<global>> packKernel(): void
PackMethodNoWeightsSharedTable
+pack(): void
+packOptimized(): void
+<<global>> packKernel(): void
PackMethodNoWeights
+pack(): void
+packOptimized(): void
+<<global>> packKernel(): void
PackMethodNoWeightsShared
+pack(): void
+packOptimized(): void
+<<global>> packKernel(): void
PackMethodSweep
+pack(): void
+packOptimized(): void
+<<global>> packKernel(): void

Figure 4.4: Diagram of different pack methods for alias table construction.

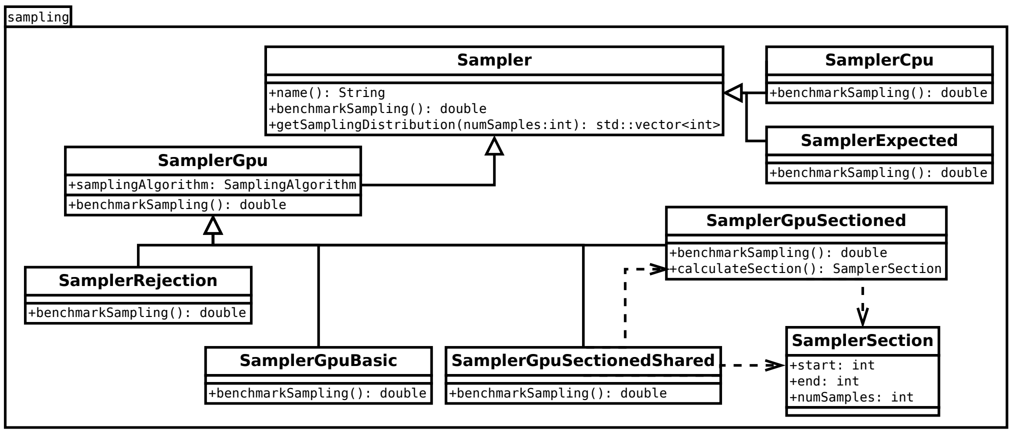

sampling
Sampler
+name(): String
+benchmarkSampling(): double
+getSamplingDistribution(numSamples:int): std::vector<int>
SamplerCpu
+benchmarkSampling(): double
SamplerExpected
+benchmarkSampling(): double
SamplerGpu
+samplingAlgorithm: SamplingAlgorithm
+benchmarkSampling(): double
SamplerGpuSectioned
+benchmarkSampling(): double
+calculateSection(): SamplerSection
SamplerRejection
+benchmarkSampling(): double
SamplerSection
+start: int
+end: int
+numSamples: int
SamplerGpuBasic
+benchmarkSampling(): double
SamplerGpuSectionedShared
+benchmarkSampling(): double

Figure 4.5: Diagram of different methods to perform weighted sampling.

4.2 Preliminaries

In order to develop test cases, we implement different alias table construction methods on the CPU. For an explanation of the algorithms, see Chapter 2. We also implement different prefix sum algorithms on the GPU.

Vose's Construction on the CPU The construction by Vose [62] is based on stacks. The most important difference of our implementation to the paper is floating point inaccuracies. If there are no light items left and the last remaining heavy item is slightly heavier than W/N, the algorithm tries to retrieve a light item that does not exist. To circumvent the problem, we handle remaining items separately as soon as one of the queues runs out of items, like in the implementation by Hübschle-Schneider and Sanders [27].

Sweeping Construction on the CPU The sweeping alias table construction [27] can be used to construct alias tables on single-threaded CPUs without a need for additional data structures. The method is the fastest of our single-threaded CPU implementations, even though in Ref. [25], the stack construction is the fastest. Because our focus lays on the GPU, we do not further investigate that behavior.

Splitting Construction on the CPU For the PSA construction [27], our CPU implementation generates all the necessary data structures for the parallel case but then executes the actual steps sequentially. It is easier to debug than GPU-only solutions and, because CUDA allows for sharing code between CPU and GPU, the implementation can be re-used on the GPU.

Prefix Sum Prefix sums play a major role in the PSA method. We therefore implement and compare the basic and the work-efficient method on the GPU. The methods are explained in Section 1.4. We also develop an in-place variant of the work-efficient prefix sum algorithm for arbitrary array lengths. This is achieved by calculating the prefix sum in batches of $\mathit{blockSize}^k$. Nvidia's `cub` library ships with a prefix sum implementation that uses the work-efficient algorithm. The developers pay special attention that the memory access is aligned with the hardware banks [2]. For tuning parameters like the block size, the developers implement case distinctions for different hardware architectures. The `cub` library is faster than our own implementation by an order of magnitude. For constructing alias tables, we therefore only use `cub`'s prefix sums.

4.3 Partitioning of Light and Heavy Items

The split and pack methods both need an array of references to the positions of light items `l` and an array of references to the position of heavy items `h`. The following section explains our considerations for that process.

Classification Kernel As a baseline method, we classify items using a dedicated kernel. The kernel checks the weight of each item and then either writes 0 (light item) or 1 (heavy item) to a temporary array. After calculating a prefix sum P of the temporary array, we use it to determine the locations of items in the `l` and `h` arrays. Because each item is either light or heavy, a single prefix sum is enough for both arrays. A heavy item i is written to `h[`P_i`]` and a light item to `l[`$i - P_i$`]`.

On-demand Classification Instead of explicitly calculating a classification array, we use a modified iterator to classify items on-the-fly during the prefix sum calculation. Additionally, we pre-calculate the division W/N on the CPU instead of in every thread on the GPU. In total, we achieve a speedup of 2.5 over the baseline method.

Cub::DevicePartition We achieve an additional speedup of 1.7 by using the `cub` library's partition function. We combine the ideas of the paragraphs above by reading the weights array and classifying items on-demand during partitioning, as well as using a pre-calculated W/N value. The library method inverts the second partition, so the sentinel elements (see Ref. [27]) need to be handled separately.

4.4 Split Method

In the following section we describe implementation details of our new split algorithms. The algorithms themselves are introduced in Section 3.1. We continue to denote the number of splits to be calculated with the variable s.

4.4.1 Baseline Method

As a baseline, we transfer the original split algorithm from Ref. [27] to the GPU. For that, we start one GPU thread for each split, like it is described in the original paper. For a small number of splits, starting s groups of size 1 is fastest because it allows better load balancing between multiple SMs. When using an optimized packing method with a large number of splits (introduced in Section 3.2.3), the method can profit from a large group size because the first memory accesses during binary search are the same and can be shared between all threads.

Algorithm 4.1: Branched memory reads

```
int x;
if (condition)
    x = array[a];
else
    x = array[b];
```

Algorithm 4.2: Optimized branched memory reads

```
int *xPtr;
if (condition)
    xPtr = array + a;
else
    xPtr = array + b;
int x = *xPtr;
```

4.4.2 Partial p-ary Split

The partial p-ary split method is specifically designed to be efficient on GPUs, so no major changes to the pseudocode in Section 3.1 are needed. Each thread stores the check results in an array that is located in the shared memory. Looking through those check results for determining the new boundaries a and b is then performed in parallel. Preliminary experiments show that a good threshold to switch to binary search is when the search range of p-ary search cannot be reduced to less than $p/8$.

4.5 Pack Method

The following section describes implementation considerations for the pack algorithms that are introduced in Section 3.2.

4.5.1 Baseline Method

For the baseline method, we build upon the CPU implementation. The original method as it is introduced by Hübschle-Schneider and Sanders [27] performs memory access in both of the `if` branches. Our optimized version only performs pointer arithmetics in branches. The actual memory operation is then done for both branches at once. Algorithms 4.1 and 4.2 illustrate the idea of moving memory reads. The optimization makes the method up to two times faster.

4.5.2 `l` and `h` in Shared Memory

For the shared memory method, we start groups with 512 threads. Those threads cooperate to copy the interesting sections of the `l` and `h` arrays to the shared memory in an interleaved way. After that, most of the threads terminate, which allows the GPU to free up resources. Only the threads of one single warp then perform the actual packing. The shared memory size per group is limited. To make it possible to copy the section's data to the shared memory, the split method needs to generate significantly more sections than when using the baseline method. We query the hardware parameters of the GPU to decide how many splits are needed to still fit the arrays into the shared memory.

4.5.3 Weight in `l` and `h` Arrays

By using C++ macros, we make it possible to switch between `l` and `h` with and without weights. Building upon the shared memory method, the weights are written during partitioning and then copied to the shared memory for packing.

