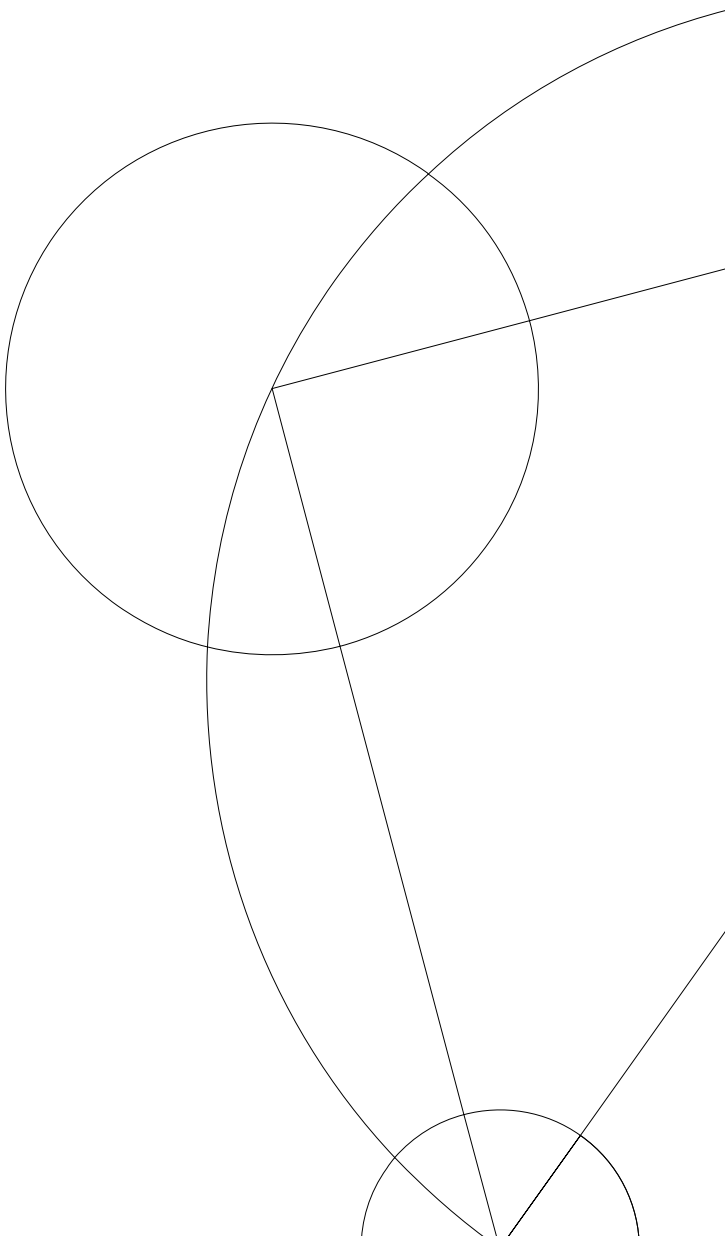
UNIVERSITY OF COPENHAGEN

Multireduce and Multiscan on Modern GPUs

Marco Eilers
eilers.marco@googlemail.com

Master's thesis

Department of Computer Science
Supervisor: Andrzej Filinski



March 2014

Abstract

With the introduction of platforms like CUDA and OpenCL, the superior computing power of modern GPUs compared to CPUs is used more and more often to accelerate general purpose computations. Data parallel primitives like reduce, scan or sort can be used as simple, deterministic building blocks for parallel algorithms, hiding the complexity of the underlying GPU implementation. The multireduce and multiscan, generalizations of the ordinary reduce and scan, have immediate applications for common algorithmic problems and show potential to be useful primitives as well. This thesis aims to find efficient GPU algorithms for both of theses problems, which currently do not exist.

In order to establish a baseline CPU implementation to which GPU algorithms can be compared, we first discuss sequential algorithms for both multireduce and multiscan. We point out performance problems of sequential algorithms for both operations which stem from poor use of various caches, and show that huge pages and partial radix sorting can be used to avoid these problems. We find that the potential improvement which can be achieved through sorting differs wildly between different CPUs, and provide a simulator to estimate the benefit of this method for any given system.

For GPUs, we systematically evaluate possible algorithms for both problems. We examine three main groups of algorithms: parallel adaptations of the sequential algorithm, GPU adaptations of existing PRAM algorithms, and sort-based conversions to simpler problems, namely segmented scan and reduce. For each of these possibilities, we discuss how the GPU memory hierarchy can be used for best performance, and which additional algorithmic improvements can be implemented for operators which are commutative as well as associative. For the multireduce, we propose an algorithm which, despite its generality, performs at least as well as the best published histogramming algorithm for all inputs and provides a 18× speedup over the CPU algorithm for small numbers of buckets. We also propose an algorithm based on sorting and segmented reduction which, in contrast to the previous algorithm, can also be used for non-commutative operators, and whose performance is on par with reduce-by-key implementations of current libraries. For large numbers of buckets, the range of available algorithms is smaller and the speedup compared to the CPU implementation can shrink to a factor of three. Like for CPUs, we establish that partial sorting can lead to better cache hit rates and better overall performance under certain circumstances.

Subsequently, we apply the insights gained in the design of multireduce algorithms to closely related problems, namely scattering and histogramming. In the latter case, the superior work distribution of our fastest multireduce algorithm results in a 40% speedup over the best currently available histogram algorithm, independently of the input data, making our solution the fastest existing histogram algorithm for GPUs.

A similar discussion is presented for possible multiscan algorithms. We observe that the only existing work-efficient PRAM algorithm for the multiscan is intrinsically unsuited for execution on GPUs and leads to poor performance. While a moderate speedup compared to the sequential algorithm is possible, the overall performance of multiscan is significantly worse than that of multireduce algorithms. We therefore conclude that the multiscan cannot be recommended as a general building block for GPU algorithms.

Contents

Chapter 1

Introduction

1.1 Motivation and Structure

In 1965, Gordon Moore [31] predicted that the number of transistors on a chip would at least double every year for the next ten years. While the exact rate of increase has been adjusted to a doubling every 18 months, this prediction, known as "Moore's law", has stood the test of time and, with some minor deviations, holds true until today. In recent years, however, the way in which this increase of computing power is expressed has changed in a major way. Until 2003, the clock frequency of processor cores doubled along with the number of transistors, thus steadily increasing the performance of sequential, single-thread algorithms. Since then, however, physical limitations have prevented clock frequencies from substantially surpassing 4 GHz [35], and today the performance of single cores no longer increases significantly. Instead, CPU manufacturers have started to increase the number of cores on a single CPU die. Since the beginning of this trend, programmers therefore have to use several CPU cores at once in order to fully utilize a system's resources.
Existing programs which were written with single core systems in mind can sometimes benefit from having multiple cores. This is the case if they employ *concurrency*, i.e. they use different threads for performing different tasks. This is the case in many programs, since the use of multiple threads has many benefits even on single core systems: They are means to ensure the responsiveness of parts of a system in the presence of other ongoing computation. Obvious applications of this are user interfaces, which are expected to react to user inputs even while the application is working on something else, or servers, which have to be able to quickly respond to many clients' requests even if some specific request triggers a lot of computation. But while the goal of concurrency is not necessarily to achieve a speedup on multi-core hardware, concurrent applications naturally benefit from a system's ability to execute several computations at once. If, however, the main or only goal of using multiple threads is a speedup on multi-core systems, this is referred to as *parallelism*. Parallelism is therefore only sensible to employ on multi-core systems, since multiple threads do not benefit an application's performance significantly if all threads have to be executed sequentially.
There are two main brands of parallelism, *task parallelism* and *data parallelism*. The former denotes a situation where different threads (and therefore different cores) perform different computations, using either the same or different data. In many cases, the border between concurrency and task parallelism is fluid, since, as pointed out before, letting a different thread handle a specific task may be sensible for reasons other than performance, yet doing so on a multi-core system will automatically result in a speedup. But task parallelism can also be employed explicitly and without concurrency in mind.
However, task parallelism comes with many problems. For one, it generally does not scale well with the number of available processors. If a number n of tasks is executed in different threads, then the application's performance can benefit from using up to n cores, but any

additional cores will be left unused. Since the number of cores in typical consumer and server CPUs is expected to rise further in the future, this is unsatisfying, especially because for most applications, there is only a very limited number of tasks that can naturally be separated from one another.
Another issue is that, in many cases, cooperation between different threads is necessary, often requiring the use of tools like locks or synchronization between threads. This is problematic because, firstly, the achievable speedup is limited if threads have to wait for other threads to release a lock, or to reach a synchronization barrier. Secondly and more importantly, it introduces non-determinism and thereby makes it much more difficult to write correct programs. Problems like race conditions and deadlocks, which simply cannot happen in single-thread code, are a major problem that many programmers find hard to avoid. This is only a problem if tasks that are executed in parallel have side effects that affect other threads, but in most applications, this is often the case.
The second brand of parallelism, data parallelism, has none of these problems. In data parallelism, the same task is performed for many pieces of data in parallel. An obvious condition for this to work is that there is a big number of data elements on which the same or similar computations have to be performed. If this is the case, however, the program can potentially make use of as many processors as there are pieces of data. Furthermore, this kind of parallelism can be exploited by programmers without having to write explicitly parallel code that uses threads. Not all algorithms can be implemented this way, but in most cases, skeletons for data parallel operations can be provided in libraries or in language primitives, so that application programmers only have to input application-specific logic, and generic code will handle the distribution of the available data among the available cores. In these cases, the application itself will be completely deterministic, thus avoiding most of the problems that plague concurrent and task-parallel programming.
While CPUs have been forced to embrace parallelism by physical limitations which have made improvement of purely sequential performance impossible, they have still stayed close to their original design: Every single CPU core is in itself a fully functional processor and is generally optimized to deliver good single thread performance. The main difference to CPUs before 2003 is that several such cores are combined on one die. This conservative approach to parallel hardware can be called *multi-core*.
There is, however, a different approach, which is often called *many-core*. In this approach, the number of processors combined on one chip is not a handful(two to eight), but tens or hundreds. In order to achieve this, the cores have to be simple in nature, sharing resources where possible, and giving up on many of the optimizations that allow optimal single thread performance, like for example out-of-order-execution. A multiprocessor designed with this paradigm in mind derives its strength from its ability to execute many instructions at the same time, which can lead to a much higher number of operations executed per time unit than a multi-core approach. The downside is usually that utilization of all cores is necessary to achieve good performance, since single thread performance can be more than ten times lower than that of traditional single-core or even multi-core CPUs [40].
A natural approach for designing such hardware is SIMD (Single Instruction, Multiple Data), where several cores concurrently execute the same instruction on different data. This allows several cores to share facilities like the instruction counter, branching hardware and instruction cache, which keeps individual cores small and simple and enables manufacturers to pack even more cores on one die. Hardware following this paradigm is obviously a great foundation to run data-parallel programs, as the hardware is specifically optimized to efficiently perform the same computations on many pieces of data.
CPUs have adopted this principle to some degree by offering SIMD instructions as part of the Streaming SIMD Extensions (SSE), which offer special instructions that perform an arithmetic operation on a number (usually between two and eight) values at once. The reliance on

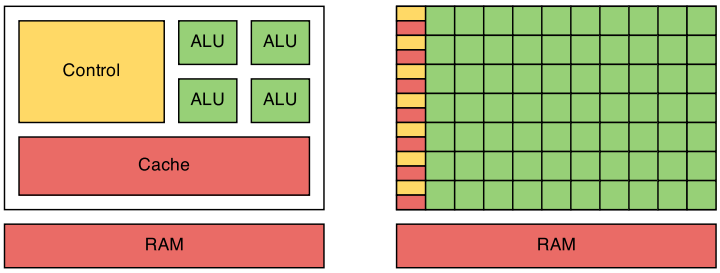


Figure 1.1: Schematic structure of CPUs as opposed to GPUs (adapted from [13])

special instructions means that programs need specific adaptations to make good use of SSE instructions, and the additional effort for this is usually only invested for computationally intensive applications. In most cases, single CPU cores therefore still perform the bulk of their work completely sequentially.

A type of hardware that follows the SIMD principle on a deeper and more fundamental level are graphics processors (GPUs) [23]. Until the late 1990s, graphics hardware consisted entirely of fixed-function pipelines with a single purpose, the acceleration of 3D rendering. They were configurable to some degree, but not freely programmable like CPUs. From 2001 onwards, manufacturers introduced programmable pixel and vertex shaders, which enabled advanced effects that were not feasible with the old fixed function architecture. In the following years, these different shader types evolved to use the same hardware, resulting in the so-called unified shader architecture in 2006. These unified shaders could be used as either pixel, vertex or geometry shaders, depending on the needs of the application. Unified shaders were therefore fully programmable processors with the flexibility to execute different kinds of tasks, and since the number of shaders increased with every new generation, GPUs slowly turned into general purpose processors with hundreds of cores working in parallel (see Figure 1.1).

Moreover, the pressure to achieve higher and higher performance, mostly for 3D video games, had resulted in a highly parallel processor design with enormous floating point performance, far beyond that of then-current CPUs. Figure 1.2 shows the peak floating point performance of CPUs and GPUs over the last few years; the best GPUs consistently outperform the best CPUs by a factor of ten to 20. Note that the performance shown for CPUs is the cumulated peak performance of several cores using SSE instructions, which increase the CPU's throughput up to a factor of eight (see [47] for a performance comparison of scalar and SIMD instructions). The peak performance of a completely sequential algorithm using a single core and scalar operations is therefore much lower than shown in Figure 1.2.

Researchers soon noticed this potential and attempted to use GPUs for applications other than 3D rendering, introducing the concept of *GPGPU* (general-purpose computing on graphics processing units). There were, however, many obstacles: The APIs that had to be used to program shaders were intended to be used for graphics processing only. Programs therefore had to be written in shader languages like GLSL or HSHL, which offer no support for custom data types, only accept inputs in the form of textures, and write results to the frame buffer in the form of pixel values. There were further architectural limits: Pixel shader routines, for example, are expected to compute the final value of a single pixel and can therefore only write to that pixel's location in the frame buffer; writes to dynamically computed locations are impossible [23].

With the release of the next generation of graphics processors in 2007, NVIDIA then introduced the Compute Unified Device Architecture (CUDA), making GPUs fully programmable using an extension of C/C++ and without thinking of data in terms of 3D rendering. From then on, programmers and researchers have strived to exploit the superior processing power

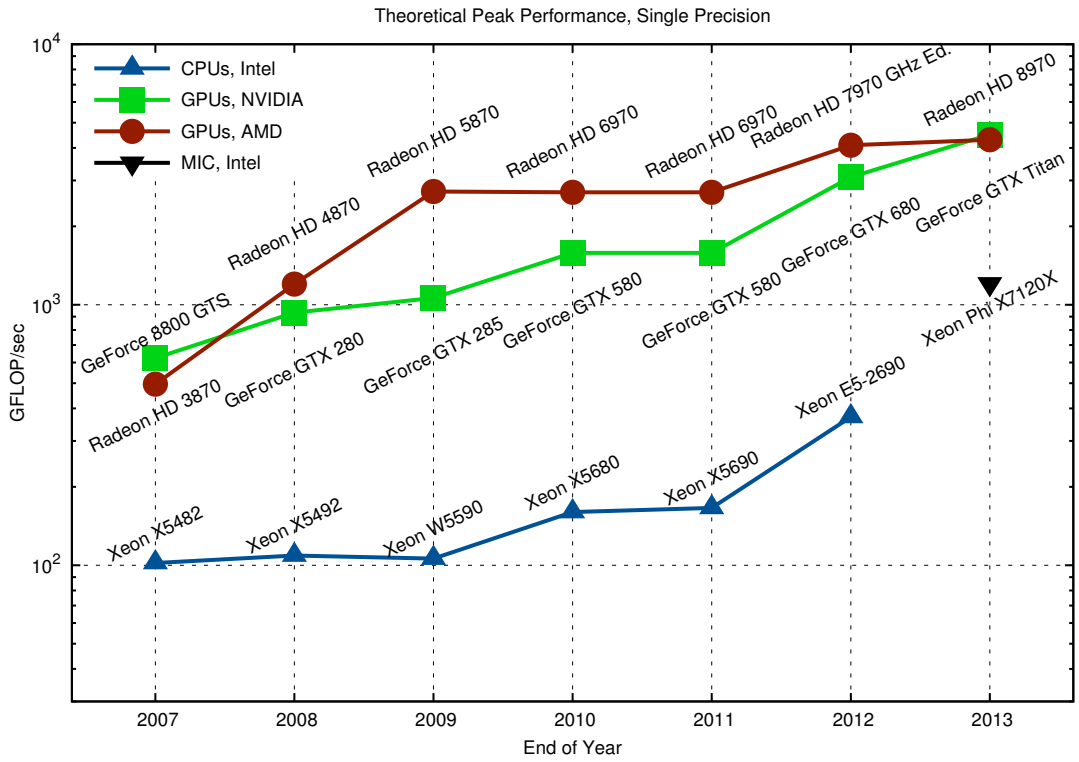


Figure 1.2: Comparison of floating point performance of CPUs and GPUs (single precision) [40]

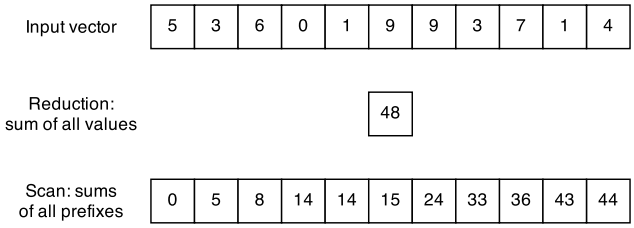


Figure 1.3: Reduction and scan of an input vector using addition

of GPUs for various purposes.

One approach for making GPU acceleration accessible to users is, as mentioned before, the use of skeletons. The Thrust library [20] is one example of this; designed to resemble the C++ Standard Template Library (STL), it offers reusable and composable skeletons that allow users to transparently use GPU acceleration without having to write GPU-specific code. In order to achieve this, it offers functions for various problems like sorting, gathering and scattering, as well as higher order functions like map, reduce and scan.

A reduction combines a vector of inputs using a binary function. If the function is, for example, addition, then a reduction of a vector of numbers will compute the sum of all numbers in the vector. The scan is a closely related operation: Instead of calculating a single reduction, it outputs a vector containing the reductions of all prefixes of the input vector. Both operations are illustrated in Figure 1.3.

The scan in particular has been a useful tool that serves as a basis for many other parallel algorithms and has been called "a miracle of efficient parallel communication" [12]. Though

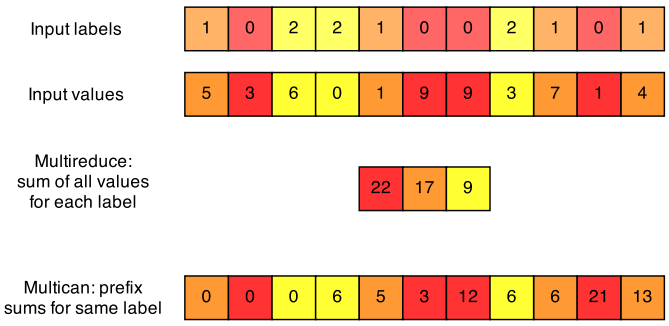


Figure 1.4: Multireduce and multiscan using addition

seemingly inherently sequential, it can be trivially distributed among many cores and efficiently computed in parallel.
The same is true for reductions: Especially when algorithms are expressed as compositions of several high level functions, one can imagine that there will often be a need to reduce a number partial results.
While these two operations have been implemented on GPUs and are almost ubiquitously used, their generalizations, multireduce and multiscan, have not. Multireduce and multiscan generalize the ordinary reduce and scan in a similar way: In addition to a single vector of input values, they accept a vector of labels. For every label, a multireduce computes the reduction of all values with that label. The multiscan, on the other hand, computes for every input value the reduction of all previous values with the same label. Both operations are illustrated in Figure 1.4 and will be defined more precisely in Section 1.2.
The multiscan has been proposed as a fundamental primitive for parallel computation several times.
In 1987, Ranade [39] proposed an abstract machine which only offers a number of set operations and a multiscan operation as primitives. In spite of this, it still subsumes many other abstract machines proposed at the time, proving the expressive power of the multiscan in combination with a small number of other primitives. Similarly to the ordinary scan, a number of algorithms can be elegantly expressed in terms of multiscans, including sorting [39] and sparse matrix vector multiplication [45]. Additionally, since an ordinary scan is just a special case of a multiscan, all applications of scans can also be expressed in terms of multiscans as well.
The multireduce likewise has a number of obvious applications. Scattering and histogramming, two operations that are ubiquitous in data parallel computing, are special cases of multireduce. A possible application for the general multireduce is MapReduce as presented by Google [14], a general framework for parallel computation consisting of two phases (see Figure 1.5). In the first, the map phase, a unary function is applied ("mapped") to a large number of input values in parallel, resulting in a number of key-value-pairs. In the reduce phase, these are then distributed to different processors by their key, and a reduction is performed on the values of each key separately. This distribution by keys, called "shuffling", obviously requires some kind of scattering, gathering or sorting. Additionally, different keys are generally thought to be processed by different processors, possibly resulting in poor load balancing among the available resources if the distribution of keys is very skewed. An efficient multireduce operation could make the shuffling unnecessary, thus saving resources and enabling a sensible use of the available processing power.
In spite of this, there has been no theoretical proposal or practical implementation of either multiscan or multireduce on modern parallel hardware. The most recent paper on the subject was written by Sheffler in 1993 and targets contemporary CRAY supercomputers.

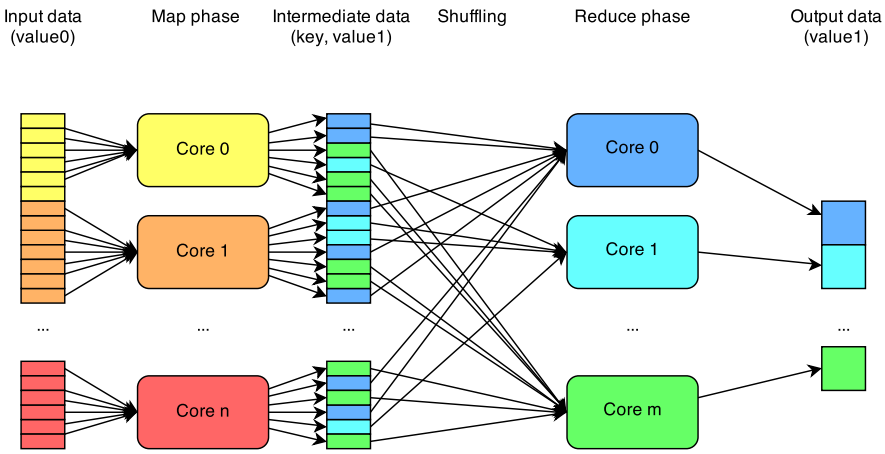


Figure 1.5: Schematic structure of the MapReduce framework

The aim of this thesis is therefore to evaluate the possibility of implementing these two functions efficiently on modern graphics processors. In order to accomplish this, we will first define the multireduce and multiscan as well as relevant related operations and special cases. Since most of these operations have have simple optimal algorithms in theory, we will then introduce the theoretical models of computers which are typically used to evaluate algorithm performance on an abstract level, so that we can later point out how these models differ from actual hardware. Based on this, we will try to find optimal sequential algorithms for multireduce and multiscan, which we will use as a baseline to evaluate the performance of the GPU implementation against. Subsequently, the architecture of GPUs will be examined in detail and the requirements that algorithms have to satisfy in order to perform well on GPUs will be explained.
We will then begin a systematic evaluation of possible algorithms, first for the multireduce, then for the multiscan, discussing relevant literature along the way. This includes GPU algorithms for related problems as well as general parallel algorithms for multireduce and multiscan, which will be evaluated for their adaptability for GPUs. Throughout this discussion, we will implement and benchmark different algorithms wherever necessary.
Since, as pointed out before, there is a number of frequently used operations which is closely related especially to the multireduce, we will apply the insights won in the previous discussion to these problems and evaluate their usefulness in these cases.

1.2 Definitions

An ordinary *reduce* (also known as a *fold*) is a higher order function that applies a given binary operator to all values in a vector. While it is not generally necessary for the input to be a vector of values (it could also be a tree or another recursive data structure), we will be working exclusively with vectors in this thesis. More formally:
Given a monoid M with an associative binary operation $\odot$: $M \times M \to M$ and a neutral element $e \in M$, a reduce operates on a vector of values $[v_0, v_1, ..., v_{n-1}]$, where each $v_i \in M$, and computes r such that

$$r = \bigodot_{i=0}^{n-1} v_i.$$

Typical choices for $\odot$ are addition, multiplication, minimum, or maximum on integer or floating point values.
A *scan*, also known as a prefix sum, takes the same inputs and computes a vector of values $[r_0, r_1, ..., r_{n-1}]$ that contains the result of the reduce of all prefixes of the input vector. There are two variants of scans: for an *exclusive* scan, the ith output element will be the reduce of the input vector up to but excluding the ith input element, such that

$$r_k = \bigodot_{i=0}^{k-1} v_i,$$

whereas for an *inclusive* scan, the ith output element is the reduce of the prefix *including* the ith element, such that

$$r_k = \bigodot_{i=0}^{k} v_i.$$

There are *segmented* versions of both reduce and scan, with an additional input vector $[f_0, f_1, ..., f_{n-1}]$, with each $f_i \in \{0, 1\}$. These serve as flags that denote the beginning of a new segment. The reduce and scan are then computed for each individual segment. If q of the input flags are set to one (and the input is therefore partitioned into q segments), and the beginnings of new segments are denoted by the set P of $\{p_0, p_1, ..., p_{q-1}\}$ (so that $f_{p_i} = 1$ if $i \in P$), and these positions are assigned so that $p_i < p_{i+1}$ for all $i < q - 1$, then the result of a segmented reduce is a vector $[r_0, r_1, ...r_{q-1}]$ such that

$$r_k = \begin{cases} \bigodot_{i=p_k}^{p_{k+1}-1} v_i & \text{if } k < q-1 \\ \bigodot_{i=p_k}^{n-1} v_i & \text{if } k = q-1 \end{cases}$$

The definition of a segmented scan is analogous. Using

$$q_k = \begin{cases} p_{q-1} & \text{if } p_{q-1} \leq k \\ p_i \text{ s.t. } p_i \leq k < p_{i+1} & \text{if } p_{q-1} > k, \end{cases}$$

the output $[r_0, r_1, ..., r_{n-1}]$ of a segmented scan can be defined as

$$r_k = \bigodot_{i=q_k}^{k-1}$$

in the exclusive case and

$$r_k = \bigodot_{i=q_k}^{k}$$

for the inclusive case.
Note that the result of segmented scans is undefined if the first flag is not set to one. Since the start of the vector must necessarily begin a new segment, there is no meaningful interpretation of the segmented scan if the flags state otherwise. This is not necessarily the case

for segmented reduces, where the beginning of the input vector may just be ignored until a set flag is encountered.
Multireduce is another generalization of the ordinary reduce operation and performs separate reductions on groups of values in a vector. In contrast to the segmented reduce, these groups of values do not have to be sections of adjacent values in the input vector, but can be distributed freely.
For a vector of values $[v_0, v_1, ..., v_{n-1}]$, where each $v_i \in M$, and a vector of integer labels $[l_0, l_1, ..., l_{n-1}]$ with each $l_i \in \{0, 1, ..., m-1\}$, a multireduce computes a vector of reductions $[r_0, r_1, ..., r_{m-1}]$ such that

$$r_k = \bigodot[v_j \mid k \in \{0, 1, ..., m-1\} \wedge l_j = k].$$

A common special case of a multireduce where all values are equal to one, i.e. $v_i = 1$ for all $i \in \{0, 1, ..., n-1\}$, and $\odot$ is chosen to be addition, is called a *histogram*.
Multiscan, which is more commonly known as *multiprefix*, generalizes the scan operation (see for example Blelloch's introduction [5]) in a similar way, and can also be described as performing multireduce for all prefixes of the input vectors.
Given the same inputs as before, the multiscan operation computes the vector $[s_0, s_1, ..., s_{n-1}]$ such that

$$s_i = \bigodot[v_j \mid j \in \{0, ..., i\} \wedge l_j = l_i]$$

for the inclusive multiscan, or

$$s_i = \bigodot[v_j \mid j \in \{0, ..., i-1\} \wedge l_j = l_i].$$

for the exclusive multiscan.
Analogous to histogramming, there is a special case of the multiscan where all values are equal to one, which we will refer to as *multitagging*.
The *scatter* operations is closely related to the multireduce and can be defined as follows:
Given a vector of values $[v_0, v_1, ..., v_{n-1}]$, and a vector of integer labels $[l_0, l_1, ..., l_{n-1}]$ with each $l_i \in \{0, 1, ..., m-1\}$, the scatter computes the vector $[s_0, s_1, ..., s_{m-1}]$ such that

$$s_i = v_j \text{ s.t. } j = max\{k | l_k = i\}.$$

If there is no such v_j (because the set $\{k|l_k = i\}$ is empty), s_i is either undefined, or may fall back to either a default value or corresponding value in an additional input vector. By this definition, if several values are scattered to the same location, the one with the highest index (i.e. the last one) wins.
Note that with this definition, the scatter can be regarded as a special case of the multireduce, where the operator $\odot$ is defined as

$$x \odot y = y.$$

Sometimes the scatter is defined less strictly, so if several values are scattered to the same location, any one of them may win. For this type of scatter, the definition for s_i is simply

$$s_i = v_j \text{ s.t. } i = l_j.$$

In this thesis, we will generally assume that a scatter follows the first definition, and point out that we are using the second definition whenever we do so.
The scatter has a mirror image, the *gather*. Although its structure is different from all of the aforementioned operations, we will still include it here and throughout the thesis because of its close relation to the scatter, and because many of the performance considerations applicable to the other operations also apply to the gather.
Given a vector of values $[v_0, v_1, ..., v_{m-1}]$, and a vector of integer labels $[l_0, l_1, ..., l_{n-1}]$ with each $l_i \in \{0, 1, ..., m-1\}$, the gather computes a vector $[s_0, s_1, ..., s_{n-1}]$ such that

$$s_i = v_{l_i}.$$

In the widespread case where $n = m$, and with the vector of labels being some permutation of the integers $[0, ..., m-1]$, gather and scatter are therefore two different ways of perform a permutation of the given vector of values. Additionally, both definitions of the scatter are equivalent in this case.

1.3 Theoretical models of computation

When evaluating the algorithmic complexity of a given algorithm, one first needs to choose an abstract machine on which the algorithm is assumed to run. In comparison to real world computers, such abstract machines are much simpler, usually offering only a very limited number of instructions and consisting of only few distinct parts, and are defined mathematically. This also means that they can offer features that real life computers cannot have in principle, like unlimited amounts of memory. Reasoning about the runtime and space requirements of a given algorithm is therefore much simpler on an abstract machine (and, since there are many more different versions of actual computing hardware than different actual machines, the result is much more generally applicable).
While this approach results in general, provable statements about the runtime of a given algorithm, it is not without flaws. It is, for example, possible that an algorithm whose theoretical complexity is optimal may be less than optimal on an actual computer, because the computer's hardware operates differently than that of the chosen abstract machine.
We will now describe the two abstract machines most commonly used for evaluating the complexity for either sequential or parallel algorithms, respectively.

- A random access machine (RAM) consists of a finite program that operates on an arbitrarily large number of registers holding integers. The size of the integers is generally assumed to be finite, but large enough to contain the index of any register. The main difference between a RAM and other similar abstract machines of its time is that the RAM offers instructions to indirectly address registers. The instruction $X_i \leftarrow X_{X_j}$, for example, copies the contents of register number X_j to register X_i. A similar instruction exists for copying an integer to an indirectly specified register. This way of addressing essentially enables the user to use the registers like a block of continuously addressed memory that can be randomly accessed, hence the name. The cost of a random memory access, although suggested differently in the original paper [11], is usually assumed to be constant.

  Additionally, the random access machine offers basic arithmetic operations (while the original paper only mentions addition and subtraction, one usually also includes multiplication and division as well as all other operations that modern processors can execute in constant time), comparisons and conditional branches, all of which also take unit time.

- A parallel random access machine (PRAM) is a generalized version of a RAM that essentially consists of an arbitrarily large number of RAMs which can communicate through shared memory of finite, but arbitrarily large size. A PRAM is usually assumed to operate in a Single-Instruction-Multiple-Data (SIMD) fashion, meaning that the same program is run on all processors. Each processor does, however, know its own index, which allows for meaningful communication through shared memory. All processors of a PRAM synchronously execute cycles of the following three phases:

  1. Load value from shared memory
  2. Perform local computation
  3. Write value to shared memory

  Each of these phases can, of course, be left out.

  An obvious problem PRAMs have that does not occur in RAMs are access conflicts, i.e. several processors reading from or writing to the same shared memory cell. There are numerous types of PRAMs that handle this problem differently:

  - Exclusive Read Exclusive Write (EREW): No two processors are allowed to read from or write to the same memory cell at once. A program that ignores this constraint is an illegal program for this type of PRAM and its behaviour is undefined.
  - Concurrent Read Exclusive Write (CREW): No two processors are allowed to write to the same memory cell at once. Reading from the same cell is allowed.
  - Concurrent Read Concurrent Write (CRCW): Both reading from the same cell and writing to the same cell are allowed. There are, again, several types of CRCW PRAMs to distinguish the results of simultaneous writes to the same cell:
    * COMMON CRCW: Writing to the same cell is only allowed if all processors write the same value. If this is not the case, the program is, again, illegal, and the behaviour is undefined.
    * ARBITRARY CRCW: One randomly chosen write will succeed; the values written to the same memory cell by other processors are discarded.
    * PRIORITY CRCW: Each processor has a fixed priority. The processor with the highest priority will succeed. (cf. [16])

Additional types of PRAMs exist [27], but those mentioned before are the most fundamental ones, and we will not discuss any others.
There are no significantly different definitions of RAMs, and where different models exist, they are generally equally powerful and can simulate each other without asymptotic performance loss, the situation is more complicated for parallel algorithms. Any algorithm that runs on a COMMON CRCW PRAM will, for example, run correctly on an ARBITRARY CRCW PRAM, while the opposite is not necessarily true. Similarly, any ARBITRARY CRCW PRAM algorithm will work on a PRIORITY CRCW PRAM but not the other way around. For parallel algorithms one should therefore always state the assumed abstract machine as well as the derived algorithmic complexity.
While both RAMs and PRAMs differ from the architecture found in real life computers in many ways, the main differences for the purposes of this thesis lie in the organization and performance of the memory. Instead of being an infinite, continuous block with equal (constant) access time to any part of it, no matter if it was accessed before or not, real life memory subsystems consist of a hierarchy of several types of memory with increasing size and increasing latency that work together in complex ways. We will discuss important differences between theoretical model and actual hardware in detail in Section 2.3 for the CPU and in Section 3.1 for the GPU.

Chapter 2

Sequential implementation

In this chapter, we will examine the how the multireduce, multiscan and related problems can be solved by sequential algorithms. Our aim is to find an optimal sequential algorithm which we can use as a baseline when we are evaluate GPU algorithms in the following chapters. We will therefore try to identify any problems which limit the algorithms' performance and look for ways to avoid them. Where necessary, we will outline background information algorithmic or hardware issues along the way. We will start out by explaining the trivial sequential algorithms for our main algorithms.

2.1 Naive approaches

All of the operations which we have defined in the last chapter share a very simple common structure. It is therefore not surprising that there are very simple algorithms for calculating all of them, which we will discuss in this section. As a simple example, we will first consider the *scatter* (see Algorithm 2.1).
The inputs are two vectors *indices* and *values* of length n. *values* may contain arbitrary values of arbitrary type (we will call the type M throughout the thesis), whereas *indices* may contain unsigned integers in the range $[0...m-1]$. The result is a vector of the same type as *values* of length m.
In order to perform the scatter, the *values* and *indices* vectors are traversed simultaneously, and for each index-value-pair, the value is copied to its corresponding index in the *result* vector. If several values have the same index, the value that occurs last in *values* will be the one found in the result. If, for some indices, there are no values, those will be left at an initial value to which we assume the *result* vector was initialized beforehand. Alternatively, if no initialization is performed, those values will just be undefined.

function scatter(int n, int m, int $indices[n]$, M $values[n]$, M $result[m]$) **:**
  for $i = 0$ **to** $n-1$ **do**
    $result[indices[i]] = values[i]$;
  end
end

Algorithm 2.1: Sequential scatter algorithm

Virtually all of the operations defined in Section 1.2 have very similar sequential implementations. By just adding new values to the current value of the result vector instead of replacing it (or performing any other operation $\odot$, as mentioned before), a scatter algorithm can be turned into a sequential multireduce algorithm, as shown in Algorithm 2.2. Since the values are now reduced up according to their indices, we called the resulting array *buckets* to better capture their function as containing the reduction of the values put into them. For the

multireduce, the initial values in *buckets* will always influence the final result, even if there are values added to each bucket; it is therefore advisable and common to initialize them to some value beforehand (often the neutral element e).

function multiReduce(int n, int m, int $indices[n]$, M $values[n]$, M $buckets[m]$) **:**
  for $i = 0$ **to** $m - 1$ **do**
    $buckets[i] = e$;
  end
  for $i = 0$ ***to*** $n - 1$ **do**
    $buckets[indices[i]] \odot= values[i]$;
  end
end

Algorithm 2.2: Sequential multireduce algorithm

In order to turn this multireduce algorithm into a multiscan, one only needs to store the intermediate values of the current bucket at every step. For an exclusive multiscan, the current bucket's value is stored in the result *before* the current value is included into the reduction; for an inclusive scan operator is applied to the current bucket value and the new value first. For the complete algorithm for an exclusive multiscan, see Algorithm 2.3.

function multiScan(int n, int m, int $indices[n]$, M $values[n]$, M $result[n]$) **:**
  M $buckets[m]$;
  for $i = 0$ **to** $m - 1$ **do**
    $buckets[i] = e$;
  end
  for $i = 0$ ***to*** $n - 1$ **do**
    $result[i] = buckets[indices[i]]$;
    $buckets[indices[i]] \odot= values[i]$;
  end
end

Algorithm 2.3: Sequential exclusive multiscan algorithm

Not only are these algorithms very simple, they also seem to be optimal (at least if one only considers sequential algorithms). If one takes into account simple optimizations that compilers typically make, like recognizing $indices[i]$ to be available after fetching it from memory once and reusing the available value, these algorithms do not waste computational power on unnecessary operations and solve the problem in a straightforward way. They fetch each input value only once, which is necessary because every input value does actually influence the result, perform only the necessary addition, and write each result value exactly once, which is obviously necessary as well. We therefore have algorithms with linear runtime and can assume the constant to be quite low.

2.2 Measuring algorithm performance

When benchmarking different algorithms throughout this thesis, we will always choose addition as the operation to be performed by the multireduce and multiscan. If the number n of inputs is not stated separately, we will always choose $n = 2^{26}$ for both CPU and GPU benchmarks. This number is large enough to utilize the full capacities of both CPUs and GPUs, but small enough to comfortably fit into main memory and GPU memory, even when additional temporary buffers of length n are needed.
Unless noted otherwise, CPU measurements will be performed on an AMD Opteron 6274 with a maximum clock frequency of 2.2 GHz, 16 kB L1 cache, 2048 kB L2 cache and 6144

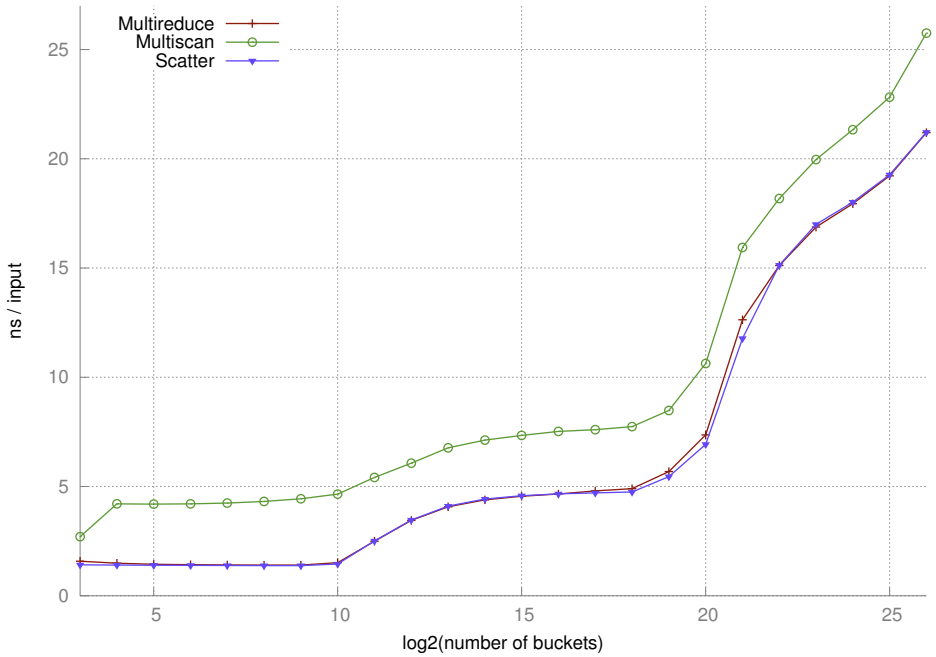


Figure 2.1: Runtime of sequential multireduce, multiscan and scatter

kB L3 cache.
Throughout this thesis, we will measure performance in nanoseconds per input, where a key-value pair constitutes an input for a multireduce or multiscan, whereas an input for a histogramming algorithm would only be one key. This measure is therefore not simply the inverse of the bandwidth per second value, which is often used in GPU benchmarking. We decided to use the time per input measure precisely because it allows us to directly state how much time one algorithm takes per input unit compared to another one, without having to take into account the number of inputs and outputs needed by any such algorithm.
If we run the aforementioned algorithm on this system, we make some interesting observations (see Figure 2.1 for the results). The performance of all three algorithms is generally very similar, however, for low values of m, the multiscan takes more than twice as much time as the scatter and multireduce. We will explain this phenomenon later, for the moment, there is another, more important observation. While the runtime of all three algorithms is relatively low and almost constant for configurations with low values of m, it increases rapidly when m surpasses 2^{19}. This behaviour should not occur if the RAM model was accurate, and it needs to be explained.
Since the pattern is the same for multireduce, multiscan and scatter, we will focus on examining only one of them, the multireduce. We will also use a different (older and less powerful) machine for this purpose for technical reasons (full administrative rights, which are needed for some measurements and for one of the solutions we will propose). This machine has an Intel Core2 Duo CPU, model number T7300, with 2 GHz. Figure 2.2 shows the multireduce algorithm's runtime for a constant number $n = 2^{26}$ of inputs, but with an increasing number m of buckets on the older machine. As before, the *indices* are evenly distributed between 0 and $m - 1$.
While the algorithm performs quite well for a low m (i.e. a small number of buckets), it quickly gets much slower when m gets bigger. More precisely, there seem to be three distinct steps where the algorithm gets significantly slower:

1. The first such step occurs between 2^{12} and 2^{15} buckets; the runtime per integer rises

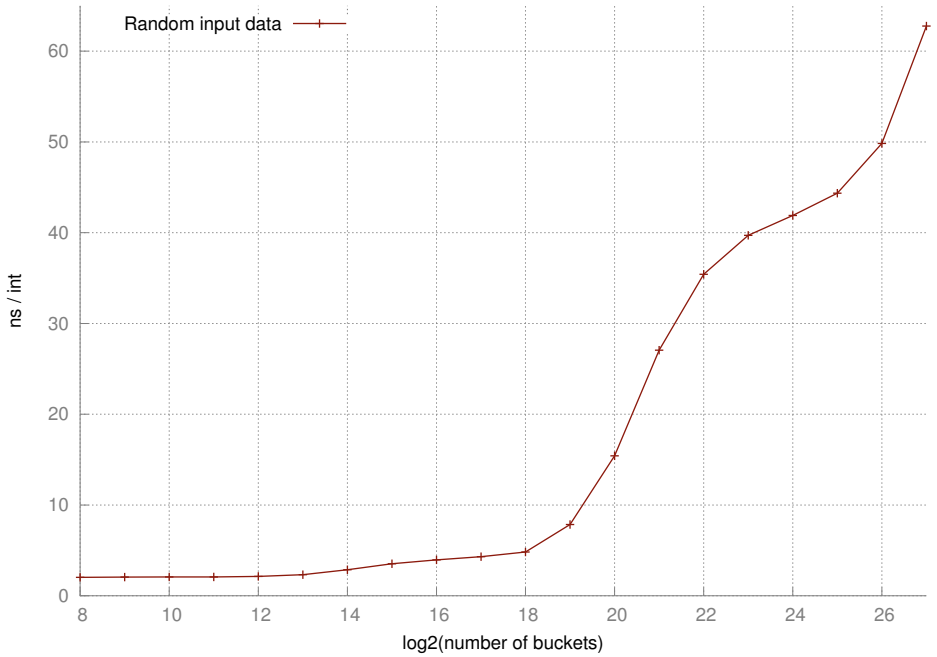


Figure 2.2: Runtime of sequential multireduce algorithm for different numbers of buckets

from slightly above 2 ns to 3.5-4 ns.

2. In the second step, between 2^{18} and 2^{23} buckets, the runtime quickly rises from about 4.5 ns to about 40 ns per integer.

3. While there seems to be a plateau between 2^{23} and 2^{25} buckets, the runtime starts rising very soon again and goes up to 62 ns per integer for 2^{27} buckets.

These three steps together result in a runtime that is more than 30× worse for high numbers of buckets than for low ones, which is a very significant lowering of the algorithm's performance. Note that very high numbers of buckets are not at all artificial for some of the mentioned algorithms. While it is true that, for example, histograms usually use a number of buckets significantly lower than those which lead to problems with this algorithm, others, like scatter (and the closely related gather), have very natural applications where $m = n$, which can result in a very high m because n can, of course, be very large.
Since the RAM model does not predict any of these three distinct slowdowns, the reasons for them must lie in in an area where actual hardware differs from the RAM model in a fundamental way. This is the case for memory access times: while the RAM model assumes constant access time independently of the access patterns, modern computers are optimized to perform well for typical access patterns, at the cost of performing worse for less typical ones. The most important tool used to achieve this goal are caches, which come in many forms. The ones that are the most relevant to our discussion are the caches used to speed up memory accesses in general, and the transaction-lookaside buffer (TLB), a cache used within the virtual address translation system. Since our goal in this chapter is to optimize the performance of the sequential algorithms for multireduce, multiscan and related operations, we must understand where their obvious performance problems stem from. We will therefore discuss the relevant parts of the memory subsystem of current computers, so that we can identify and solve the specific problems which lead to the bad performance of our algorithms.

2.3 CPU memory subsystem

In this section, we will sketch the architecture of the memory subsystem of current computers. We will, however, only explain the parts of the memory subsystem relevant for our problem, i.e. caches in general and the TLB as part of the virtual address translation system in particular. Unless stated otherwise, we will be summarizing the relevant information presented in [36].

2.3.1 Caches

Current computers typically employ a hierarchy of different types of memory with varying size and speed. The goal such a hierarchy is supposed to achieve is to create the illusion of a single, large pool of memory with very high speed. The necessity to design complex hierarchies to create this illusion arises from the fact that faster memory tends to be more expensive (per amount of storage) than slower memory, which means that it is not economically realistic to use large amounts of the fastest available memory in most cases. Additionally, there are some technological considerations: For processors, for example, it makes sense to place some memory on chip, where the access latency can be kept to a minimum. Naturally, this is only an option for very small amounts of memory, which can therefore only be used to speed up execution, not to store significant amounts of data.
For these reasons, current PCs and servers typically use several levels of memory, with the largest having relatively high access latencies, and each level being smaller and faster than the one above. The main reason why this architecture can result in the illusion of one very fast, very large memory in many cases is that memory accesses usually follow a pattern called the principle of locality, of which there are two types:

- *Spatial locality* is the principle that if one memory location is accessed, nearby locations will tend to be accessed soon.
- *Temporal locality* is the principle that if one memory location is accessed, the same location will likely be accessed again soon.

One can exploit these principles to accelerate access to a memory M_1 by using a second layer of memory M_0, which is faster and smaller than M_1, and storing those locations of M_1 in M_0 which are expected to be referenced in the near future. M_0 is the called a *cache* for M_1, and it is of course possible to create several layers of caches such that M_i caches accesses to M_{i+1}, which in turn caches accesses to M_{i+2} etc. Current PCs and server computers usually have at least two, often three levels of cache to speed up accesses to main memory.
M_0 checks if it currently contains a copy of l. Caches always store data in fixed-sized blocks starting at fixed offsets, called *cache lines*, and can store a fixed number of these blocks. Therefore M_0 actually needs to check whether it currently has the cache line which contains l. If a cache is *direct mapped*, then there is exactly one place in the cache where any specific cache line can be stored, meaning that the cache only has to check for entry if it contains the currently requested block. If so, it can directly answer the incoming request. This is called a *cache hit*.
The alternative to direct mapped caches are set associative or fully associative caches. An n-way set associative cache allows any block to be stored in one of n different locations, meaning that n locations may have to be checked to determine if a requested block is currently cached. Usually some kind of simple hash function is used to map memory addresses to locations; a popular option is to simply take a certain range of bits from the address. In a fully associative cache, any block may be stored at any location in the cache, and therefore all locations have to be checked when a request comes in. This check can generally be done in parallel for all relevant locations, but that does, of course, require more complex hardware.

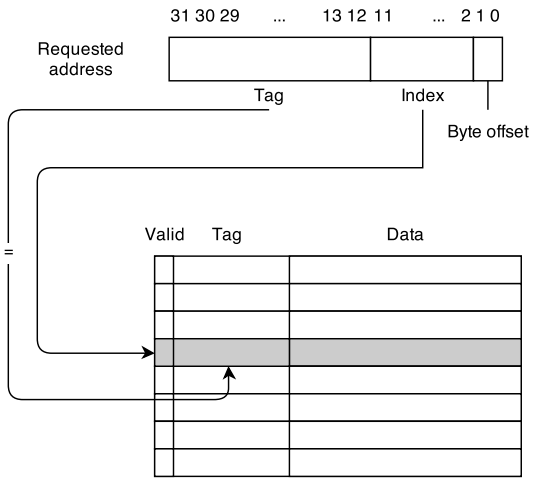


Figure 2.3: Example implementation of a direct mapped cache (adapted from [36])

A possible implementation of a direct mapped cache is shown in Figure 2.3. The requested address is stripped off the bits denoting the position of the requested word within a cache line. A part of the remaining address is then used as the index at which the requested block of data is to be stored in cache. If this part consists of ten bits, as in the example, the cache must therefore be able to hold 2^{10} blocks at once. The rest of the address is used as a tag which is stored along with the cached data. For each request, the tag of the requested address is compared to the tag saved at the given index, and if they are equal, the cache contains the requested location.
If the check determines that the requested location is currently not cached (i.e. a *cache miss*), it sends a request to the M_1 (which may be another cache or main memory, but is in any case slower than M_0). When it gets the requested cache line, it has to determine where to store it. In case of a direct mapped cache, there is only one possibility; for set-associative and fully associative caches, there are several, from which one has to be chosen. A commonly used scheme for selecting a cache line is LRU (least recently used), which selects the entry which whose last access is the least recent one, but there are other schemes, including many fast approximations of LRU. After selecting a location, the incoming data is stored at said position and the original request is answered.
It may also be necessary to store the data which was previously occupying the selected location in higher level memory depending on the used write policy: A *write through* cache will forward all write requests to higher level memory in addition to writing the value in cache as well, meaning that the data in held in cache and that in higher level memory are always consistent. In contrast to that, a *write back* cache will only perform writes in cache. This means that, when a cache line is evicted from cache because another block of data takes its place, data which has been changed in cache needs to be written back to higher level memory, which in turn means that cache needs to store an additional bit for every location which indicates if the data at this location has been changed and needs to be stored again on eviction of the cache line.
Current caches have some additional functionality in addition to the ones mentioned so far. An important aspect is that they try to detect simple memory access patterns in order to predict which locations will be accessed next, and speculatively *prefetch* those locations to cache. This presupposes, of course, that data accesses indeed follow certain patterns and that these are simple enough for the cache to detect. Techniques for doing this exist both for sequential accesses as well as for strided accesses (see for example [46] for an overview),

but more complicated access patterns will likely not be prefetched (correctly) on current hardware.

2.3.2 Virtual address translation

An additional task of the memory subsystem is the translation between virtual and physical memory addresses. Current computers use virtual address spaces which are usually larger (but can also be smaller) than the address spaces of the used hardware. Their main purposes are to prevent different processes from interfering with each other's memory by giving each process its own address space, and to allow them to address more memory than is physically available in main memory.
In order to do this, virtual memory is divided into pages. Each page may either be in main memory or stored to some other medium like a harddrive at any time. Virtual memory addresses then consist of a page number and a page offset. Each process uses its virtual memory address space as if it were actual physical memory. When a page is accessed, the memory management unit needs to translate the virtual address to a physical one. For this purpose, each running process has its own *page table*, which stores the physical address for each page currently in memory. If a requested page is not in main memory, but stored on disk, a *page fault* is generated and the page is loaded into memory. The page table for each process is stored in main memory.
Since this translation has to occur for every single memory access, quick access to the page table is vital for good performance. It therefore makes sense to cache the physical addresses of the most recently accessed pages. This is done by the transaction-lookaside buffer (TLB), which works just like a cache for main memory in most ways. Again, there are various possibilities for the associativity of a TLB, and the address-to-cache-location mapping outlined above can also be used with page numbers in the TLB. Another similarity of main memory caches and TLBs is that current systems often have several levels of them. A request for a certain page's address can then either result in a *TLB hit* or a *TLB miss*. In the latter case, the page's physical address (if any) must be retrieved either from the next level TLB or, if there is no next level, from the page table in a process called a page walk.
There are various ways to implement the page walk and, in fact, many ways to design the page tables themselves (e.g. in form of multiple levels [3]), which we will not discuss here as they are not relevant to our further course of action. More information on both caches and virtual memory can for example be found in [36].
The main point to take away from this is that there is another level, in addition to main memory caches, which makes use of locality to accelerate memory accesses, meaning that violating the locality principle can entail a performance penalty. There are therefore at least two significant differences between real hardware and the RAM model, which assumes constant access time to any memory location at all times.

2.4 Improving the naive algorithm

Knowing this, we can hypothesize that the first two performance drops are the results of L1 and L2 cache misses. This is a very common phenomenon, and we will verify if this is indeed the problem and then try to solve it at a later time.
First, we will deal with the third drop in performance, which is much more mysterious, since it starts to have a large impact when $m \geq 2^{25}$ and therefore 2^{27} bytes are needed for storing buckets in memory, a size which is too large to fit into any currently used cache. We hypothesize that this drop stems from an increasing number of TLB misses.
On the test system, a normal page has a size of 4 kB and can therefore hold 1024 4-byte integer values. This means that the buckets placed in main memory by the multireduce

$\log_2 m$	Miss rate
16	0.027%
18	0.356%
20	14.163%
22	18.858%
24	20.697%
26	27.252%
28	26.902%

Table 2.1: TLB load miss rate for simple multireduce for different numbers of buckets (m)

algorithm are distributed over at least $\lfloor \frac{m}{1024} \rfloor$ pages. For $m = 2^{25}$, this means that the page table contains 2^{15} entries just for the buckets. Reliable information about the details of current processor's TLB sizes is rare, however, one source [22] suggests that the data TLB of the test system can hold 256 entries. Based on this, we would expect a performance drop starting around $m = 2^{18}$, since at this point the buckets would be distributed over 256 pages and would therefore completely fill the TLB. For higher choices of m, there will likely be a TLB miss for every bucket access.
Since each iteration of the loop requires fetching a label, a value and the current value of a bucket, and one of these requires a random access, we would expect a TLB load miss rate of one third. There is one store per iteration, but it is to a location which has just been fetched, so that TLB misses for stores should not be a problem for this algorithm.
If we measure the data TLB load miss rate for different values of m, we get the data in Table 2.1. Miss rates do indeed rise to almost one third of all loads above $m = 2^{26}$, which suggests that third drop in performance really does stem from TLB misses. While our prediction that the miss rate should rise rapidly for $m > 2^{18}$ is confirmed by the data, there is one curiosity: The miss rate remains around 27% even for higher numbers of buckets, but we have observed that performance keeps dropping for higher numbers of m. A possible explanation for this is that the cost of a page walk increases when even more pages are used. Since modern CPUs use additional caches for storing page table entries in addition to L2 cache [3], the reason for this may well be more internal cache misses during the page walk. Since there is little to no official information on the specifications of such caches in actual hardware, however, we cannot test this assumption.
A way to avoid TLB misses is the use of larger pages (called huge pages in Linux-based operating systems), which have a size of 2048 kB instead of the normal 4 kB and therefore reduce the number of necessary page table and TLB entries for a given amount of data. Figure 2.4 shows that the third performance drop disappears when huge pages are used, which both confirms that TLB misses are the reason for said drop, and shows a way to avoid this problem.
Since we have explained the third drop in performance, we will now go back to the first two drops. Increasing m means increasing the number of buckets that may potentially be accessed during each step of the algorithm. A natural hypothesis is therefore that a drop in performance occurs each time a cache is no longer big enough to fit all the needed data, which means that data has to be stored in or retrieved from the next level of memory, which has higher access latency. Additionally, a miss in one cache means that an entire cache line needs to be fetched from the next level, not just the value that was actually requested. If, for very large choices of m, a lot of accesses result in cache misses, this means that the available memory bandwidth is not used efficiently to fetch the data that is needed, but is mostly wasted for data which will not be used.
Note that this problem of overflowing caches only occurs when we increase m, but not when

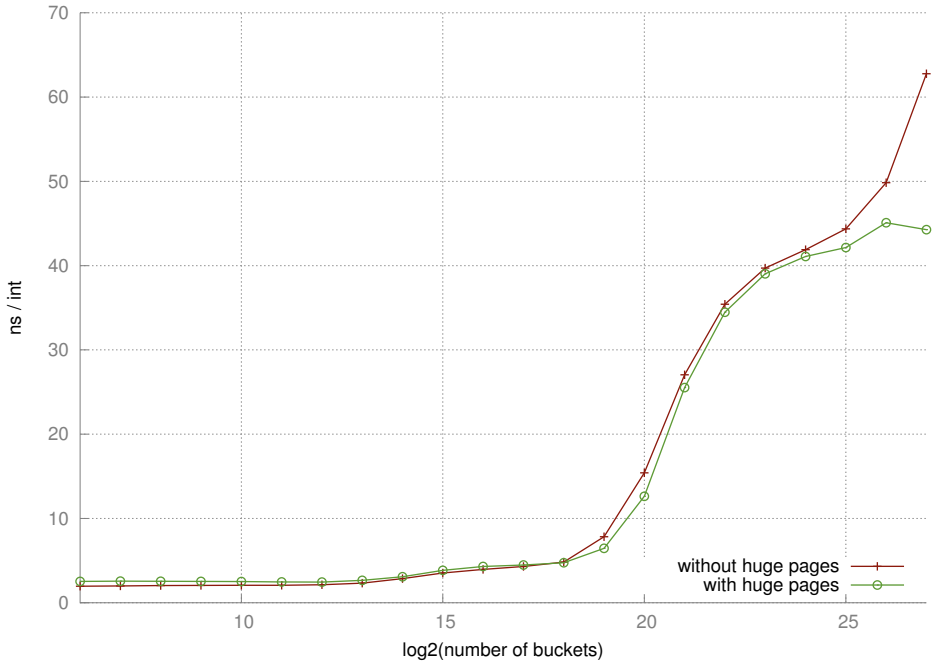


Figure 2.4: Runtime of sequential multireduce algorithm with and without huge pages

we increase n: The vectors of length n, no matter how large they are, are always accessed sequentially. The cache therefore only ever needs to prefetch the next value of all such vectors, no matter how large the actual vectors are. The buckets, however, are *not* accessed sequentially. At every iteration of each of the algorithms, any one of m buckets can be read from and/or written to.

The machine on which the performance data was gathered has an L1 data cache of 32 kB, or 2^{13} 32-bit integers. If we take into account that the buckets are not everything that needs to be kept in cache (the next indices and values need to be prefetched into L1 cache as well, for example), we would expect that all buckets fit into L1 cache up until 2^{12} buckets, and that performance drops from that point onwards. This is, indeed, where the first drop in performance occurs.

The L2 cache on the employed machine, which is used as both data and instruction cache, has a size of 4096 kB. It could, in theory, contain 2^{20} 32-bit integers. However, again, the buckets are not everything that needs to be cached. In this case, we also need to keep in mind that this cache is used for instructions as well, and since the cache is, of course, not fully associative, the available space may not always be perfectly used. So it seems sensible that the L2 cache would work fine up until 2^{18} buckets, but be too small for bigger values of m, as we observe in the actual data.

We would then expect a noticable, but relatively small difference in performance between the sizes that fit into L1 and L2 cache, respectively, and a much bigger difference between sizes that fit into L2 cache and those that don't, since for the latter, the algorithm would constantly have to access completely uncached main memory, which is much slower than both caches.

We can test the hypothesis that cache misses are the main cause for the performance decrease for high choices of m by testing the same algorithm with non-random data (see Figure 2.5 for results). As we predicted, the performance in this case stays almost constant, regardless of the choice of m. The first two drops in performance completely disappear with sequential input data, and so does, in fact, the third one (even without using huge pages).

We will try to solve the problem of cache misses by using a sorting approach to make the

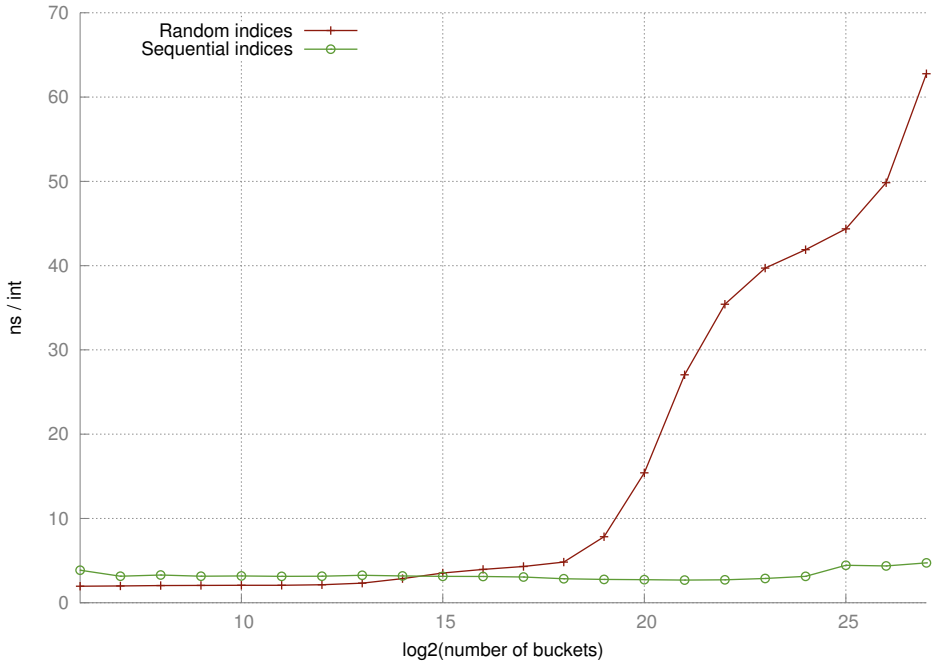


Figure 2.5: Runtime of sequential multireduce with random and sequential input labels

input to the actual multireduce algorithm more closely resemble sequential input data, which, as shown before, leads to optimal performance. The idea of sorting data in order to avoid slow random access is not new and has been used in other areas before. One example are disk scheduling algorithms, which try to minimize the time needed to serve disk reading and writing requests. The elevator algorithm achieves this by sorting incoming requests by their locations, and then serving them all in a single sweep over the disk [9].
For our algorithm, we will use radix sort, a stable sorting algorithm with a runtime complexity of $\mathcal{O}(dn)$ for n inputs consisting of d bits. In contrast to comparison-based sorting algorithms, radix sort can only be used to sort values stored in positional notation, which is the case for the binary representation typically used for (unsigned) integer values. The idea behind it is that a vector of numbers can be sorted by first sorting it according to the least significant digits, then repeating the process on the resulting vector for each of the remaining digits from least significant to most significant. While radix sort has slightly worse locality than other sorting algorithms [24], it allows us to sort the input data only partially, thereby saving computation time compared to a full sort. This is significant because the sorting process must, of course, take less time than is gained through the subsequent avoidance of cache misses in order to be useful.
Radix sort allows us to partly sort the input labels and values by the labels, resulting in partly ordered input vectors: If the input has $m = 2^d$ buckets, we choose some number $b \leq d$ of bits by which we sort. We then calculate an offset $o = d - b$, and sort by b bits, starting with the oth least significant bit. This will save computation time compared to doing a full sort of all 32 bits of an integer, but it saves even more, by not completely sorting the input labels. Instead, this approach will result in continuous sections of labels which differ at most by 2^o. More formally, for every segment s there exists a value k_s such that for every label l_i in s, $k_s \leq l_i < k_s + 2^o$. If we choose b in such a way that 2^o buckets fit in cache, then all buckets needed by a specific segment can be fetched to cache once the segment starts, and all subsequent update of buckets in the segment will very likely hit the cache. A simple radix sort algorithm is shown in Algorithm 2.4; an illustration of a first round of a radix sort can be seen in Figure 2.6.

```
function getOffsets(int n, int base, int keys[n], int buckets[base], int offset, int run) :
    for i = 0 to base − 1 do
        buckets[i] = 0;
    end
    for i = 0 to n − 1 do
        int currentBucket = keys[i]/(run × base × 2^offset) mod base;
        buckets[currentBucket]++;
    end
    // scan bucket values
    int sum = 0;
    for i = 0 to base − 1 do
        int currentValue = buckets[i];
        buckets[i] = sum;
        sum += currentValue;
    end
end

function radix(int base, int n, int buckets[base], int keys[n], int offset, int run, int
resultKeys[n], M values[n], M resultValues[n]) :
    for i = 0 to n − 1 do
        int currentBucket = keys[i]/(run × base × 2^offset) mod base;
        int index = buckets[currentBucket];
        resultKeys[index] = keys[i];
        resultValues[index] = values[i];
        buckets[currentBucket]++;
    end
end

function radixSort(int indices[n], M values[n], int runs, int base, int o, int n) :
    int resultKeys[n];
    M resultValues[n];
    for run = 0 to runs − 1 do
        if run > 1 then
            resultKeys := keys;
            resultValues := values;
        end
        M buckets[base];
        getOffsets(n, base, keys, buckets, offset, run);
        radix(base, n, buckets, keys, offset, run, resultKeys, values, resultValues);
    end
end
```

Algorithm 2.4: Radix sort algorithm which sorts by b bits in $runs$ rounds if $runs \times base = 2^b$, with an offset of o bits

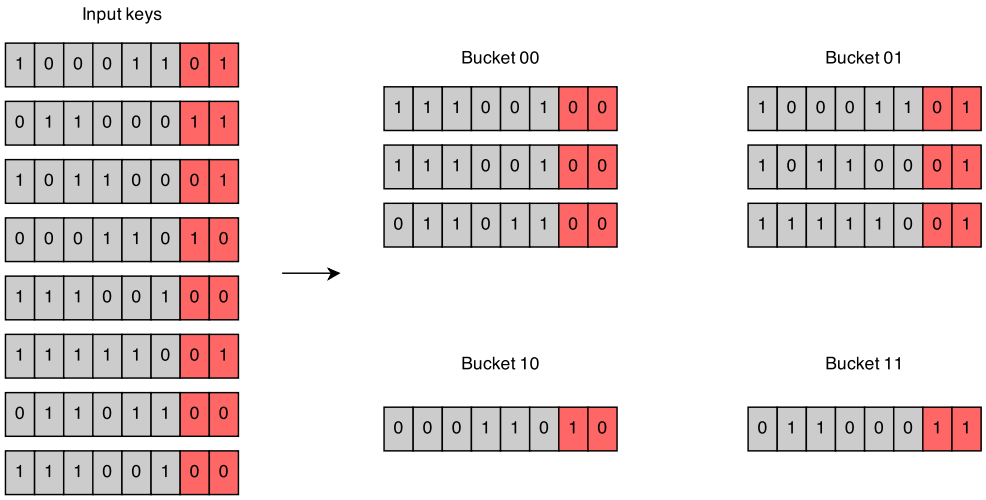


Figure 2.6: Illustration of the first round of radix sort with a base of 2 and an offset o of zero

In order to speed up the multireduce algorithm using a radix sort, all we have to do is find an appropriate number of bits to sort by, sort the input labels and values, and then apply the multireduce function to the partially sorted labels and values. For scatter, the procedure is exactly the same, since it is a special case of the multireduce. For the multiscan and the gather operation, however, the algorithm needs to be a little more complicated as shown in Algorithm 2.5 and 2.6.

```
struct IndexValue {
  int index;
  M value;
};
```

function sortMultiScan(int *indices*[*n*], *M values*[*n*], *M result*[*n*], int *runs*, int *base*, int *n*)
:
  IndexValue *ivalues*[*n*];
  for $i = 0$ **to** $n - 1$ **do**
    $ivalues[i].index = i$;
    $ivalues[i].value = values[i]$;
  end
  int *resultKeys*[*n*];
  IndexValue *resultValues*[*n*];
  for $run = 0$ **to** $runs - 1$ **do**
    if $run > 1$ **then**
      $resultKeys := keys$;
      $resultValues := values$;
    end
    M buckets[*base*];
    radixSort(*indices*, *values*[*n*], *runs*, *base*, *n*)
  end
  multiScan(*resultKeys*, *resultValues*, *result*, *n*);
end

Algorithm 2.5: Sorting algorithm for multiscan

The performance of a sort-based multireduce compared to the simple implementation ob-

```
function sortGather(int indices[m], M values[n], int runs, int base, int n, int
resultKeys[n], M resultValues[n]) :
    M buckets[base];
    M bucketsCopy[base];
    int temp[n];
    getOffsets(n, base, keys, buckets, offset, run);
    buckets := bucketsCopy;
    radix(buckets, keys, o, run, n, resultKeys, values, resultValues);
    for i = 0 to n − 1 do
        temp[i] = values[temp[i]];
    end
    for i = 0 to n − 1 do
        int currentBucket = keys[i]/(run × base × 2^offset) mod base;
        result[i] = temp[bucketsCopy[currentBucket]];
        bucketsCopy[currentBucket]++;
    end
end
```

Algorithm 2.6: Gather with one-run radix sort

viously depends to a great degree on the properties of the memory subsystem of the used hardware, especially the size and latency of caches and the main memory. In order to evaluate its applicability for various hardware configurations, we implemented a simulator for cache and memory behaviour, which can be configured to simulate arbitrarily large hierarchies of write-back caches with different sizes and associativities, given the access times for each one of them. It uses a simple approximated LRU replacement mechanism and performs rudimentary prefetching.

For the test system, we created a pool of pointers, each pointing to a random pointer in the pool, and measured the average time needed for following one pointer to another (after a warmup period which allows the pool to be cached). If the pool fits in L1 cache, this measures the latency of L1 cache; if it is far larger than L1 cache but fits into L2, it will measure the latency of the latter, and so on.

The results of the simulation, as well as the actual performance of both the original approach and the sorting approach, can be seen in Figure 2.7. We configured the sorting algorithm to sort the input data if $m < 2^{21}$, and sort by $\log_2 m - 18$ bits. These parameters were empirically chosen and result in the best performance on the test hardware; for lower choices of m, the time needed for sorting cancels out all gains made in the subsequent multireduce computation.

In terms of actual performance, the results show that the sorting approach does indeed result in better performance for many buckets. The difference, however, is not a qualitative one, as the general pattern of the curve remains the same. At its best, the sorting algorithm needs two thirds of the time of the original algorithm.

The simulation results, however, obviously do not agree with the real life performance, apart from the most basic pattern. The simulation predicts that both the simple algorithm and the sorting algorithm should perform much worse than they actually do. It also predicts a non-monotonic performance curve of the sorting algorithm: According to the simulator, the sorting algorithm should perform better for $m = 2^{21}$ (the lowest m for which sorting is performed) than for $m = 2^{20}$ (the highest m without sorting). That means that according to the simulator, we should already sort for $m = 2^{20}$, whereas in reality, this actually reduces performance. We will have to find out if the simulator is simply inaccurate or if something unexpected is happening in the CPU or in memory.

A sign that suggests that the latter might be the case is the fact that even in the worst case,

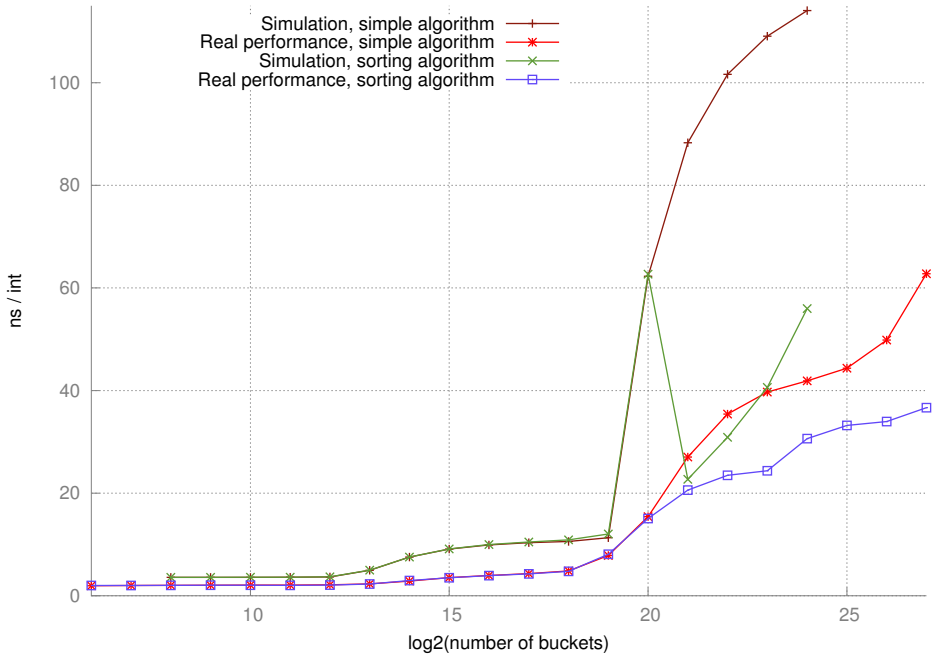


Figure 2.7: Actual runtime vs. simulation results of simple and sorting multireduce

the runtime of the simple multireduce algorithm per input is lower than the time needed for a single uncached main memory access (63 ns vs. 111 ns). Since we expect one uncached main memory read for almost every loop iteration for high values of m, this should be impossible, and we would in fact expect the runtime to rise over 111 ns per input, as the simulator predicts.

We found that the most likely reason for this behaviour is that, on real hardware, memory accesses are not performed completely sequentially, but are pipelined. This means that while the CPU is still waiting for the bucket value in one loop iteration, it already requests the value of the next one, thus reducing the average latency. Since we gathered the input latencies using pointer jumping, which is necessarily sequential, this would explain the difference. In order to test this assumption, we have manipulated the multireduce algorithm to force sequential execution of the bucket reads. We do this by letting the choice of bucket in each iteration depend on the value of the bucket read in the previous iteration, so that it is impossible for the CPU to request the next bucket while it is still waiting for the value of the previous one. Figure 2.7 shows that this change does indeed affect performance in a major way.

The simulation is now much closer to the real performance. The main difference is that the simulated performance drops more abruptly when L2 cache can no longer hold all buckets. A possible explanation for this is that the cache line replacement algorithm or the prefetching algorithm used by the real CPU are different from the relatively simple ones used by the simulator. Since LRU leads to bad cache hit rates for some very common access patterns, current CPUs usually utilize more sophisticated replacement policies [38], so that we should expect small differences between real performance and a simulation using LRU.

Note that with enforced sequential data fetching, the real performance of the sorting algorithm also has a bend at $m = 2^{20}$, meaning that *if* data fetching was indeed sequential, we should in fact sort for $m = 2^{20}$ instead of $m = 2^{21}$, as the simulation suggests. This shows that the simulator works accurately enough to find optimal configurations for the sorting algorithms if the hardware it simulates does not violate the simulator's assumption that all memory accesses are carried out sequentially.

We can now go back to our primary goal for this chapter, which is to find an optimized

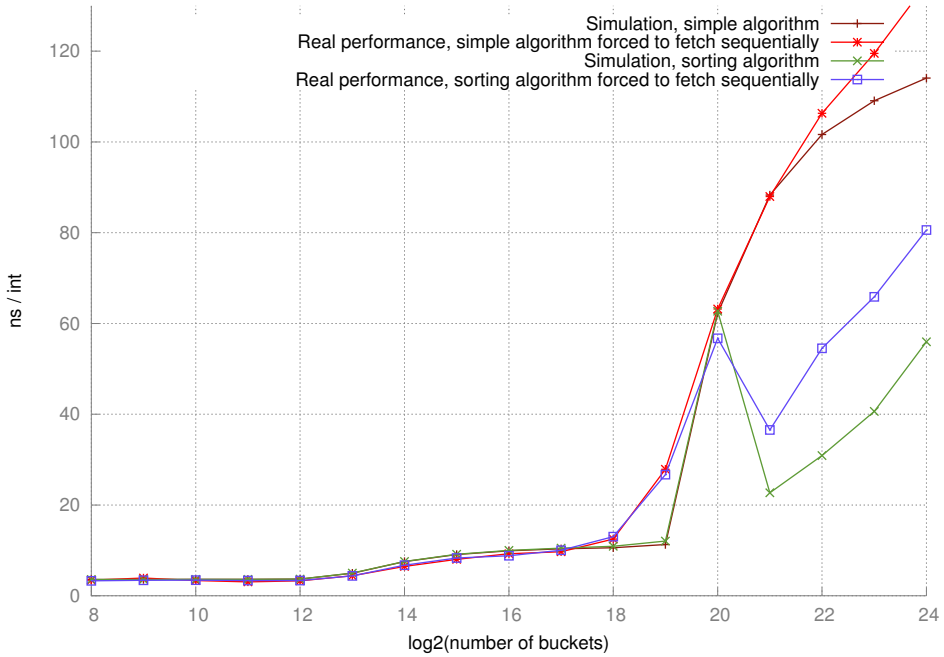


Figure 2.8: Actual runtime vs. simulation results of simple and sorting multireduce when sequential data fetching is enforced

sequential algorithm for multiscan and multireduce. We have shown that the sorting approach can lead to a speedup for the multireduce on our older test system. However, we also expect the relative performance of the simple and the sorting approach to vary between systems with different cache hierarchies, and the system we have used for testing so far is too old to serve as a good representative of current CPUs. We have therefore benchmarked the sorting approach on the newer Opteron system as well. The results showed that on this system, which has more and bigger caches, the sorting approach does not actually result in a speedup for any configuration, neither for the multireduce nor for any of the related operations. While we have shown that the sorting approach can be useful on certain CPUs, we must conclude that its usefulness will always depend on the employed system.