4.5.4 Chunked loading

The chunked pack method reduces the number of splits required by loading chunks of the `l` and `h` arrays to shared memory on demand. We copy data from the global memory when either the `l` or `h` chunk is used up. If the other chunk is used up by more than 50%, we also load more data for that one. This avoids threads running out of data early. In preliminary experiments, we find the optimal number of splits to be around $10 \cdot |\mathit{Threads}|$. Because we copy new chunks even if the items are not used up, some leftover items are overwritten. To reduce the number of global memory read operations, a modification of the algorithm re-uses the already loaded items and moves them to the beginning of the shared memory region. The performance of the modification is worse than the original chunked method because it requires more synchronization and adds more branches that are not entered by all threads. We therefore do not reuse items and simply load the whole chunk from global memory. Because the memory banks can be accessed in parallel, this is not a big problem. Using asynchronous copy operations is not covered here but looks promising with preliminary tests showing a 5% speedup.

4.5.5 Uncompetitive methods

In the following section, we present our implementations and optimizations of pack methods that are uncompetitive.

Alias Table in Shared Memory A problem of storing the final alias table sections in shared memory is that threads of multiple groups can write to overlapping sections of the alias table. Additionally, depending on the distribution of weights, the memory range that is written to can be too large for the shared memory. Our implementation for the method is only a proof-of-concept that assumes that the table can be packed without any overlap and is still slower than the baseline.

Pre-filled Alias Table The alias table's rows are stored as structs with weight, alias and padding, while the input weights are a tightly packed array. The fastest method to copy the input weights to the table is to copy them to the GPU normally and then use a kernel that fills the table. An alternative, a `memcopy2D` operation with spread, is significantly slower. This applies to both copying from the host to the device and copying from the device to the device.

Sweeping Pack Method On GPUs, even when a loop is only executed on one thread, all other threads of the same warp have to wait. This makes the sweeping pack method 4.5 times slower than the baseline pack method. We also develop an optimized version where the loop is shared for both of the `if` branches. That method still is 3.7 times slower than the baseline method.

Reordering `l` and `h` Arrays For reordering, we use a number of `memcpy` calls (device to device). We perform preliminary experiments with different arrangements, for example round-robin or random distribution. The performance of the reordered pack method is, depending on the weight distribution, between 1.04 times and 2.2 times slower than the baseline method.

Precomputed `weight[l[i]]` For constructing the precomputed array, we start a kernel that loads items from the `l` and `h` arrays in an interleaved way. The simple kernel directly writes to a new array.

4.6 PSA+

The PSA+ algorithm (see Section 2.4.5) simultaneously speeds up every step of the PSA algorithm because it reduces the number of items that need to be taken care of. It greedily packs parts of the alias table without building the full data structures like prefix sums. The remaining items that cannot be packed greedily are then handled with the normal PSA algorithm.

Custom Partition Our PSA implementation partitions items from the input distribution to the `l` and `h` arrays using the `cub` library. For PSA+, we implement our own partitioning step to be able to directly pack the items there. The algorithm reads the weights array in an interleaved fashion and determines if items are light or heavy. It then uses a group-local atomic variable to determine indices for storing items and weights in a shared memory array. After that, the items are written back to the global memory in an interleaved fashion. The performance of our re-implementation is similar to the method from the `cub` library.

Pack while Partitioning Before writing the `l` and `h` arrays back to the global memory, we can now perform greedy packing. We do that by assigning each thread the same number of light and heavy items. After packing greedily, the progress of the individual threads is stored. All threads of a group then work together to only copy items back to the global memory that are not handled yet. A complication of PSA+ is that the greedy packing step can leave some items partially packed, meaning that some of a heavy item's excess weight is already distributed to light items. Using a competitive pack method, the items in the `l` and `h` arrays are saved together with their weight (see Section 3.2.3). For the last remaining heavy item, our PSA+ algorithm updates the item's weight, re-classifies it and then write it to either the light or the heavy array. This avoids the need for an additional data structure for partially packed items.

4.7 Sampling

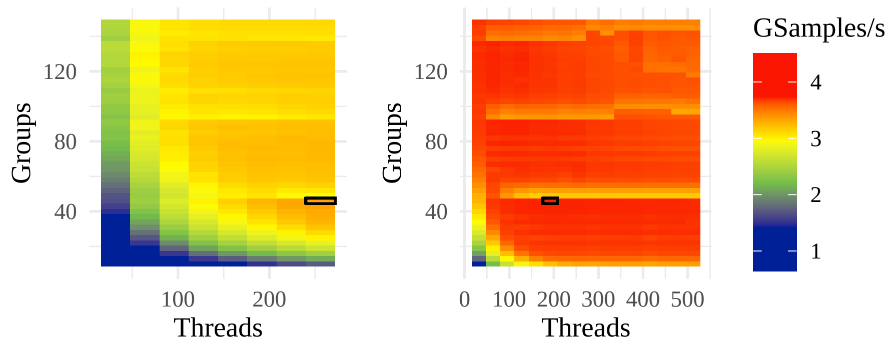
In the following section we describe precautions to be taken when implementing our new sampling algorithms.

4.7.1 Random Number Generator

For sampling, a random number generator is the most important requirement. Some generator algorithms are presented in Section 1.1. To be able to compare sampling throughput between multiple generators, we use C++ templates that allow for exchanging the generator implementation. We implement templates for the xorwow and Mersenne Twister generators from Nvidia's `curand` library.

Mersenne Twister internally uses synchronization, so either all threads of a group must draw a random number from the same state or none. While this is not a problem for the baseline sampling method, it causes problems with determining the sections for sectioned sampling (see Section 3.4.2). To split up the table, we repeatedly need to set a seed during kernel runtime. We therefore cannot use the Mersenne Twister generator for sectioned sampling.

Preliminary experiments show that plain xorwow (without an alias table) can sample with around 26 GSamples/s and Mersenne Twister can sample with around 25 GSamples/s. These numbers can therefore be seen as an upper bound for alias table sampling. We execute our final benchmarks with the faster and more flexible xorwow generator.



(a) Mersenne Twister

(b) Xorwow

Figure 4.6: Sampling throughput of the baseline method, depending on the number of groups and the number of threads per group of the CUDA grid. Uniform random weight distribution with a table size of 10^6 items.

4.7.2 Baseline Method

The baseline method follows the algorithm described by Walker [63]. Each thread draws samples independently of other threads. On CPUs, reducing the number of branches (and therefore jumps) is vital. Contrarily, our tests for sampling on GPUs show that this is less important. Memory access, which is the bottleneck of the method, is only performed in one of the branches. To make sure that the compiler does not optimize away drawing the samples, we sum up the samples (modulo 2) in a local variable.

Group size For determining the grid size of the kernel, we perform preliminary experiments. Figure 4.6 shows the influence of the number of groups and number of threads on sampling throughput with the baseline method. We observe peaks for group sizes that are multiples of the number of SMs in the GPU. The xorwow generator reaches its maximum throughput with 46 blocks and 192 threads each. The Mersenne Twister generator reaches its maximum throughput with 46 blocks and 256 threads each.[1] For Mersenne Twister, the number of threads per group is limited to at most 256. For the evaluations in Chapter 5, we therefore use above grid sizes.

[1]The measurements are different to the GTX 1650 Super, where the best group size differs more between the PRNGs. There, the maximum for xorwow is achieved at 19 blocks and 256 threads each. The maximum for Mersenne Twister is achieved at 136 blocks and 32 threads each.

One-PRNG-per-kernel-call-per-thread Xorwow needs to generate PRNG subsequences in each thread, which has more startup overhead than using independent seeds. The One-PRNG-per-kernel-call-per-thread (pK-pT) scheme (see Section 1.1.2) allows for skipping the generation of subsequences. We implement the scheme using `SHA1` and `MD5` hashes for seed generation. Preliminary experiments show that this yields a throughput improvement of 0.2% for drawing from alias tables. Because the improvement is not significant but increases the complexity of random number generation, we do not develop the approach further.

4.7.3 Sectioned Sampling Methods

Because each CUDA group runs on a single SM, all its threads share the same L1 cache. We therefore split the alias table by group, not by thread. For determining the sections, we start with the whole alias table as one large section and then recursively split it in half. We then randomly determine how many samples need to be drawn from each half. While an implementation of the binomial distribution is available in the C++ standard libraries, it is not available in Nvidia's `curand` library. For $N > 30$, the normal distribution is a good approximation of the binomial distribution [37] and computationally much easier to evaluate. As $N \gg 30$ always holds in our case, we use normally distributed random deviates. We choose the size of the sections so that they completely fit into the L1 cache. The RTX 2080 uses 96 kB of L1 cache [1], which is enough for 6144 rows. The number of groups must therefore be large (well over 1000 even for 10^7 items) to achieve optimal cache utilization.

4.8 Performance Optimization

In the following section we describe general performance improvements that do not change the algorithms conceptually. Because the changes are responsible for significant speedups, we still document them here.

4.8.1 Managed Memory Allocations

Using manual instead of managed memory transfer is key to consistently achieve good performance. This is because optimizations in one kernel's memory access can negatively affect other kernel's running time when using managed allocations. These allow for direct sharing of memory address regions between CPU and GPU with automatic transfers when accessing memory on the other device. This leads to unpredictable behavior. Using explicit memory transfers alleviates this at the cost of increased design complexity.

Algorithm 4.3: Memory access without compiler aids

```
TableRow row;
rows[index] = row;

STG.E.64.SYS [R4] R52
STG.E.64.SYS [R4+0x8] R2
```

Algorithm 4.4: Memory access with compiler aids

```
TableRow row;
*reinterpret_cast<int4*>(&rows[index])
    = *reinterpret_cast<int4*>(&row);

STG.E.128.SYS [R12] R4
```

4.8.2 Compiler Aids for Memory Access

Preliminary versions[2] of the pack method write too much data to the L2 cache. A reason for this excessive amount can be found when inspecting the generated assembler code using Nvidia NSight Compute. Assigning the alias table rows is not fully optimized by the compiler and generates two 64 bit store operations instead of a single 128 bit store operation. Using two 64 bit transactions is inefficient because the GPU's memory uses transactions of 32 bytes. When only storing 64 bit without coalescing, 75% of the bandwidth is wasted. By casting the `TableRow` struct to an `int4`, we trick the compiler into using one single 128 bit store operation instead. Casting only works because the alias table array is properly aligned in the memory, which is ensured when allocating. Algorithms 4.3 and 4.4 give an example for the C code we use and the assembler code that is generated from it.

Memory transfer measurements

To determine the effect of the compiler aid, we provide preliminary measurements of the memory transfer volume in the paragraphs below. For construction, we use the p-ary split method and the shared memory pack method with $N = 5 \cdot 10^7$ items and $s = 580.000$ splits. The measurements are performed on a GTX 1650 Super.

Split Method The split method theoretically writes $s \cdot 16$ B $= 9$ MB to the global memory. Our measurements show a transfer of 13 MB (18 MB to the L2 cache). The split method's global memory read operations are performed by the search operation. In each iteration, it reads 4 `double` values (prefix sum of light and heavy items in two positions). Each of the s threads roughly need $\mathcal{O}(\log(|\mathtt{h}|))$ steps, so an estimate for the theoretical read volume is $\log_2(0.5 \cdot N) \cdot s \cdot 4 \cdot 8$ B $+ s \cdot 8$ B ≈ 434 MB. The actual amount that is read from the global memory is 1050 MB (620 MB to the L2 cache). The difference can be explained with

[2]Measurements of versions without the optimization are not included in this document.

the fact that the GPU uses memory transactions of 32 bytes, so large parts of the transactions are not actually used. The cache hit rate is rather small (4% L1, 15% L2) but that is expected for a search operation that only accesses items once. While the compiler aid can be applied to reading the `l` and `h` array's items, the effect is insignificant.

Pack Method The pack method theoretically needs to write the alias table once, resulting in $N \cdot 16$ B $= 762$ MB of global memory writes. Our measurements show total writes to the global memory of 849 MB. The amount of memory written to the L2 cache in our preliminary versions[3] is 3075 MB and therefore significantly higher. Using the compiler aids, we are able to reduce this to 1490 MB, which also makes the pack operation nearly 50% faster. This is still double the expected value. The reason again are the 32 byte transactions of the GPU. Because threads of the same warp write to unrelated memory locations of the alias table, the operations cannot be coalesced and half of the transactions' data is wasted. The pack method reads 9 MB of splits and the complete `l` and `h` arrays with weights, resulting in $N \cdot 16$ B $= 762$ MB from global memory. The measured amount from the global memory is 778 MB and the amount from the L2 cache is 774 MB. The cache hit rate (66% L1, 67% L2) is significantly higher than with the split method. The compiler aids for memory access reduce the SOL value[4] of the pack method from 61% down to only 37%. Even though the SOL value is decreased in this case, the performance gets better. The reason is that we write the same data but with more efficient operations.

Sampling When using the compiler aids, the baseline xorwow sampling method gets up to 2 times faster and the cached sectioned method 30% faster. The sectioned shared method is not influenced by the change because most of its memory operations are served from shared memory. Drawing 10^9 samples from an alias table of size 10^6, the baseline sampling method reads 79 GB of data from the global memory. The cached sectioned method reduces that to 77 GB. The shared memory method reduces the data transfer by two orders of magnitude down to 340 MB. The raw table size is $10^6 \cdot 16$ B = 15 MB. The rest of the memory operations are caused by the PRNG initialization.

[3]Measurements of versions without the optimization are not included in this document.

[4]The speed of light value compares the achieved memory throughput with the theoretical maximum. Usually, a larger SOL value is better.

4.8.3 Multiple Streams

A common performance improvement that is suggested in Nvidia's best practices guide [3] is to use multiple streams. Streams allow for executing different kernels at the same time instead of serializing them.

Prefix sums Using the competitive alias table construction methods, only the prefix sums can be executed in parallel. Profiling an implementation with multiple streams shows that the GPU still executes the prefix sums nearly serialized. The time improvement is measurable but below 1%. Because the support for multiple streams complicates the code without a real benefit, we do not use multiple streams for prefix sums in our implementation.

Interleaved Split and Pack Nvidia's best practices guide [3] also suggests to use parallelism by performing work in an interleaved fashion. When implementing an algorithm that consists of two tasks, the usual pattern is to first process all items with one task and then all with the other task. In some cases, it can be more efficient to run the first task with a few items at a time and then already run the second task in parallel. When using our shared memory pack method, a large number of splits needs to be calculated. We implement additional parallelism by changing the split and pack methods to only look at a small number of splits at a time. After 10% of the splits are done, we record an event which then triggers the pack step of that section to run on another stream. Preliminary experiments show that even though the method actually executes in parallel, it does not make the total construction time faster.

4.8.4 GPU Architecture Parameters

Hard-coded values like block dimensions or actual worker threads have an effect on the performance. Optimizing such parameters is an ongoing research topic [19]. We choose a simple method to optimize our parameters using a Python script. Iteratively, the script modifies one of the parameters at random. It then recompiles[5] and tests the new version. If the new parameters lead to better performance than the previous minimum, it updates the parameters. If the performance is worse, it keeps the old ones. Following the idea of simulated annealing, the script slowly decreases the step size.

Our optimizer script is able to achieve significant speedups by fine-tuning only a few parameters. Table 4.1 shows measurements using 10^7 items with the shared

[5]Making the values configurable at run-time decreases the performance because the values need to be passed to every kernel.

Distribution	SMW[6]	Group size				Time
		p-ary	Split	Pack	PSA+	
Power law[7]	64	512	512	512	–	3.87 ms
	32	547	523	552	–	3.46 ms
Uniform weights	64	512	512	512	–	4.90 ms
	24	519	510	551	–	3.91 ms
Uniform PSA+	64	512	512	512	512	3.33 ms
	11	532	512	490	512	2.75 ms

Table 4.1: Results of the GPU architecture parameter optimization.

memory pack method and partial p-ary split. To our surprise, often values that are not multiples of the warp size work best. The number of worker threads that actually pack items (in contrast to just help with copying) has by far the most impact. Unfortunately, the best parameters depend on the input distribution. While it is possible to achieve a speedup of up to 1.25 for uniform random weights by fine-tuning the parameters, other distributions get slower when using that configuration. The parameters therefore need to be optimized for the actual use-case. Repeating the optimization process of the uniform weight distribution multiple times yields similar parameter values.

4.8.5 Cache configuration

Because L1 cache and shared memory are located in the same hardware area, CUDA allows for expressing a preference for the cache configuration. Developers can choose to dedicate a higher portion of the available storage to the L1 cache or to the shared memory. The limited sectioned method is the only one that is influenced significantly, so we use a manual cache configuration for its kernel.

4.9 Verification

In the following section we describe different checks and how they help us to ensure that table construction is correct and sampling is unbiased.