In terms of our search for an optimal algorithm for the multireduce and multiscan, this means that the simple algorithm we started out with is indeed the fastest algorithm on our best system, and we will use this algorithm to evaluate the performance of GPU algorithms in the next chapters.

When first benchmarking the performance of multireduce, multiscan and scatter, we noticed that the best-case time for calculating the multiscan is almost twice as long as that needed for the multireduce, which performs exactly the same calculations but fewer memory transactions. Since, as we now know, all buckets fit into L1 cache in the best case, and sequential reads from the input arrays can be assumed to be cached in L1 as well, the reason for this cannot be cache behaviour. This leads us to conclude that memory bandwidth is the limiting factor for the sequential performance of both algorithms; an assumption which we confirmed by other benchmarks.

As a result of this, multi-core algorithms for the same problems cannot lead to a significant performance increase, as long as the available bandwidth does not increase when several cores are used, which is only the case if the different cores are physically placed on different sockets, which is not the case for most CPUs. Since any multi-core CPU algorithm would therefore never result in an improvement based on actual algorithmic differences, we will not consider parallel CPU algorithms in this thesis.

2.5 Conclusion

We have shown that multireduce and similar algorithms suffer from poor cache hit rates and TLB misses for large numbers of buckets. While using huge pages can prevent the latter, a more general solution is to partially sort the input data to make it more closely resemble sequential input data. Our benchmark shows that sorting can, in fact, lead to an increase in performance on certain systems with small caches, but the improvement is highly dependent on the used CPU. One can easily imagine scenarios where the expected improvement of such an approach is much larger, e.g. when working on external media, or in fact any other data source which has significantly lower latencies for sequential access patterns and/or frequently used data. For these cases, the existing simulator can be used to estimate if and how much a sorting approach can improve performance.
In addition to being useful for performance reasons, the algorithm can also be used to increase the lifetime of certain types of external media: SSDs, which can only handle a limited number of writes to each cell, suffer much more from random access writes than from sequential ones.
A natural topic for follow-on work would be to extend the simulator to account for the superior possibilities offered by modern memory and CPUs, i.e. to properly model pipelined main memory reads and writes, and to use more sophisticated cache management routines. Such a simulator should be suitably parameterized in order to be able to accurately simulate both current and older memory architectures as well as special cases like disks.
Since we concluded that memory bandwidth is the main factor limiting the performance of multireduce and related algorithms, no improvement can be expected from parallel CPU algorithms, and we will therefore not pursue them in this thesis.
Concerning our search for the optimal sequential algorithm for multireduce and multiscan, our conclusion is that the simple algorithms are in fact the fastest ones for current CPUs with large caches. These will therefore be the algorithms to which we will compare the GPU algorithms to be developed throughout the next chapters.

Chapter 3

GPU Fundamentals

The remaining chapters of this thesis will be devoted to developing fast algorithms for multi-reduce and multiscan for GPUs. Compared to algorithm design for CPUs, GPU performance depends on special hardware characteristics to a much larger degree. In order to have an informed discussion of specific algorithms in the following chapters, it is therefore vital to first introduce the main concepts used in GPU programming, and then elaborate on the preconditions for efficient use of GPU resources. This is the main focus of this chapter.

3.1 Modern GPU architecture and the CUDA programming model

The general idea behind the design of GPUs has already been highlighted in Section 1.1, but a more in-depth knowledge of the actual hardware is necessary in order to understand the conditions under which code can be executed efficiently on GPUs. While explaining these details, and in fact throughout the remainder of this thesis, we will be using the terminology that is generally employed in CUDA programming as opposed to other frameworks like OpenCL. However, since they are both intended to be able to be used on the same hardware, most of the core concepts are identical or at least very similar, and are simply named differently. We will try to keep the hardware descriptions as general as possible, but some digressions about specifics of the current generation of GPUs may be necessary. In these cases, we will presuppose the traits of current NVIDIA GPUs, but we will make it clear to the reader that these things may change in the future. Additionally, we will only discuss the features of CUDA relevant to our later algorithm discussion. A more complete overview can be found in the CUDA C Programming Guide [13], which the following discussion is based on.

Each CUDA GPU consists of several streaming multiprocessors (SMs) (see Figure 3.1). An SM, in turn, contains a number of scalar processors (SPs). As mentioned before, groups of these processors share a common control logic and can therefore only execute the same instruction at the same time. An SM also contains registers for the SPs to work with. Each SM has a small amount of shared memory which can be used cooperatively by the SMs' SPs and can also be used as L1 cache, as well as a constant cache (abbreviated to CC in Figure 3.1) and a read-only data cache. The functions of the L1 cache will be discussed in detail at a later point; the constant cache and read-only data cache do not play a role for our purposes and are only mentioned for completeness. All streaming multiprocessors have access to the global DRAM, which is cached by a shared L2 cache. L1 cache does *not* cache global memory reads and writes, since it is private to each SM and would have to handle cache inconsistencies otherwise.

The structures offered by CUDA on a programming level generally mirror the structure of the hardware. In Figure 3.2, each of the main structures has the same color as the part of

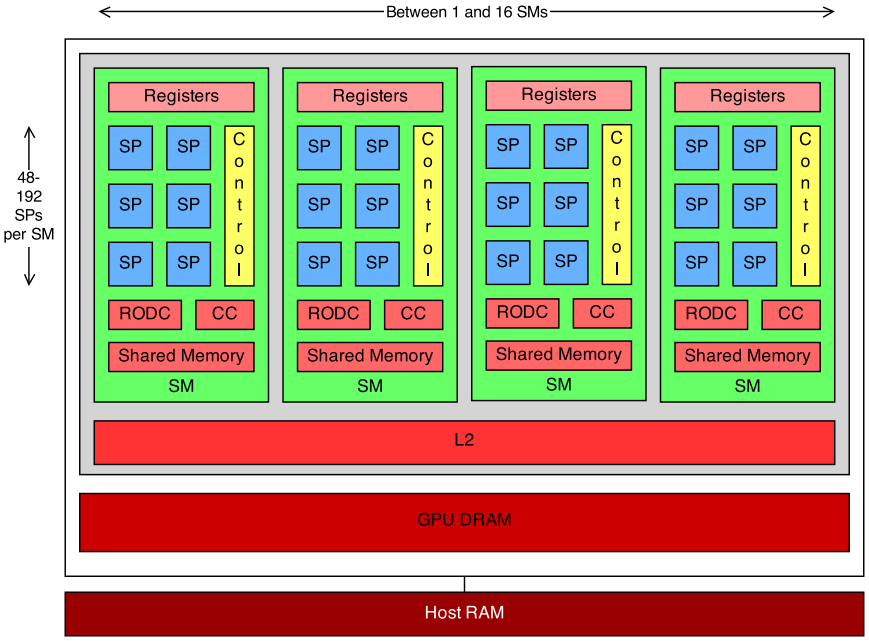


Figure 3.1: Structure of modern CUDA-enabled GPUs (adapted from [41])

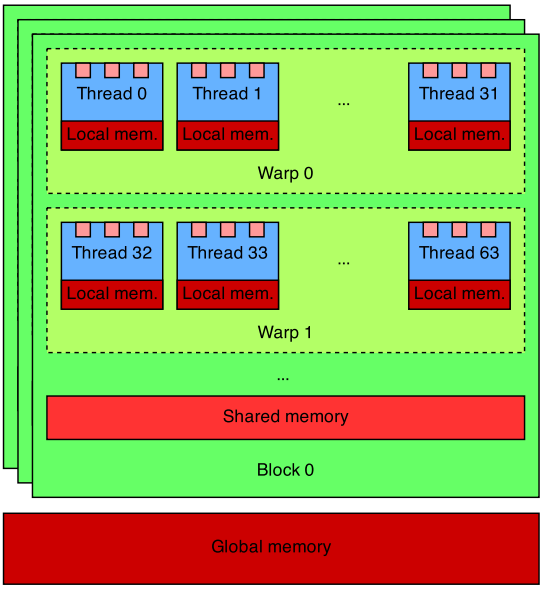


Figure 3.2: CUDA programming model

the GPU hardware it corresponds to in Figure 3.1.
On the lowest level, there are single threads, each with a specific ID, which execute instructions as specified by the programmer. A thread has two ways to store its private data during execution. The first one are registers, which are allocated to threads according to their needs. Registers are the fastest type of memory available, with very low latency, although there is a certain latency after storing a value in a register before said value can be read again. If a thread needs more private space than the available registers can offer, local memory is used instead. Local memory is physically placed in the GPU's DRAM and therefore has a high latency. Local memory accesses are, however, cached in L1 cache; since the data is private to each thread, there is no problem with cache incoherences. Another case where local memory is used instead of registers is for arrays which are indirectly addressed, i.e. arrays which cannot be transformed into a number of normal variables by the compiler due to indirect addressing.
CUDA code is written on a thread level. A function written for execution on a GPU is called a *kernel*, and on invocation of a kernel, all threads execute the same kernel function. The behaviour of different threads can be distinguished based on their thread ID, which is privately known to each thread.
Threads are organized in blocks. All threads in a block have access to a shared memory. From a hardware point of view, L1 cache and shared memory are identical (and simply called shared memory in Figure 3.1), they have identical performance which is not much worse than that of registers, though the official documentation makes no guarantees in terms of latency times. The programmer can specify the amount of said memory that should be used as shared memory as opposed to L1 cache for each kernel before it is called.
While shared memory allows threads of the *same* block to cooperate, variables in global memory can be accessed by threads of *all* blocks. They are stored in off-chip DRAM. As mentioned before, this DRAM has high latency, but it also has high throughput. Accesses to global memory are cached in L2 cache *only*. There is even less information about the performance of L2 cache than in the case of L1/shared memory. Since there is only one global L2 cache, as opposed to dedicated L1/shared memory for every SM, one can expect the L2 performance to be a lot worse than L1, although it must still be a lot faster than DRAM, since there would be no reason for it to exist otherwise. Global memory is where the input data for each kernel is initially found, and it is where any results must be written if they are supposed to be used after the kernel has finished. Since global memory is the only means of communication between the GPU and the host, CUDA offers special functions for copying data between the computer's main RAM and the GPU's DRAM, which can then be accessed through loads from global memory.
When writing CUDA code, the programmer has to write the kernel code (e.g. the code every thread should execute) and, in the kernel launch call, specify the number of blocks to be executed, as well as the number of threads in each block. During execution, blocks are then allocated to SMs. This means that all the threads of the blocks allocated to a given SM need to share the registers available on that SM. The state of each thread is maintained on an SM for the entire lifetime of its block, which allows for fast switching between threads. The number of registers used by the threads of a block therefore limits the number of resident blocks on any SM at any given moment. The same is true for shared memory: If blocks use an amount s of shared memory, then only $\lceil \frac{shared_{SM}}{s} \rceil$ blocks can be resident on an SM at any time, where $shared_{SM}$ is the amount of shared memory per SM. When the execution of a block has finished, a new block can then take its place.
In terms of actual execution, blocks are further divided into warps. On current hardware, a warp consists of 32 threads of the same block. The assignment of threads to warps is based on the thread ID, and since the latter does not change, each thread will belong to the same warp during its entire lifetime. During each cycle, an SM's scheduler will select one (or, on modern

hardware, several) active warps to be executed. A warp is active if its next instruction can be executed during the current cycle, meaning that the warp is not, for example, waiting for data from DRAM to come in. The SPs of an SM will then concurrently execute the same instruction for all the threads of a warp. On a warp level, GPUs are therefore strictly SIMD machines. Different warps, however, may execute different instructions (e.g. by branching on the thread ID). The same is, of course, true for different blocks. As a whole, GPUs therefore do not adhere to the SIMD principle. Instead, NVIDIA calls the model of parallelism used by GPUs SIMT (single instruction, multiple threads).
The SIMT model has direct consequences for the performance of branch execution. Different warps can execute different branches without any loss of performance. If, however, different branches are executed within a warp (a phenomenon called *branch divergence*), only one branch can be executed at a time. The SPs responsible for executing threads that do not take part in the current branch will then be deactivated, and the remaining SPs execute the branch. The execution of different branches is serialized. This obviously has consequences for performance: Not only does a number of the available SPs necessarily lay idle for some time if there is branch divergence within a warp, but the total execution time will be the time needed for executing all all branches taken by at least one thread sequentially.
As we have seen, the number of threads which can actually be executed at once by an SM is only a small multiple of the warp size. Nevertheless, programmers should strive to place several hundred threads on each SM at once (either by using many blocks, or by using blocks with many threads). The reason for this is a central concept called *latency hiding*. As mentioned before, the latency of global memory accesses is hundreds of cycles (again, detailed information is unavailable) and cannot be completely hidden by caches. Instead, CUDA GPUs achieve maximum performance by having a high number of resident threads, and therefore a high likelihood that enough warps will be active at the start of every cycle. Since all resident threads keep their data stored in the SM's registers during the lifetime of a block, switching between different warps has no additional costs.
The order in which warps are scheduled to be executed is generally undefined, as is the order in which blocks are allocated to be executed on SMs. If a kernel's correctness depends on some instruction having been executed by all threads in a block at some point, this block's threads need to be synchronized, for which CUDA offers a primitive. Note that synchronization is only necessary and possible on a block level: On a warp level, there is no need to synchronize, since warps are strictly SIMD. On a global level, however, synchronization may be desirable in many cases, but CUDA does not offer any way to synchronize execution between more than one block.
For cases where several threads or blocks need to cooperate on the same data, CUDA offers atomic updates for both global and shared memory. If there are conflicting atomic updates to the same location, these updates will be serialized in an undefined order. While atomic updates are slower than normal writes, their performance has improved dramatically on recent GPUs, thus making them viable for use even for high performance algorithms [33]. On current GPUs, atomic operations are also available in shared memory.

3.1.1 GPU performance requirements

The aforementioned information usually suffices to be able to write correct code. In order to maximally utilize the GPU's performance, however, some additional aspects need to be taken into consideration. Again, we will only give an overview of the information needed for the discussion of our algorithms; a more complete overview can be found in [34]. The first is that the limiting factor for GPU algorithm performance is often a different one than for corresponding CPU algorithms. While GPUs currently have a significantly higher memory bandwidth than CPUs (see Figure 3.3), the difference is not as big as it is for pure computing power (a factor of about seven for memory bandwidth, 10-20 for computing power). The

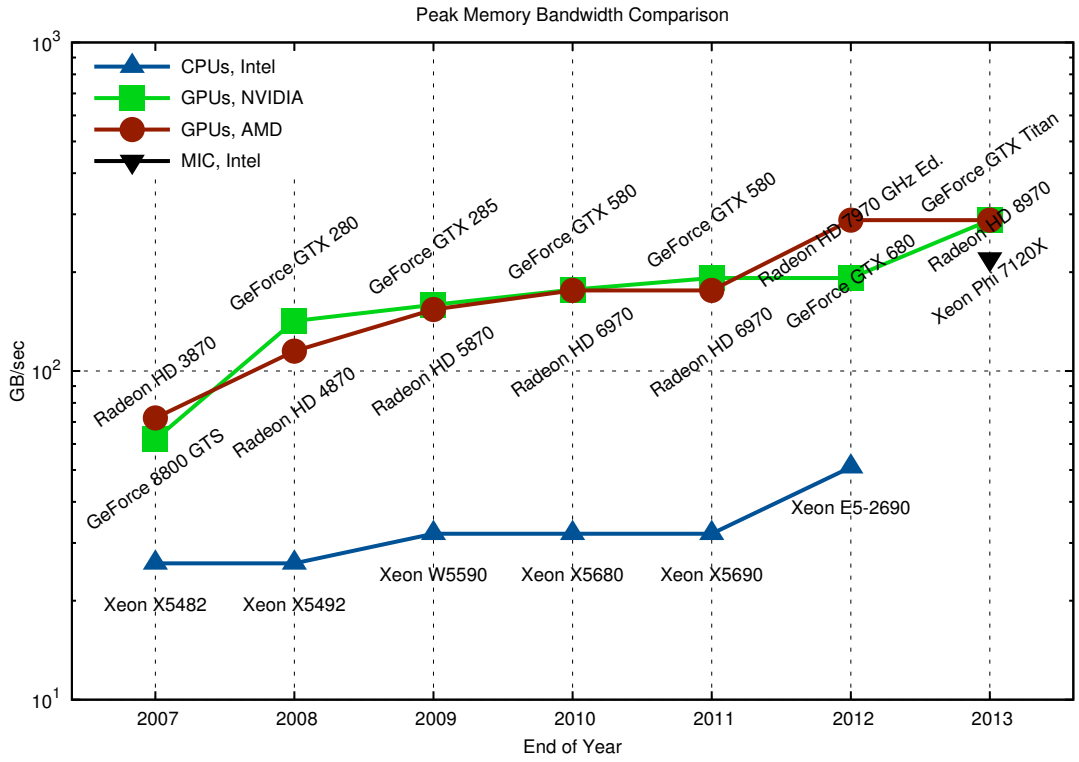


Figure 3.3: Memory bandwidth comparison of CPUs and GPUs [40]

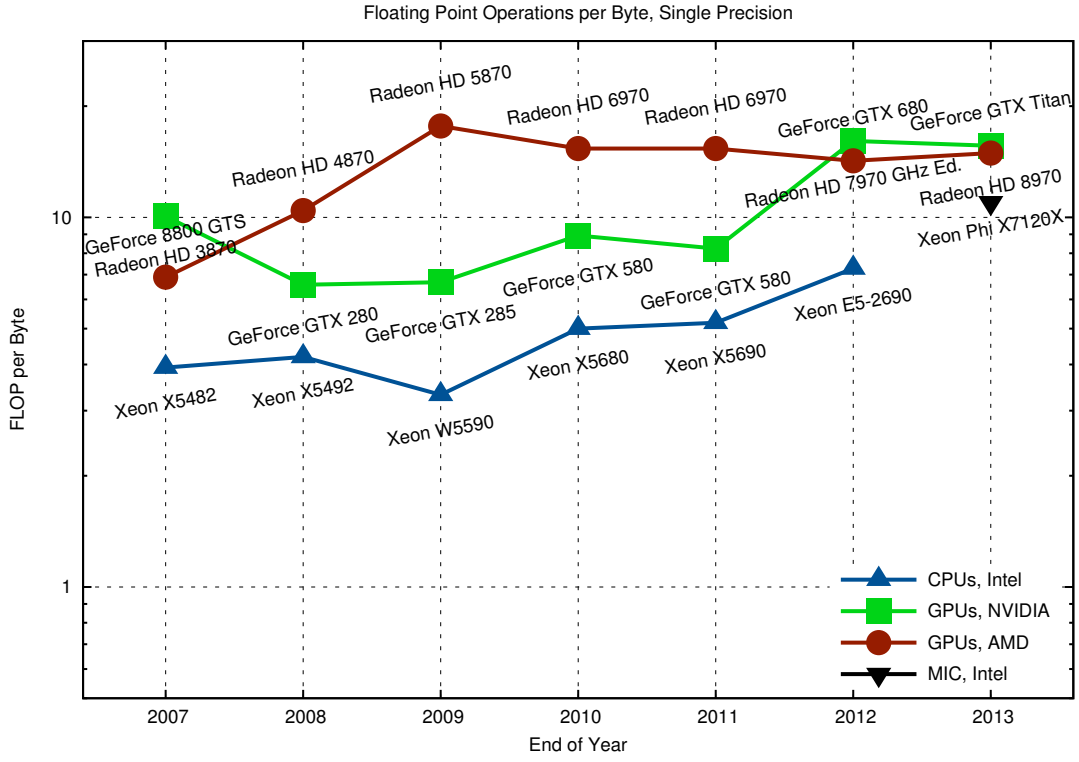


Figure 3.4: Floating point performance per bandwidth of CPUs and GPUs [40]

relation of available computing power to available memory bandwidth is therefore different, as shown by Figure 3.4: GPUs can perform about twice as many floating point operations for every byte they can transfer (again, assuming that all CPU cores are used and performing SSE instructions). This is significant because, as we have already seen, the memory bandwidth available on CPUs is not always enough to keep even one core working continuously. GPU algorithms will therefore tend to work more instead of storing intermediate results.
Note, however, that the memory bandwidth shown in Figures 3.3 and 3.4 is the maximal theoretically achievable bandwidth. Our discussion of the performance problems of the sequential algorithms has shown that achieving or even getting close to this bandwidth is not always trivial even in the CPU case. For GPUs, different types of memory require additional conditions to be fulfilled in order to optimally access the memory. These conditions are:

- Accesses to DRAM (i.e. global memory) should be *coalesced*. This means that consecutive threads of the same warp should access consecutive 4-byte-words of global memory, i.e. if thread i accesses word j, then thread $i+1$ should access word $j+1$, assuming threads i and $i+1$ belong to the same warp. If they are not in the same warp, their memory accesses will usually not be performed at the same time anyway and therefore cannot influence each other's performance. Memory transactions are always carried out in blocks of 32, 64 or 128 bytes, meaning that access patterns fetching smaller blocks will always waste bandwidth. For optimal performance, the block of consecutive memory locations that is accessed should also be aligned with its size, i.e. the start address of the block should be a multiple of its size. If this condition is fulfilled, the entire block can be transferred from DRAM to the SM in one (or few, for bigger blocks) transaction. Otherwise, more transactions will be necessary, in the worst case one for each requesting thread, and most of the transferred data will be unused.

  Recent GPUs have relaxed the conditions for optimal access to some degree, but they changed only in some details, and discussing the changes here would be too specific to the current architecture of NVIDIA GPUs.

  While local memory is also physically stored in DRAM, the CUDA runtime automatically arranges local memory variables in such a way that the same variable for consecutive threads is placed in consecutive memory locations, so that accesses are automatically coalesced if different threads access the same local variable.

- Shared memory consists of several banks (currently 32, meaning there is one bank for every thread of a warp). During each cycle, each bank can execute one memory transaction. Shared memory can therefore be used optimally if all threads in a warp access different memory banks in every instruction, because every bank can handle one request. Another optimal case is several threads accessing the exact same location: While there are several requests to the same bank, only one value is actually read and can be broadcast to all requesting threads. Problems occur if several threads of a warp access different locations from the same bank at once. This is called an n-way bank conflict, where n is the number of different locations on the same bank accessed at once. In this case, n transaction have to be serialized, increasing the cost of the overall transaction by a factor of n. In the worst case, n can be the number of threads in a warp, so that bank conflicts can increase the cost of a shared memory access by a factor of 32 on current GPUs. Since all threads of a warp generally execute at once, this means that the entire warp has to wait until all transactions have been executed.

  Successive 4-byte-words in shared memory belong to successive memory banks. Bank conflicts can therefore be avoided by using the same access pattern as for coalesced global memory accesses, i.e. thread i accessing word j, thread $i+1$ accessing word $j+1$ etc. More generally, any access pattern will avoid bank conflicts if $\lceil \frac{address}{wordSize} \rceil \mod nBanks$ is different for each thread of a warp.

In principle, the same problem could occur for global memory, which also consists of several physical parts called partitions. The equivalent of bank conflicts for DRAM partitions is called *partition camping*. On modern GPUs, however, a pseudo-random permutation is used to map global memory sections to different partitions. One can therefore expect partition accesses to be randomly distributed in the average case, so that partition camping should not be a problem. On the other hand, it is of course always possible to just have bad luck and consistently hit the same partition. Unless programmers find out the nature of the used hash function, there is no way to actively avoid this possibility.

The other reason that partition camping is usually not a big problem, whereas bank conflicts can be, is that accesses to global memory are expected to take hundreds of cycles. There is therefore very little impact if they need a few cycles more for some requests. Shared memory, on the other hand, is expected to be a fast alternative to global memory and used for exactly that reason, which is why shared memory bank conflicts typically do matter.

To summarize, a modern GPU differs in many important ways from any kind of PRAM. The main differences are:

1. The memory hierarchy: While CUDA devices do offer a global memory accessible by all threads, this memory has high latency. In order to create a fast kernel, one therefore tends to use shared memory, which is only accessible by threads of the same block. This is a concept that is usually not found in PRAM models. The size of the shared memory is very limited, which forces programmers to use algorithms that only use small amounts of memory repeatedly. The use of shared memory (and, for that matter, local memory or registers) also limits the ability of an SM to execute several blocks at once: The number of concurrently executed blocks is limited by the demand of each block of each of these resources; if one block uses all the shared memory an SM possesses, only one block at a time can be run on each SM.

   The assumption used in PRAMs that memory accesses generally have unit cost, which is of course invalid in GPUs because of the wildly different latencies of global and shared memory and the existence of a cache for global memory (which reduces cost for recently accessed parts of DRAM), is further shattered by the fact that different memory access patterns lead to vastly different performance. An example is coalescing global memory accesses: if all threads of a warp access a continuous area of memory in a certain order, this can reduce the number of necessary memory transactions by a factor of 32.

   While global memory access patterns can often be predicted directly from a high level description of an algorithm, a similar problem is much harder to spot. As mentioned before, simultaneous accesses to the same bank in shared memory results in a bank conflict (except in some special cases), which again means that the requests have to be served sequentially. This means that the worst case step complexity of n shared memory accesses is n for both writes and reads. Determining whether parallel reads or writes can potentially access the same memory bank, however, requires knowledge of the size and structure of the used data types, their positioning in memory, and the number and structure of the memory banks in the used hardware (which has changed in the past and may well change again). Any algorithm designed to run on a PRAM that does not specifically account for all of these things (which is basically any algorithm designed to run on a PRAM), even if it uses the cache well and can fit its data into shared memory, can potentially see all its concurrent memory accesses getting serialized.

2. The understanding of SIMD: PRAMs are generally expected to run in a SIMD fashion. While GPUs follow the SIMD paradigm more strictly than PRAMs in some ways (e.g.

branch divergence; only one branch of execution can be run at a time, so different branches are executed sequentially, which of course prolongs the runtime), they are less strictly SIMD in other ways: All blocks of threads may execute independently of each other in any order, with their execution being interleaved or not. CUDA does not support synchronization between blocks. If an algorithm depends on a certain step being done before another one on a global level (i.e. more than one block is needed), one therefore has to run several distinct kernels in order to enforce synchronization, which causes additional overhead for dispatching the kernels and causes other problems, e.g. because of the fact that shared memory contents cannot be reused across several consecutively executed kernels.

Writing one's own global synchronization methods is in most cases impossible and in all cases bad for performance: If several blocks wait for another block, which is not currently being executed on any SM, the program may wait indefinitely because the waiting blocks currently running prevent the other block from ever being executed. Even if one could guarantee that all blocks can be dispatched to the SMs concurrently, the waiting blocks would spend valuable computing time (and possibly memory bandwidth) in waiting loops, since there is no way to mark a block as being idle until some event happens [48].

Even within blocks, the order in which different warps are executed is arbitrary. While there is a possibility to synchronize all threads of one block, this synchronization comes with a performance penalty. Perhaps more importantly, it may cause warps to stay inactive while waiting for other warps to arrive at a synchronization barrier. This reduces the number of warps that can be executed at any given time and therefore restricts the SM's ability to hide memory latency by switching between different warps, in effect causing an SM to be idle while waiting for certain warps.

Essentially, GPUs' inability to execute several branches at once, along with the undefined order of execution on a block and warp level, which can be fought to some degree but not without performance overhead, may strongly affect the performance of PRAM algorithms that depend heavily on branching and/or all operations being globally executed in a specific order.

3.2 Fundamental GPU algorithms

In the following discussion of algorithms for multireduce and multiscan, we will repeatedly make use of other algorithms fundamental to GPU computing. While we will discuss parallel algorithms for scan and reduce in the respective chapters of their generalized versions, we will briefly sketch two algorithms here: GPU radix sorting as well as a multi-pass scheme for gather and scatter.

3.2.1 Radix sort

As many other algorithms, GPU radix sort is based on scans. There are several implementations which differ in algorithmic details, like the radix sort offered in the CUDA SDK [41], SRTS radix sort [30], which is the fastest published GPU radix sort algorithm, and GRS [2], which is optimized for key-value-sorting with large values. The main idea behind the algorithm, however, is always the same, and is shown in Algorithm 3.1 and illustrated in Figure 3.5.

For every round of sorting n inputs by b bits at once, 2^b integer flag vectors of length n are created. For each element of the input, the corresponding flag in flag vector V_i is set if the selected radix bits have the value i. A simple example: If $b = 4$, and the second round of sorting is running, the currently selected bits of a binary number 0100 1001 0010 are 1001,

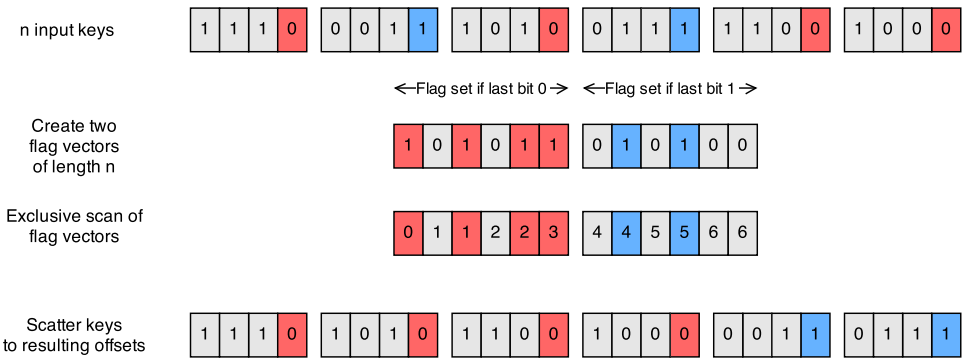


Figure 3.5: First round of GPU radix sort with $b = 1$ (adapted from [30])

function radixRun(int $in[n]$, int $out[n]$, int n, int run, int b) **:**
    int $flagVectors[2^b \times n]$;
    parfor $i = 0$ **to** $n - 1$ **do**
        for $j = 0$ **to** $2^b - 1$ **do**
            $flagVectors[j \times n + i] = 0$;
        end
    end
    parfor $i = 0$ **to** $n - 1$ **do**
        int $currentBits = in[i]/(2^b \times run) \mod 2^b$;
        $flagVectors[currentBits \times n + i] = 1$;
    end
    int $scannedFlags[2^b \times n]$;
    Scan($flagVectors$, $scannedFlags$, $+$);
    parfor $i = 0$ **to** $n - 1$ **do**
        int $currentBits = in[i]/(2^b \times run) \mod 2^b$;
        int $offset = scannedFlags[currentBits \times n + i]$;
        $out[offset] = in[i]$;
    end
end

Algorithm 3.1: Parallel radix sort algorithm for a single run sort by b bits

which is 9 in decimal. Therefore the flag corresponding to said number in the 9th flag vector is set to one, and to zero in all other vectors. All flag vectors concatenated into one vector are then scanned (exclusively). In the last step, the final offset is calculated for each input number. For input i, this is done by selecting index i in the flag vector where the ith flag was set in the first step. The input is then scattered to this location. Most of the work done by this algorithm is done in the scan of an array of length $n \times 2^b$, leading to an overall work complexity of $\mathcal{O}(n \times 2^b)$.
We will use this opportunity to introduce our notation for GPU algorithms, which we will use for all GPU-specific parallel algorithms; for the description of generally applicable parallel algorithmic concepts, we will still use the **parfor** notation as above.
As mentioned before, GPU code is written in the form of kernels, which are then executed concurrently by a number of threads, which are organized into blocks. The number $nBlocks$ of blocks and the number $nThreads$ of threads per block have to be specified for every kernel call. This means that GPU functions necessarily consist of at least one kernel as well as a host function which calls the kernel. We will generally define the host function first and the kernel(s) afterwards. The syntax we will use for a kernel call is as follows:
kernelName«$nBlocks$, $nThreads$»(arg_0, arg_1, ..., arg_{k-1});
This will start a kernel execution with $nBlocks$ blocks, each consisting of $nThreads$ threads, each of which executes the kernel function $kernelName$. Each executing thread can then refer to its ID $threadId$ as an implicit parameter anywhere in its code which is in the range $[0...nThreads-1]$. Likewise, the identifier $blockId$ denotes the ID of each thread's block, and the identifiers $nBlocks$ and $nThreads$ can be accessed within the kernel code as well. This is necessary because the function parameters arg_0, ..., arg_{k-1} of the actual kernel function are identical for all threads.
Arrays handed to kernel functions as arguments are assumed to be located in global memory. Shared memory variables are declared within kernels and have the modifier "shared" in front of the variable declaration. All variables declared within kernels which are not shared are local to each thread. Occasionally, we will use an additional modifier "local" if it makes the code more intelligible.
Since we want to demonstrate general algorithmic ideas, not write runnable code, we will generally assume that the number of input elements can be evenly divided between threads or blocks; more specifically, we assume that all divisions which have the purpose of dividing input data between entities result in integer values. For the same reason, we will not explicitly state synchronization statements. Instead, we will assume an implicit synchronization after every statement, meaning any thread in a block can only execute statement $i+1$ if statement i has been executed by all threads in the same block.
If we transfer the radix sort algorithm sketched above to the kernel format, the result is Algorithm 3.2: Where before there were **parfor** statements working on n items at once, we now have n threads which all work in one block for simplicity. Since the scan may be carried out by any number of threads which is not necessarily equal to n, it is necessary to split the algorithm into three parts.

3.2.2 Gather and scatter

The optimal PRAM algorithm for gather is obvious. For gathering n values, n processors can fetch their respective labels in parallel, fetch the values that belong to it, and store it in their output locations. Every processor does linear work, and the algorithm has constant depth. A similarly obvious algorithm can do the scatter with the same work and step complexity on an CRCW PRAM; a PRIORITY PRAM could even prioritize some indices over others in cases where several values are scattered to the same location.
For the gather, the PRAM algorithm transfers relatively well to GPUs at least in principle. Current GPUs can broadcast values that are requested by several threads (meaning that

```
function radixRun(int in[n], int out[n], int n, int run, int b) :
    int flagVector[2^b × n];
    setFlags«1, n»(in, flagVector, run, b);
    int scannedFlags[2^b × n];
    // choose scanBlocks and scanThreads for optimal performance Scan«scanBlocks,
    scanThreads»(flagVector, scannedFlags, n);
    scatter«1, n»(in, out, scannedFlags, run, b);
end
kernel setFlags(int in[n], int flagVector[2^b × n], int run, int b) :
    for i = 0 to 2^b − 1 do
        flagVector[i × n + threadId] = 0;
    end
    int currentBits = in[threadId]/(2^b × run)  mod 2^b;
    flagVector[currentBits × n + threadId] = 1;
end
kernel scatter(int in[n], int out[n], int scannedFlags[2^b × n], int run, int b) :
    int currentBits = in[threadId]/(2^b × run)  mod 2^b;
    int offset = scannedFlags[currentBits × n + threadId];
    out[offset] = in[threadId];
end
```

Algorithm 3.2: Parallel radix sort algorithm for n threads

the cost of such concurrent accesses does not increase), so there is no problem with several threads that have the same label reading the same corresponding value. One would make a few sensible changes to the algorithm: Since GPUs can only execute a limited number of instructions at any time anyway, there is no benefit in creating a thread for each input label. Instead, one should process a number of values, resulting in an algorithm like Algorithm 3.3.

```
function gather(int n, int m, int labels[n], M values[m], M result[n]) :
    // choose nBlocks and nThreads for best performance
    gatherKernel«nBlocks, nThreads»(n, m, labels, values, results);
end
kernel gatherKernel(int n, int m, int labels[n], M values[m], M result[n]) :
    int blockWork = n/nBlocks;
    int *blockLabels = &labels[blockId × blockWork];
    int *blockResult = &result[blockId × blockWork];
    for i = threadId to blockWork step nThreads do
        int currentLabel = blockLabels[i];
        blockResult[i] = values[currentLabel];
    end
end
```

Algorithm 3.3: Simple GPU gather algorithm

Every thread fetches one label at a time in a strided manner to coalesce the memory accesses. Then every thread fetches the corresponding value and, again coalesced, stores them in their output location. Every block does this for a range of input labels, thereby fetching a contiguous part of the input label vector sequentially.
This algorithm seems reasonably efficient, but it has one weakness. The load of the values is not coalesced (unless the input assigns consecutive labels to consecutive threads). Even worse, there is also no pattern in the access of the values. In cases with many input values (a large m in terms of the definitions used before), when not all of them fit in cache, there

will therefore be uncoalesced reads from uncached global memory.
The situation is similar for the scatter, except in this case there are uncoalesced, uncached writes. There are two additional problems here: The resulting value in any location scattered to by several threads is arbitrary; all that is certain is than one of the values that was supposed to be scattered to a specific location will end up there. This algorithm therefore cannot be used if there are any requirements concerning which value 'wins' if several are written to one location.
He et al. [19] have proposed an algorithm that partly overcomes the primary weakness of the simple scatter and gather algorithms discussed above, the fact that there are uncoalesced accesses to potentially uncached memory if the number m of buckets is large. While they do not address the coalescing problem, they propose a multi-pass scheme to achieve a higher cache hit rate. While they never describe their parallel algorithm in detail, their approach is essentially as follows: If not all buckets fit into cache at once, traverse the input several times instead of only once. During each traversal, only read and write values for a subset of all labels that fits into cache. Thereby the cache hit rate is increased at the cost of having to traverse the input more than once. They also present a formula for estimating which number of passes will deliver the highest performance.
The multi-pass approach has its merits if one can reasonably expect the distribution of labels to be about even, and if m is only a little larger than the number of buckets that would fit into cache. If the first condition is not fulfilled and only a small number of labels is actually used, the single-pass algorithm already uses the cache well, and doing multiple passes is just a waste of time. If the second condition is not fulfilled, the number of passes necessary to achieve a high percentage of cache hits will be so high that the overhead for traversing the input many times will cancel out any performance gains due to better cache utilization.

3.3 Expectations for a GPU implementation

Expectations for the speedup of GPU algorithms over their CPU-based counterpart are often high, since several publications have published speedups of one, two or even three orders of magnitude for different problems. Lee et al. [25] have, however, concluded that even for problems with large amounts of data parallelism, GPUs are only 2.5× faster than optimized CPU implementation on average.
Since we use a completely sequential CPU implementation as our baseline, whereas the optimized CPU algorithms used by Lee make use of several cores and, wherever possible, modern CPUs' vector instructions, we should be able to achieve more than a 2.5× speedup. Our hope is to find algorithms which are at least five times faster than the CPU implementation, since at this point, the speedup is large enough to justify the added effort of performing computation on the GPU.
While we noted that the sequential CPU algorithm seems to be limited by memory bandwidth, and that current GPUs only offer seven times as much memory bandwidth than high end CPUs, these high end CPUs often use more than one socket, each of which has only a fraction of the officially stated memory bandwidth. Since a sequential algorithm can naturally only take advantage of the memory bandwidth provided by one socket, the memory bandwidth available to a GPU algorithm is likely more than seven times as big as that available to a sequential CPU algorithm.
We can turn to existing GPU implementations for related problems to get an idea of the speedup for the specific problems we are working on. The problems most closely related to multireduce and multiscan for which there are existing algorithms are reduce-by-key and scan-by-key. Both are variations of segmented reduce and segmented scan, respectively, which take labels instead of flags as inputs in addition to the values, and where the start of the next segment is denoted through a change of the label. Both Thrust and Modern GPU offer

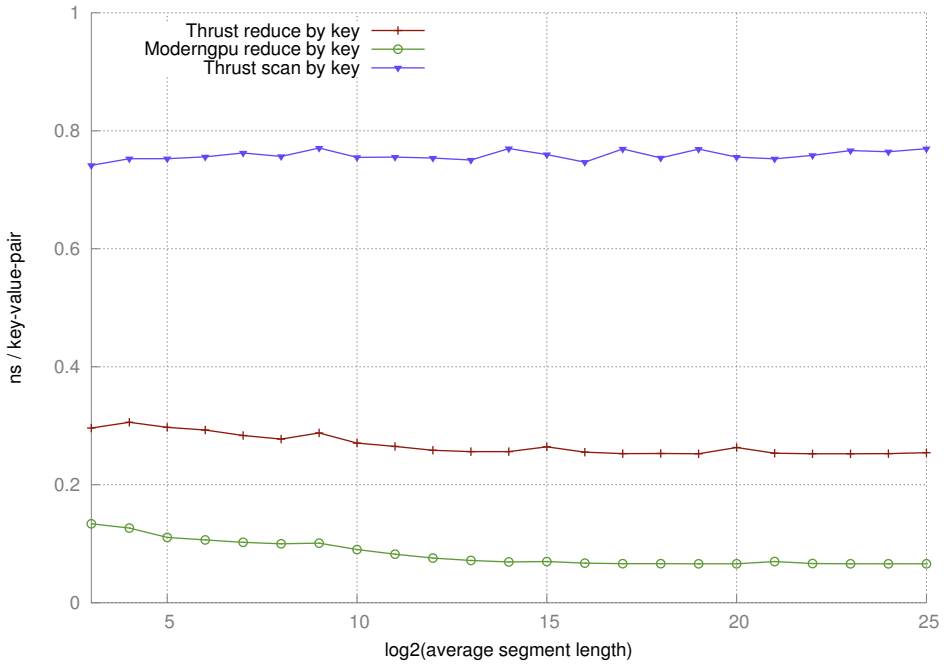


Figure 3.6: Performance of existing algorithms for reduce by key and scan by key

implementations of reduce-by-key, whereas only Thrust offers a scan-by-key function. The performance of both algorithms for different average segment lengths is shown in Figure 3.6. The performance of Thrust's reduce-by-key is basically constant for different segment lengths and never slower than 0.3 ns per input. The scan-by-key, in contrast, takes around 0.75 ns per input for any segment length. Compared to the 2.5 ns per input and the 6 ns per input which the sequential algorithms for multireduce and multiscan need even in the best case, the GPU by-key-algorithms are about eight times faster in both cases. While the reduce-by-key implementation provided by the Modern GPU library needs less than 0.1 ns per input for large segments, this implementation makes some additional assumptions about the inputs and is therefore not a completely generally applicable algorithm.
Since both multireduce and multiscan are more general versions of the by-key-algorithms benchmarked here, we could confidently call any multireduce or multiscan algorithm fast if it performs as well or better than these two existing implementations.

3.3.1 Using libraries

In the next part of this thesis, we will discuss and implement different approaches to implementing multireduce and multiscan, some of which build upon existing algorithms.
To implement all of these combinations, a lot of different helper functions will be needed (e.g. segmented reduce and radix sort on a global and block level). Implementations for all of them exist in various libraries like CUDPP [1], Thrust [20], Modern GPU [12] and CUB [28], some of which use well-documented algorithms, whereas others simply provide a black box function without providing any information on what it does internally. However, all of them are optimized functions, often developed by NVIDIA researchers, which is why it makes sense to use them wherever possible (which also saves a lot of time). The downside of this is that they might not be perfectly suited for the context we will use them in for a number of reasons:

- The format in which they expect their inputs or return their outputs might not be the

one we would use, so that a conversion is necessary.

- Since they work as black boxes, it might be difficult to intertwine several such functions in cases where this would save work.
- They might not offer features we would optimally like to use.

In spite of this, the amount of work necessary to implement even one of these functions and optimize it to a degree where it could compete with the performance of existing implementations is unreasonable high for the scope of this thesis. We will therefore adapt existing libraries if the expected result is worth the effort, and otherwise use the existing functions as they are and suggest what could be done differently.
The results of our benchmarks will be useful as an indication which algorithm should be used in what scenario, but will probably not reflect the best performance that could possibly be achieved with the given approach. The same attitude will be used for the parts not taken from libraries: We will optimize them to a reasonable degree, trying to avoid obvious pitfalls like bank conflicts, branch divergence and uncoalesced memory accesses, but the goal is not to get the best possible implementation in each case.

3.4 Measurements

All measurements of GPU performance throughout this thesis are performed on a Geforce GTX 690, which is a dual card in the sense that it contains two actual GPUs. Using multiple GPUs at once in CUDA is possible, but requires additional programming effort and may give rise to performance phenomenons only seen in multi-GPU configurations. Since we wanted to measure algorithm performance for the more standard case of one GPU, we therefore did not take advantage of the second GPU core. When using only one GPU, the GTX 690 performs almost exactly like a GTX 680, with the only difference being a slightly lower clock frequency. The GTX 690 is based on the Kepler architecture and supports Compute Capability 3.0. It has eight SMs with up to 48 kB of shared memory each, and a theoretical memory bandwidth of 192 GB/s. We use version 5.0 of CUDA, and the most recent available versions of all referenced libraries.

Chapter 4

Multireduce on GPUs

4.1 Adapting ordinary reduce

A natural starting point when looking for an efficient algorithm for the multireduce problem are the existing algorithms for the ordinary reduce, which we will therefore describe here.
The general idea of parallel reduction is very simple. Assuming that the number n of input elements is a power of two, the algorithm works as described in Algorithm 4.1 and illustrated in Figure 4.1.

function reduce(M in[n], int n) **:**
  for $i = 1$ **to** $\log_2 n$ **do**
    parfor $j = 0$ **to** $\frac{n}{2^i} - 1$ **do**
      $in[j] = in[2j] \odot in[2j+1]$;
    end
  end
  return $in[0]$;
end

Algorithm 4.1: Parallel reduce

This algorithm, which can be easily extended to handle different input lengths, has a depth of $\mathcal{O}(\log n)$ and a work complexity of $\mathcal{O}(n)$ and is therefore work-efficient.
The reduction algorithm provided in the CUDA SDK is an adaptation by Harris [17] of this algorithm. Harris shows how to implement this algorithm on GPUs with good performance, i.e. how to avoid branch divergence and bank conflicts and unroll loops where possible. In more recent work, Martín et al. [26] further improve on the existing algorithm with an optimized work distribution between different blocks and warps, but do not change the algorithm itself.
There is, of course, one obvious way to adapt the traditional algorithm to perform a multireduce, but it is just as obvious why this approach is not always an efficient one. Instead of scalar values, one could let the algorithm run on vectors of these values with length $nBuckets$. Input elements could be converted to this format by starting out with a vector a consisting only of zeroes, and then setting $a_{l_i} = v_i$. If the input operator is $\odot$, the modified $\odot'$ would be defined such that $a \odot' b = [a_0 \odot b_0, a_1 \odot b_1, ...]^T$. The result of the traditional algorithms would then be a vector c containing the results for each label for the multireduce. In order to get the actual result r_i for each input value v_i, one would have to select the value c_{l_i} corresponding to the label for each element, which could, again, be done in parallel.
While this algorithm has the same depth as the original elements (since all additional work can be done in parallel), the work performed by it would be of the order $\mathcal{O}(n \times nBuckets)$, which is undesirable since $nBuckets$ may potentially be quite large. Unless $nBuckets$ is regarded as

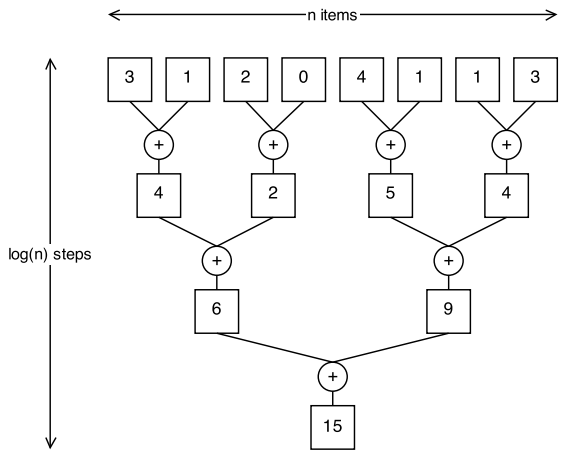


Figure 4.1: Parallel reduce algorithm, illustrated for addition on integers

fixed, this algorithm is therefore not work-efficient, since the sequential algorithm has a work complexity of $\mathcal{O}(n)$, no matter how large m is. While it may well be the fastest multireduce algorithm if $nBuckets$ is very low, it will not perform well in general, which is why we will have to look for alternative approaches.

4.2 Overview of possible solutions

Since adapting the traditional reduce algorithm does not seem to be an option, other approaches must be found in order to implement multireduce efficiently.
Assume for a moment that a multireduce algorithm is given. Then the obvious way to compute the multireduce of a given vector of label-value-pairs is to apply the algorithm to the entire vectors and return the result. There is, however, a different possibility: The algorithm could be applied to several segments of the input in parallel, resulting in several partial results. These results could then be accumulated into the final results. A scheme for doing this is shown in Figure 4.2.
The input arrays are partitioned into $nSegments$ parts of equal length. A multireduce algorithm is then applied to each part in parallel, which writes the partial results back to global memory. If the labels in the input data are in the range $[0...nBuckets-1]$, each segment will write $nBuckets$ to global memory. We will call the $nBuckets$ buckets containing the partial reductions for one input segment a *bucket set*, although it is not necessarily a set in the mathematical sense. $nSegments$ bucket sets (or $nSegments \times nBuckets$ total buckets) are therefore written to global memory after the first step. These can then be combined into the final result in a second step, using a normal reduction for each bucket. Since we assume the operation $\odot$ to be associative, this will always compute the correct result if each SM works on a continuous segment of the input, and the final reduction is carried out in the correct order.
This simple approach, which we will refer to as *partitioning*, has two main advantages:

1. It introduces an additional level of parallelism. Even if the algorithm used for each part is completely sequential, several parts can be worked on concurrently, which is why this approach would be the first choice when implementing a parallel multireduce on the CPU.

   The potential amount of additional parallelism is quite large, since the input can, in principle, be divided into arbitrarily short segments. The downside of having a large

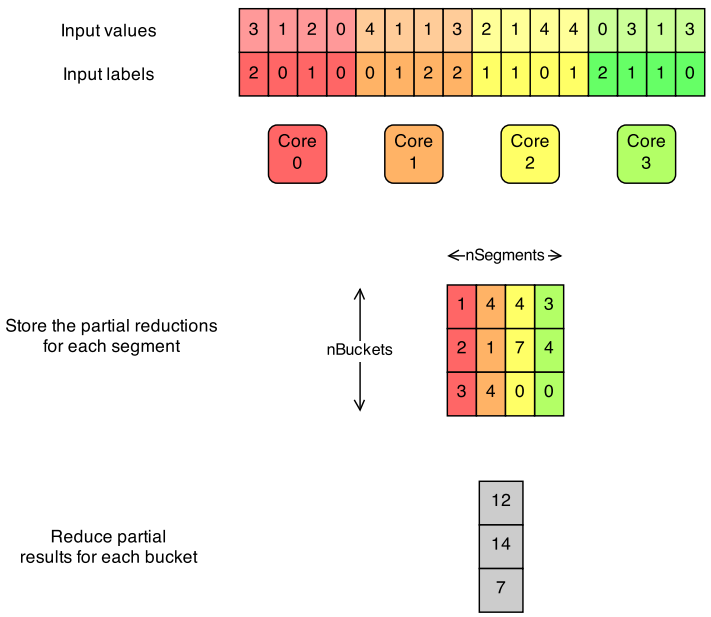


Figure 4.2: Partitioning scheme for multireduce

number of segments is that a large amount of space is needed to store the results for each block, and that reducing the partial results to the final result takes longer the more segments there are: $nBuckets$ reductions of $nSegments$ elements each have to be carried out in the final step.

2. It allows the algorithm used internally to use resources that are only available on a block-level, like synchronization or shared memory. The former is actually a precondition for many algorithms to work correctly, while the latter can be helpful to speed up the execution of many algorithms.

There are often advantages as well as disadvantages to performing an algorithm exclusively within shared memory: While it is much faster than global memory, its size is also very limited, which is a problem for some algorithms. Therefore there seem to be three sensible ways to implement almost any given multireduce algorithm on GPUs:

1. Without partitioning, meaning that the algorithm is applied to the entire input at once.
2. With partitioning, where the algorithm uses global memory to work on each block.
3. With partitioning, where the algorithm uses shared memory to work on each block.

We should therefore evaluate each algorithm we can find on these three levels, but we still need algorithms to do the actual work.
As mentioned before, working with several segments at once is a way to introduce parallelism even for sequential algorithms. One can therefore take the simple, sequential multireduce algorithm as a basis, and try to further adapt it to the capabilities of GPUs.
Alternatively, one could attempt to convert the multireduce problem to a different one for which there are existing efficient GPU algorithms. Ordinary scan and reduce are closely related to the multireduce and, as we have seen before, can be implemented very efficiently on GPUs, so it would be sensible to try and convert a multireduce to a combination of these problems. As it turns out, this is possible: By sorting the input keys and values, a multireduce can be turned into a segmented reduce.

In the final reduction step of the partitioning approach, we need to perform an ordinary reduction of the partial results of each bucket. In order to achieve good performance, we would obviously like to perform the reductions for all buckets in parallel. Since there seems to be no library which offers a function to reduce several vectors at once, we will use a segmented reduce function instead, and simply mark the start of a new bucket as a new segment. We will choose the segmented reduce function provided by the Modern GPU library, which seems to be the fastest of its kind [12].
In the first step, the multireduce algorithm used on a block level stores its result for each bucket in an array as shown in the second step of Figure 4.2. Since $nBuckets$ reductions of $nSegments$ elements are necessary, we then create a flag vector $[f_0, f_1, ..., f_{n-1}]$ where f_i is set iff $\exists k.i = k \times nSegments$, and feed this along with the partial results array to ModernGPU's segmented reduction function. The results of said function are then the final reductions of each bucket.
Now that the framework in which we will use our algorithms is defined, we will discuss the algorithms themselves, starting with adaptations of the sequential algorithm.

4.3 Adaptation of sequential algorithm

In this section, we will discuss different ways to adapt the sequential multireduce algorithm to the GPU. We will start out by giving an overview of the work that has been done in this area. While there are no publications on adapting the multireduce itself for GPUs, there have been a number of publications on the special case of histograms, all of which use an approach one might call an adaptation of the sequential histogram algorithm. In the remainder of this section, we will then try to apply the core concepts of the discussed histogram algorithms to our multireduce implementation.

4.3.1 Related work

The first proposals for GPU histogramming (for example Scheuermann and Hensley [42]), written before the introduction of CUDA, still describe algorithms in terms of using textures and shaders with OpenGL, and have significantly worse performance than later CUDA implementations; we will not discuss them here. The first two implementations that used CUDA were proposed by Podlozhnyuk [37]. Like all of the other algorithms we will discuss in this section, their general idea is to create several partial histograms for parts of the input data in parallel and later combine them into one. The algorithms differ, next to some minor details, in the number of partial histograms, the number and distribution of threads and blocks, the way they are stored in memory, and the different kinds of collisions that can occur as a result of this setup. All proposals are, of course, also a product of their time: When CUDA was first introduced, the capacities of GPUs in terms of shared memory and their ability to perform atomic operations was very limited compared to their modern equivalents, which of course affects the algorithms that can be developed for them.
Both of Podlozhnyuk's proposals were designed for CUDA devices with 16 kB of shared memory per SM, which offer atomic operations for global memory, but not for shared memory. The number of memory banks was assumed to be 16, and memory accesses were assumed to be performed per half-warp, meaning that a maximum of 16 threads at a time would access shared memory distributed among 16 banks.
The first approach targets histograms with 64 bins. It creates one private partial histogram for every thread in shared memory, with 192 threads in one block. In order to save space, each bucket uses only eight bits, meaning that every thread can only process a maximum of 255 inputs before one of the buckets can potentially overflow. Like all other published histogram algorithms with the exception of Shams and Kennedy [44], inputs are also assumed to be byte

sized (which is possible because all labels can only be values between 0 and 63 anyway) and each thread fetches four of them (i.e. one word) at once. Additionally, global memory reads are coalesced between the threads of a warp. Because each thread works on its own partial histogram, there can be no write conflicts, and atomic operations are not needed. In order to avoid bank conflicts, the partial histograms are placed in memory in such a way that each thread's histogram is placed entirely on one bank, and the threads in every half-warp use different banks. There are therefore no bank conflicts, independently of the input data. In the last step, the partial histograms are reduced to a global histogram using atomic addition operations in global memory.
In this algorithm as in all others, the partial histograms are combined in the end using hardware atomics.
Podlozhnyuk's second approach targets histograms with 256 bins, again stored in shared memory. Since Podlozhnyuk assumes that 192 threads are necessary to make good use of the GPU's resources, but shared memory is not big enough to store 256-bin histograms for 192 threads (unless the number of bits per bin is very low), this approach only creates one partial histogram for every warp. Writing to the same histogram with all threads of a half-warp at once means that there can be write conflicts between the threads, and every such write conflict is also a bank conflict. As mentioned before, CUDA devices did not support atomic updates of shared memory at the time Podlozhnyuk's paper was written, so an atomic update process was developed in software. This causes some overhead even in cases without write conflicts, as each thread has to check whether its value has been successfully written, and it causes writes to be serialized if several of them target the same bin, meaning that in the worst case, when every thread in a half-warp gets the same index, 16 updates are executed sequentially, leading to very bad performance in the worst case. In the best case, however, this algorithm utilized the full capacities of the GPU, and is therefore to this day the fastest algorithm for randomly distributed inputs.
Podlozhnyuk's algorithms were later generalized by Shams and Kennedy. In contrast to all other algorithms discussed here, they do not necessarily expect their histogram to be performed on image data and therefore also support more than 256 bins. This also means that they expect input data in the form of 32 bit integers instead of byte sized inputs, since more bits are needed to address higher numbers of bins. Both of the two algorithms they present are loosely based on Podlozhnyuk's.
Their first algorithm essentially combines the concepts of Podlozhnyuk's 256-bin algorithm and the multi-pass scatter/gather algorithm discussed before. Like the former, it stores one partial histogram for each warp in shared memory and sequentializes updates to the same location in software. In order to support more than 256 bins, the total number of bins is divided into several partitions if necessary; the algorithm then runs over the input data several times and ignores the bins not belonging to the current partition. A tradeoff needs to be made between the number of warps that are run concurrently on each SM and the number of partitions: More concurrent warps means a better utilization of the hardware, but it also means that each warp has less space for storing its partial histogram, thus requiring more partitions and consequently more passes over the input data. Shams and Kennedy also provide a formula to estimate which configuration is optimal. The performance of this approach is obviously data dependent. While it is able to deal with any number of bins in theory, performance gets very bad for very large numbers of bins simply because the input data has to be traversed a large number of times.
Their second approach is loosely analogous to Podlozhnyuk's 64-bin algorithm, with one important change. Again, each thread has its own private partial histogram. Since shared memory is only large enough for 64 bins per thread for a reasonably high number of threads, Shams and Kennedy place the partial histograms in global memory, using 32 bits for each bin and thus removing the restriction of having each thread process only a limited number of

input elements. As a result, there are no write conflicts and, since bank conflicts are only a problem in shared memory, no bank conflicts either. This method is therefore completely data independent. It does, however, have the obvious problem that it performs many uncoalesced writes to global memory. Shams and Kennedy address this problem by trying to reduce the amount of memory accesses by buffering writes in shared memory. Essentially, they store a thread-private mini-histogram in shared memory, using a very small number b of bits for each bin. How many bits can be used is, again, dependent on the number of bins (which is therefore not unlimited with this algorithm), the number of threads per block and the amount of shared memory that is available. Writes to global memory are only performed when a bin in shared memory overflows, thus reducing the number of total global memory writes by a factor of 2^b. Again, there is a tradeoff between the number of bits per bin and the number of threads per block; more threads and therefore more parallelism better utilize the GPU's resources, but also mean that less bits can be used and the number of global memory accesses increases.

Nugteren et al. [32] improved on these approaches with two algorithms, one of which is data independent. They regard image processing as the primary use case for histogramming, and therefore only support 256 bins. They also assume a high correlation between the values of adjacent pixels, as images tend to have. Since their paper was written later than the ones discussed so far, they assume a modern GPU architecture with 48 kB of shared memory, distributed among 32 banks, execution of all memory accesses for a full warp at once (not only for a half-warp, as before), and hardware support for atomic operations on shared memory. Nevertheless, their first algorithm does not depend on these features and is designed to work on older GPUs as well.

For said first proposal, they take Podlozhnyuk's 256 bin approach with per warp partial histograms as a baseline and suggest three ways to improve on it. The first two approaches exploit the assumption that adjacent input values are often the same if the input data is an image. Since identical adjacent values will lead to write conflicts in Podlozhnyuk's approach, so that writes must be serialized, this is a major weakness of the algorithm. Nugteren et al. therefore suggest different methods for shuffling the input data before applying Podlozhnyuk's algorithm. They also suggest using the now available atomic operations on shared memory instead of the software-based approach in the original algorithm. The actual structure of Podlozhnyuk's algorithm, however, remains untouched.

Their second algorithm is a variation of Podlozhnyuk's 64 bin approach adapted to work with 256 bins. It uses per thread histograms in shared memory, with each bin using 16 bits of space. Each block consists of one warp (32 threads). One block therefore needs $32 \times 256 \times 2$ bytes, or 16 kB, so three such blocks fit into a current GPU's SM's shared memory at once. They compare several ways to distribute the histograms across shared memory and choose one that results in completely data independent performance.

Each thread in a warp is assigned its own bank. The thread's private histogram is then placed exclusively on this bank, so that bank conflicts between the threads in one warp can never occur. The distribution of buckets among banks in shared memory is illustrated in Figure 4.3. This distribution is the main contribution of Nugteren's proposal, since the underlying algorithm is still very simple, as demonstrated by Algorithm 4.2: The algorithm's complexity stems mainly from complicated address calculations; from a more high level point of view, all it does is fetching and binning data.

Since this approach not only avoids bank conflicts, but any write conflicts between different threads, no atomic operations on shared memory are needed, and the performance is completely independent of the input data. The downside is that this algorithm only allows for very limited parallelism (96 threads per SM), and therefore performs worse than Podlozhnyuk's algorithm in many cases. In terms of worst case performance, it is nevertheless a big improvement over previous algorithms.

```
function histogram(int n, int indices[n], int result[256]) :
    // let nSMs be the number of the GPU's SMs
    int nBlocks = nSMs × 3;
    int nThreads = 32;
    int partialResults[256 × nBlocks];
    int threadWork = n/(nBlocks × nThreads);
    nugterenHistogram«nBlocks, nThreads»(indices, partialResults, threadWork);
    Accumulate all partial results in partialResults to result using a second kernel
end
kernel nugterenHistogram(int indices[n], int output[nBlocks × 256], int threadWork) :
    shared int16 histograms[256 × nThreads];
    int *blockIndices = &indices[blockId × nThreads × threadWork];
    char currentBytes[4];
    int *currentWord = (int*) &currentBytes[0];
    for i = 0 to threadWork − 1 do
        // load one word
        currentWord = blockIndices[nThreads × i + threadId];
        // bin it as four separate values
        for j = 0 to 3 do
            byte current = currentBytes[j];
            int address = 2 × (threadId + nThreads × (current/2)) + (current mod 2);
            histograms[address]++;
        end
    end
    for i = threadId to 256 step nThreads do
        int bin = 0;
        for j = 0 to nThreads − 1 do
            bin += histograms[2 × j × nThreads × (i/2) + (i mod 2)];
        end
        output[blockId × 256 + i] = bin;
    end
end
```

Algorithm 4.2: Main kernel of Nugteren's histogram algorithm

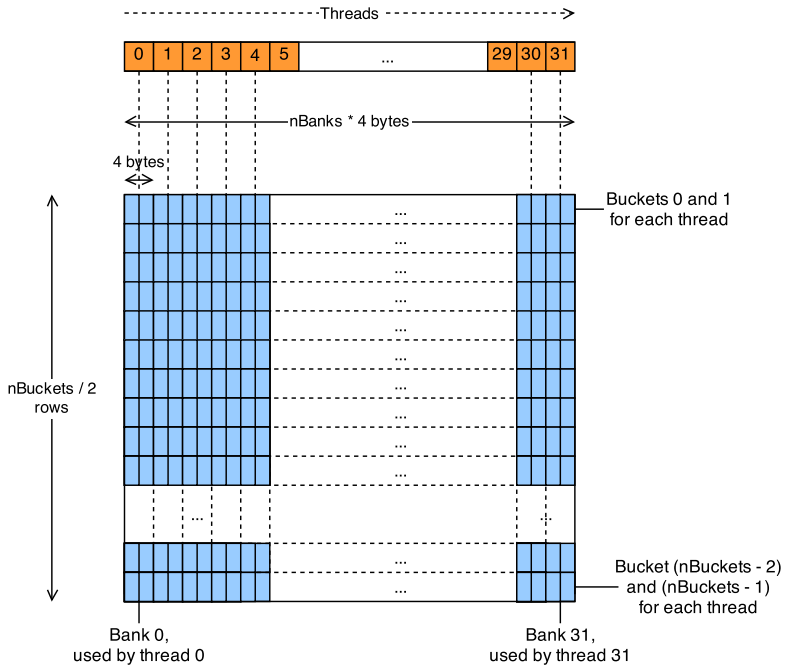


Figure 4.3: Memory layout of Nugteren's histogram algorithm

The most recent attempt to improve histogram performance on the GPU by Brown and Snoeyink [8] is called TRISH (Threaded Registers Independent Strided Histogram). It is somewhat similar to Nugteren's second proposal and, like the latter, is designed to work with 256 bins and used per-thread-histograms in shared memory. TRISH is an attempt to use more parallelism on several levels in order to improve on Nugteren's performance. Firstly, it allows more threads to be executing at once by only using 8 bits per bucket in shared memory with 64 threads per block, thus using 16 kB per block, so that 192 threads in three blocks can reside on every SM at once on current GPUs, twice as many as in Nugteren's algorithm. Like Nugteren, TRISH arranges the buckets in such a way that there cannot be any bank conflicts within a warp. A high level view of the algorithm can be found in Algorithm 4.3.
Each thread fetches a word of input data, which represents four byte-sized input labels. It then bins these input labels using its own private histogram in shared memory. Up to this point, TRISH is virtually identical to Nugteren's algorithm. Because TRISH uses only eight bits per bucket in shared memory, an overflow can occur after binning 256 labels. Since labels are fetched in blocks of four, this means that only 63 words can be fetched and processed before an overflow is possible. After 63 input words, the buckets in shared memory are therefore accumulated and the results are stored in private registers. Since there are 64 threads in a block, and TRISH uses 256 buckets, this means that each thread needs four registers so that all buckets can be stored. For each of those four buckets, each thread then iterates over all partial results for that bucket in shared memory, accumulates their values into its private register bucket and resets the shared memory buckets to zero. When this process has finished, 63 new words can be fetched etc., until the input has been processed completely.
Throughout the algorithm, Brown and Snoeyink try to improve instruction level parallelism, thus giving the compiler the possibility to rearrange instructions for better performance. In order to achieve this, they unroll loops when the total number of iterations is known, and they batch instructions, e.g. by first fetching several input words from global memory at once and subsequently binning them all. The main advantage of the latter is that the latency for

```
function histogram(int n, int indices[n], int result[256]) :
    // let nSMs be the number of the GPU's SMs
    int nBlocks = nSMs × 3;
    int nThreads = 64;
    int partialResults[256 × nBlocks];
    int threadWork = n/(nBlocks × nThreads);
    trishHistogram«nBlocks, nThreads»(indices, partialResults, threadWork);
    Accumulate all partial results in partialResults to result using a second kernel
end
kernel trishHistogram(int indices[n], int output[nBlocks × 256], int threadWork) :
    int *blockIndices = &indices[blockId × nThreads × threadWork];
    shared byte histograms[nThreads × 256];
    // local buckets stored in registers
    // local bucket i is used to store bucket i × nThreads + threadId
    local int buckets[4];
    Initialize buckets to all zeros
    for i = 0 to threadWork do
        for i ∈ [0...62] do
            Read 4-byte-word of data from blockIndices[i × nThreads + threadId]
            for j = 0 to 3 do
                Bin byte j of fetched data to histograms
            end
        end
        for j = 0 to 3 do
            int currentBucket = j × nThreads + threadId;
            Accumulate values of bucket currentBucket in shared memory histograms to
            local buckets[j]
            Set bucket currentBucket to zero for all histograms in shared memory
        end
    end
    for i = 0 to 3 do
        int currentBucket = i × nThreads + threadId;
        output[blockId × 256 + currentBucket] = buckets[i]
    end
end
```

Algorithm 4.3: Main kernel of TRISH histogram algorithm

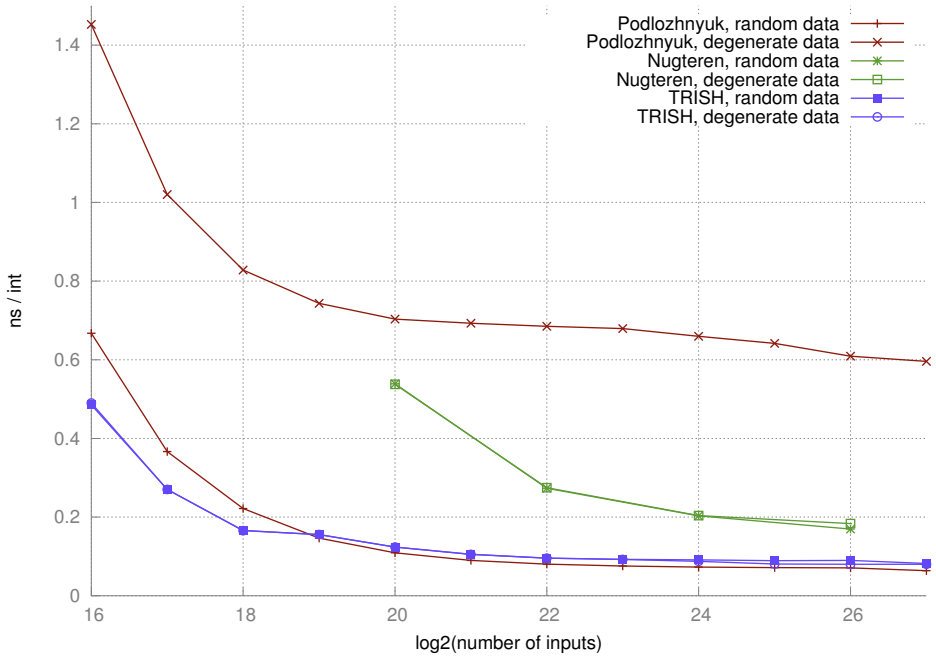


Figure 4.4: Performance comparison of Podlozhnyuk, Nugteren and TRISH

fetching four words at once is not nearly as high as the added latencies for fetching one word, processing its content, and then fetching the next one four times. A disadvantage is that this increases register usage for storing several inputs at once. Since the amount of concurrent threads is already severely limited by the amount of available shared memory, however, this does not actually affect performance.
When accumulating buckets into local registers, TRISH uses vector parallelism to add two 16-bit values at once in a single 32-bit addition. This helps save computation throughout the algorithm. Some other optimizations like using bit shifting instead of multiplication and division are nothing new and do not change the actual algorithm, so we will not discuss them here.
The respective performance of the three main approaches can be seen in Figure 4.4. Podlozhnyuk' algorithm performs best for random input data, but worst for degenerate data. Nugteren managed to improve the worst case, but is far away from Podlozhnyuk's best case, whereas TRISH performs almost as well as Podlozhnyuk's best case for any input data and is therefore a massive improvement over the previous approaches.
The main conclusions we can take from these proposals are the following:

- Adaptations of the sequential algorithm are the predominant approach to computing histograms on GPUs, which suggests that it is also a reasonable way to approach a multireduce algorithm.
- As with most GPU algorithms, one of the main goals is to create as much parallelism as possible.
- For low numbers of buckets, approaches using shared memory have the best performance.
- Algorithms which work on global memory can handle higher numbers of buckets and can achieve reasonable performance as well.

- When using shared memory, avoiding bank conflicts should be a main concern and can be sensible even if the amount of parallelism is lowered in the process.
- Serialized writes can lead to disastrous performance for degenerate input data and should therefore be avoided at all costs.

We will use these general conclusions as well as concrete ideas from the mentioned histogramming algorithms while searching for an optimal multireduce algorithm throughout the next sections.

4.3.2 Multireduce with non-commutative operators

```
function multiReduce(int n, int m, int labels[n], M values[n], M buckets[m]) :
    // choose nBlocks and nThreads for best performance
    miniMultiReduce«nBlocks, nThreads»(n, m, labels, values, buckets, n/nBlocks);
end
kernel miniMultiReduce(int n, int m, int labels[n], M values[n], M buckets[m], int
blockWork) :
    int *blockLabels = &labels[blockId × blockWork];
    M *blockValues = &values[blockId × blockWork];
    shared int currentLabels[nThreads];
    shared M currentValues[nThreads];
    for i = 0 to blockWork step nThreads do
        int offset = i × nThreads + threadId;
        currentLabels[threadId] = blockLabels[offset];
        currentValues[threadId] = blockValues[offset];
        // only thread 0 performs the actual multireduce
        if threadId == 0 then
            for j = 0 to nThreads − 1 do
                buckets[currentLabels[j] × nBlocks + blockId] += currentValues[j];
            end
        end
    end
    Write buckets to global memory
end
```

Algorithm 4.4: Minimal multireduce kernel

The simplest possible approach for implementing multireduce for GPUs is an adaptation of the sequential algorithm with minimal changes, as shown in Algorithm 4.4. In each block, a number $nThreads$ of threads cooperatively fetch input labels and values to shared memory with coalesced memory accesses. Once the data is fetched, all threads but one go inactive. The only active thread then processes the input data stored in shared memory using the sequential multireduce algorithm discussed in Section 2.
This algorithm is obviously not optimal. Since all but one thread stay idle while the actual work is performed, only $\frac{1}{nThreads}$ of the available processing power can be used during the actual computation. It also offers no parallelism within a block, meaning that this algorithm can only be used within the partitioning scheme discussed before; otherwise, only a single thread on the entire GPU would perform the entire calculation. The only advantage this approach has is that the amount of work it does is minimal, not just asymptotically, but absolutely.

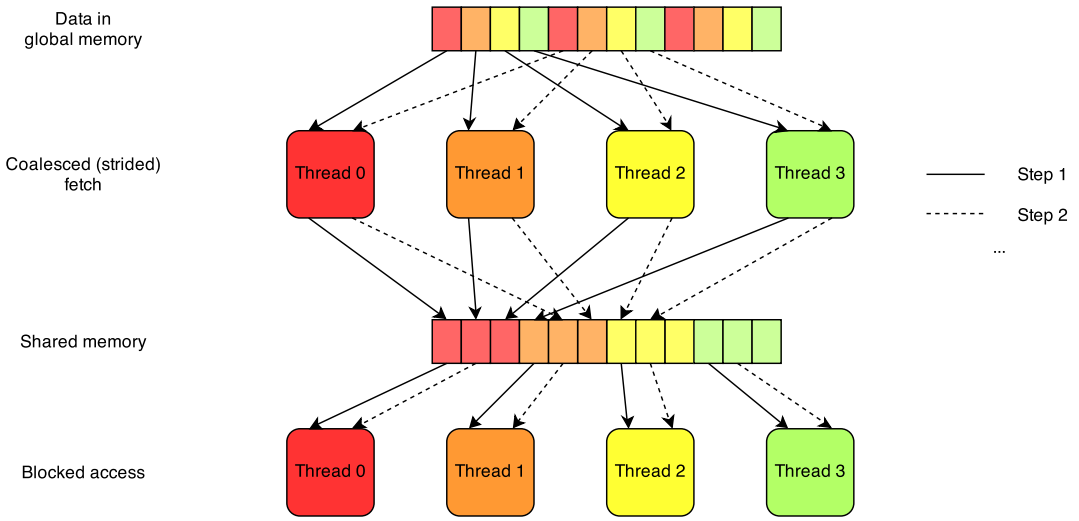


Figure 4.5: Baxter's transposition algorithm with $nItems = 3$ and $nThreads = 4$

Obviously, it would be desirable to let the remaining threads in each block participate in the actual work as well, as is the case in all histogram algorithms discussed before. If, however, the operator used is not commutative, then the input elements of a segment have to be processed in the order in which they occur in the input vector, which means that there is no trivial way to let several threads work on one segment. The solution to this problem would be to let every thread in a block work on a different segment (basically using the partitioning trick again, this time on a block level).

1. In order to achieve coalesced memory access, adjacent threads need to request adjacent words. A solution where every thread loads data from its own segment will therefore necessarily have uncoalesced access.

2. Each thread works on its segment completely independently from the other threads, meaning that it also needs its own bucket set. Since a block optimally has a very high number of threads, this would result in a very large number of intermediate sets of buckets, which all have to be reduced in the end and which have to be stored somewhere in the meantime.