Manual Comparison For a manual test, we assign a weight distribution that is easy to recognize and check for humans. We then generate the alias table, sample from it on the CPU and plot the resulting distribution. This provides a

[6]Number of shared memory worker threads that perform the actual packing.
[7]Shuffled power law distribution with exponent=1.

first, quick look at the behavior of the generated table. Many of the problems during development can be debugged by visualizing the sampled distribution, for example when experiencing sections that are skipped during construction.

Summing up the Table Entries In order to automatically verify that an alias table is correct, we sum up the weights of items and aliases. We then compare the result with the input distribution. This allows for ensuring table validity without manual comparisons. These unit tests are executed automatically every time a table is constructed in debug mode. Additionally, we test the construction with various weight distributions on every program run.

Sampling Distribution To test the sampling step on the GPU, we pass an array to the sampling method and count the actual sampled items. To avoid a performance impact, the test is removed in release builds using macros. For an automated unit test, we verify that the maximum difference between expected and sampled distribution is small enough. A proper statistical Kolmogorov-Smirnov test [38] on the sampled and expected distribution results in p-values of 0.9959 (D=0.026, 10^6 samples), 0.6654 (D=0.046, 10^5 samples) and 0.1725 (D=0.07, 10^4 samples). We can therefore not assume that the distributions are different.

Chapter 5

Evaluation

In the following chapter we perform measurements comparing our newly developed methods among each other, as well as with the parallel CPU implementation by Hübschle-Schneider and Sanders [27].

5.1 Experimental Setup

For the evaluation, we use multiple different machines, including standard consumer hardware. The machines are specified in the following paragraphs. Unless otherwise noted, our measurements are performed on machine A, which is consistently 2 times faster than on Machine B. We therefore only provide measurements on Machine B for the rare cases where the machines exhibit different behavior. The GPU code is compiled using `nvcc` on CUDA 11.1. The measurements are performed by using GPU timers, which have a negligible impact on the performance.

Machine A (RTX) The GPU that we use for most measurements is a Nvidia GeForce RTX 2080, which has 2944 CUDA cores and 8 GB of global memory with up to 448 GBps of memory bandwidth [1]. The driver version is 455.45. The host features an Intel Core i5-750 processor with 4 cores and 4 threads, as well as 2 memory channels [28]. The operating system we use is Ubuntu 16.04.

Machine B (GTX) For comparison, we also perform measurements on a machine that is equipped with a cheaper Nvidia GeForce GTX 1650 Super. The card has 1280 CUDA cores and 4 GB of global memory with up to 192 GBps of memory bandwidth [45]. The driver version is 450.66. The host features an Intel Xeon 1230-v3 processor with 4 cores and 8 threads, as well as 2 memory channels [29]. The operating system we use is an up-to-date Arch Linux as of 2020-10-07.

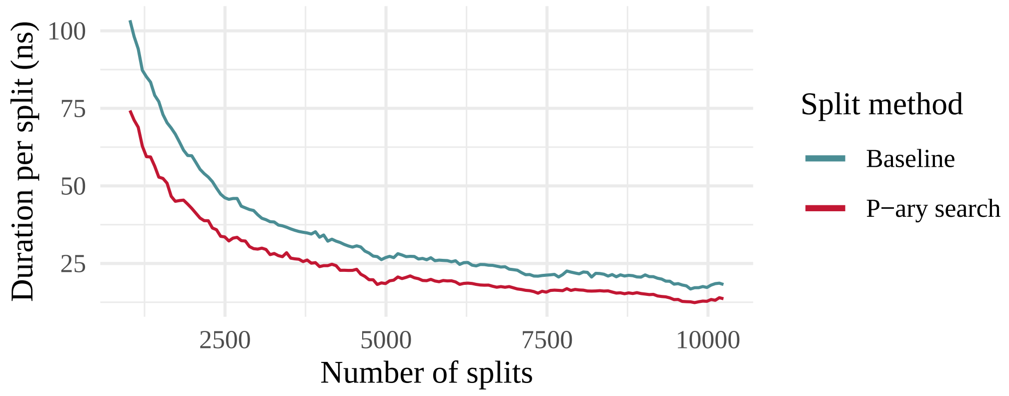


Figure 5.1: Time needed for determining a single split using different split algorithms. Using 10^7 input items with uniform random weights.

Machine C (Intel) For providing comparisons with a powerful CPU, we use a machine with four Intel Xeon Gold 6138 CPUs (4×20 cores with 160 hyper-threads) and 256 GB of DDR4 RAM [25]. The machine does not feature a GPU and runs Ubuntu 20.04. The code is compiled with g++ 9.3.0.

Machine D (AMD) Another machine with a powerful CPU that we measure on has a single-socket AMD EPYC 7551P CPU (1×32 cores with 64 hyper-threads) and 256 GB of DDR4 RAM [25]. The machine does not feature a GPU and runs Ubuntu 20.04. The code is compiled with g++ 9.3.0.

5.2 Construction

In the following section we evaluate the performance of alias table construction on the GPU using our new algorithms.

5.2.1 Splitting

A comparison of our split methods is plotted in Figure 5.1. On the x-axis, the figure shows s, the number of splits. The inverse methods are omitted as they are entirely uncompetitive. Independently of the number of splits, the partial p-ary split method is up to 1.5 times faster than the baseline method. This reduces the running time of the overall shared memory table construction by up to 15%.

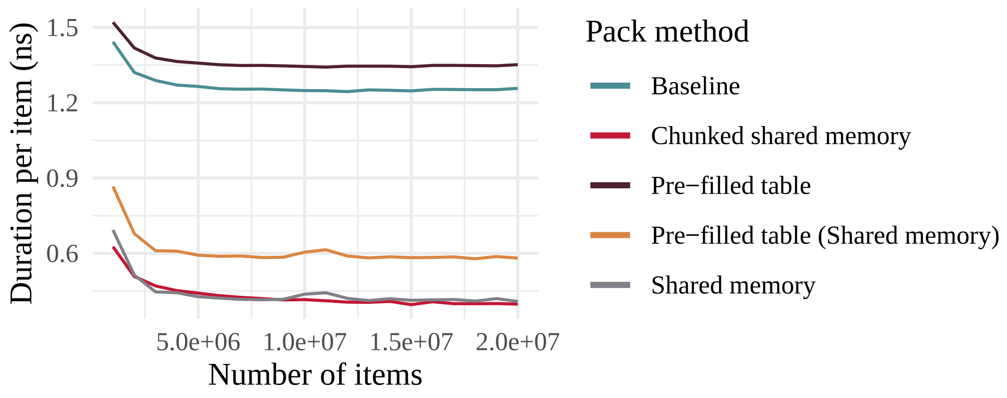


Figure 5.2: Construction duration for a single item using different pack algorithms. The time includes both splitting and packing. Items use uniform random weight.

Method	Duration per item
Sweeping	2.41 ns
Precomputed `weight[l[i]]`	1.55 ns
Re-ordering `l` and `h`	1.58 ns

Table 5.1: Construction duration for a single item using uncompetitive methods. The input is 10^7 items with uniform random weights.

5.2.2 Packing

In Figure 5.2 we compare the construction speed between our pack methods. Because the pack method has an influence on the number of splits to calculate, the figure shows the full construction time including p-ary split and the respective pack method. The shared memory methods exhibit a clear improvement over the methods that access the global memory directly. Uncompetitive methods are omitted from the plot for better readability but are listed in Table 5.1.

`l` and `h` in Shared Memory The shared memory pack method achieves a performance improvement over the baseline method even if it requires to spend more time in the splitting step. Because of the better alignment, the method reduces the total amount of data that is read from the L2 cache by 57%.

Weight in `l` and `h` Arrays The pack method with weights in `l` and `h` achieves a speedup of 15% in comparison to the shared memory method without weights. The pack step gets 2 times faster while the split and partition steps get 2 times slower. The reason for that is that including the weights increases the size of the elements of `l` and `h` items from 8 bytes to 12 bytes (+4 padding). This means that the shared memory can hold fewer items and therefore more splits need to be calculated. The partition method gets slower because it needs to perform more memory write operations.

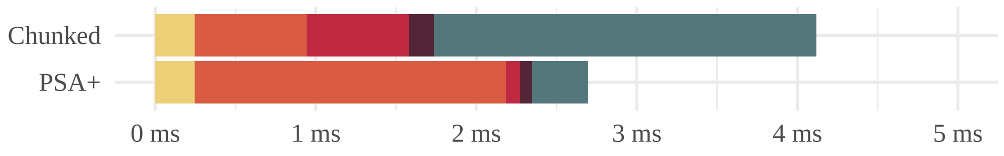
Chunked Method The chunked pack method itself is slower than the shared memory method because it uses memory access that can be coalesced not as well. At the same time, the method speeds up the splitting step significantly because the section size is no longer limited by the available shared memory. In total, the chunked pack method is slightly faster than the shared memory method for large N. On the GTX 1650 Super, this difference is a bit more visible. The chunked method is closer to the original idea of Hübschle-Schneider and Sanders [27] where the number of splits matches the number of executing threads.

5.2.3 PSA+

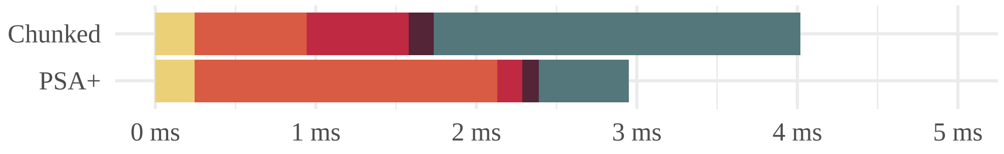
In cases where items have uniform random weights, PSA+ can greedily handle around 90% of the items. Therefore, the standard PSA method only needs to be executed on a fraction of the input data. A reason why Hübschle-Schneider and Sanders' [27] algorithm can pack a higher fraction of the items greedily is that our section size for packing greedily is limited by the shared memory and therefore rather small. We introduce a threshold for the number of light and heavy items that are needed before greedy packing is attempted. This makes it possible to reduce the performance impact of distributions where greedy packing is not promising. Using uniform random weights with 10^7 items, PSA+ achieves a speedup of 1.5 to PSA and using a shuffled power law distribution with exponent 0.5, it achieves a speedup of 1.4. On the GTX 1650 Super, it achieves a speedup of up to 2.2. When using a higher value for the exponents or not shuffling the input, our PSA+ implementation gets slower. Figure 5.3 shows a profiling output comparing PSA and PSA+ with different weight distributions and $N = 10^7$ items. While PSA+ does not work for all distributions, it achieves significant speedups when supported.

5.2.4 Weight Distributions

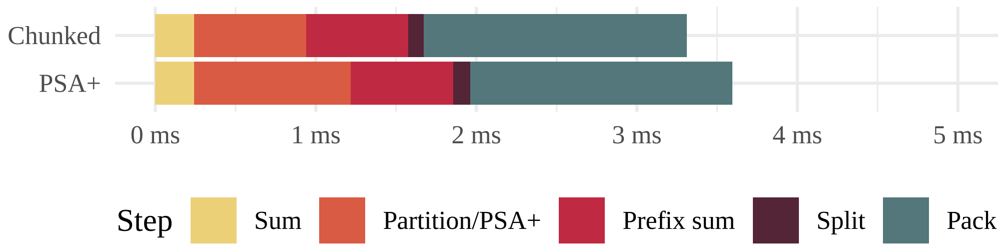
Construction speed is strongly influenced by the weight distribution. For uniform random weights, the split kernel reads up to 7 times more data from the device storage than for the power law distribution. The more balanced the number of light and heavy items, the more items need to be considered in the split step's search operation. Using the chunked pack method, the influence of the split method can



(a) Uniform weights

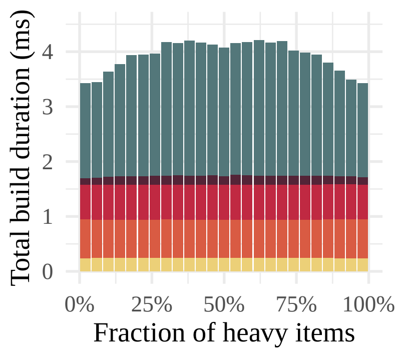


(b) Power law (exponent=0.5, shuffled)

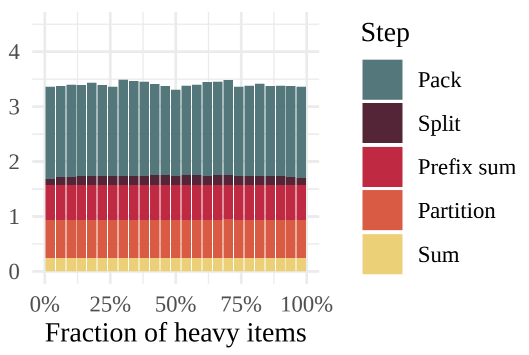


(c) Power law (exponent=1)

Figure 5.3: Construction duration of an alias table of size 10^7. Comparison of different weight distributions with either PSA or PSA+.



(a) Heavy items at random positions



(b) Heavy items sorted to the beginning

Figure 5.4: Construction duration of an alias table with 10^7 items and varying fraction of heavy items. Because we use the chunked shared method, the variance of the split duration is rather small.

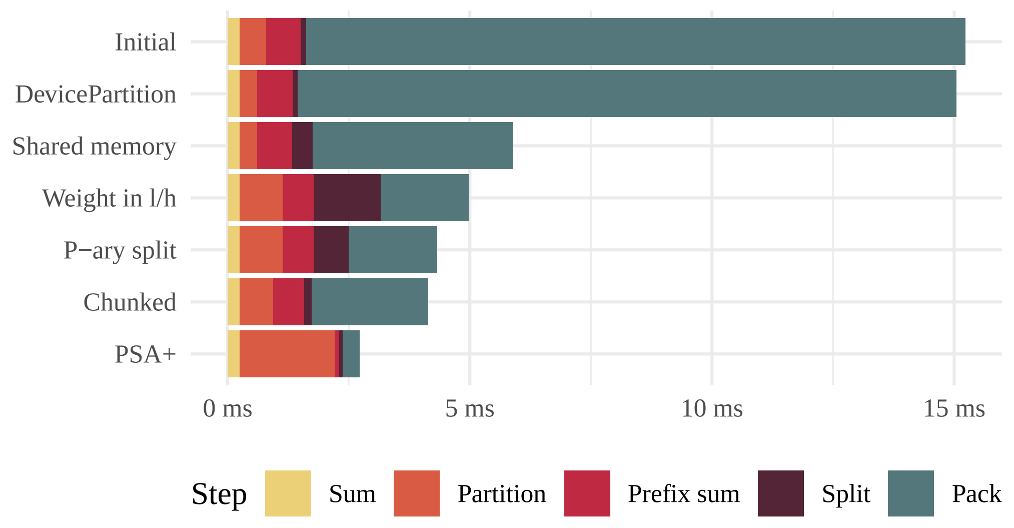


Figure 5.5: Construction duration of different methods that are developed in this thesis. Measurements taken from a table of size 10^7 with uniform random weights.

be mostly eliminated. The pack step is dependent on the fraction of heavy items only if the items are located at random positions throughout the input weights. Figure 5.4 shows measurements with items of weight either 1 or 2, where the fraction of items with weight 2 is varied.

5.2.5 Progress during this Thesis

Over time, the performance of our algorithms improves significantly. Between our first implementations and the baseline, we achieve a speedup of 10 using memory access and group dimension optimizations. We then change the algorithms conceptually to improve the performance. Figure 5.5 shows the milestones of achieving an additional speedup of 4 to the baseline.

5.2.6 Comparison with Hübschle-Schneider and Sanders

As displayed in Table 5.2, our GPU-based chunked method achieves a speedup of 34 over the CPU-based PSA method [27].[1] Figure 5.6 shows that our method is faster even when the input weights are present on the host and the resulting alias table is transferred back to the host. In fact, our construction method is faster than the time it takes to transfer a constructed alias table to the GPU.

[1]For the speedup we compare the RTX 2080 card on machine A with the CPU of machine B. Machines A and B are both consumer devices but the CPU of machine B is faster.

Method	Machine	Build duration	
		10^7	10^8
Ref. [27]	C	–	83.1 ms
	D	–	151.5 ms
	B (CPU)	136.1 ms	1126.7 ms
Ours	A (RTX 2080)	4.0 ms	32.8 ms
	B (GTX 1650 Super)	8.1 ms	–[2]

Table 5.2: Construction duration comparison with different machines and Hübschle-Schneider's method [27]. Input are 10^7 and 10^8 items with a shuffled power law distribution.