A variation of the first problem which is common to many GPU algorithms is this: $nThreads$ threads need to fetch a contiguous block of $nThreads \times nItems$ words of data in such a way that thread i has the words $i \times nItems$ to $i \times nItems + nItems - 1$, meaning that thread zero has the first $nItems$ words, thread one has the second $nItems$ words and so on. In other words, the input data is needed in a blocked fashion, but can only be efficiently fetched in a strided (i.e. coalesced) fashion.
This is obviously similar to our problem, in the sense that every thread in a block needs to fetch a contiguous block of memory, but doing so in a naive way would result in uncoalesced memory accesses. The only difference is that in our case, the blocks of memory for all threads do not form a contiguous block, since the length of a segment will generally be larger than $nItems$. We will examine how the more common problem is typically solved and then try to adapt the solution to our scenario.
The general solution to this common problem is to *transpose* the input data through shared memory. We will examine the algorithm proposed by Baxter [4] shown in Algorithm 4.5 and illustrated in Figure 4.5.

```
// threadId is between 0 and nThreads − 1
// each thread needs to fetch a block of nItems words from global array blockData
shared M sharedValues[nItems × nThreads];
local M values[nItems];
for i = 0 to nItems − 1 do
    int index = nThreads × i + threadId;
    values[i] = blockData[index];
end
for i = 0 to nItems − 1 do
    sharedValues[nThreads × i + threadId] = values[i];
end
for i = 0 to nItems − 1 do
    values[i] = sharedValues[nItems × threadId + i]
end
// do something with values
```

Algorithm 4.5: Data transposition algorithm by Baxter [4]

In a first step, *nThreads* threads load *nThreads* words from every segment cooperatively, interleaving the reads to have coalesced access. The results are stored in shared memory in their original order. This is done *nItems* times, using $nThreads \times nItems$ words of shared memory in total. Each thread can then fetch its *nItems* words of data from shared memory, which does not require coalesced access for optimal performance. Bank conflicts can occur at this stage if *nItems* and the number of banks (currently 32) are not relatively prime. Assuming, for example, that $nItems = 8$ and that the fetch for $values[0]$ from thread zero hits bank zero, then thread one will hit bank eight, thread two will hit bank 16, etc., and threads four, eight, twelve, 16, 20, 24 and 28 will also hit bank zero, thus resulting in an eight-way bank conflict. This can be prevented by simply choosing *nItems* relatively prime to the number of banks.

This algorithm works well for transposing contiguous blocks of input data. However, in our case, we do not need to fetch a contiguous block. Instead, every thread needs to fetch a block of *nItems* elements from its own segment. We have adapted Baxter's algorithm to account for this; the result is shown in Algorithm 4.6.

Each thread still fetches one word from memory. To do this, it first calculates which segment it should fetch data from (*currentSegment*), which of the *nItems* words for this thread it should fetch in the current iteration (*offset*) and how many words from this segment have already been fetched in previous iterations (*segmentOffset*). It then fetches the right word and stores it directly in shared memory; the intermediate storing in a local array in Baxter's algorithm is not actually necessary. This is done iteratively for $segmentLength/nItems$ iterations, i.e. until the whole segment has been fetched and processed.

The problem of bank conflicts is the same in this case as in Baxter's original algorithm. If, however, we choose *nItems* relatively prime to 32, a different problem occurs. Global memory transfers on CUDA GPUs always occur in blocks of 32, 64 or 128 bytes. Memory bandwidth is therefore always wasted if *nItems* is not a multiple of eight. This problem is magnified by the fact that the offset from which a block of *nItems* words is fetched from every segment is a multiple of *nItems*. If *nItems* is not a multiple of eight, the block fetched in each new iteration is therefore probably not aligned to 32-byte-boundaries, which may result in even more necessary transactions, and therefore even more wasted bandwidth. Multiples of eight, however, are obviously never relatively prime to 32. With the current algorithm, we therefore have a choice between bank conflicts and wasted memory bandwidth.

We therefore improved the previous algorithm so that *nItems* can be chosen to be a multiple of eight without causing bank conflicts; Algorithm 4.7 is the result and is illustrated in

```
// threadId is between 0 and nThreads − 1
// each thread needs to fetch a block of nItems words from global array blockData
shared M sharedValues[nItems × nThreads];
local M values[nItems];
int segmentOffset = 0;
for segmentOffset = 0 to segmentLength step nItems do
    for i = threadIdto nItems × nThreads step nThreads do
        int currentSegment = i/nItems;
        int offset = i  mod nItems;
        int index = currentSegment × segmentLength + segmentOffset + offset;
        sharedValues[i] = blockData[index];
    end
    for i = 0 to nItems − 1 do
        values[i] = sharedValues[nItems × threadId + i];
    end
    // do something with values
end
```

Algorithm 4.6: Adaptation of Baxter's transposition algorithm for fetching from different segments

Figure 4.6.

```
// threadId is between 0 and nThreads − 1
// each thread needs to fetch a block of nItems words from global array blockData
shared M sharedValues[(nItems + 1) × nThreads];
local M values[nItems];
int segmentOffset = 0;
for segmentOffset = 0 to segmentLength step nItems do
    for i = threadId to nItems × nThreads step nThreads do
        int currentSegment = i/nItems;
        int offset = i  mod nItems;
        int index = currentSegment × segmentLength + segmentOffset + offset;
        sharedValues[(i/nItems) × (nItems + 1) + (i  mod nItems)] = blockData[index];
    end
    for i = 0 to nItems − 1 do
        values[i] = sharedValues[(nItems + 1) × threadId + i]
    end
    // do something with values
end
```

Algorithm 4.7: Improved transposition algorithm for fetching from different segments

The data intended for thread i is now stored in shared memory at the offset $i \times (nItems + 1)$ instead of $i \times nItems$. This way $nItems$ can be chosen to be $k \times 8$, but in terms of shared memory accesses, the algorithm behaves as if $nItems = k \times 8 + 1$ in the previous algorithm, thus resulting in optimal bandwidth utilization without bank conflicts in the last step. The downside is that $nThreads$ additional words of shared memory are needed for each thread, which can potentially limit the amount of possible parallelism. For our purposes, however, the improved algorithm was always faster than the original one.
Our baseline for a multireduce algorithm for non-commutative operators is therefore an algorithm which uses Algorithm 4.7 to fetch both labels and values from global memory. In the following sections, we will discuss the histogramming process itself, i.e. we will replace the "do something with *values*" line in the aforementioned algorithm.

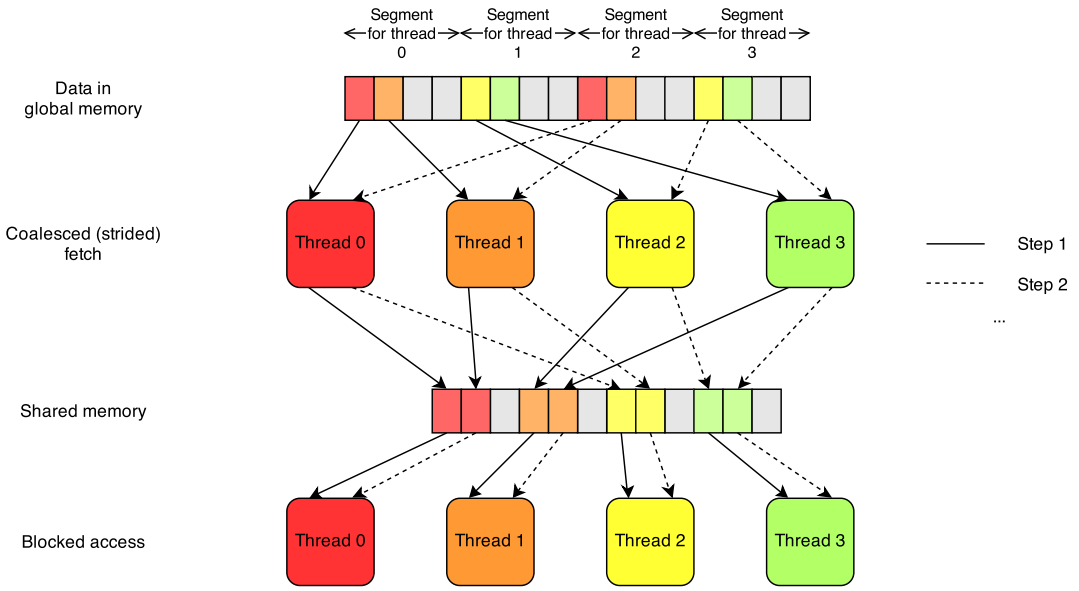


Figure 4.6: Transposition scheme for non-contiguous segments with $nItems = 2$, $nThreads = 4$ and a segment length of 4

4.3.3 Commutative operator

The problem becomes much simpler if the operator in question is commutative as well as associative. In this case the order in which inputs are processed does not matter, meaning that threads do not have to work on contiguous segments of the input, but can fetch data in any order. There is therefore no need to transpose inputs before using them. Additionally, one does not necessarily need to use a different set of buckets for every thread. It might, of course, still be advantageous to use several such sets for performance reasons, but in principle, all threads can work on the same set of buckets if we can ensure that their writes do not interfere with each other (e.g. using atomic updates). The very bad performance of Podlozhnyuk's 256-bin algorithm in the worst case suggests, however, that we should avoid a scenario where a great number of threads need to update the same memory location at the same time.
We will now discuss several ways to implement this algorithm.

4.3.4 No partitioning

At the beginning of this chapter we stated that any multireduce algorithm can be used either directly on the entire input, or on a smaller segment after partitioning the input. While this remains true in principle, the situation is somewhat complicated for this algorithm: Our adaptation of the sequential algorithm partitions the input data on a block level and thus essentially plays the same trick again which we have used on a global level before. There is therefore no meaningful definition of using this algorithm "without partitioning". Additionally, we have assumed throughout the previous sections that the results of the multireduce for single segments will be combined into a global result in a second step. This, however, is only the case if we use a partitioning scheme. It is therefore not possible to use this algorithm without a partitioning scheme.

4.3.5 Partitioning in global memory

Non-commutative operators

For non-commutative operators, each thread works on its own input segment using its own set of buckets. This has the advantage that there is no need for atomic operations, since there can never be any write conflicts. When searching for the best configuration for this algorithm, the only parameters that can be influenced are the number of blocks ($nBlocks$), the number of threads per block ($nThreads$), and the number of items processed by a thread in each run ($nItems$). $nItems$ and $nThreads$ are limited by the amount of available shared memory (of which $nThreads \times (nItems + 1)$ words are used per block for transposing the input). $nThreads$ is additionally limited by the amount of available DRAM, since $nBlocks \times nThreads \times nBuckets$ words are needed for storing intermediate results. For high numbers of buckets, this means that $nThreads$ has to be very low, and the possible amount of parallelism drops.
Since every thread can update any of its buckets in every step, bucket updates will necessary be uncoalesced most of the time. The arrangement of bucket sets in global memory that is required for the final reduction of all partial results does, however, have the effect that consecutive threads updating identical labels will lead to coalesced access. This algorithm may therefore perform slightly better for degenerate input data.
In their global memory histogram algorithm, Shams and Kennedy try to reduce the number of writes to global memory buckets by keeping mini-buckets of only a few bits in shared memory, and only writing to global memory when a mini-bucket overflows. This approach cannot be used for a multireduce, however, since it presupposes that the values written to a bucket can be kept in very few bits. While this is the case in histogramming, where every update just increments a bucket's value, multireduce deals with 32-bit values throughout, so that this improvement cannot be applied to a multireduce. We therefore have a very simple algorithm as shown in Algorithm 4.8, which is simply the sequential algorithm which stores its results in such a way that they can be reduced by the final step of our partitioning scheme. Note that this algorithm assumes that it runs within the main loop of the transposition algorithm 4.7 and therefore works with two local arrays $labels$ and $values$ of length $nItems$.

// int $labels[nItems]$ is a local array of labels fetched by the transposition algorithm
// M $values[nItems]$ is a local array of the corresponding values
// M $buckets[nBuckets \times nThreads \times nBlocks]$ is an array in global memory
for $i = 0$ **to** $nItems$ **do**
  int $offset = labels[i] \times nThreads \times nBlocks$;
  $buckets[offset] \odot= values[i]$;
end

Algorithm 4.8: Multireduce algorithm to be used inside the main loop of a transposition algorithm

Since we cannot fully utilize the available memory bandwidth due to uncoalesced accesses, a high cache hit rate would be particularly helpful. The cache hit rate of this algorithm, however, can be problematic. Both with large numbers of buckets and with large numbers of blocks or threads, not all buckets will fit into cache. Finding an optimal configuration for this algorithm is therefore non-trivial and best done empirically. We will make the following assumptions:

- We will assume that the distribution of threads among blocks does not have a major influence on the overall performance, as long as there is at least one block for each SM.
- We will also assume that a random distribution of inputs among buckets is the worst case for this algorithm, since there are no write conflicts that could impact performance

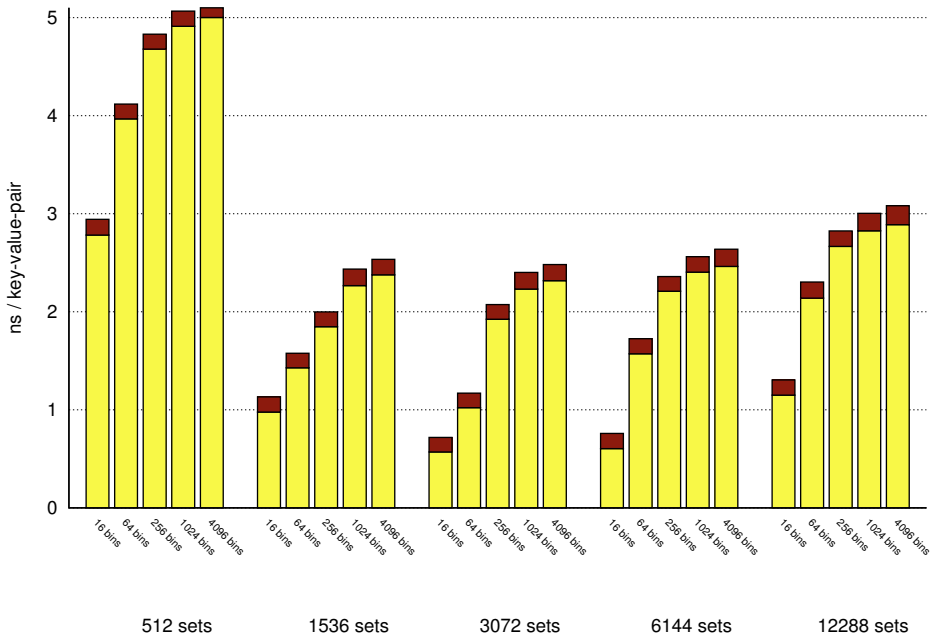


Figure 4.7: Performance for random data with different combinations of $nBuckets$ and $nSets$. The red part denotes the time used for reducing the partial results of all blocks.

negatively for degenerate, and the cache hit rate will necessarily be lower if all buckets from each bucket set are accessed randomly.

- We also assume that a larger $nItems$ is generally good for the algorithm's performance if it does not impact the amount of parallelism, as it means that more data is fetched at once, and more data can be processed by each thread at once.

All of these assumptions have been confirmed through small experiments for different parameter combinations.
We will therefore test the algorithm with a constant number of inputs, with a varying number of buckets and a varying total number of threads, and the largest possible $nItems$ in each case. The results for random input data can be seen in Figure 4.7.
The overall performance is disappointing. The single best result is 0.758 ns per key-value-pair, with 6144 threads distributed among 16 blocks (and therefore 6144 bucket sets) and only 16 buckets. Generally, lower numbers of buckets result in better performance, as does a higher number of threads for up to 6144 threads. The final reduce phase takes roughly constant time in all cases, but the main phase contributes far more to the algorithm's cost.
This pattern is what we expect if the cache hit rate is the most important limiting factor for the main algorithm's performance, as soon as a certain amount of parallelism is reached. We can confirm this by running the same configurations with a degenerate input distribution. In this case, the number of buckets is irrelevant to the performance, so we can just set it to 256 in all cases. We will not measure the final reduce phase this time, since its cost is almost constant. If cache misses are indeed the main limiting factor for the algorithm's performance, we expect much better performance for degenerate data. The result can be seen in Figure 4.8.
While the performance is still not optimal, it is a major improvement over the previous case. The most likely explanation for this behaviour is that the cache hit rate is indeed a major problem, although the coalescing of updates to the same bucket may also play a role. There are several possible ways to increase the cache hit rate for non-degenerate input data:

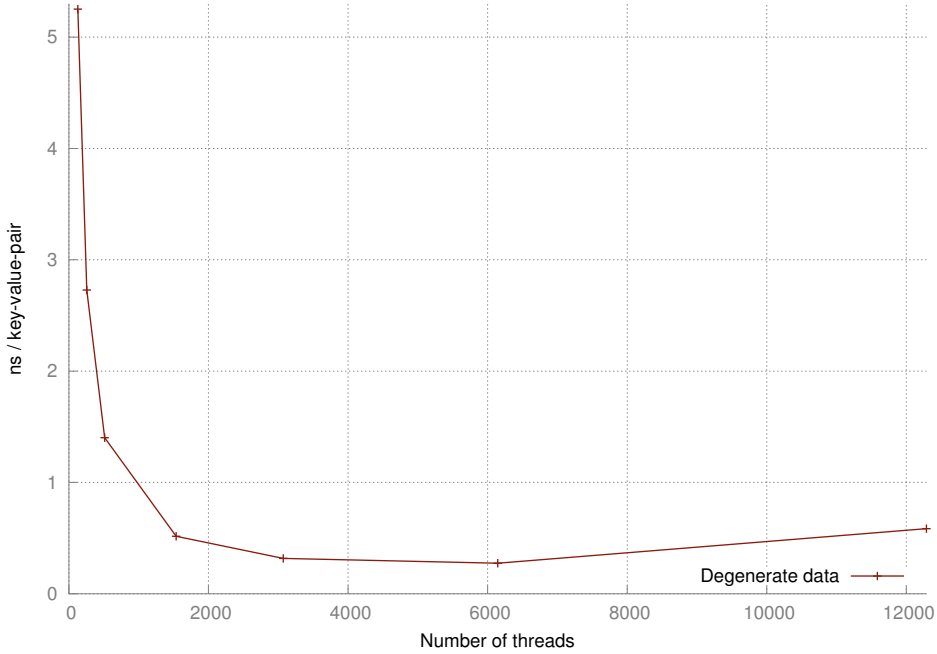


Figure 4.8: Performance of the main computation for degenerate input data

1. One could reduce the amount of parallelism even further, thus having less overall bucket sets which need to be cached.
2. One could employ a multi-pass scheme as seen before in the gather/scatter algorithm.
3. One could sort the input data by its labels in order to ensure that adjacent key-value pairs very likely use the same bucket, which should bring the performance closer to the degenerate case, at the cost of additional sorting.

The first idea does not seem promising in this case, since the previous benchmarks have shown that a large number of threads is actually necessary to achieve good performance.
The second approach may well result in an improvement, but a multi-pass algorithm whose number of passes depends linearly on the number of buckets is not work-efficient, and will not yield good results for very large numbers of buckets.
We will therefore follow the third approach. Since the slowdown for random data compared to the degenerate case is massive (a factor of two even for only 16 buckets), we expect that the input data needs to be almost completely sorted in order to profit from improved cache hit rate. Performance seems to be optimal in both cases with 3072 threads, so we will use this number of threads and sort the input completely for a first try.
We will use SRTS sort for this purpose, and we will exploit the fact that we know the total number $nBuckets$ of buckets, by only sorting by the first $\log_2 nBuckets$ bits of the labels (since we know that all other bits are always zero). This way, we can save computing time compared to a sort by all 32 bits, while still getting fully sorted output. We have adapted STRS sort to support partial sorting for this purpose. The input data is completely sorted on a global level, and the sorted data is then fetched and processed by the transposition algorithm.
Figure 4.9 shows the result. Even with fully sorted inputs, the performance is much worse for random data than for degenerate input data. A possible explanation for this is that coalesced updates, not cache usage, is the main factor improving performance in the degenerate case. Another possible reason is closely related: If a cache line is fetched which contains bucket

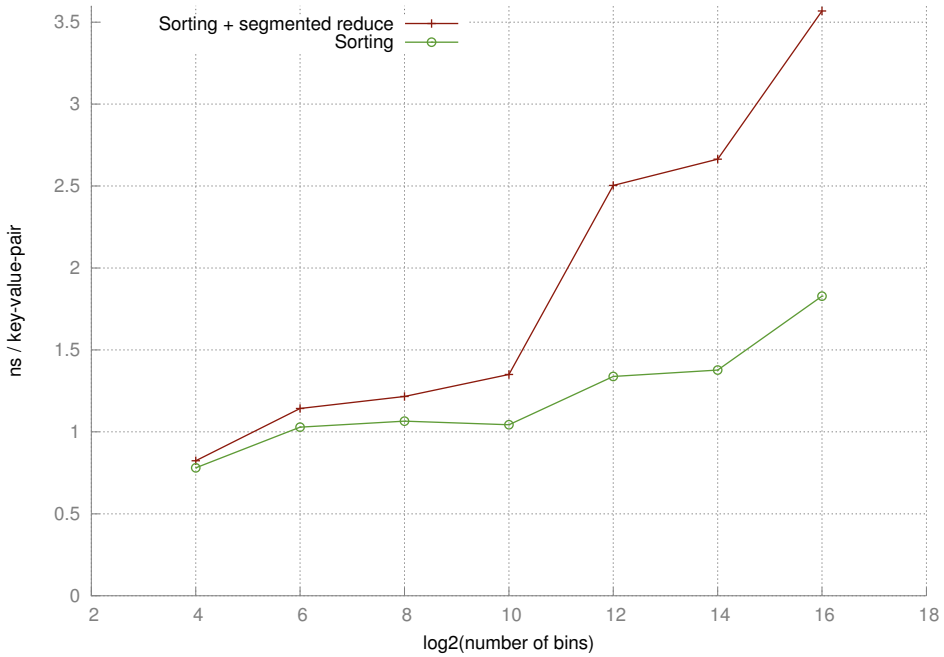


Figure 4.9: Performance of the main computation for sorted input data

i for some thread, the space before and/or after that cache line contains bucket i for other threads, meaning that one cache line will serve several threads if they all write to the same bucket. Sorted input, however, only guarantees that the buckets which single threads write to in subsequent iterations will likely be identical; different threads still very likely work on different buckets, thus potentially needing an entire cache line for themselves.
The absolute performance, while better than in the unsorted case, is also disappointing; the algorithm needs more than 2.5 ns per input for more than 4096 buckets and is therefore only twice as fast as the CPU implementation; this marginal speedup is not enough to justify the additional effort needed for moving the computation to the GPU. We will therefore no longer pursue this approach for non-commutative operators.

Commutative operators

If the used operator is also commutative, the number $nSets$ of bucket sets that are kept in global memory can be arbitrarily chosen without compromising the algorithm's correctness. There is, however, a tradeoff: If more than one thread works on a bucket set, all updates have to be atomic, which results in additional cost for every update, even if there are no actual write conflicts. It can also result in bad performance for degenerate input data, similar to Podlozhnyuk's 256-bin algorithm: If many or all indices hit the same bucket, and several threads have to share a bucket, then updates likely have to be serialized and performance suffers. On the other hand, less total bucket sets mean that there is more cache space for every bucket set, which is likely to increase cache hit rate. This is especially important if the indices in the input data are roughly equally distributed, whereas it does not play a big role in degenerate cases, since only one or very few buckets from each set have to be kept in cache in these cases.
The distribution of bucket sets among threads is simple: Since all updates of a warp are executed at once, the write conflicts between them should be kept to a minimum. Threads are assigned to warps by their thread ID, so a simple way to find the right bucket set for each thread is to calculate $threadId \mod nSets$. This distribution performs far better than, for

example, using one set of buckets for each block. The latter would result in a 32-way write conflict for *every* write if the input data is degenerate. The resulting algorithm is shown in Algorithm 4.9.

func multiReduce(int n, int $indices[n]$, M $values[n]$, int $nSets$, int $nBuckets$, M $buckets[nBuckets \times nSets]$) **:**
    // choose $nBlocks$, $nThreads$ and $nSets$ for best performance
    int $threadWork = n/(nThreads \times nBlocks)$;
    globalMultiReduce«$nBlocks$, $nThreads$»(n, $indices$, $values$, $nSets$, $nBuckets$, $buckets$, $threadWork$);
end

kernel globalMultiReduce(int n, int $indices[n]$, M $values[n]$, int $nSets$, int $nBuckets$, M $buckets[nBuckets \times nSets]$, int $threadWork$) **:**
    int *$blockIndices$ = &$indices[blockId \times nThreads \times threadWork]$;
    M *$blockValues$ = &$values[blockId \times nThreads \times threadWork]$;
    M $myBuckets$ = &$buckets[threadId \mod nSets]$;
    for $i = threadId$ **to** $threadWork \times nThreads$ **step** $nThreads$ **do**
        atomicAdd($myBuckets[blockIndices[i] \times nSets]$, $blockValues[i]$);
    end
end

Algorithm 4.9: Main kernel for commutative algorithm using partitioning in global memory

For non-degenerate cases, we expect the performance to be worse because of lower cache hit rates, just like in the non-commutative case.
The results for degenerate and random input data can be found in Figures 4.10 and 4.11, respectively.
For degenerate input data, the algorithm performs badly (ca. 1 ns per input) if all threads work on the same set of buckets, as expected, and improves if more bucket sets are used. The best performance (0.17 ns per input) is reached with eight bucket sets shared across all threads. The number of threads has no impact on the performance in this case.
For random data, the performance for low numbers of buckets is relatively high (0.10 ns per input in the best case). Using more bucket sets has no positive effect in this case. From a certain point on, the performance gets radically worse and quickly drops to 1 ns per input and worse. This is expected to be the result of a dropping cache hit rate. The used GPU has 512 kB of L2 cache, which suffices to store 2^{17} 4-byte integers, so we would expect the cache hit rate to drop if 2^{17} or, since not only the buckets will be caches, more likely 2^{16} buckets are accessed frequently. Note that the total number of buckets in global memory is the product $nSets \times nBuckets$.
Table 4.1 shows the data in more detail; parameter combinations which result in 2^{16} or more total buckets in global memory are shaded red. These are, in fact, the configurations which need 0.5 ns and more per input, whereas smaller numbers of total buckets need a maximum of 24 ns.
Since the bad cache hit rate significantly diminishes the algorithm's performance for high numbers of buckets, we will try to use a sorting approach in this case as well. We cannot make many reasonable assumptions about the performance with different numbers of buckets, bucket sets, and different amounts of sorting, since all those factors work together in a very complex way. Additionally, we do not just want to optimize the algorithm for random distributions, but we have to consider the degenerate case as well, which will not profit at all from sorting and will simply take longer than without sorting.
We have therefore tested the algorithm for any number of buckets greater than 14, with 1, 2, 4 or 8 bucket sets (since more than eight no longer improve performance in any case), and for any number of sorting bits. The results are far to extensive to show here in detail,

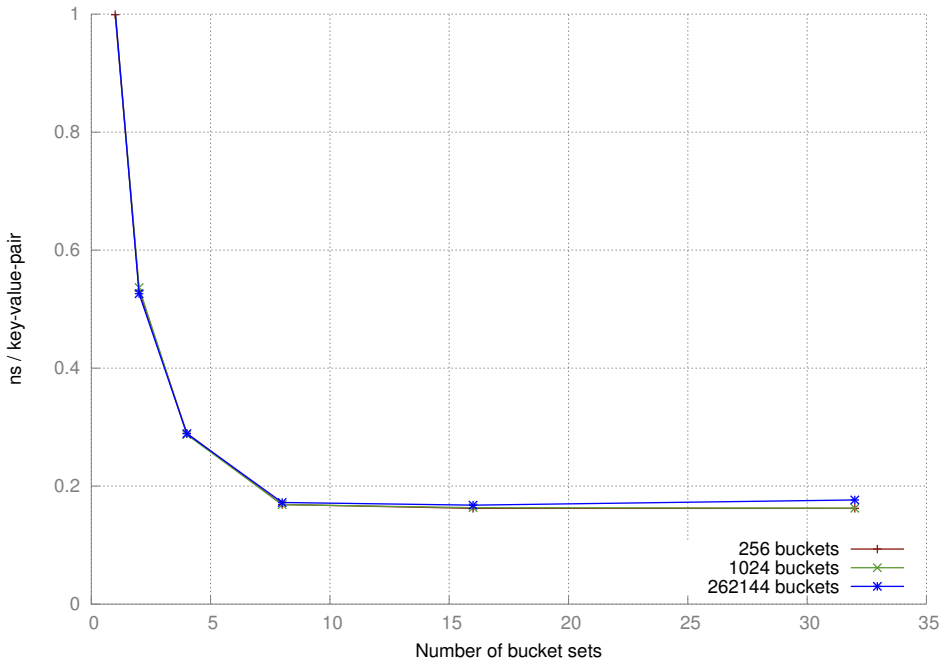


Figure 4.10: Performance with commutative operator for degenerate input data

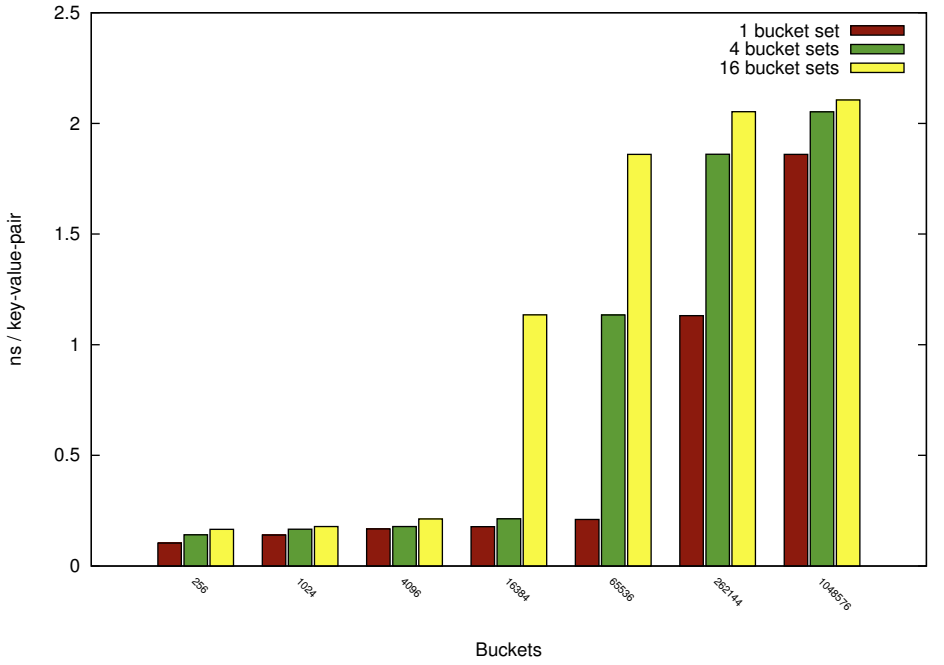


Figure 4.11: Performance with commutative operator for random input data

Sets \ Buckets	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
2^0	0.126174	0.157151	0.18861	0.195116	0.227656	1.148403	1.881422
2^1	0.144953	0.175636	0.201207	0.213474	0.527933	1.636045	2.020782
2^2	0.167653	0.191249	0.203438	0.241097	1.159856	1.886282	2.07879
2^3	0.177879	0.203151	0.206308	0.527598	1.636025	2.018021	2.111483
2^4	0.190973	0.206772	0.238288	1.160555	1.885561	2.078099	2.132013
2^5	0.199893	0.207527	0.524508	1.636431	2.014978	2.115052	2.148898

Table 4.1: Performance for commutative operators with different numbers of bucket sets and buckets

$\log_2 nBuckets$	$\log_2 nSets$	$\log_2 tBuckets$	$sortBits$	Rest comp.	Random	Degenerate
14	2	16	0	16	0.241097	0.288879
16	1	17	2	15	0.791643	0.778114
18	3	21	15	6	1.095509	0.983847
20	2	22	15	7	1.188077	1.107018
22	1	23	15	8	1.31336	1.359951
24	2	26	20	6	1.618827	1.431664

Table 4.2: Best case performance with previous sorting

but the best case for each number of buckets is shown in Table 4.2. By best case, we mean the configuration with the best worst-case performance, meaning that a configuration which performs decently for both random and degenerate data will win over one that performs well for random data but badly for degenerate input. $tBuckets$ refers to the total number of buckets, i.e. $nBuckets \times nSets$. The times for both random and degenerate data are given in ns per key-value-pair. 'Rest complexity' denotes $\log_2 tBuckets - sortBits$.
While these results seem almost random, there is a pattern which becomes more obvious when looking at the complete data set. Generally, a rest complexity of around seven seems to deliver good performance, and other combinations of $nBuckets$, $nSets$ and $sortBits$ which result in a similar rest complexity tend to have good results as well (as long as $nSets$ is larger than one; otherwise, the degenerate case performs badly). The fluctuation in the actual rest complexity can by explained by the sorting algorithm, which works fastest if $sortBits$ is a multiple of five.
The rest complexity can therefore be used as a measure to choose a good value for $sortBits$ and $nSets$ for a given number $nBuckets$. There is, however, some amount of unpredictability in the algorithms performance for different parameter combinations, which of course limits its general usability; an algorithm whose optimal parameters have to be determined empirically for each system can hardly be recommended for general use. Nevertheless, if properly configured, the performance of this algorithm is much higher than that of the sequential CPU implementation, particularly for high numbers of buckets, where cache misses diminish the CPU's performance.

4.3.6 Partitioning in shared memory

We have seen that an adaptation of the sequential algorithm can deliver relatively high performance when implemented in global memory, even though it uses a lot of uncoalesced global memory accesses. We will not try to achieve even better performance by using shared memory instead. Since most of the histogramming algorithms presented in this chapter work in shared memory, it seems reasonable to choose one of them as a starting point.

We have seen that the TRISH algorithm, which has the best performance of all histogram algorithms, builds on the memory layout of Nugteren's second algorithm. We have also seen that almost all of TRISH's improvements depend on the ability to store buckets as 8 bit integers. For a multireduce, this is not possible: The buckets' values are not incremented by one, as is the case with histograms; instead, a value from the input is added to it. This value can be any 32 bit integer, which obviously will not fit into anything but a 32 bit bucket. It is true that we assume the multireduce's output to be an array of 32 bit integers, which means that most input values should not use all of their 32 bits. Otherwise an overflow would likely occur, making the result useless for most purposes. Nevertheless, any single value may still be arbitrarily big, as long as the other values added to the same bucket are sufficiently few or small. Additionally, one should not forget that multireduce is a framework for many operations, not just addition, and for operations like min or max, there is absolutely no problem with large input values. The same is true for floating point values, whose space requirement remains constant even if they are added up.
Since Nugteren's algorithm is essentially TRISH without the modifications that do not apply to multireduce anyway, and beats every other presented algorithm in terms of worst-case performance, it seems sensible to use Nugteren's algorithm as a baseline and try to adapt it to perform a multireduce instead.
We will, unlike Nugteren, generally assume that input labels are 32 bit integers, as we have before. Since the multireduce does not generally work on image data, there is no reason to restrict oneself to addressing only 256 buckets. While 16 bits would likely suffice for most applications (and any algorithm that places multiple histograms in shared memory cannot handle more buckets anyway), using 32 bit integers in all cases keeps the input format general, so that this algorithm could easily be used in combination with different ones which support even higher numbers of buckets without changing the input format.
Our baseline is therefore an algorithm which takes 32-bit input labels and values, stores one bucket set of up to 256 buckets in shared memory, and does so in such a way that the buckets for every thread in a warp uses a different bank. Since each bucket uses 32 bits of storage, and adjacent 32-bit words of shared memory belong to different banks, this is trivial (see Figure 4.12); a block consisting of $nThreads$ threads uses $nThreads \times nBuckets$ words of shared memory and the bucket with label b for thread $threadId$ is located at offset $b \times nThreads + threadId$.
As before, we can make different improvements to this baseline depending on the commutativity of the used operator.

Non-commutative operators

As stated before, a non-commutative operator requires that each thread keeps its own histogram, and that input data is transposed in shared memory in order to have coalesced global memory reads. If we choose the method discussed in Section 4.3.2 to do this as we did before, we have to choose a number $nItems$ of items to be processed per iteration per thread again, and $nItems \times nThreads$ additional words of shared memory will be needed for transposing. There is very little opportunity for improvement in this case. Apart from loop unrolling, which is done by the compiler anyway, none of TRISH's improvements of Nugteren's algorithm can be applied to this case. The performance of this algorithm for different choices of $nItems$ can be seen in Figure 4.13.
The result shows that more buckets result in worse performance, which is expected, because more buckets mean more shared memory use per thread, and therefore less parallelism. A low choice of $nItems$ seems to be preferable in general, as this allows more parallelism, with one exception: In the case of 256 buckets, only one block of 32 threads can run on each SM in all cases because of the space requirement of the bucket sets; in this case, the amount of parallelism is the same for all configurations, and the one with $nItems = 32$ performs best

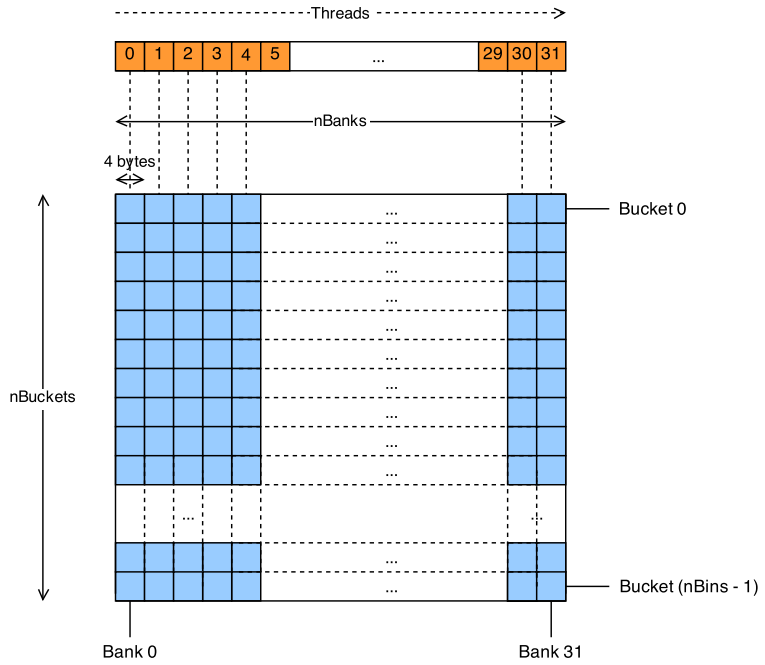


Figure 4.12: Shared memory layout of the baseline multireduce baseline

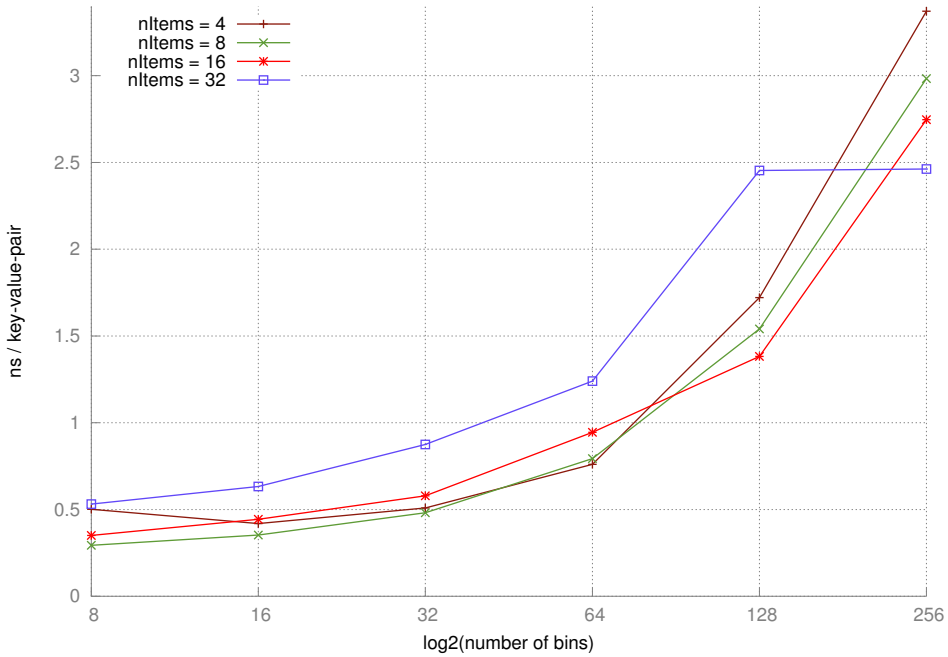


Figure 4.13: Performance of the shared memory algorithm for non-associative operators, data independent

because it processes more data at once than the others.
More importantly, the performance of this multireduce algorithm is much worse than that of Nugteren's original algorithm, in the sense that performing the multireduce takes a much longer amount of time per label-value-pair than it takes to perform the histogram per input value. The cost per input with 256 buckets, for example, is 0.27 ns in Nugteren's algorithm, and 2.46 ns in our case.
Three main reasons are likely the reason for the bad performance:

1. Additional shared memory space required for transposing inputs limits parallelism, though the impact of this should be relatively low for small choices of $nItems$.