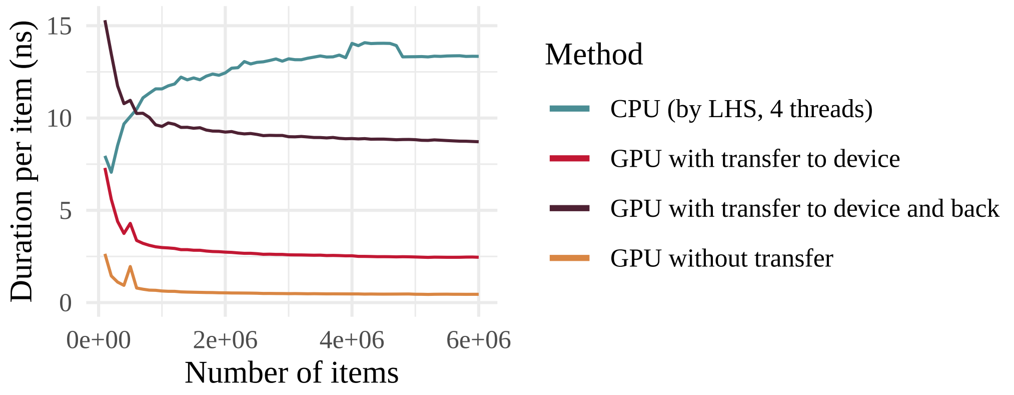


Figure 5.6: Construction duration of a single item when including or excluding memory transfers. Comparing uniform random weights with different table sizes.

5.2.7 Struct-of-Arrays

Unless otherwise noted, we use array-of-structs storage for the alias tables because Hübschle-Schneider and Sanders [27] describe it as faster on CPUs. We can confirm the CPU result on the GPU. Constructing an alias table that is stored as struct-of-arrays with the shared memory method is around 25% slower than array-of-structs. The array-of-structs method uses a struct with a content of 12 bytes, which is padded to 16 bytes by the compiler. When memory usage is a critical constraint, the struct-of-arrays method can have the advantage that its two arrays are each tightly packed.

[2]While the alias table itself can be stored on the GTX 1650 Super, its 4 GB of memory are not enough to store the temporary data structures for alias table construction.

5.3 Sampling

Figure 5.7 shows a comparison of the baseline sampling method and the three sectioned methods with $N = 10^6$. Figure 5.8 shows the same comparison when drawing samples from a table with 10^7 items. For drawing only a few samples, the baseline method is fastest. It does not need any preprocessing and quickly reaches its maximum speed. The shared memory methods have a significant overhead for determining the sections or copying data. If the number of samples drawn is increased, the investment pays out and the sectioned methods can generate up to 15 GSamples/s.

Figure 5.9 visualizes which method is best when varying both table size and number of samples. The sectioned methods are better when drawing many samples from a small table. While the shared memory method can achieve higher peak throughputs, the sectioned limited method is more generic and achieves a decent throughput in more cases.

5.3.1 Comparison with Hübschle-Schneider and Sanders

In Table 5.3 we compare the sampling throughput of our sectioned limited method with the CPU implementation of Hübschle-Schneider and Sanders [27]. We assign shuffled power law distributed weights to each item. For sampling, the GPU has a clear advantage because its memory can handle random access operations much more efficiently. On the tested consumer machines, our parallel GPU method has up to 56 times more throughput than Hübschle-Schneider's method.[3] Even on the powerful machine C, they achieve no more than 2 GSamples/s, which we can easily outperform using consumer hardware.

5.3.2 Space-efficient rejection sampling

Bringmann's space-efficient rejection sampling algorithm [14] (see Section 2.8) first builds a data structure in linear time and then samples from that in expected constant time. Our implementation shows a similar construction speed as alias tables. Sampling is significantly slower, having a throughput of up to 0.9 GSamples/s. This is caused by the fact that if one thread needs to reject a sample and draw again, the other threads in the same warp have to wait. The method is not a good fit for GPUs and sectioned sampling cannot be applied.

[3]For the speedup we compare the RTX 2080 card on machine A with the CPU of machine B. Machines A and B are both consumer devices but the CPU of machine B is faster.

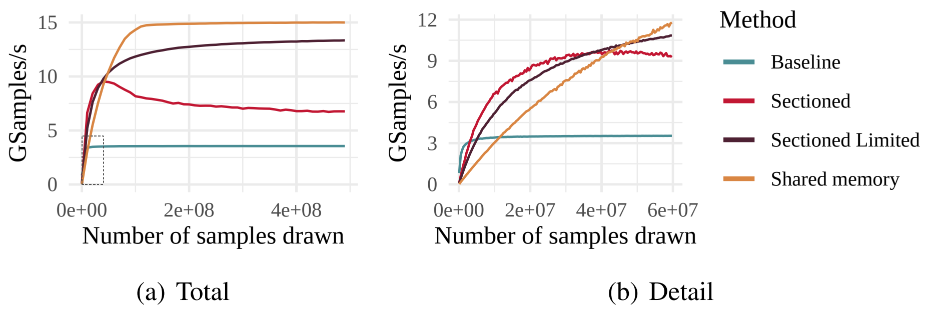


Figure 5.7: Comparison between sampling methods depending on the number of samples drawn. Input is a uniform random weight distribution of size 10^6.

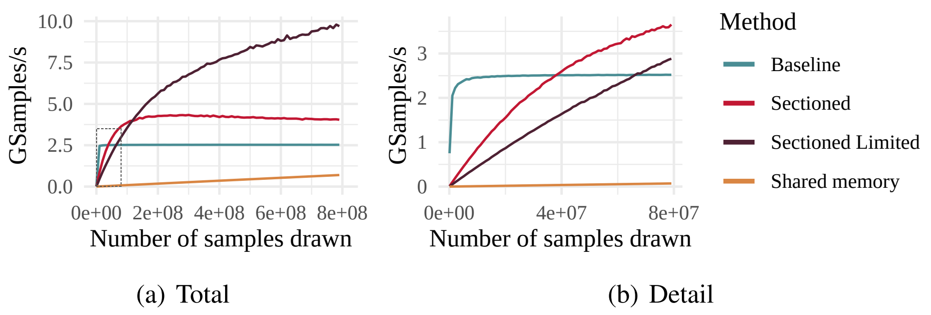


Figure 5.8: Comparison between sampling methods depending on the number of samples drawn. Input is a uniform random weight distribution of size 10^7.

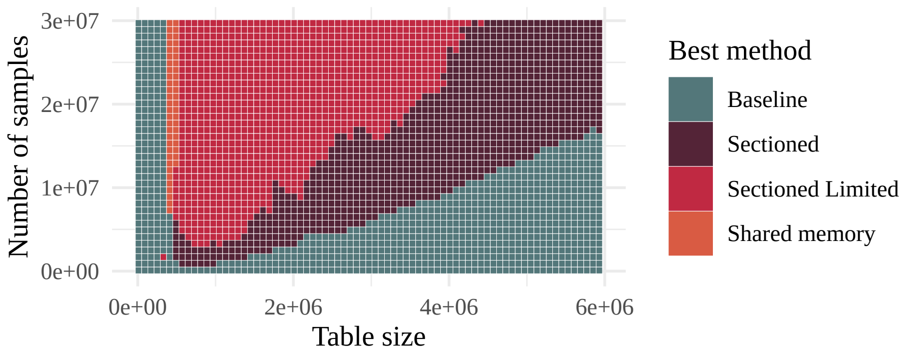


Figure 5.9: Comparison which method has the highest throughput depending on table size and number of samples drawn. The input distribution are uniform random weights.

Method	Machine	MSamples/s			
		10^6	10^7	10^8	10^9
Ref. [27]	C	–	–	2231	1675
	D	–	–	910	909
	B (CPU)	238	171	163	–
Ours	A (RTX 2080)	13438	10148	2442	–
	B (GTX 1650 Super)	6223	3474	1060	–

Table 5.3: Sampling throughput comparison with Hübschle-Schneider's method. Drawing 10^9 samples from a table of varying size. For $N \leq 10^7$, we use our sectioned limited sampling algorithm. For $N = 10^8$, we use our baseline sampling algorithm.

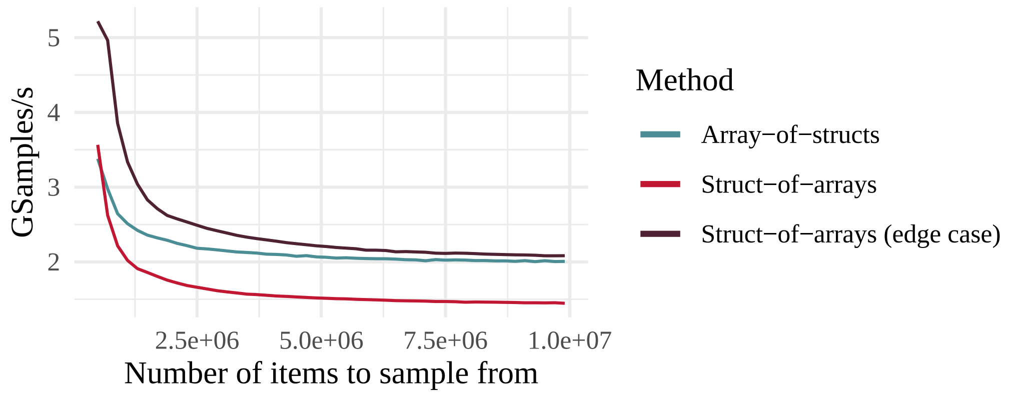


Figure 5.10: Sampling throughput of array-of-structs and struct-of-arrays. We assign special weight distributions where either no item has an alias (edge case) or all items have the next item as their alias (others).