2. Limited parallelism because of higher space requirements for buckets in shared memory. This is very likely a main factor, since the fact that we use 32-bit buckets where Nugteren uses 16 bits means that we can only have half as many resident threads at most, and probably less because of the extra space requirement for transposing.

3. Using 32-bit labels and additional values. While Nugteren's algorithm is not even close to using the full bandwidth of current GPUs, fetching eight times as much data from global memory as Nugteren's algorithm for every processed labels will necessarily result in much worse performance.

Since there seems to be no way to improve this algorithm any further, we have to conclude that this approach will not lead to a fast multireduce algorithm.

Commutative operators

If the used operator is commutative, there are more opportunities for optimization. Since there is no need for transposing the input, one can apply TRISH's improvement of batch fetching labels and values to registers. Because of this, and because the entire shared memory can be used for storing buckets, a simple implementation of this takes 1.53 ns per input for $nBuckets = 256$, which is better than in the non-commutative case (2.46 ns), but still much worse than Nugteren's algorithm (0.27 ns).
One thing we should note when evaluating both Nugteren's performance and the one of our simple multireduce is that both underutilize the hardware's capabilities. Since they both use one complete bucket set per thread, with 16 bits per bucket in Nugteren's case and 32 bits in the multireduce case, the number of threads that can run on a single multiprocessor is severely limited. A single one of Nugteren's histograms with 256 16-bit buckets takes up 512 bytes of shared memory. 48 kB of shared memory would therefore be enough to contain 96 such threads, or three warps. This is not nearly enough to fully exploit the capabilities of a current GPU, which needs hundreds of concurrent threads per SM to be fully utilized. In the case of the multireduce with 32 bit buckets, the situation is even worse. We either need to limit the number of buckets to 128, or reduce the number of warps to one. The former would make the algorithm less usable, while the limits parallelism and therefore performance performance, as we have seen.
We can also observe that the primary change in TRISH, using 8 bit buckets, is designed specifically to allow for more concurrent warps and thus better utilize the hardware. While this obviously works to some degree, as TRISH consistently outperforms Nugteren, the authors of TRISH report a GPU hardware utilization of 12.5%, which is still far from optimal.

Per Bank Bucket Sets We propose an alternative model with the following properties:

- There is one bucket set for each shared memory bank. The buckets are placed in shared memory in such a way that each bucket set lies occupies only one bank, meaning that

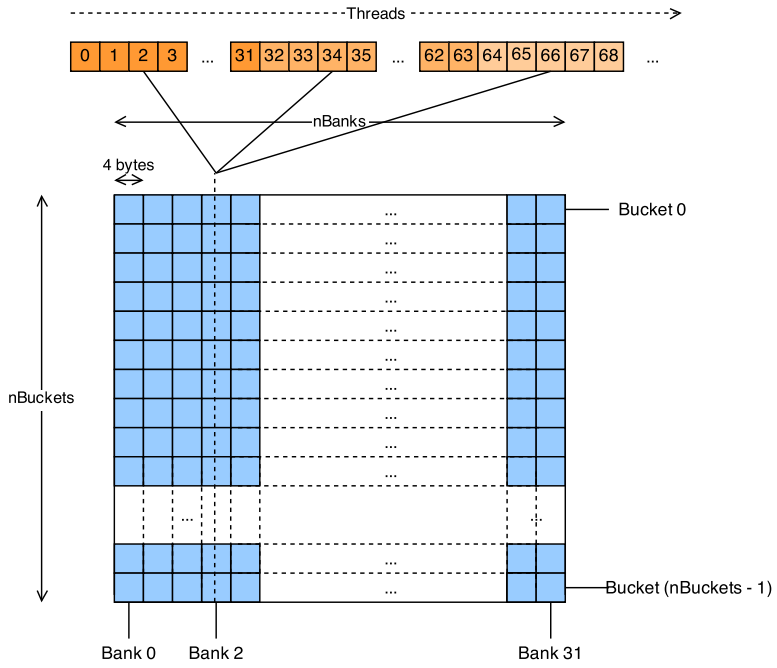


Figure 4.14: Memory layout of per-bank-multireduce algorithm

accesses to different bucket sets will be served by different banks. This is also the case in Nugteren's algorithm as well as in TRISH.

- Instead of using one thread per bucket set, as TRISH and Nugteren do, we use several. This means that multiple threads have to work on the same bucket set.
- We distribute the bucket sets among threads such that thread i uses bucket set number $i \mod nBanks$, where $nBanks$ is the number of shared memory banks (i.e. 32 on current hardware).

The resulting shared memory layout is illustrated in Figure 4.14. This approach has the following consequences:

1. There are no conflicts (neither actual write conflicts nor bank conflicts) between the writes of one warp, since $threadId \mod nBanks$ is distinct for each thread in a warp. This presupposes that the number of memory banks is identical to the number of threads per warp, which is the case on current hardware and is unlikely to change.
2. There may be write conflicts between threads that belong to the same block, but to different warps. In order to prevent threads from interfering with each other's updates, we need to use atomic updates on our shared memory bucket sets. This comes with a performance penalty even if there are no actual write conflicts. However, the number of writes that has to be sequentialized is at most the number of warps per block, not the number of threads per warp, as was in Podlozhnyuk's algorithm.
3. Several threads can work on the same part of shared memory. This is significant because for every value they write to shared memory, each thread has to fetch two values from global memory. During the latency of this memory access, the thread would otherwise be idle and the part of shared memory assigned to it would be unused.
4. Per bank bucket sets therefore increase the amount of possible thread level parallelism per unit of shared memory.

The main point here, and the reason why this seems to be a better approach than other algorithms where several threads share a histogram and can therefore generate write conflicts, is this: Threads of one warp are guaranteed to execute their memory accesses at the same time. It is therefore important to avoid potential conflicts between them, as those conflicts will necessarily impact performance. Threads of different warps, on the other hand, may be active concurrently or they may not. In fact, only a limited number (usually four on current NVIDIA GPUs) of warps can ever be active at the same time on each SM. Additionally, it is guaranteed that every warp spends a significant amount of its runtime waiting for data from global memory. During this time, another warp can work on the same data without generating any write conflicts.
In contrast to Nugteren and TRISH, this approach is obviously data dependent, since write conflicts are possible. However, we expect an improvement over the simple multireduce approach described earlier even in the worst case of all labels being equal. The reason for this is that each thread will be idle for a significant part of its lifetime, meaning that, even if all writes go to the exact same location, several threads updating the same bucket concurrently will result in higher throughput than only one thread doing so.
In a better scenario where all labels are used about equally often and the distribution is random, we would expect write conflicts to be rare even if two warps write to the same bucket set at the same time. Write conflicts will obviously be less likely the more buckets there are, which means that we can increase the number of warps per block even more for larger numbers of buckets. This increased parallelism can help counter the fact that larger bucket sets mean less histograms fit in shared memory, and therefore less blocks can reside in an SM. Because of this balancing effect, we would expect to find almost constant performance for different numbers of buckets, as long as at least one bucket set fits into shared memory.
The problem of possibly bad performance for degenerate input data can be solved by aggregating requests to the same bucket. Since we already fetch several labels and values at once, only a small adaptation is needed to compare pairs of them and, if they are equal, update the corresponding bucket only once. This has has very little impact on the performance for random data, but can completely remove the performance disadvantage for degenerate input.
We will make one final optimization. We have so far written the results for each segment to separate locations in global memory, as laid out in our partitioning scheme. This is necessary if the used operator is commutative, since in this case the order of the final reduction is important. It also makes sense if our main kernel works directly on global memory and there are good reasons for using more than one bucket set, as was the case in the previous section. Here, however, none of these conditions apply. Instead of using a second step to reduce the partial results, we will therefore use atomic updates to write the results for each segment directly to global memory.
The complete algorithm resulting from these considerations is shown in Algorithm 4.10.
Figure 4.15 compares the performance of multireduce with per-bank bucket sets (our solution) and per-thread bucket sets (the choice Nugteren and TRISH made for their histogram algorithms). For more than 16 buckets, the per-bank version performs a lot better, even though the primary change is very simple; in fact, its performance stays virtually constant around 0.08 ns, which is much better than Nugteren's algorithm and even slightly better than TRISH, which takes around 0.89 ns per byte on the same GPU. Note that this is despite both Nugteren and TRISH exploiting special attributes of histogramming.
We therefore have a very competitive multireduce algorithm for up to 256 buckets if the used operator is commutative.

Extension for more than 256 buckets The limited size of shared memory prevents it from being used for any number of buckets, not in the sense that performance would suffer too much, but in the sense that it simply cannot be run. Nevertheless, we would like to

```
func multiReduce(int n, int nBuckets, int nBanks, int indices[n], M values[n], M
buckets[nBuckets]) :
    // choose nBlocks for best performance
    // nThreads can be chosen arbitrarily, but this choice is fastest
    int nThreads = nBuckets × 4;
    int threadWork = n/(nThreads × nBlocks);
    perBankMR(n, nBuckets, nBanks, indices, values, buckets, threadWork);
end

kernel perBankMR(int n, int nBuckets, int nBanks, int indices[n], M values[n], M
buckets[nBuckets], int threadWork) :
    shared M blockBuckets[nBanks × nBuckets];
    int *blockIndices = &indices[blockId × nThreads × threadWork];
    M *blockValues = &values[blockId × nThreads × threadWork];
    M myBuckets = &blockBuckets[threadId  mod nBanks];
    for i = threadId to (threadWork × nThreads) step (2 × nThreads) do
        // fetch two labels and values at once
        int index1 = blockIndices[i];
        int value1 = blockValues[i];
        int index2 = blockIndices[i + nThreads];
        int value2 = blockValues[i + nThreads];
        // check if labels are equal
        bool equal = index1 == index2;
        value1 = value1 ⊙ (value2 × equal);
        value2 = value2 × (!equal);
        // if so, perform only one update
        atomicOp(myBuckets[index1 × nBanks], value1, ⊙);
        if value2 then
            atomicOp(myBuckets[index2 × nBanks], value2, ⊙);
        end
    end
    // accumulate partial results and write them to global memory
    for i = threadId to nBuckets step nThreads do
        int reduction = e;
        int j = threadId;
        do
            reduction ⊙= blockBuckets[i × nBanks + j];
            j = j + 1  mod nThreads;
        while  (j != threadId) ;
        atomicOp(buckets[i], reduction, ⊙);
    end
end
```

Algorithm 4.10: Kernel for per-bank-bucket-set multireduce

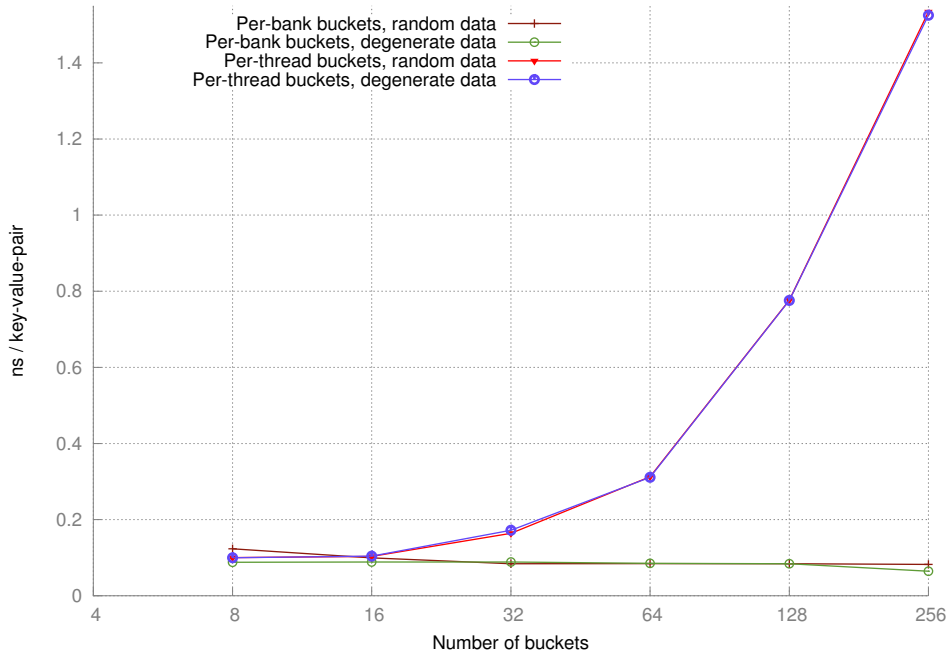


Figure 4.15: Performance of per-bank buckets vs. per-thread buckets multireduce

exploit this algorithm's good performance for higher numbers of buckets as well. One way to do this is the multi-pass scheme used by the scatter/gather approach described earlier: One could run over the input data several times, each time only processing the values whose label is in the current 256-bucket range. In addition to the obvious advantages (more buckets can be used) and disadvantages (many runs cost time; not useful if the number of buckets is much higher than 256, as the time needed for the many runs is too high), there are a few considerations that apply especially if the multi-pass approach is chosen for this particular algorithm:

- Since the single-pass algorithm still does not fully exploit the GPU's theoretical memory bandwidth, the concurrent (atomic) writes to shared memory seem to be the bottleneck that prevents even better performance, even in the case of random data distribution. In the multi-pass scheme, less such writes are performed. To some degree, this should accelerate every single run through the data, so that a two-pass multireduce with 512 buckets should take less than twice the time needed to do a 256 single-pass multireduce. If the concurrent writes are the only factor that is preventing better performance, and if the number of writes that is actually performed can be kept about equal to the 256-bin case (maybe by increasing the number of warps per block even further), the performance difference between 256 buckets and 512 buckets may actually be quite small. At some point, though, the additional cost for performing multiple passes will lead to significantly worse performance.

- With degenerate input data, one pass actually does all the work and should perform exactly as in the single-pass case or only slightly worse, since it has to check if the current label is within the current partition for every input. All other passes, however, simply read the input data, check if it belongs in the current partition, and then discard it. Since this should not take very long, the performance for a (small) number of passes should not be significantly worse than the single pass performance in cases with degenerate input data.

The other natural idea to extend the algorithm to more than 256 buckets is to partially sort the input data, so that it consists of segments which contain only values within a 256-bucket range. A block can then run over (a part of) the input and every time the data switches to a new segment, write the current bucket values to global memory, set the buckets in shared memory back to zero, and start working on the next segment. Compared to the original algorithm, this has the overhead of resetting buckets in shared memory, which probably cannot be avoided, and of constantly checking if the segment has switched. The latter is actually a slightly bigger problem, since we would prefer to fetch a number of labels and values from global value at once in several threads. Including a mechanism to check the segment and abort if necessary between processing several elements would nullify many of the advantages of a partly unrolled loop and make the implementation unnecessarily complicated. We have therefore decided to include a first step, which determines the offsets of each segment before the main algorithm is run. The main kernel, sketched in Algorithm 4.11, can then use this information and will know how many elements it has to process, so that it can process batches of input without performing constant checks.
The resulting algorithm executes the following steps:

1. It sorts the input data, again using SRTS sort.

2. It calls an additional kernel to calculate the offsets of each segment. The result is a vector $[o_0, o_1, ..., o_p]$ such that $p = \frac{nBuckets}{256}$, and o_i denotes the offset of the segment with the labels between $i \times 256$ and $(i+1) \times 256$.

3. It then calls an adapted version of the multireduce kernel to calculate the multireduce for every segment

This results in a lot of additional work compared to the original algorithm, but the amount of additional work is almost constant with respect to the number of buckets in the input data. The only significant change for more buckets are that the sorting algorithm has to sort by more bits, and that segments tend to change more often, so that the buckets in shared memory have to be reset more often. We therefore expect the sort-based extension perform worse than the multi-pass version for (relatively) low numbers of buckets, and better for higher ones.
The actual performance is shown in Figure 4.16, and is just what we expected. The multi-pass extension needs slightly less than twice the original time for twice as many buckets, while the performance of the sort-extension is almost constant, and does not surpass one ns per input. While the sorting extension is faster than the CPU implementation, the algorithm running in global memory performs better in all cases.

4.4 Sort-based approach

We have now discussed several ways to adapt the sequential multireduce algorithm for GPUs. Alternatively, a multireduce can also be performed using a sort-based approach, which we will discuss in this section. This approach consists of three main steps:

1. Sort all label-value-pairs by their labels.

2. Transform the sorted labels into segment start flags. A change of labels encodes the start of a new segment.

3. Perform a segmented reduction of the sorted values using the segment start flags. The result is the multireduce of the initial input.

```
func multiReduce(int n, int indices[n], M values[n], int nBuckets) :
    // choose nBlocks1, nBlocks2, nThreads1, nThreads2 for best performance
    int offsets[nBuckets/256]; getOffsets«nBlocks1, nThreads1»(n, indices, offsets);
    int threadWork = n/(nBlocks2 × nThreads2);
    sortExtensionMR«nBlocks2, nThreads2»(n, nBuckets, indices, values, offsets,
    threadWork);
end

kernel sortExtensionMR(int n, int nBuckets, int indices[n], M values[n], int
offsets[nBuckets/256], int threadWork) :
    int blockOffset = blockId × nThreads × threadWork int *blockIndices =
    &indices[blockOffset];
    M *blockValues = &values[blockOffset];
    shared int blockBuckets[nBanks × nBuckets];
    int curSegment = blockIndices[0]/256;
    int processed = 0;
    int offset = blockOffset;
    int segLength;
    for processed = 0 to threadWork step segLength do
        int segLength = offsets[curSegment + 1] − offset;
        Set all blockBuckets to zero
        Fetch and process segLength input labels and values
        Atomically add blockBuckets values to global buckets
        curSegment++;
        offset += curSegment;
    end
end
```

Algorithm 4.11: High-level view of the kernel for the sort-extension

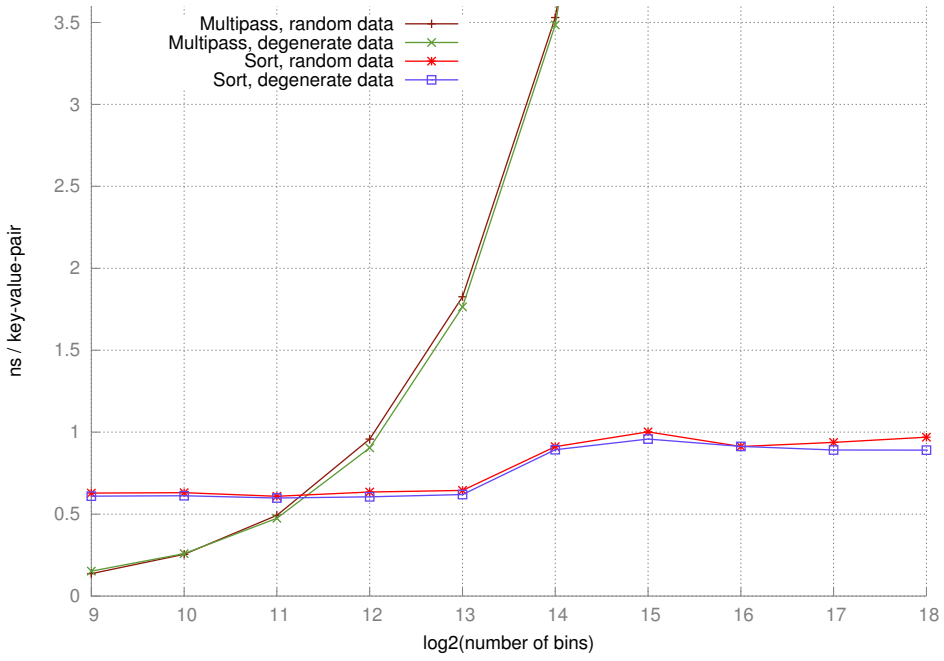


Figure 4.16: Performance of multi-pass vs. sort-based extension of per-bank multireduce

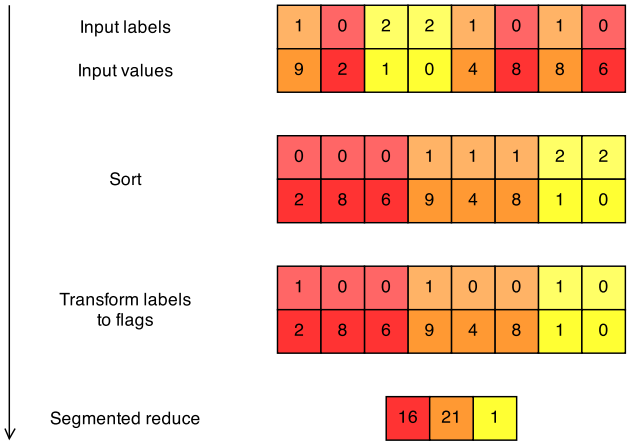


Figure 4.17: Multireduce by sort and segmented reduction, illustrated for addition on integers

function sortMultiReduce(int n, int $nBuckets$, int $indices[n]$, M $values[n]$, M $results[nBuckets]$) **:**
    stableSort($indices$, $values$);
    bool $flags[n]$;
    parfor $i = 0$ **to** $n - 2$ **do**
        $flags[i] = indices[i]! = indices[i + 1]$;
    end
    $flags[n - 1] = 0$;
    segmentedReduce($flags$, $values$, $result$, $\odot$);
end

Algorithm 4.12: Sort-based multireduce algorithm

This algorithm is expressed slightly more formally in Algorithm 4.12 and illustrated in Figure 4.17.
In order for this algorithm to deliver correct results with non-commutative operators, the used sorting algorithm needs to be stable, meaning that the order of values with the same key is not changed during the sort. Fortunately, this is always the case with radix sort. In theory it would be possible to use a non-stable sort algorithm for commutative operators if that were more efficient. However, since radix sort is the fastest sort algorithm available on GPUs, this is not the case. One can conceive of a case where the labels are not integers and therefore radix sort (which assumes that labels are stored in positional notation) cannot be used. In this case commutative operators might result in an advantage, but even that is doubtful, since the best comparison-based GPU sorting routines (e.g. Modern GPU's merge sort [12]) are also stable.
We will, however, assume the labels to be unsigned integers in any case, and will therefore not make a general distinction between commutative and non-commutative operators.
As before, there are different ways to implement this general algorithm based on the partitioning scheme, which we will discuss in the following sections.

4.4.1 No partitioning

In order to use this algorithm without partitioning, one only has to perform a radix sort of the entire input, followed by a transformation applied to every pair of adjacent indices, which computes the beginnings of segments, followed by a segmented reduction. Highly optimized implementations of all these operations are available in existing libraries.
The fastest published sorting algorithm, SRTS sort, has been discussed before. Both the Thrust library and Modern GPU offer a reduce-by-key functionality which already combines the last two steps. The Modern GPU implementation outperforms Thrust by a large margin, which is why we will use their implementation.
A natural approach to improving this algorithm would be to avoid writing intermediate results to global memory. Between the second and third step, this can be easily implemented: Where the segmented reduce would otherwise fetch a flag, it can simply fetch two labels, compare them, and proceed with the flag. Since data is fetched in blocks, and one of these two labels is also needed by another thread, this can be implemented with next to no overhead if threads share labels, e.g. by using shared memory. In fact, Modern GPU's implementation already does this.
Between the first and second step, leaving out the global store is not really feasible. Since (SRTS) radix sort ends with a scatter operation, adjacent labels cannot be easily and efficiently identified until the scatter is completed and the labels are in global memory. At this point, there is no opportunity to save memory bandwidth any more.
The performance of this algorithm for different numbers of buckets and for both randomly distributed and degenerate input data is shown in Figures 4.18 and 4.18, respectively.
Performance is slightly better for degenerate data. For only 16 buckets, the algorithm needs only 0.5 ns per input, but performance worsens for more buckets, again in steps determined by the used sort function, and is already over 1 ns per input for 4096 buckets. This algorithm is therefore clearly outperformed by variations of the sequential adaptation for commutative operators. Since those algorithms performed much worse for non-commutative operators, however, this sorting algorithm may well be the best option for those cases.

4.4.2 Partitioning in global memory

While the sorting algorithm can, of course, be implemented as part of the partitioning scheme and working in global memory, it seems unlikely that this would result in a performance advantage over either the shared memory version or the one without partitioning. The

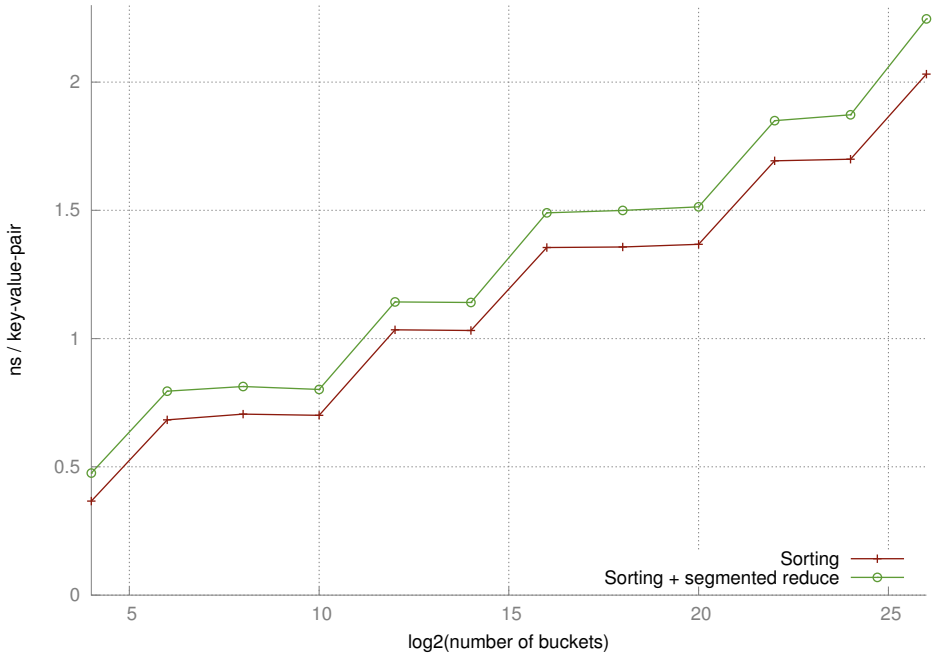


Figure 4.18: Performance of global sort & segmented reduce for random inputs

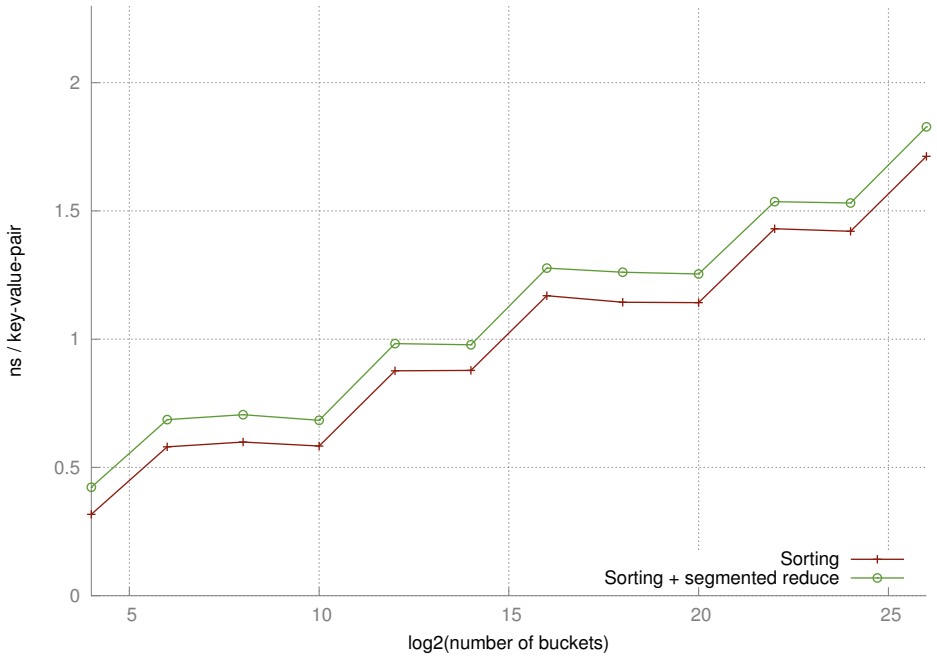


Figure 4.19: Performance of global sort & segmented reduce for degenerate inputs

radix sort, the transformation to flags and the subsequent segmented reduce are already highly parallelized algorithms, which would not benefit from additional parallelism. If it were advantageous to perform either the sort or the segmented reduce on a block level and combine the results later, existing library functions for sorting and segmented reduce would already do that.
The partitioning approach has some overhead in its final reduction step, and this overhead is only worth having if one can reasonably expect improved performance of the main computation in exchange. Since the latter is not the case here, we will not consider this option any further.

4.4.3 Partitioning in shared memory

The situation is different for a partitioning approach which is implemented exclusively in shared memory. Shared memory can be used to store the intermediate results after the sort and after the transformation from labels to segment start flags. In contrast to the global approach, which writes the results of the sort to global memory, which then have to be read again for the subsequent reduce-by-key, a shared memory implementation can calculate a multireduce while reading each key and value only once, and only writing the final results to global memory.
This, however, means that all data and intermediate results have to be stored in either shared memory or in registers for the entire runtime of each block. Efficient use of these resources is therefore vital, so we must first evaluate how much space is needed for each step, which again requires that we know the components we are going to use.
In this case, the range of available library implementations is very small. CUB offers a block-level radix sort, and Modern GPU has a block-level segmented reduce. The latter internally uses Modern GPU's segmented scan to compute a segmented reduce. Since the input-output behaviour of Modern GPU's segmented reduce algorithm is not optimal for our purposes, we will implement our own version of the segmented reduce which more closely fits our needs. We will, however, generally follow the pattern used in Modern GPU's segmented reduce function, meaning that we also use Modern GPU's segmented scan function in our algorithm.
Both the sorting function and the segmented scan need temporary storage in shared memory, and both accept inputs as function parameters, meaning that the input data does not have to be stored in shared memory. Both libraries are explicitly high-performance libraries which have been released in 2013, so we expect their performance to be close the best that is currently achievable.
To understand what is necessary to combine these two functions, we need to take a closer look at their input and output and their space requirements:

- CUB's radix sort can process several inputs per thread. It needs shared memory for storing temporary data; the required amount depends on the number of threads in the block and the number of bits sorted by in one step. Input labels and values must be stored in local arrays, which are given to the function as parameters. The radix sort does not address these arrays indirectly, which means that they can be stored in registers, and slow local memory is not needed (unless register usage is generally too high). When the sorting algorithm has completed, the local arrays of thread i will contain the key-value pairs which belong between those of thread $i-1$ and $i+1$.

  More formally: If threads $[t_0, t_1, ..., t_{nThreads-1}]$ simultaneously call the sort function, with thread i handing it a local vector $labels_i[nItems]$ as an argument, this can be interpreted as a single vector $allLabels[nThreads \times nItems]$, where $allLabels[i] = labels_{\frac{i}{nItems}}[i \mod nItems]$. This vector is then sorted, so that after the sort $allLabels[i] \leq allLabels[i+1]$ for all i. When the sorting has completed, $labels_i = [allLabels[i \times$

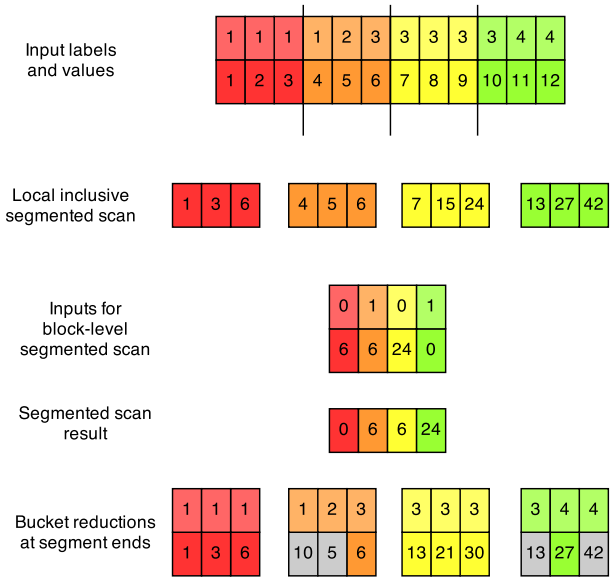


Figure 4.20: Parallel segmented reduce using segmented scan

$nItems], allLabels[i \times nItems], ..., allLabels[i \times nItems + nItems - 1]]$. The procedure for the vector of values is equivalent.

- Modern GPU's segmented scan processes one flag-value-pair per thread. It also requires shared memory as temporary storage. Inputs are accepted as function arguments, meaning that thread i calls the function with value v_i and flag f_i as arguments, and will get the corresponding result of the segmented scan, r_i, as a return value. For segment starts (whose segmented scan result value is necessarily zero in the exclusive case), the returned value is the reduction of the *previous* segment. The reduction of the last segment is written to a local variable through a pointer.

  More formally: If threads $[t_0, t_1, ..., t_{nThreads-1}]$ simultaneously call the segmented scan function, and thread i hands it a value v_i and a flag f_i, this can be interpreted as a value vector $[v_0, v_1, ..., v_{nThreads-1}]$ and a flag vector $[f_0, f_1, ..., f_{nThreads-1}]$, on which an exclusive segmented scan is the computed, resulting in a vector $[r_0, r_1, ..., r_{nThreads-1}]$ which contains the segmented scan of the input vectors. For each thread i, the function then returns a value x_i such that $x_i = r_i$ if $f_i = 0$, and $x_i = r_{i-1} \odot v_{i-1}$ otherwise.

Since both of these functions have to be executed by the same block, they both have to use the same number $nThreads$ of threads. The radix sort works most effectively if it processes several elements per thread (the default value is four). This means that in order to use the available resources efficiently, each block has to process several inputs per thread. While the segmented scan processes only one element per thread, the algorithm we will use to compute the segmented reduce can process several.
The entire algorithm is shown in Algorithm 4.13. Note that the calls to both library functions may look as if they were executed locally and independently by every thread, or alternatively as if they started a new kernel. Neither is the case. In both cases, all existing threads call a function which cooperatively performs a task across all threads in the block, i.e. the sorting and the segmented scan, respectively.
The complete algorithm works as follows: First, each thread fetches $nItems$ labels and values form the input data, which are then sorted across threads using the CUB function. Once labels and values are sorted, all threads write their new first label to shared memory, so that thread $i-1$ can fetch the first label of thread i. This is needed later for the segmented reduce.

Input : int $blockIndices[n]$, M $blockValues[n]$
Output: M $results[nBuckets \times nBlocks]$
local int $indices[nItems]$;
local int $values[nItems]$;
Fetch $nItems$ indices from $blockIndices$ to $indices$
Fetch $nItems$ values from $blockValues$ to $values$
cub::sort(indices, values);

// exchange first labels shared int $nextValues[nThreads]$;
$nextValues[threadId] = threadId == 0 \ ? \ nBuckets : labels[0]$;
int $myNext = nextValues[threadId + 1 \mod nThreads]$;

// first phase of segmented reduce int x;
bool $f = false$;
int $localScan[nItems]$; int $next = indices[0]$;
for $i \in [0...nItems - 1]$ **do**
  $x = i \ ? \ op(x, values[i]) : values[i]$;
  $localScan[i] = x$;
  int $current = next$;
  $next = i = nItems - 1 \ ? \ myNext : indices[i + 1]$;
  bool $flag = current \neq next$;
  $values[i] = flag$;
  if *flag* **then**
    $x = identity$;
  end
  $f = f \ || \ flag$;
end

// second phase
int y = mgpu::segmentedScan(x, f);

// third phase
for $i \in [0...nItems - 1]$ **do**
  $localScan[i] \mathrel{+}= y$;
  if $values[i]$ **then**
    $result[indices[i] \times nBlocks + blockId] = localScan[i]$;
    $y = 0$;
  end
end

Algorithm 4.13: Complete sort-based multireduce algorithm in shared memory

The segmented reduce algorithm has three phases (illustrated in Figure 4.20). The first phase computes a local inclusive scan of the local values, and simultaneously computes the input for the second phase, the segmented scan. x is essentially the value which is carried over to the next part, i.e. that the reduction of the last segment within the current thread's data. f denotes if a new segment starts inside the current thread's data. x is used as the value and f as the flag that is given to the segmented scan. The flags for each single element are stored in the array *values* to save register space, and to avoid recomputing all flags in the second phase. An if-statement is used in this part and again later, but since there is no else-branch, this is simply a case of predicated execution, and not real branch divergence.
In the third phase, the result y of the segmented scan is added to all elements of *localScan* which have the same label as the first element of the current thread. If the current element is the last of a segment, the new value of *localScan* is the reduction of the current segment and is written to global memory.
One problem is finding the optimal value for $nItems$: A higher choice for $nItems$ will mean that each thread can process more data in one run, at the cost of higher register usage. One approach to doing this is to calculate the number of registers each thread can use without lowering the amount of parallelism. Since there are 48 kB of available shared memory, and the temporary storage used about 36 bytes per thread, an SM can hold up to 1365 threads based on shared memory consumption. Since an SM has 65,536 registers, this leaves 48 registers for every thread. We reach 48 registers per thread if we set $nItems = 8$. In practice, however, using more registers always pays off in this case, possibly because the sorting algorithm gets more efficient the larger $nItems$ is, as is suggested in the CUB documentation. The highest setting for $nItems$ that does not lead to register spilling to slow local memory is $nItems = 12$, and the algorithm consistently performs best with this setting, so this is what we will choose for our benchmarks.
If we run the algorithm with different numbers $nThreads$ of threads per block and different numbers $nBucket$ of buckets, we get the results shown in Figure 4.21. The algorithm takes less than 0.2 ns per input for up to 2^8 buckets, which is close to the best algorithms we have seen so far, and under 0.5 ns per input for up to 2^{12} buckets for random input data. With more buckets, the performance for degenerate data remains acceptable, while it drops off sharply for random data.
The main reason for this is that, assuming random data, if $nBuckets$ is larger than the product $nThreads \times nItems$, it becomes likely that all or most labels in any given segment are distinct. In this case, neither the sorting nor the segmented reduce actually contribute to solving the multireduce problem, and the actual work is done during the final step, the reduction of partial results. In this case, the algorithm essentially degenerates to the multireduce performed as a reduction of vectors as outlined at the beginning of this chapter, which is already a bad algorithm for large numbers of buckets. To make matters worse, there is a significant overhead for the sort and segmented reduction of each segment without any real benefits.
Another issue prevents the algorithm from being used for very large numbers of buckets altogether: Lower numbers of threads per block as well as more buckets mean that more global memory space is needed to store the bucket sets for all threads. At some point, there is simply not enough memory to store all needed buckets, which results in an allocation error, meaning that there are strict limits within which the algorithm can be used that are not related to its performance.
At the start of this section, we mentioned that this method can be used for both commutative and non-commutative operators. This is true in principle, however, for non-commutative operators, we need to transpose the input data using Baxter's algorithm as we did before. The reason for this is that the radix sort, which is stable, will preserve the order of elements with the same label during the sorting process, and if the order in the input is wrong because

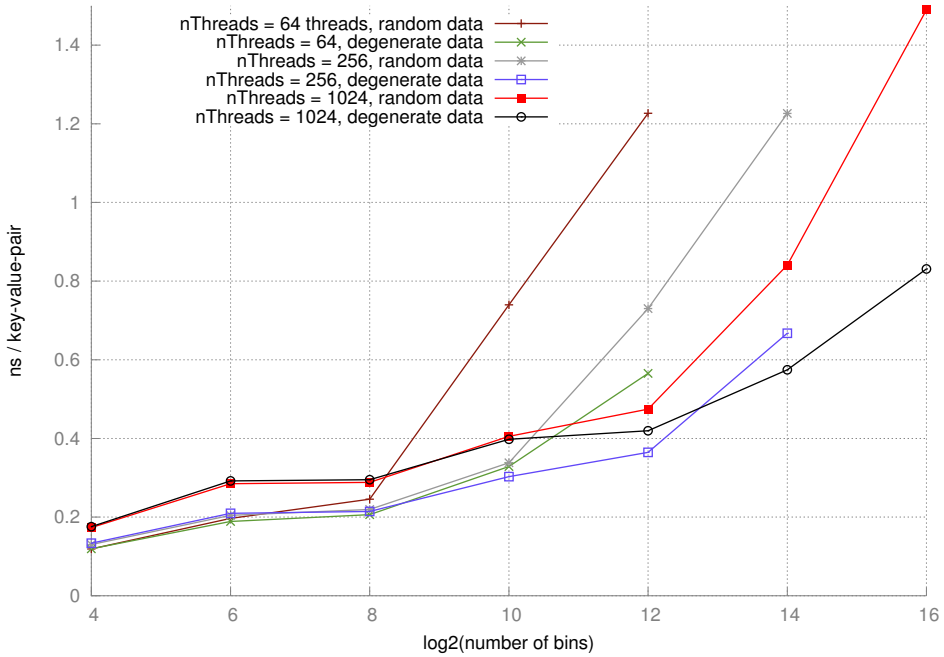


Figure 4.21: Performance of the sort and segmented reduce algorithm in shared memory for different numbers of threads per block

of strided fetching, the result can be wrong as well. Luckily, we can simply insert Baxter's algorithm for the lines which fetch the labels and values. Figure 4.22 shows that this does not seem to affect the algorithm's performance at all, so that this algorithm can be used for non-commutative operators without any additional limitations.