5.3.3 Struct-of-Arrays

Sampling from the struct-of-arrays storage scheme is slower than from array-of-structs. Only in the edge case of an alias table where all items have weight $T_i^w = W/N$ (see Section 2.4.7), struct-of-arrays is slightly faster. Because of the two arrays, sampling an item requires two memory operations. Figure 5.10 shows performance measurements of the baseline sampling method when using the array-of-structs and the struct-of-arrays storage.

5.4 Power Usage

In 2015, data centers already used around 1% of the global power consumption. Until 2025, this amount is expected to increase to up to 4.5% [9]. An important cost factor of high performance clusters is cooling, as 1 W of power consumption requires up to 1 W of cooling [41]. Reduced power usage results in less heat being produced, reducing the amount of cooling needed. For comparing the CPU and the GPU implementation, a simple speedup is not enough. The two devices differ heavily in their architecture. Comparing the power usage is a better metric than hardware cost because it is not influenced by market decisions and gives an estimate of the run-time cost.

In this thesis, our main goal is to improve the performance. While more efficient code usually leads to less power usage, we do not explicitly optimize our implementation for low power usage. Techniques like undervolting or throttling are not evaluated. We also do not use techniques like power aware task scheduling [34]. For a study of what instructions and what memory access operations require most energy on GPUs, we refer to Ref. [18]. In cases where energy efficiency is crucial, FPGAs can provide a comparable performance to GPUs while having a better energy efficiency [41].

Experimental Setup The power usage of GPUs is influenced by many factors, for example by the temperature [49]. The experiments in this section are executed at room temperature. Before each measurement, we wait for the system to idle and cool down. The supply voltage is kept at the factory defaults. The total power usage of the GPU, including memory and voltage converters, can be measured with Nvidia's `nvtop` utility [49]. It does not contain CPU and RAM, so we use an external power measurement device.[4] When idle, machine A has a power consumption of 50 W and its RTX 2080 has a power consumption of 1 W.

Construction During construction of a table with 10^8 items, the GPU of Machine A has a power consumption of 167 W. The power consumption of the whole system is 263 W. This is a 213 W increase to the idle state, while the measurements with `nvtop` only show an increase of 166 W. Running the GPU code therefore also increases CPU and RAM power usage. To compensate for different hardware setups, we calculate the power usage with the difference between idle and load state using external measurements. The time to construct a single alias table is 32.8 ms. Constructing an alias table therefore uses 213 J/s $\cdot$ 32.8 ms $\approx$ 7 J. Measurements on the other machines are displayed in Table 5.4.

[4]For machines A and B, we use a BaseTech Cost Control 3000. For machines C and D, we use a Voltcraft Energy Check 3000.

Machine	Power usage
A (RTX 2080)	7 J/10^8 items
B (GTX 1650 Super)	–[5]
B (CPU implementation [27])	47 J/10^8 items
C (CPU implementation [27])	45 J/10^8 items
D (CPU implementation [27])	25 J/10^8 items

Table 5.4: Power usage of constructing an alias table of size 10^8, depending on the hardware used. Shuffled power law distribution with exponent=1.

Machine	Power usage
A (RTX 2080)	69 J/GSample
B (GTX 1650 Super)	69 J/GSample
B (CPU implementation [27])	511 J/GSample
C (CPU implementation [27])	242 J/GSample
D (CPU implementation [27])	181 J/GSample

Table 5.5: Power usage of sampling from an alias table of size 10^8, depending on the hardware used. Shuffled power law distribution with exponent=1.

Sampling When drawing 10^9 samples from an alias table of size 10^8, the GPU has a power consumption of 130 W. The external measurement shows 216 W for the whole system. At a sampling frequency of 2.4 GSample/s, this means that sampling needs 166 J/s / 2.4 GSample/s $\approx$ 69 J/GSample. Measurements of the other systems are displayed in Table 5.5. While the RTX 2080 has more than double the TDP than the GTX 1650 Super [1, 45], its increased sampling speed leads to exactly the same power usage per GSample.

Summary For construction, our GPU implementation needs only 14% of the power that the CPU implementation of Ref. [27] needs on a personal computer (Machine B). When comparing to the powerful 4-socket server (machine C), our implementation needs 15% of the power. For sampling, our implementation only needs 13% of the power on a personal computer and 28% on the server. This makes our method suitable for use in data centers where power usage must be considered carefully.

[5]The card does not have enough memory to hold temporary data structures during construction for table size $N = 10^8$. An extrapolation based on measurements with smaller N results in $\approx$ 7 J/10^8 items.

Chapter 6

Conclusion

In this thesis, we present new algorithms that make construction of and sampling from alias tables applicable on GPUs. We are able to achieve significant speedups, even if the algorithm by Hübschle-Schneider and Sanders [27] that we build upon is not efficient on GPUs. Our algorithms are more energy efficient than the CPU implementations.

Construction We introduce a new search algorithm, partial p-ary search, that enables fast splitting. Our pack method with chunked loading to the shared memory adapts the memory access pattern to be more efficient on GPUs. We demonstrate a speedup of 33.2 over the CPU implementation [27]. We even achieve significant speedups when including the memory transfers.

Sampling For a large number of samples, our sectioned limited sampling algorithm is significantly faster than previously known algorithms. This is achieved by dividing the alias table into sections which can then be sampled in a more cache-efficient way. While the original method by Hübschle-Schneider and Sanders [27] can sample with up to 2 GSamples/s on an expensive 4-socket server, we demonstrate up to 13 GSamples/s on consumer hardware.

Future Work We plan to test the fast GPU alias table construction in real-world applications like graph generation (see Section 2.9). Additionally, we plan to evaluate partial p-ary search on its own, which, to our knowledge, is not covered in the literature yet. It would be interesting to evaluate our algorithms on an Apple M1 processor, where CPU and GPU share the same memory pool and therefore do not need transfers [10]. We also plan to experiment with using block-wise prefix sums [25] to reduce the memory volume.

Chapter 7

Appendix

7.1 List of Figures

7.2 List of Tables

7.3 List of Algorithms

Bibliography

[1] Nvidia turing gpu architecture. https://images.nvidia.com/aem-dam/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, 2018. Accessed: 2020-12-14.

[2] Cub: cub namespace reference. https://nvlabs.github.io/cub/namespacecub.html#abec44bba36037c547e7e84906d0d23ab, 2020. Accessed: 2020-08-17.

[3] Cuda c++ best practices guide. https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf, 2020. Accessed: 2020-07-15.

[4] curand | nvidia developer. https://developer.nvidia.com/curand, 2020. Accessed: 2020-07-15.

[5] Green 500. https://www.top500.org/lists/green500/2020/06/, 2020. Accessed: 2020-10-06.

[6] Top500. https://www.top500.org/lists/top500/2020/06/, 2020. Accessed: 2020-10-06.

[7] Christos Alexopoulos and George S Fishman. Capacity expansion in stochastic flow networks. *Probability in the Engineering and Informational Sciences*, 6(1):99–118, 1992.

[8] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[9] Anders Andrae. Total consumer power consumption forecast. *Nordic Digital Business Summit*, 10, 2017.

[10] Apple. Apple m1 chip - apple. `https://www.apple.com/mac/m1/`, 2020. Accessed: 2020-12-12.

[11] Nikolaus Binder and Alexander Keller. Massively parallel construction of radix tree forests for the efficient sampling of discrete probability distributions. *arXiv preprint arXiv:1901.05423*, 2019.

[12] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[13] Guy E Blelloch. Prefix sums and their applications. Technical report, Citeseer, 1990.

[14] Karl Bringmann and Kasper Green Larsen. Succinct sampling from discrete distributions. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 775–782, 2013.

[15] FB Brown, DA Calahan, and WR Martin. A discrete sampling method for vectorized monte carlo calculations. *Trans. Am. Nucl. Soc.;(United States)*, 38(CONF-810606-), 1981.