4.5 Application to related problems

4.5.1 Scatter with sorting

During our discussion of the adaptation of the sequential multireduce algorithm in global memory, we have seen that partially sorting input data for a multireduce can result in a performance improvement for large numbers of buckets. It seems natural to try to speed up scatter algorithms using the same method, since scattering to a large number of locations is a relatively common task on GPUs. This is evidenced by fact that He et al. [19] published a paper to specifically address the problem of cache misses in GPU scattering. Their multi-pass solution, however, is not work efficient and can therefore not be recommended for use with very large numbers of buckets. We have therefore tested if sorting can be used as a more generally applicable method to speed up scattering on GPUs.
The performance of a naive GPU scatter algorithm for random inputs with different numbers of buckets is shown in Table 4.3. For up to $nBuckets = 2^{16}$ buckets, it needs between 0.15 and 0.25 ns per second; if $nBuckets$ grows larger, slightly more than one ns per input is needed. The most likely explanation for this is that for $nBuckets > 2^{16}$, the buckets no longer fit into L2 cache.
We have tested the sort-scatter algorithm with different combinations of $sortBits$ and $nBuckets$ and concluded that the performance of the actual scatter phase for high numbers of buckets can be brought back down to the range of 0.1 to 0.2 ns per input if $\log_2 nBuckets - sortBits \leq 8$. This is valid for a simple scatter algorithm where every block works on a different, contiguous segment of the input. The reason why we have to sort by a relatively large number of

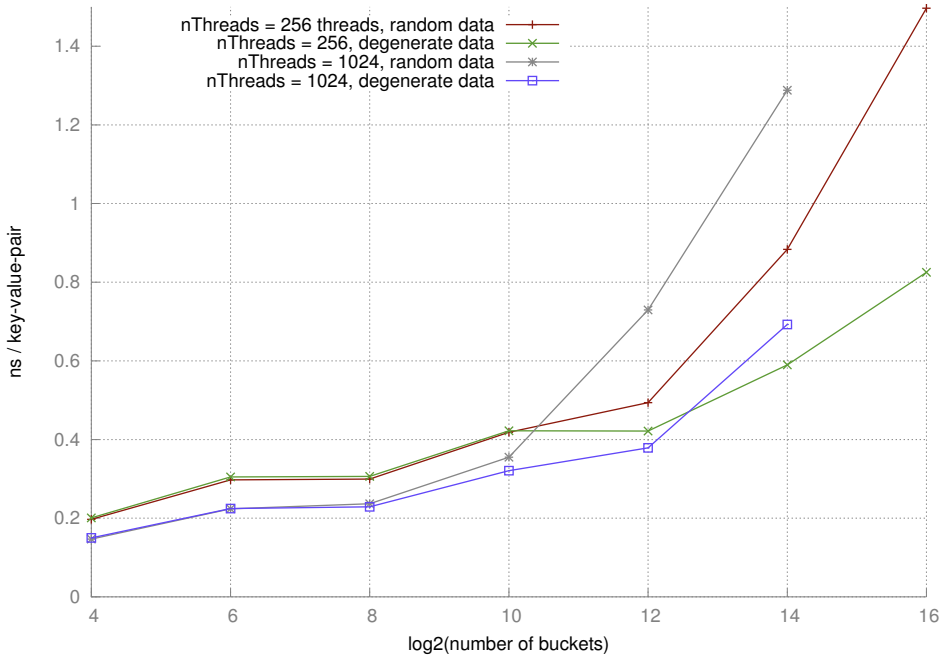


Figure 4.22: Performance of the sort and segmented reduce algorithm in shared memory with transposed input data

$\log_2 nBuckets$	ns per input
8	0.112742
10	0.154659
12	0.185251
14	0.200674
16	0.218198
18	0.590086
20	0.917882
22	1.007348
24	1.029998
26	1.063898

Table 4.3: Scatter performance for different numbers of buckets

bits is that different threads are working on different segments of the input at the same time, so that the buckets for the current segments of *all* threads have to be in cache for optimal performance, as opposed to only *one* set of buckets before. Therefore the range of buckets in every segment must be very small in order for the cache to be able to hold all current buckets for all threads.
We have tried to mitigate this problem by letting the threads of all blocks access the entire input data in a strided fashion instead of letting each block have its own segment. We hoped that the resulting access pattern is closer to a sequential traversal. However, the scheduling hardware ultimately controls which warps or blocks execute at what time, so that there is no way to guarantee that the input data is in fact accessed in a nearly sequential manner. Nevertheless, this approach, helped to raise the limit for an acceptable speed of the actual scatter phase to $\log_2 nBuckets - sortBits \leq 9$.
We have mentioned before that SRTS sorting is fastest if $sortBits$ is a multiple of five. In fact, sorting by any number $sortBits$ of bits which is not a multiple of five is generally not significantly faster than sorting by the next higher multiple of five. Sorting by five bits takes about 0.4 ns per key-value-pair.
Sorting could improve performance if, for some number of buckets, the time needed by the naive scatter algorithm is longer than that needed to sort by sufficiently many bits to bring the performance of the scatter phase back to a low level. More precisely, sorting can help if for some $nBuckets = 2^b$, $scatterTime_{nBuckets} > 0.2 + 0.4 \times \lfloor (b-9)/5 \rfloor$ (where $scatterTime_n$ is the time needed per input by a naive scatter to n buckets).
This is not the case, and we therefore have to conclude that sorting cannot help to speed up the scatter at least on our GPU.

4.5.2 Histograms

We have shown before that existing histogram algorithms can be adapted to perform a multireduce, but that many possible optimizations which can speed up histogram calculations cannot be applied to the more general multireduce. Nevertheless, our best multireduce algorithm slightly outperforms the best published histogram algorithm in terms of time per input, and does so for any input data distribution. It therefore seems obvious to try and transfer the ideas which helped us develop a fast multireduce algorithm back to the special problem of histograms.
We adapted the best histogram algorithm TRISH to make use of the two main concepts which make our multireduce algorithm fast, while leaving the structure of the algorithm unchanged. In particular, this means:

- We increased the number of threads per block from 64 to 512. Since there are still only 64 histograms in shared memory, we needed to make sure that for shared memory accesses, each thread behaves as if it were one of only 64. Since the behaviour of single threads can be differentiated only through use of the thread ID, this could be achieved by a simple trick: Wherever accesses to shared memory are concerned, $threadId \mod 64$ is used instead of the actual thread ID; for all other purposes each thread uses its actual thread ID as before.
- To prevent data corruption through conflicting writes, we made bucket updates atomic.
- To prevent bad performance for degenerate input data, we made use of the fact that four inputs are always processed at once, and bundle updates if several inputs target the same bucket.

The latter is even more effective when applied to histogramming in general and TRISH in particular than for the general multireduce, for two closely related reasons:

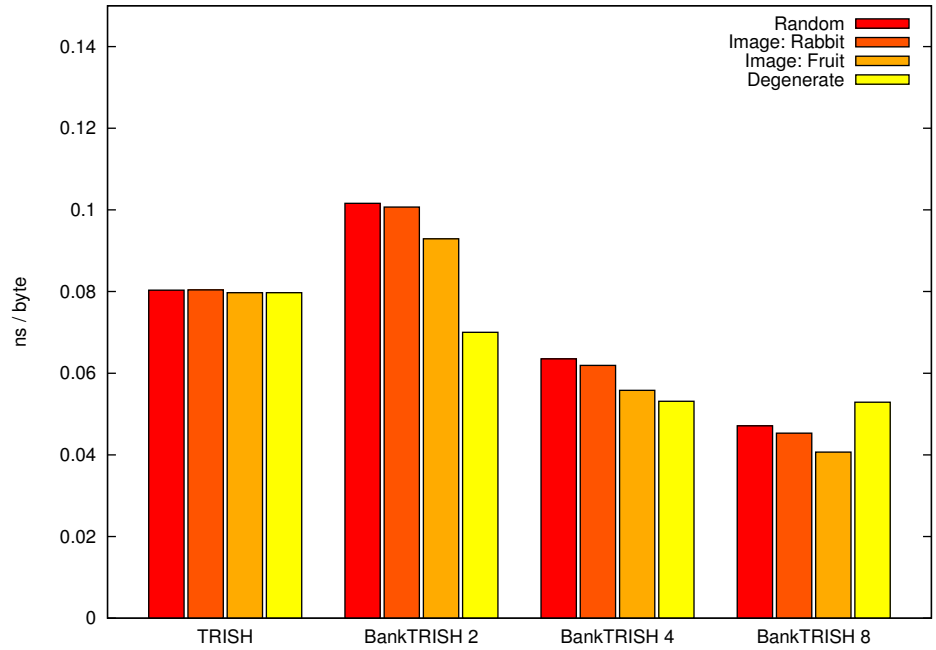


Figure 4.23: Runtime of original TRISH compared to BankTRISH with factor 2-8

1. TRISH (like most other histogram algorithms) fetches input data in words of four bytes. For image data, this means that each thread has the color values for a row of four adjacent pixels. Neighboring pixels, however, have a high correlation and are likely to have the same value in normal image data. While bundling updates mostly improves performance for completely degenerate input data in the case of the multireduce and leads to a minimal slowdown for input data distributions, we expect it to actually improve histogramming performance for normal image data as well.

2. The fact that buckets stored in shared memory are also byte sized, but updates to them always write an entire word, means that updates cannot only be bundled if they actually target the exact same bucket, but even when they target the same group of four buckets.

The resulting performance for random and degenerate input data as well as two actual images (we used two images used by Nugteren et al. to evaluate their algorithm), compared to the original TRISH implementation, can be seen in Figure 4.23. The line for "BankTRISH n" denotes our optimized version of TRISH (which lets several threads work on the same shared memory bank, hence the name) running with $n \times 64$ threads per block. Our implementation performs best for $n = 8$, and outperforms TRISH by 33% in the worst case and 40% on average in this configuration. Our implementation also outperforms both Nugteren's and Podlozhnyuk's implementations for both random and degenerate inputs; a comparison of all three algorithms in terms of performance can be found in Appendix A.
Our adaptation of the TRISH algorithm is therefore faster than any published histogram algorithm for all inputs. This is despite the fact that we made only the most basic changes to TRISH and did not make any attempts to increase the algorithm's performance aside from a basic transfer of two core ideas.

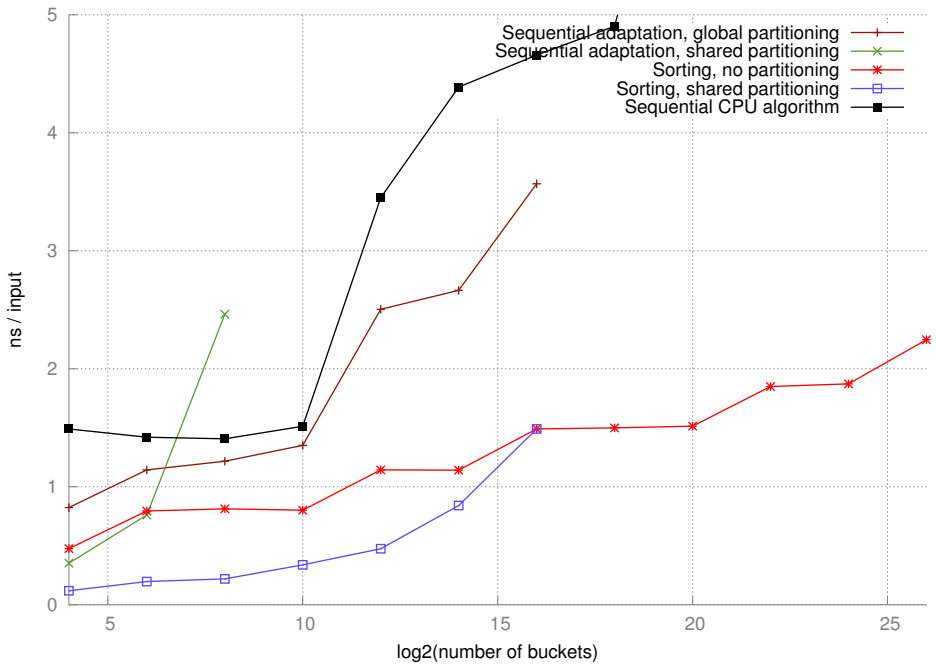


Figure 4.24: Comparison of multireduce algorithms for non-commutative operators

4.6 Conclusion

We have evaluated a number of different approaches for computing the multireduce on GPUs. In particular, we used two main algorithms, an adaptation of the sequential multireduce algorithm, and a sort-based approach. We combined both of these algorithms with different variations of a partitioning scheme, which allows work to be distributed on different levels of the GPU structure.

Our overall conclusion has to be a positive one: For all configurations, we found algorithms which perform three times better than our baseline CPU implementation in the worst case, and nine times better on average. The precise speedup for any given configuration depends to a large degree on the commutativity of the operator.

Figure 4.24 shows the performance of different algorithms with non-commutative operators for random input data. The speedup of the best GPU based algorithm over the CPU varies between a factor of three and a factor of twelve; the average speedup for any number of buckets is a factor of six. Sort-based algorithms outperform adaptations of the sequential algorithm by a large margin. The reason for this is that an enforced one-to-one relationship of threads to bucket sets hurts the cache hit rate (in global memory) and the amount of parallelism (in shared memory) of implementations based on the sequential algorithm. While we made some non-trivial algorithmic improvements by generalizing a data fetching algorithm by Baxter and transferring the partial sorting approach we used on the CPU to the GPU, these improvements could not overcome the structural problems of the algorithm for non-commutative operators.

The sorting algorithms, on the other hand, perform well independently of the used operator. The approach using shared memory in particular performs on the level of Thrust's reduce-by-key operation for up to 1024 buckets, needing only a third of a nanosecond per input, compared to 1.5 ns for the CPU algorithm. For larger numbers of buckets, the sorting algorithm without partitioning performs best. This is a general pattern: Due to the limited size, shared memory algorithms can often only be used for a limited number of buckets, but deliver great performance for those cases. Algorithms using global memory, on the other

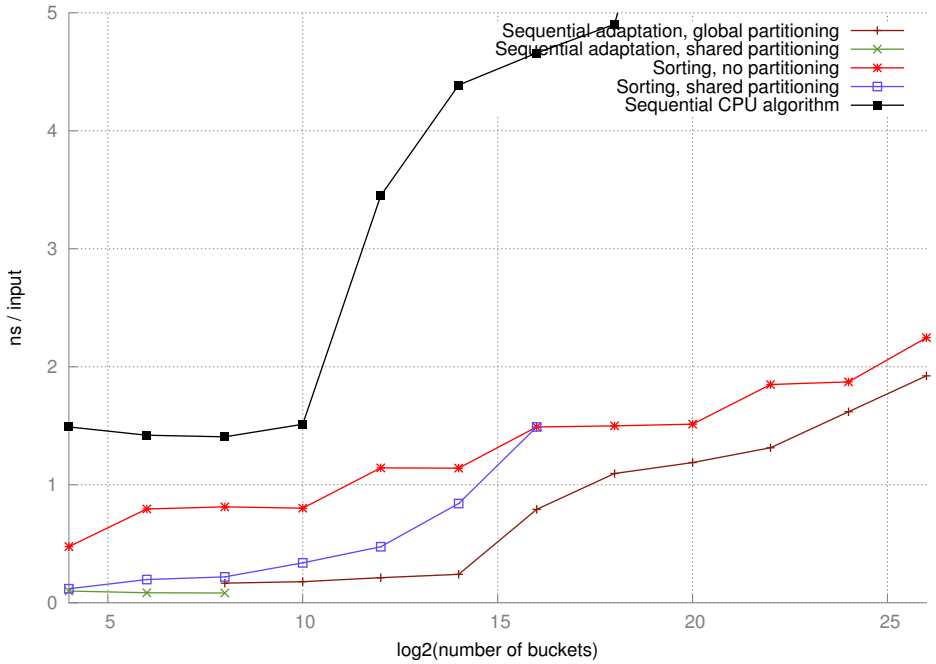


Figure 4.25: Comparison of multireduce algorithms for commutative operators

hand, have inferior peak performance, but scale better with the number of buckets. The absolute performance of the global sorting algorithm, for example, is not stellar (more than 2 ns per input for 2^{25} buckets), but it is a great improvement over the CPU algorithm especially for large numbers of buckets, since the CPU's performance drops to over 20 ns per input in these cases.

If the used operator is commutative, the adaptations of the sequential algorithm perform much better, while the performance of the sorting based approaches remains the same (see Figure 4.25). As soon as the ratio of threads and buckets sets can be varied at will, sequential-based algorithms are no longer hindered by low amounts of parallelism and cache misses, and outperform sort-based algorithms due to their simplicity. The average speedup over the CPU in this case is a factor of eleven; for different numbers of buckets, it varies between four and 17.

Our most convincing implementation is an adaptation of the sequential algorithm which uses a partitioning approach and works in shared memory (Section 4.3.6). We used an existing histogramming algorithm as a basis and extended it with a superior mapping between threads and shared memory, which results in a much higher possible amount of parallelism. As a result, this algorithm performs on par with or even better than the fastest published histogram algorithm, while being much more generally applicable. With an effective memory bandwidth of 100 GB/s, the algorithm does not completely utilize the theoretical memory bandwidth of 192 GB/s of a single GPU of the Geforce GTX 690, but any further improvements will necessarily stay under a factor of 2. The numbers may well be even better on different GPUs, as the GTX 690 has significantly less SMs (and therefore shared memory) than previous and subsequent GPUs, which directly affects this algorithm's performance.

For the adaptation of the sequential algorithm working in global memory, the partial sorting algorithm, which did not result in a significant improvement for non-commutative operators, significantly increased performance for large numbers of buckets.

We applied two of the improvements we made to multireduce algorithms to two of its special cases, the scatter and the histogram. While we had to conclude that a sorting algorithm cannot improve the performance of scattering on a GPU, we successfully applied the core idea

behind our best multireduce algorithm to the best existing histogram algorithm, resulting in a speedup of 30-40 % for all inputs. Our approach is therefore the fastest currently published histogramming algorithm for GPUs.

Chapter 5

Multiscan on GPUs

5.1 Adapting ordinary scan

function *Scan(M data[n])* **:**
  for $i = 0$ ***to*** $\log_2 n - 1$ **do**
    parfor $j = 0$ ***to*** $n - 1$ ***step*** 2^{i+1} **do**
      $data[j + 2^{i+1} - 1] = data[j + 2^i - 1] \odot data[j + 2^{i+1} - 1];$
    end
  end
  $data[n - 1] = 0;$
  for $i = \log_2 n - 1$ ***to*** 0 **do**
    parfor $j = 0$ ***to*** $n - 1$ ***step*** $2^i + 1$ **do**
      $M\ temp = data[j + 2^i - 1];$
      $data[j + 2^i - 1] = data[j + 2^{i+1} - 1];$
      $data[j + 2^{i+1} - 1] = temp \odot data[j + 2^{i+1} - 1];$
    end
  end
end

Algorithm 5.1: Parallel scan algorithm (adapted from [18])

The basis for the most-used parallel scan algorithms is Blelloch's tree-based algorithm from 1990 [6], a variation of which is shown in Algorithm 5.1 and illustrated in Figure 5.1. The algorithm is divided into two phases, the *upsweep* and the *downsweep*. The upsweep is essentially a reduction which uses the algorithm presented in Section 4.1 and keeps some of

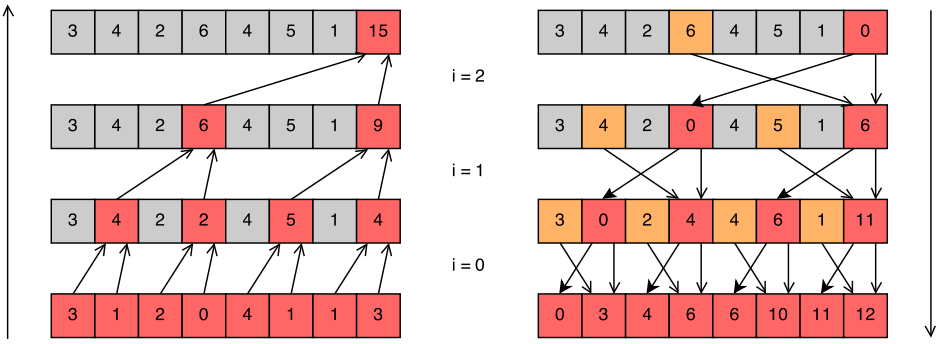


Figure 5.1: Work-efficient parallel scan algorithm as described by Blelloch (adapted from [18])

its intermediate results, and the downsweep uses these intermediate results to compute the scan. Like the parallel reduction, this algorithm has a work complexity of $\mathcal{O}(n)$ and a depth of $\mathcal{O}(\log n)$ and is therefore work efficient.
For GPUs, Horn [21] proposed the first scan algorithm in 2005, which, however, was not work-efficient. In 2007, Harris et al. [18] and Sengupta et al. [43] both proposed work-efficient CUDA implementations based on Blelloch's algorithm. Dotsenko [15] further improved on their implementation, but the algorithm still remained fundamentally the same. Merrill and Grimshaw [29] presented an overview of all existing scan algorithms in 2009 and further improved on the previous results.
In the previous chapter, we briefly described a way to use an ordinary reduce algorithm to perform a multireduce using vectors of the original elements. The same approach can be used to perform a multiscan with a scan algorithm, and the adaptations are completely analogous. The drawback of such an approach is identical, too: while it is a good algorithm for small (single digit) numbers of buckets, it gets very inefficient as the number of buckets gets larger.

5.2 Overview of possible solutions

The multiscan has been proposed as a fundamental primitive by a number of publications. It may therefore seem surprising that actual PRAM algorithms for the multiscan are quite rare. Here we will briefly discuss the main proposals and why most of them fail to present an algorithm which is usable today.
Ranade [39] proposed an abstract machine called the Fluent abstract machine, intended to avoid many of the pitfalls that come with coordinating large numbers of processors on traditional machine models. The purpose of said machine is to completely hide interprocessor communication and the corresponding complexity from the programmer. In order to accomplish this, it offers only a very limited set of instructions:

- The multiprefix instruction $\texttt{MP}(A, v, \odot)$, where A is a location in shared memory, v is a value of type T and $\odot$ is a binary associative operator of type $T \times T \to T$. If the previous value of A was $v_{original_A}$, this call will return $v_{original_A}$ and set A to $v_{original_A} \odot v$.

  If, however, each processor $P_i \in \{P_0, P_1, ..., P_{m-1}\}$ calls $\texttt{MP}(A_i, v_i, \odot)$ simultaneously, and for each P_{j_k} in a subset $\{P_{j_0}, P_{j_1}, ..., P_{j_{n-1}}\}$ of these processors $A_{j_k} = C$, and $j_0 < j_1 < ... < j_{n-1}$, then the final value of location C will be $v_{original_C} \odot v_{j_0} \odot v_{j_1} \odot ... \odot v_{j_{n-1}}$, and the value returned to P_{j_k} is $v_{original_C} \odot v_{j_0} \odot ... \odot v_{j_{k-1}}$.

  This multiprefix instruction working on registers is closely related to the multiscan function working on vectors which we want to implement on GPUs. Assume that $[v_0, v_1, ..., v_{n-1}]$ is a vector of values, $[l_0, l_1, ..., l_{n-1}]$ is a vector of labels in the range $\{0, 1, ..., m-1\}$ as in the definition above, and A is an array of length m, initialized to contain all zeros. If processor P_i calls $\texttt{MP}(A[l_i], v_i, \odot)$, the returned value will be equal to r_i as in the above definition for the exclusive multiscan.

- Convenience functions for memory manipulations, `READ` and `WRITE`, which are defined in terms of multiscan operations.

- Primitive set functions, including basic functions like inserting elements into sets and deleting them from it as well as membership tests, unions and intersections, but also including an `APPLY` function, which applies a unary function to all elements of a set.

While the fact that this instruction set suffices to perform general computations shows that the multiprefix is very powerful as a language primitive, Ranade's proposal offers little help in finding an efficient parallel algorithm for the multiscan. His proposed implementation requires special purpose hardware whose structure, a network of independent nodes with a complex

message passing and routing system, does not resemble GPU hardware or even PRAMs at all. In a different proposal, Cohn [10] proposes an implementation of the multiprefix instruction for a hypercube, which is also not applicable to current hardware.
Blelloch and Maggs [7] seem to suggest that a work-efficient PRAM algorithm for the multiscan exists, and refer to a paper by Matias and Vishkin [27]. Said paper, however, only seems to show that a *fetch-and-add* operation can be implemented efficiently on a PRAM using a probabilistic algorithm.
The fetch-and-add is a non-deterministic variation of the multiprefix. The `MP` instruction of the Fluent abstract machine which we just discussed can be turned into a fetch-and-add operation by dropping the condition that $j_0 < j_1 < ... < j_{n-1}$. Without this condition, all values written to the same location can be added up (or more generally, reduced) in any order. The fetch-and-add can also be defined as relation (not a function, since it is non-deterministic) on vectors. Analogously to the way we used the `MP` instruction to implement a multiscan of a vector above, the same can be done for a fetch-and-add.
Fetch-and-add is therefore not identical to the multiprefix (although Matias an Vishkin use the two terms interchangably, which may explain the confusion), and it cannot be used for the most prominent applications of the multiscan (e.g. radix sorting or sparse matrix vector multiplication). Moreover, it is inherently deterministic. Since our main goal is to create a primitive multiscan function which can be used by other algorithms without considering (or even knowing about) the underlying parallel implementation, non-determinism is exactly what we want to avoid, and we will not consider the fetch-and-add-algorithm any further.
The only existing work-efficient multiscan algorithm for PRAMs is therefore the one published by Sheffler [45], which we will present later in this chapter and discuss its applicability to current hardware. Like for the multireduce, we will also consider algorithms based on adapting the sequential multiscan algorithm as well as a sort-based approach.
For the sort-based approach and the adaptations of sequential algorithm, many of the occurring problems and solutions will resemble the ones which have been encountered for the multireduce, and discussing them again in full detail would be redundant. Wherever this is the case, we will therefore refer to the corresponding discussion in the multireduce chapter and only point out the adaptations that have to be made to perform a multireduce instead.
As before, there are three main ways to use any of these algorithmic approaches: Applying it directly to the entire input, or partitioning the input into several segments and applying the algorithm to all of these segments in parallel, which can be done using either global or shared memory. The structure of the latter approach for the multiscan is shown in Algorithm 5.2 and illustrated in Figure 5.2.
The input data is partitioned into *nSegments* segments. The multiscan for each segment is then computed in parallel, and the partial bucket values for each segment are written to global memory. The *nSegments* partial results for each of the *nBuckets* buckets are then scanned in the order of the segments they belong to, thus calculating the reductions for all buckets up to every segment start. In the final step, the results of the original multiscan for each segment are iterated through again, and the value of the corresponding scanned bucket is added to each result.
In the previous section, we have seen that the multireduce can be performed faster if it works with commutative operators. If this is the case, data can be fetched in strided order, which optimally exploits the available memory bandwidth, and can be processed directly without being redistributed among threads. More importantly, a commutative operator allows several threads or even blocks to work on a common bucket set, whereas a non-commutative operator enforces the use of a separate bucket set for every input segment. For these reasons, multireduce algorithms which use non-commutative operators are significantly slower.
For the multiscan, there is no distinction between commutative and non-commutative operators. Here, we *always* need to process data in a specific order and use dedicated bucket

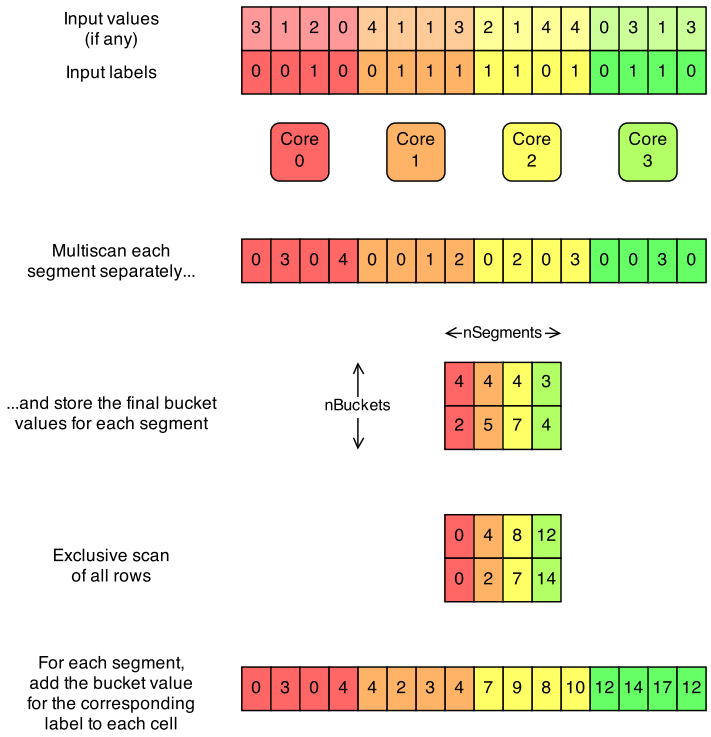


Figure 5.2: Parallel multiscan algorithm using partitioning with $n = 16$, $nSegments = 4$ and $nBuckets = 2$

function *partMultiScan(int labels[n], M values[n], M result[n])* **:**
  // int *nSegments* is chosen for best performance
  int $segLength = \frac{n}{nSegments}$;
  M *buckets*[$nSegments \times nBuckets$];
  parfor $i = 0$ **to** $nSegments - 1$ **do**
    int $offset = i \times segLength$;
    multiScan(&*labels*[*offset*], &*values*[*offset*], &*result*[*offset*], &*buckets*[i], $\odot$);
  end
  shared M *scannedBuckets*[$nSegments \times nBuckets$];
  parfor $i = 0$ **to** $nBuckets - 1$ **do**
    scan(&*buckets*[$i \times nSegments$], &*scannedBuckets*[$i \times nSegments$], $\odot$);
  end
  parfor $i = 0$ **to** $nSegments - 1$ **do**
    parfor $j = 0$ **to** $segLength - 1$ **do**
      int $offset = i \times segLength + j$;
      $result[offset] = scannedBuckets[labels[offset] \times nSegments + i] \odot result[offset]$;
    end
  end
end

Algorithm 5.2: Parallel partitioning strategy for multiscan

sets for each segment, meaning that we will generally expect the resulting performance to resemble that of multireduce for non-commutative operators, even if the multiscan itself uses a commutative operator. The reason for this lies in the nature of the multiscan, which calculates a result for every prefix of the input data. A partial result which has been obtained by processing, for example, every kth element up to index i of the input can therefore be used used for a multireduce with a commutative operator without any problems; several such partial results only need to be combined in the end. For the multiscan, the same partial result would be much less useful, as it cannot be used at all to calculate a result for any index before i, and the same is true for partial results which have been acquired in the process (i.e. the results after processing every kth element up to index $i-k$, $i-2k$ etc.). In other words, where for the multireduce with a commutative operator every thread can fetch and process elements of the input data in a strided order, and the results can simply be accumulated in the end, we would need to accumulate temporary results for *all* threads after *every* processed element for the multiscan, which would obviously ruin any performance advantage gained through strided data fetching.
If the idea of the underlying algorithm is similar, a multiscan algorithm will therefore generally closely resemble a multireduce algorithm which works for non-commutative operators following the same principle, independently of the operator used by the multiscan.
When implementing and benchmarking the partitioning scheme for different algorithms, we will use a function from the CUDPP library to scan the bucket values. This function is ironically called "multiscan", but it simply performs several ordinary scans in parallel. We use it to scan the partial results for all buckets in parallel.
The last step, which adds the results of the scan to the results of the multiscan in the first phase, is trivial to implement with reasonable performance. The only major design decision one has to make is whether a block should fetch all scanned bucket values to shared memory so that it has quick access to them, which might limit the amount of parallelism because of the shared memory demand of each block, or if they should be fetched from global memory for each element. In practice, we found that the shared memory version has better performance if $nBuckets < 2^{10}$, and the global memory version is faster otherwise, so we will use this configuration during the following benchmarks.

5.3 Adaptation of sequential algorithm

The simple sequential algorithm for multiscan differs from the one for the multireduce in only one detail: Before adding a new input value to the current bucket, the bucket's current value is written to the output vector. As a result, the options for adapting the sequential multiscan algorithm for GPUs are virtually identical to those for the multireduce, which we discussed in Section 4.3. The only exception is that we cannot make any improvements for commutative operators for the reasons outlined in the previous section.
The algorithms in this section would therefore be identical to those for the multireduce with non-commutative operators, only with an additional step writing out the current bucket values. This means, however, that the performance of these algorithms would be on the level of their multireduce counterparts at best, and probably slightly worse because of the additional writes. Since the performance of the non-commutative multireduce algorithms was already worse than any algorithm that could be recommended for general use, and the algorithm's implementation is trivial, we will not discuss it or its performance in detail here. Plots showing the algorithm using partitioning in both global and shared memory can be found in Appendix B.

5.4 Adapting Sheffler's algorithm for GPUs

The only existing parallel algorithm for multiscan designed for PRAMs is the one published by Sheffler in 1993 [45], which has a work complexity of $\mathcal{O}(n)$ and a step complexity of $\mathcal{O}(\sqrt{n})$. We will use Sheffler's terminology, which assumes a plus-multiscan, in our algorithm description. The algorithm does, however, work for any associative operator and also does not assume commutativity.

```
struct spinerec {
  int rowsum;
  int spinesum;
  int multisum;
  spinerec* spine
};
```

Listing 5.1: Spinerec structure as used by Sheffler's multiscan algorithm

For an input of length $n \times n$, and a multiscan using m buckets, we allocate an array of `spinerecs` (see Listing 5.1) of length m called *buckets*, and one of length $n \times n$ called *nodes*. We assume that the input labels are stored in an array called *labels* and the values in an array called *values*. First, we initialize all nodes and buckets in parallel.

parfor $i = 0$ **to** $n^2 - 1$ **do**
  $nodes[i].rowsum = e$;
  $nodes[i].spine = \&buckets[labels[i]]$;
  $buckets[labels[i]].spine = \&buckets[i]$;
  $buckets[labels[i]].rowsum = e$;
end

After this, the *spine* pointer of all buckets are pointing to themselves, and the *spine* pointers of all nodes point to the bucket belonging to their label. From now on, we will imagine the nodes to be arranged in a square of $n \times n$ spinerecs, ordered from left to right and from bottom to top as depicted in Figure 5.3 for $n = 3$. In the first phase, a *spinetree* is built for each label, the spine pointers of the nodes of each label are arranged into a tree structure. The following steps are followed for the nodes of every row in parallel, starting at the top and moving down from there:

// spinetree phase
for $r = n$ **to** 1 **step** -1 **do**
  parfor $i = (r-1) \times n$ **to** $(n \times r) - 1$ **do**
    $nodes[i].spine = \&buckets[labels[i]].spine$;
    $buckets[labels[i]].spine = \&nodes[i]$;
  end
end

Note that in this phase, all threads perform potentially conflicting updates on the spine pointer of the buckets. The algorithm therefore presupposes a CRCW-PRAM. Since no assumption is made about the result of such a write, however, except that one of the writes succeeds, an ARBITRARY CRCW PRAM suffices.
If $m = 2$ and all labels of even nodes are equal to zero, whereas labels of odd nodes are one, the spine pointers could look as depicted in Figure 5.3 after this phase. The spineree is created in order to allow conflict free updates in the following phases.

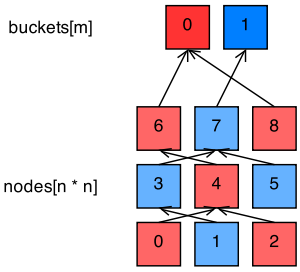


Figure 5.3: Spinetree for $n = 3$, $m = 2$ and all labels $l_i = i \mod 2$

In the rowsum phase, the following steps are executed for the nodes of every column in parallel, starting with the left one and progressing to the right.