[16] David Burke, Abhijeet Ghosh, and Wolfgang Heidrich. Bidirectional importance sampling for illumination from environment maps. In *ACM SIGGRAPH 2004 Sketches*, page 112. 2004.

[17] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[18] Sylvain Collange, David Defour, and Arnaud Tisserand. Power consumption of gpus from a software perspective. In *International Conference on Computational Science*, pages 914–923. Springer, 2009.

[19] Thanh Tuan Dao and Jaejin Lee. An auto-tuner for opencl work-group size on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 29(2):283–296, 2017.

[20] Donald e Knuth. *The art of computer programming 2: Seminumerical algorithms*. Addison-Wesley Publ. Company, 1969.

[21] George S Fishman and Louis R Moore III. Sampling from a discrete distribution while preserving monotonicity. *The American Statistician*, 38(3):219–223, 1984.

[22] Bruno Galerne, Ares Lagae, Sylvain Lefebvre, and George Drettakis. Gabor noise by example. *ACM Transactions on Graphics (TOG)*, 31(4):1–9, 2012.

[23] Zeinab Hmedeh, Harry Kourdounakis, Vassilis Christophides, Cédric Du Mouza, Michel Scholl, and Nicolas Travers. Content-based publish/-subscribe system for web syndication. *Journal of Computer Science and Technology*, 31(2):359–380, 2016.

[24] Daniel Horn. Stream reduction operations for gpgpu applications. *GPU gems*, 2(36):573–589, 2005.

[25] Lorenz Hübschle-Schneider. Communication-efficient probabilistic algorithms: Selection, sampling, and checking. 2020.

[26] Lorenz Hübschle-Schneider and Peter Sanders. Linear work generation of r-mat graphs. *Network Science*, pages 1–8, 2019.

[27] Lorenz Hübschle-Schneider and Peter Sanders. Parallel weighted random sampling. *arXiv preprint arXiv:1903.00227*, 2019.

[28] Intel. Intel® core™ i5-750 processor (8m cache, 2.66 ghz) product specifications. `https://ark.intel.com/content/www/us/en/ark/products/42915/intel-core-i5-750-processor-8m-cache-2-66-ghz.html`, 2009. Accessed: 2020-12-15.

[29] Intel. Intel® xeon® processor e3-1230 v2 product specifications. `https://ark.intel.com/content/www/us/en/ark/products/65732/intel-xeon-processor-e3-1230-v2-8m-cache-3-30-ghz.html`, 2012. Accessed: 2020-11-21.

[30] Tim Kaldewey, Jeff Hagen, Andrea Di Blas, and Eric Sedlar. Parallel search on video cards. In *First USENIX Workshop on Hot Topics in Parallelism (HotPar'09)*, 2009.

[31] Ares Lagae, Sylvain Lefebvre, George Drettakis, and Philip Dutré. Procedural noise using sparse gabor convolution. *ACM Transactions on Graphics (TOG)*, 28(3):1–10, 2009. Presentation: `https://www.youtube.com/watch?v=SqXsm44CCeU`.

[32] Pierre L'Ecuyer and Richard Simard. Testu01: Ac library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS)*, 33(4):1–40, 2007.

[33] Kaiwei Li, Jianfei Chen, Wenguang Chen, and Jun Zhu. Saberlda: Sparsity-aware learning of topic models on gpus. *ACM SIGPLAN Notices*, 52(4):497–509, 2017.

[34] Keqin Li. Performance analysis of power-aware task scheduling algorithms on multiprocessor computers with dynamic voltage and speed. *IEEE Transactions on Parallel and Distributed Systems*, 19(11):1484–1497, 2008.

[35] Wentian Li. Random texts exhibit zipf's-law-like word frequency distribution. *IEEE Transactions on information theory*, 38(6):1842–1845, 1992.

[36] George Marsaglia et al. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003.

[37] MA Martínez-del Amor. *Accelerating membrane systems simulators using high performance computing with GPU*. PhD thesis, Ph. D. thesis, University of Seville, 2013.

[38] Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

[39] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[40] Richard L. Mitchell and CR Stone. Table-lookup methods for generating arbitrary random numbers. *IEEE Transactions on Computers*, (10):1006–1008, 1977.

[41] Sparsh Mittal and Jeffrey S Vetter. A survey of methods for analyzing and improving gpu energy efficiency. *ACM Computing Surveys (CSUR)*, 47(2):1–23, 2014.

[42] Siddhant Mohanty, AK Mohanty, and F Carminati. Efficient pseudo-random number generation for monte-carlo simulations using graphic processors. In *Journal of Physics: Conference Series*, volume 368, page 012024. IOP Publishing, 2012.

[43] Hubert Nguyen. *Gpu gems 3*. Addison-Wesley Professional, 2007.

[44] Nvidia. Nvidia cuda compute unified device architecture - reference manual. `http://developer.download.nvidia.com/compute/cuda/2_0/docs/CudaReferenceManual_2.0.pdf`, 2008. Accessed: 2020-11-20.

[45] Nvidia. Geforce gtx 1650 super graphics card. `https://www.nvidia.com/en-us/geforce/graphics-cards/gtx-1650-super/`, 2020. Accessed: 2020-11-14.

[46] Manuel Penschuck, Ulrik Brandes, Michael Hamann, Sebastian Lamm, Ulrich Meyer, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in scalable network generation. *arXiv preprint arXiv:2003.00736*, 2020.

[47] Ken Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985.

[48] Carolyn L Phillips, Joshua A Anderson, and Sharon C Glotzer. Pseudo-random number generation for brownian dynamics and dissipative particle dynamics simulations on gpu devices. *Journal of Computational Physics*, 230(19):7191–7201, 2011.

[49] Danny C Price, Michael A Clark, Benjamin R Barsdell, Ronald Babich, and Lincoln J Greenhill. Optimizing performance-per-watt on gpus in high performance computing. *Computer Science-Research and Development*, 31(4):185–193, 2016.

[50] Greg Ruetsch and Brent Oster. Getting started with cuda. `https://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf`, 2008. Accessed: 2020-07-15.

[51] Mutsuo Saito and Makoto Matsumoto. Variants of mersenne twister suitable for graphic processors. *ACM Transactions on Mathematical Software (TOMS)*, 39(2):1–20, 2013.

[52] Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade, and Carsten Dachsbacher. Efficient parallel random sampling—vectorized, cache-efficient, and online. *ACM Transactions on Mathematical Software (TOMS)*, 44(3):1–14, 2018.

[53] Riyanarto Sarno, Virendra C Bhavsar, and Esam MA Hussein. Generation of discrete random variables on vector computers for monte carlo simulations. *International Journal of High Speed Computing*, 2(04):335–350, 1990.

[54] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. Graphjet: real-time content recommendations at twitter. *Proceedings of the VLDB Endowment*, 9(13):1281–1292, 2016.

[55] J Cole Smith and Sheldon H Jacobson. An analysis of the alias method for discrete random-variate generation. *INFORMS Journal on Computing*, 17(3):321–327, 2005.

[56] Guy L Steele Jr and Jean-Baptiste Tristan. Using butterfly-patterned partial sums to optimize gpu memory accesses for drawing from discrete distributions. *arXiv preprint arXiv:1505.03851*, 2015.

[57] Scott Sullivan. Lda algorithm description. `https://www.youtube.com/watch?v=DWJYZq_fQ2A`, 2017. Accessed: 2020-07-09.

[58] Myles Sussman, William Crutchfield, and Matthew Papakipos. Pseudorandom number generation on the gpu. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 87–94, 2006.

[59] J Teuhola and O Nevalainen. Two efficient algorithms for random sampling without replacement. *International Journal of Computer Mathematics*, 11(2):127–140, 1982.

[60] Eric Veach and Leonidas J Guibas. Optimally combining sampling techniques for monte carlo rendering. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 419–428, 1995.

[61] Jeffrey Scott Vitter. Faster methods for random sampling. *Communications of the ACM*, 27(7):703–718, 1984.

[62] Michael D. Vose. A linear algorithm for generating random numbers with a given distribution. *IEEE Transactions on software engineering*, (9):972–975, 1991.

[63] Alastair J Walker. An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):253–256, 1977.

[64] Tony Warnock. Random-number generators. *Los Alamos Science*, 15(1987):137–141, 1987.

[65] SJ Wilderman and YK Dewaraja. Method for fast ct/spect-based 3d monte carlo absorbed dose computations in internal emitter therapy. *IEEE transactions on nuclear science*, 54(1):146–151, 2007.