// rowsum phase
for $c = 1$ **to** n **do**
  parfor $i = c - 1$ **to** $n^2 - 1$ **step** n **do**
    $nodes[i].spine \rightarrow rowsum \odot= values[i]$;
  end
end

After this, the *rowsum* each spine element will contain the sum of the values of all nodes that share its label in the row below (or several rows below, if there is no node with the same label in the row directly below). Now, we do the following for the nodes of every row in parallel, starting at the bottom:

// spinesum phase
for $r = n$ **to** 1 **step** -1 **do**
  parfor $i = (r - 1) \times n$ **to** $(n \times r) - 1$ **do**
    if $nodes[i].rowsum \mathrel{!}= e$ **then**
      $nodes[i].spine \rightarrow spinesum = nodes[i].spinesum + nodes[i].rowsum$;
    end
  end
end

The *spinesum* of each spine element should now contain the sum of the values in all rows below that have the same label. In the final step, the following steps are executed for every row in parallel, starting with the bottom row:

for $c = 1$ **to** n **do**
  // multisum phase
  parfor $i = c - 1$ **to** $n^2 - 1$ **step** n **do**
    $result[i] = nodes[i].spine \rightarrow spinesum$;
    $nodes[i].spine \rightarrow spinesum \odot= values[i]$;
  end
end

While it seems reasonable in general that this algorithm will compute a multiscan, there is one detail which deserves some additional attention. The if-clause in the spinesum phase seems somewhat suspect. Its purpose is to prevent nodes which are not spine elements from overwriting their spine element's *spinesum*, and it achieves this goal, since *rowsum* has been initialized to the neutral element for all nodes, and has only been changed in later steps for spine elements. The purpose of the statement is therefore not related to the algorithm's performance, it is necessary to ensure the algorithm's correctness.

However, if the condition is stated the way it is in Sheffler's algorithm, it has an unintended side effect. While non-spine elements are guaranteed to have a *rowsum* equal to e, spine elements are not guaranteed to have a *rowsum* different from e. If all values which are summed up to be the *rowsum* of a certain spine element are equal to the neutral element, then this spine element will not update it's spine elements *spinesum* in this step. This is a problem because its own *spinesum* may not be equal to the neutral element, but it would not make its way up the spine in this iteration. The algorithm as stated by Sheffler therefore works correctly only if all values are distinct from the neutral element. Nevertheless, we will continue to evaluate its applicability to GPUs.
Some general observations:
The algorithm depends strongly on the use of pointers that make up the spine for each bucket. Following pointers frequently means a lot of memory accesses. It also features a very small amount of arithmetic operations, meaning that there is very few computation to be done between the memory accesses. This is especially important because, as mentioned before, this algorithm is intended for PRAMs, and therefore assumes SIMD execution, which has to be enforced through block-level synchronization on GPUs. In fact, its correctness depends on synchronization of all threads after each step, i.e. after each iteration of the sequential loops in each phase except for the initialization. This need for very frequent synchronization means that there will be little freedom to schedule another warp of the same block while a given warp is waiting for data from memory, and memory access latency therefore cannot be hidden effectively. Because of the large number of memory accesses, this will necessarily reduce the maximum achievable performance.
In addition to frequently accessing spine elements, the algorithm uses both the label and the value for each node twice. This means that they either have to be fetched from DRAM twice, obviously resulting in additional cost, or they have to be stored somewhere in the meantime. Storing them in private registers is problematic because the access pattern is not the same for the value accesses; in one phase, they are done by row, in the other phase by column, so a natural algorithm would have different threads needing the same value. For the same reason of different access patterns, it is impossible to have coalesced reads for the values both times. It therefore seems sensible to store both labels and values in shared memory after fetching them once, unless shared memory is needed for other purposes, in which case using more shared memory than absolutely necessary may limit parallelism.
The main problems, frequent memory accesses and synchronizations, lie at the core of the algorithm and cannot be avoided without basically abandoning the algorithm altogether, and we will therefore not try to do so. Instead, we will implement the algorithm for GPUs making only minor adaptations wherever they can obviously lead to better performance. While we do not expect it to perform very well, the alternatives discussed so far do not fully exploit the GPU's capabilities either, so Sheffler's algorithm may still be a valuable contribution.
As for all other algorithms we have discussed, we will now discuss if and how this algorithm can be implemented either with or without the partitioning scheme we have used throughout.

5.4.1 No partitioning

We have alread mentioned that, during each of the four main phases (excluding the initialization), the algorithm's correctness depends on the fact that steps are executed for one row or column after they have been executed for all nodes of the previous row or column. But on GPUs, this is impossible on a device level. As a result, this algorithm can only be implemented for GPUs if all threads are in the same block. That, however, means that only one of the GPU's SMs can be used, which prevents the algorithm from achieving good optimal performance to begin with.
It is therefore much more reasonable to let several blocks use the algorithm in parallel by using the partitioning scheme.

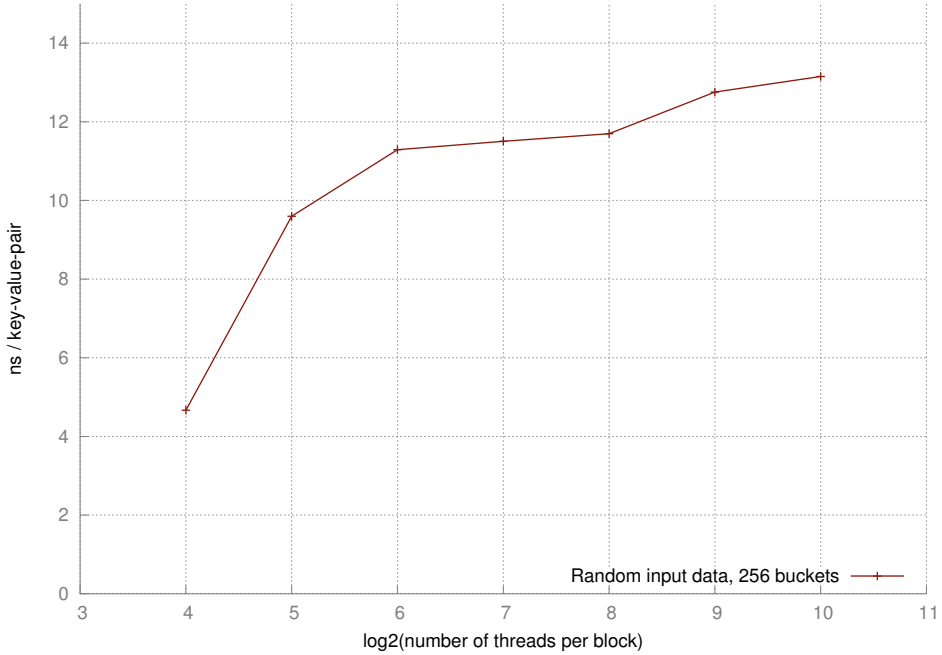


Figure 5.4: Performance of Sheffler's multiscan algorithm in global memory for different numbers of threads per block

5.4.2 Partitioning in global memory

Every instance of the algorithm, i.e. every block of threads if a partitioning scheme is used, needs to store its own vector of spinerecs. Since each spinerec used 12 bytes of memory (assuming 32-bit pointers), and each block processing $segLength$ input elements requires $n + nBuckets$ spinerecs, a large amount of memory may be needed for this data structure. This suggests that it may be best placed in global memory. It is not possible to store either only the spinerecs for the input elements or only the spinerecs for the buckets in global memory and the others in shared memory, since this would mean that the spine pointer of every node could in principle refer to either global or shared memory. This is impossible, since the compiler needs to be able to determine which level of memory is accessed by any instruction at compile time.
Memory accesses to each nodes's spine pointer are frequent, and the pointer may refer to any element in a higher row than the current one. Spine accesses are therefore unpredictable and cannot possibly be coalesced. This will necessarily affect performance negatively if nodes and buckets are stored in global memory. The point made before (that threads have very little to do between memory reads and writes) further amplifies this fact.
Apart from storing labels and values in shared memory, there are no further obvious improvements which could be made to the algorithm when implemented in global memory. Finding the best configuration for the algorithm, however, is not completely trivial. Each block can in principle work on any number of inputs, meaning that we can partition the input data into arbitrarily many segments, as long as the segment length is a square of a natural number. The length of each segment directly determines the number of threads per block (or the other way around): If a block works on $segLength$ inputs, then $nThreads = \sqrt{segLength}$, or equivalently $segLength = nThreads^2$. Figure 5.4 shows the algorithm's performance for different choices of $nThreads$ for $nBuckets = 256$.
We can see that performance is best for very low numbers of threads and therefore few elements per segment; the best configuration seems to be $nThreads = 16$ and therefore

$segLength = 256$. The algorithm did not run successfully for even shorter segments, since shorter segments mean more partial results, and the amount of global memory does not suffice to store all partial results for even smaller segment sizes.
The fact that the algorithm performs better with fewer threads per block can be explained from two perspectives. First of all, small segments mean few threads per block and low shared memory usage per block, which means that many blocks can run in parallel on each SM. This may be advantageous because, as mentioned before, the frequent synchronization prevents the warp scheduler from hiding latency efficiently. Synchronization, however, affects only the warps of one block, meaning that with more blocks per SM, the warp scheduler has more freedom.
A more likely explanation for the performance pattern is the following: At 256 inputs per segment with 256 buckets, each bucket will likely have one element in each segment. This means that most of the actual work is not performed by Sheffler's algorithm, but by the subsequent two phases of the partitioning algorithm; Sheffler's algorithm mostly serves to rearrange the inputs for each segment into a pattern which the subsequent two phases can work with (and does so in a very inefficient fashion). In effect, this means that this algorithm performs best when it is not really used and others do most of the work. This, and the fact that it needs more than 4 ns per input even with few buckets, should lead us to abandon this approach and consider alternatives.

5.4.3 Partitioning in shared memory

In order to reduce the waiting time between memory accesses, it would make sense to store *nodes* and *buckets* in shared memory. As mentioned before, a spinerec uses 12 bytes of memory, so the 48 kB of shared memory on each SM are enough to contain 4096 spinerecs. Since the length of *nodes* needs to be a square, and one would want to avoid to have warps consisting of much less than 32 threads (since that would automatically mean that a number of SPs will remain unused a lot of the time), one would prefer to either have $nThreads = 32$, and have three blocks per SM, or have $nThreads = 62$, which would leavejust enough space for 256 buckets, and have one block per SM. More than 62 threads per block would not leave enough empty space for realistic numbers of buckets.
Both cases not be optimal for different reasons. In the case of $nThreads = 62$, there are only 62 threads running on every SM. That is not nearly enough to fully exploit the SM's capabilities, which is built to host hundreds of threads at once. The fact that there is not much freedom in the warp scheduling choices is a theoretical problem here, since there can only be two total warps on each SM, so that there will not be any kind of latency hiding anyway.
In the case of $nThreads = 32$, there can be three warps per SM, which can be scheduled independently. This is obviously better than the other scenario, but only by a small margin. In addition the algorithm itself becomes suspect for small choices of $nThreads$. We have already seen that in this case, other phases to most of the work. But apart from that, there is another problem: Since the algorithm's depth is the square root of the length of the input, one could achieve a factor $nThreads$ speedup at most over the sequential algorithm for every single segment. For this potential speedup, the Sheffler algorithm performs a lot of additional work, with a lot of additional initialization and four phases, each of which requires at least as much work as one step of a simple sequential algorithm. This additional work is very likely to compensate for the theoretical speedup if $nThreads$ is small.
In addition to all this, the algorithm also has memory access problems. There does not seem to be a possibility for it to avoid a potentially large number of bank conflicts. Spines can point to any node above the current one, making it impossible to design a memory layout that definitely avoids bank conflicts in the accessed spine elements. Furthermore, the nodes are accessed both per-row and per-column, thus making any layout that puts different rows or

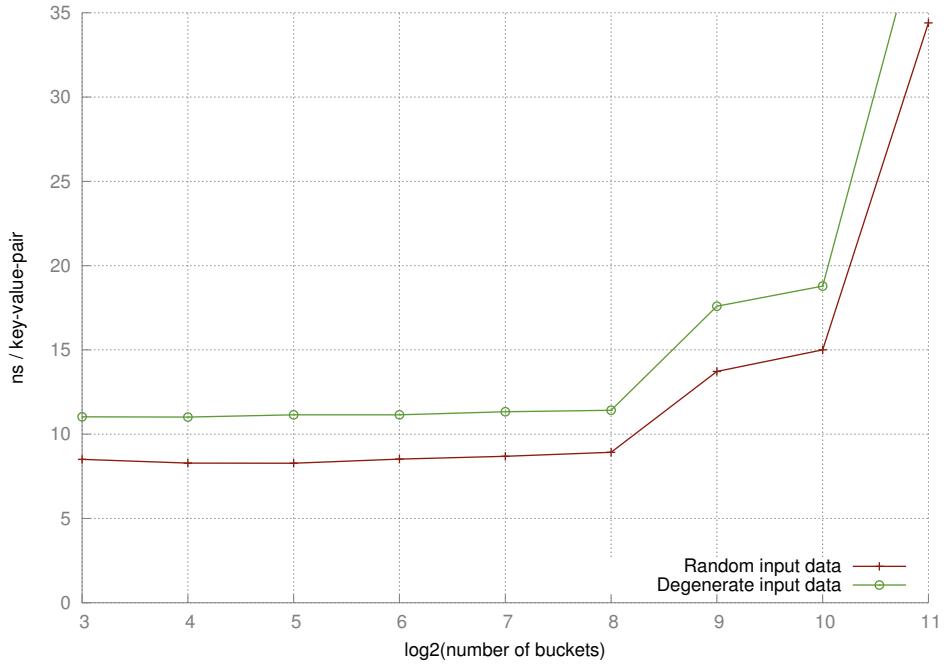


Figure 5.5: Performance of Sheffler's multiscan algorithm in shared memory

columns in different banks and thereby avoids bank conflicts for one access pattern a definite worst case for the other one.
In spite of these problems, we have implemented the algorithm for shared memory as well. The resulting performance is shown in Figure 5.5. The performance is even worse than the best case of the global memory implementation, most likely due to the severely limited parallelism. It also performs slightly worse than the sequential implementation in all cases and cannot be used at all for large numbers of buckets because of shared memory limitations. We will therefore not consider this approach any further.

5.5 Sort-based approach

Just like sorting can convert a multireduce problem to a segmented reduce problem, it can also reduce multiscan to segmented scan. The procedure for this is slightly more complex for the multiscan and described in Algorithm 5.3 and illustrated in Figure 5.6. First, a new vector of pairs is created, which stores the value and the index of each position. These pairs are then sorted by their labels. Without the index information in each pair, the original index of each value (which is needed in the last step) would be lost at this point. Subsequently, the labels are converted to segment start flags, and the values are extracted from the index-value pairs, so that a segmented scan can be performed on the values using the flags. The scanned values are then scattered back to their original locations as specified by the index field of the index-value-pairs.

```
struct IndexValue {
  int index;
  M value;
};
```

Using index-value pairs is a general way to solve the problem of remembering original indices; however, there are others. If both the values and the indices are known to only use a certain number of bits, then the two can be packed into one integer using some bit level encoding.

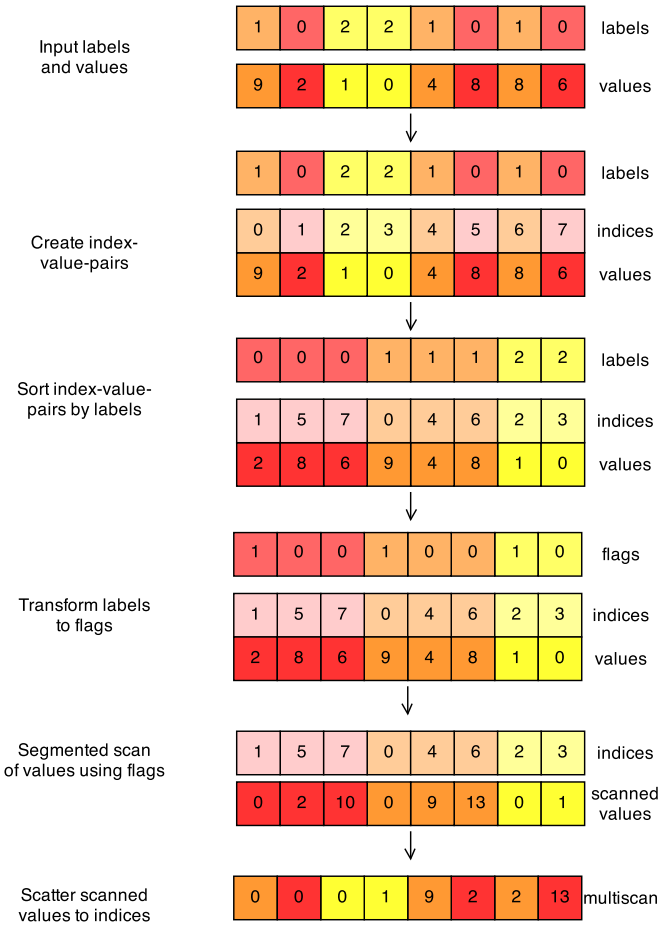


Figure 5.6: Illustration of sort-based multiscan

function *multiScan(int labels[n], M values[n], M results[n], int n)* **:**
    IndexValue $iValues[n]$;
    parfor $i = 0$ ***to*** $n - 1$ **do**
        $iValues[i].index = i$;
        $iValues[i].value = values[i]$;
    end
    stableSort($labels$, $iValues$);
    bool $flags[n]$;
    parfor $i = 0$ ***to*** $n - 1$ **do**
        $values[i] = iValues[i].value$;
    end
    parfor $i = 0$ ***to*** $n - 2$ **do**
        $flags[i] = labels[i]! = labels[i + 1]$;
    end
    $flags[n - 1] = 0$;
    $M\ scannedValues[n]$;
    segmentedScan($values$, $flags$, $scannedValues$, n, $\odot$);
    parfor $i = 0$ ***to*** $n - 1$ **do**
        $result[iValues[i].index] = scannedValues[i]$
    end
end

Algorithm 5.3: Sort-based multiscan algorithm

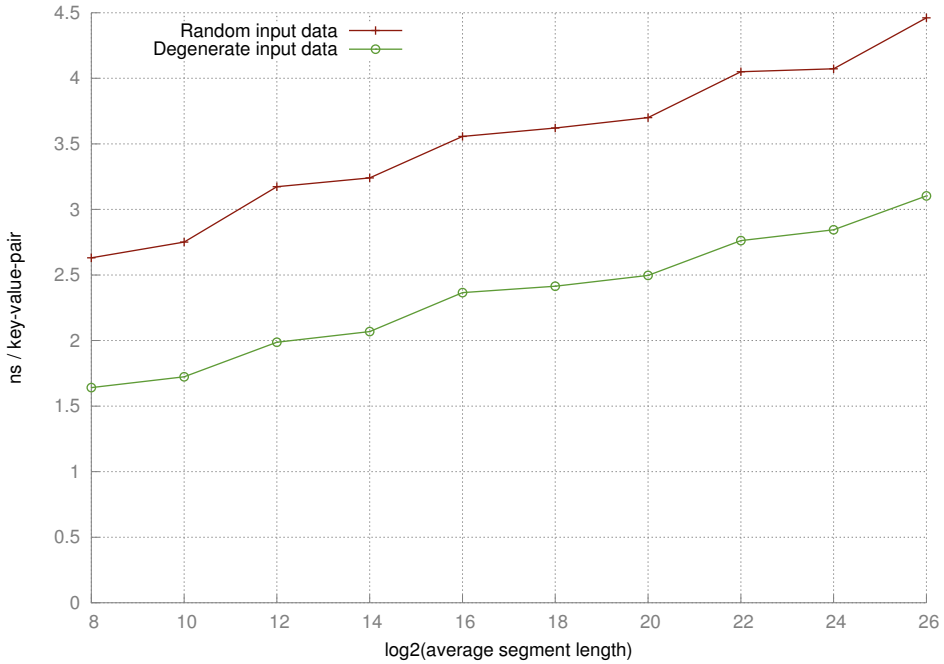


Figure 5.7: Performance of global sorting multiscan algorithm

Before or during the segmented scan, however, the original values would need to be extracted again.
It is also possible to perform the segmented scan directly on the index-value-pairs using a suitable operator, but only if the multiscan to be performed (and therefore the segmented scan) is inclusive, since otherwise the first and last indices in each segment are lost in the result.

5.5.1 No partitioning

For the version without partitioning, we have implemented the sort-based approach using SRTS radix sort, sorting only by as many bits as necessary, and Thrust's scan-by-key-function, which converts the labels to flags and performs the segmented scan in a single step. The resulting performance is shown in Figure 5.7.
Compared to the same approach for the multireduce, this multiscan algorithm performs comparatively bad for very low numbers of buckets, needing more then 2.5 ns per input for 256 buckets. This is likely due to the large amount of work necessary for converting values to index-value-pairs and back, as well as the overhead for scattering results in the last step. However, similarly to the multireduce case, the algorithm only gets gradually slower for large numbers of buckets. Since many other algorithms do not perform well, or work at all, for very many buckets, this algorithm may therefore well be the first choice for many buckets.
It is interesting that the algorithm needs about 1 ns more per input if the input is random than if it is degenerate. For the multireduce, a similar difference existed, but it was not nearly as large. The most likely reason for this is that both the sort step and the scatter step make much better use of caches if all labels are identical, since both essentially just copy large, contiguous blocks of data from one location to another in this case.
We did not try using bit-level encoding instead of explicit index-value-pairs, since we assume that the algorithm will generally be used for large input vectors and using full 32-bit integers. We did, in fact, implement a version of this which performs the scan directly on the index-value-pairs, thus saving one conversion at the expense of additional calculations during the

scan, but this resulted in worse performance than the original approach.
One possible improvement which we did not implement is converting the values to index-value-pairs during the first run of the radix sort. This would save one run over all input values, and it would mean that the radix sort algorithm only has to fetch values instead of index-value-pairs for the first run. This would, however, require a significant alteration of the existing code and would likely only result in a minor speedup, which is why we did not attempt to implement this modification.

5.5.2 Partitioning in global memory

As was the case for the multireduce, we reasoned that any partitioning strategy using a sort-based algorithm would necessarily be faster in shared memory, and that using global memory offers no significant advantages over either the partitioning version in shared memory or the global implementation.

5.5.3 Partitioning in shared memory

As for the multireduce, we will again implement our algorithm using the sort and segmented scan functions of the CUB and Modern GPU libraries, respectively.
For the reasons discussed before, we will transpose input data before using it, just like we do for the multireduce for non-commutative operators.
In this case, however, we need to perform a segmented scan instead of a segmented reduce in the second step of the algorithm. This can be done with only a few adaptations to the multireduce algorithm, which uses Modern GPU's segmented scan function to compute a segmented reduction. The problem with the segmented scan function is, however, that it computes the segmented scan for $nThreads$ inputs per block, but each block actually has $nThreads \times nItems$ input elements and needs a segmented scan of all of them. We will therefore use a similar method as in the multireduce algorithm to use a segmented scan function for $nThreads$ elements to compute a segmented scan of $nThreads \times nItems$ elements.
The algorithm to accomplish this is shown in Algorithm 5.4, where the sort and segmented scan functions are still defined like in Section 4.4.3.
We first fetch and transpose the input data for the current block using Baxter's algorithm. Instead of having a local array of values, we need an array of index-value-pairs for the multiscan. We then sort the index-value-pairs by their labels.
Now the segmented scan algorithm begins, as before by first letting each thread fetch the first label of the next thread through shared memory. We then perform a local exclusive scan of the thread's values. The input to the second phase, Modern GPU's segmented scan function, is calculated exactly as it is for the multireduce.
In the last phase, we have to combine the results of the local exclusive segmented scan with the result of the second phase. Single results are obtained by adding the results of the local scan to the segmented scan result. In the multireduce algorithm, we obtain the values of buckets by adding the result of the segmented scan to the result of the inclusive local scan at the end of each segment. Here, the local scan is exclusive, so we also have to add the last value of the segment in order to get the same result.
This means that we still need the intitial values in the last phase of the computation, which in turn means that we cannot use the local value array to store flags inbetween, and flags therefore need to be computed twice.
Compared to the non-commutative multireduce algorithm, this one has to spend additional work writing results for every element to global memory, and a small amount of extra work for recomputing labels. We therefore expect it to show performance slightly worse than, but comparable to that of said algorithm. Figure 5.8 shows this algorithm's performance.

```
kernel sortMultiScan(int inLabels[n], M inValues[n], M results[n], M
buckets[nBuckets × nThreads], int n) :
    int blockLabels[nThreads × nItems];
    M blockValues[nThreads × nItems];

    // transpose input data shared int transposeSpace[nItems × nThreads];
    local int indices[nItems];
    local IndexValue values[nItems];

    for i = 0 to nItems − 1] do
        int index = nThreads × i + threadId;
        transposeSpace[index] = blockValues[index];
    end
    for i ∈ [0...nItems − 1] do
        int index = nItems × threadId + i;
        values[i].value = transposeSpace[index];
        values[i].index = index;
    end

    Fetch labels in the same fashion

    cub::sort(indices, values);

    // exchange first labels
    shared int nextValues[nThreads];
    nextValues[threadId] = threadId == 0 ? nBuckets : labels[0];
    int nextLabel = nextValues[(threadId + 1)  mod nThreads];

    // first phase of segmented scan int x;
    bool f = false;
    int localScan[nItems];
    int next = indices[0];
    for i ∈ [0...nItems − 1] do
        localScan[i] = x;
        x = i ? x ⊙ values[i].value : values[i].value;
        int current = next;
        next = i == nItems − 1 ? nextLabel : indices[i + 1];
        bool flag = current! = next;
        if flag then
            x = identity;
        end
        f = f || flag;
    end

    // second phase
    int y = mgpu::segmentedScan(x, f);

    // third phase
    next = labels[0];
    for i = 0 to nItems − 1 do
        int first = next;
        next = i == nItems − 1 ? nextLabel : indices[i + 1];
        M y2 = y ⊙ localScan[i];
        result[values[i].index] = y2;
        if first! = next then
            buckets[indices[i] × nBlocks + blockId] = y2 ⊙ values[i].value;
            y = 0;
        end
    end
end
```

Algorithm 5.4: Complete sort-based multiscan kernel working in shared memory

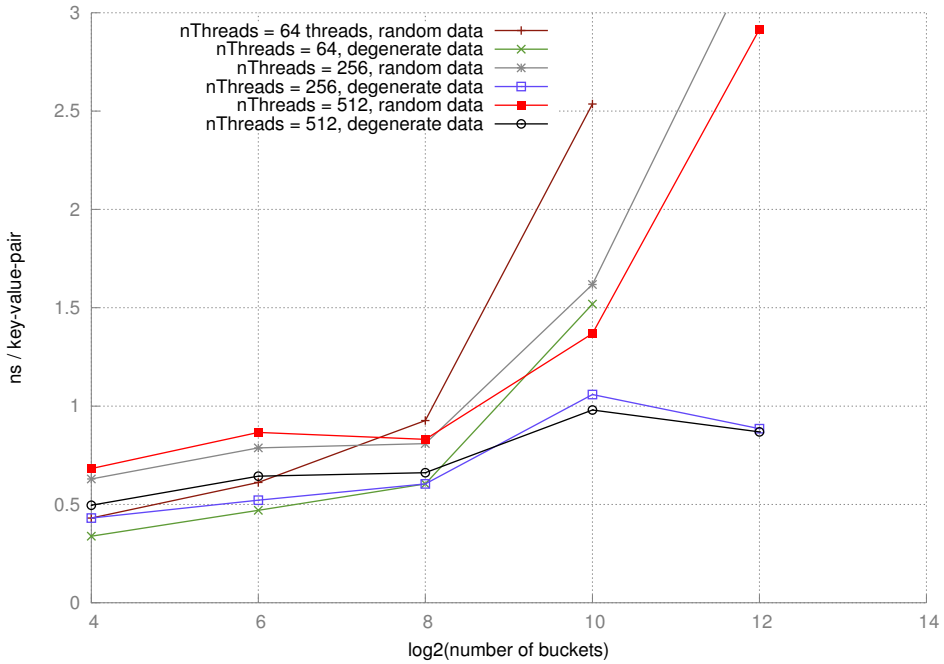


Figure 5.8: Performance of sorting multiscan algorithm in shared memory

For $nBuckets \leq 2^8$, the algorithm needs less than 0.75 ns per input and is therefore on par with Thrust's scan-by-key function and the fastest of our multiscan algorithms. While performance for degenerate input data remains good even for more buckets, it deteriorates quickly for random input data. Compared to the corresponding multireduce algorithm, this one has slightly lower performance throughout, which is expected because of the extra work necessary for performing a scan. In addition, the multiscan implementation used a lower $nItems$, which may explain part of the generally lower performance, and definitely explains why the performance starts to drop faster as $nBuckets$ grows than it does for the multireduce.

5.6 Conclusion

Figure 5.9 shows the the worst case performance of all multiscan algorithms we implemented in this chapter.
Compared to the very good results we achieved for the multireduce, the performance of our multiscan algorithms is disappointing. Sheffler's PRAM algorithm is obviously not a good fit for GPUs. Among many other problems, it suffers from a lack of parallelism, a high memory demand, frequent synchronizations and frequent, unpredictable memory access, all of which are unproblematic on a PRAM, but seriously hinder performance on a GPU. As a result, it is clearly outperformed even by the sequential CPU implementation.
While the sorting approaches outperform all other algorithms by some margin, they, too, perform much worse than they do for the multireduce. The reason for this is that the sorting algorithm itself is much more complicated for the multiscan than it is for the multireduce. The version without partitioning requires a number of additional steps, each of which reads and writes large amounts of data. When implemented using the partitioning scheme in shared memory, the performance it somewhat similar to the multireduce, but higher demand for registers limits the amount of parallelism. Nevertheless, the sorting algorithm in shared memory delivers performance on par with Thrust's scan-by-key function for low numbers of buckets, and the approach without partitioning clearly outperforms the CPU algorithm as

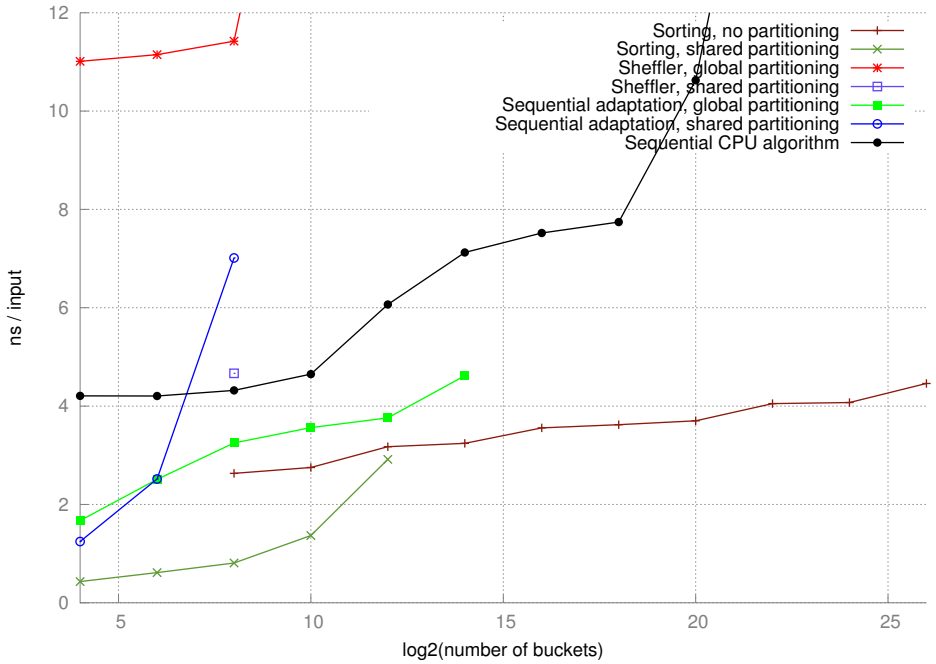


Figure 5.9: Performance comparison of different multiscan algorithms

soon as the latter starts to suffer from cache misses.
We did not discuss the adaptations of the sequential algorithm in detail, since they are virtually identical to the non-commutative case of the multireduce algorithms, which performs bad compared to other multireduce algorithms. Compared to other multiscan algorithms, however, this approach looks relatively competitive.
All in all, we have to admit that none of our multiscan algorithms outperforms the CPU implementation by a large enough factor to warrant the effort for moving the computation to the GPU. We also have to conclude that, with these algorithmic options, the multireduce operation is too slow to be used as a general building block for other algorithms.

Chapter 6

Conclusions and future work

6.1 Summary and conclusion

In this thesis, we have discussed algorithmic approaches to implementing the multireduce and multiscan operations, both of which have a number of widespread applications that could make them useful building blocks for more complicated algorithms. We have seen that an entire family of closely related problems exists around these two general operations, and that all of them can be solved by similar algorithmic approaches, which have similar performance bottlenecks.
Our examination of the sequential algorithms revealed that the performance of all of these operations suffers from cache and TLB misses for large numbers of buckets. As a result, the simple algorithms which are optimal according to the RAM model can actually be improved on for some hardware configurations. We showed that the use of huge pages can alleviate the problem of TLB misses and, more importantly, that partial radix sorting of the input data can avoid cache misses on all levels and therefore improve overall performance.
However, we also had to conclude that the usefulness of sorting depends entirely on the used hardware, and that especially high-end systems with many and large caches are unlikely to profit from this approach. This makes the ability to predict the benefit of the sorting approach for any given system especially valuable.
We therefore provide a cache simulator which, given the access latencies for the existing layers of memory, calculates the expected runtimes of both approaches. In its current state, this simulator does not take into account memory pipelining and therefore does not give perfect results for RAM access times, but it can still be useful for cases where the expected speedup of the sorting approach is much higher, e.g. gathering and scattering data to or from disk.
Subsequently, we attempted to find efficient algorithms for the aforementioned operations for modern GPUs in order to capitalize on their superior computing power. We first discussed the structure of GPUs and the requirements which algorithms must fulfill in order to fully exploit the GPU's capabilities, pointing out the differences between GPUs and the PRAM model in particular.
While there is a way to use the existing (very efficient) parallel implementations of reduce and scan to compute a multireduce and multiscan, we argued that that approach is only sensible for very low numbers of buckets and will result in bad performance otherwise, meaning that different algorithmic approaches are needed.
For the multireduce, we evaluated adaptations of the sequential algorithm as well as a sort-based conversion of the multireduce problem to a segmented reduce problem. We pointed out the possibility of implementing either of these approaches on several levels in the CUDA memory hierarchy, and implemented and compared all sensible combinations for both approaches. We surveyed the literature on related algorithms (most prominently histogramming) and transferred successful concepts to our implementations wherever possible.

While we achieved significant speedups over the CPU implementation for all configurations, a central insight we gained concerning the multireduce is that commutative operators allow for a much wider range of optimizations than non-commutative ones. As a result, we have created a separate range of algorithms which specifically exploit commutativity, and thereby reach an order of magnitude speedup over the sequential implementation on average, whereas the average speedup for non-commutative operators is a factor of six.
The most convincing multireduce algorithm we found is an adaptation of the sequential algorithm which exploits commutativity. Based on the general idea of an existing histogram algorithm, we used a superior distribution of shared memory between threads which vastly increases the amount of parallelism. As a result, our multireduce algorithm performs as well as, or slightly better than, the best existing histogramming algorithm, even though the latter uses a number of histogram-specific optimizations which cannot be applied to the general multireduce problem. It works around 18× faster than the sequential implementation. We showed that this algorithm can be extended to work on any number of buckets with acceptable, but significantly diminished performance.
A number of other approaches show good results for some input configurations; in particular, an algorithm which works with both commutative and non-commutative operators outperforms existing library functions for related but simpler problems for up to 1024 buckets. Generally however, performance gradually decreases for all approaches as the number of buckets grows.
A significant insight gained while evaluating different multireduce algorithms is that partial radix sorting with the aim of raising the cache hit rate can lead to performance improvements on GPUs as well as on CPUs. This result would have been even more significant if it could have been extended to the scatter operation, where large numbers of buckets are very frequent, but it did not result in a speedup in that case.
Histogramming is another related problem which can benefot from our work on the multireduce. While our general multireduce is already slightly faster than the fastest existing histogram algorithm, we also created a version of the multireduce algorithm optimized for histogramming. The result is consistently around 40% faster than the currently fastest published histogram algorithm.
The algorithms we found for the multiscan have been altogether less convincing than those for the multireduce. The main problem here is that the order in which elements have to be reduced is fixed for the multiscan, which is the same problem that also prevents further speedup of the multireduce for non-associative operators. This problem remains in spite of a sophisticated fetching mechanism, which we generalized from an existing proposal.
The only existing work-efficient PRAM algorithm for the multiscan, while adaptable to GPU hardware in principle, shows bad performance compared to all other approaches (consistently worse than the CPU). The main problem is that two main assumptions of the PRAM model, random memory access at constant, low cost and general SIMD execution, do not generally hold for GPUs, resulting in a lot of idle time for each thread and a need for frequent, costly synchronizations.
We therefore conclude that the multireduce can be recommended for use as a primitive operations and as a building block for other algorithms, especially if the number of buckets is relatively low and if the used operator is commutative. The latter is the case for all of the most typical operators used in scans and reductions. The multiscan, however, does not have a sufficiently fast implementation to be recommended for general use.

6.2 Future work

We suggest the following areas for possible further research:

- The main objective of this thesis was to evaluate different algorithmic approaches, not

to optimize a specific one. As a result, most algorithms presented here can almost certainly be sped up further through low-level improvements or improved work distribution between blocks and threads. This is especially true for the fast histogram algorithm, since histogramming is not the main topic of this thesis and less effort has therefore been spent on optimizing it. Nevertheless, histogram calculation is an important topic in GPU computing, as the large number of publications on this topic shows, which is why it seems worthwhile to further optimize this algorithm in particular.

- Since it is now certain that the multireduce can be calculated very efficiently on modern hardware, it seems sensible to try to apply it to different problems than the ones mentioned in this thesis. The field of (sparse) matrix and vector operations seems like an especially promising field in this regard.

- Gathering and scattering data from non-sequential locations on disk or similarly slow media is common in many applications, yet there seems to be little to no literature on accelerating such accesses on a software level. We have shown that even a simple approach like partial sorting can lead to speedups. Considering the ubiquity of this use case, further research should try to find more sophisticated ways to exploit caching in these scenarios.

- As mentioned before, the cache simulator used to predict the relative performance of traditional and sort-based scattering and gathering does not yet simulate modern memory with complete accuracy. The simulator could be extended to properly simulate memory pipelining and virtual address translation.

- While both the multireduce and the multiscan are probably best distributed as a library function in a CUDA context, other programming languages like NESL would demand for a different integration into the language. Since we have shown that a multireduce can be implemented efficiently on GPUs, different options for making the operations available in different language contexts should be developed.

Bibliography

[1] Cuda data parallel primitives library. http://cudpp.github.io/.

[2] Shibdas Bandyopadhyay and Sartaj Sahni. GRS - GPU radix sort for multifield records. In *High Performance Computing (HiPC), 2010 International Conference on*, pages 1–10. IEEE, 2010.

[3] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip, don't walk (the page table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 48–59, New York, NY, USA, 2010. ACM.

[4] Sean Baxter. Turning the crank on streaming algorithms. Presentation slides, November 2013. http://nvlabs.github.io/moderngpu/cascon.pptx.

[5] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990. http://repository.cmu.edu/cgi/viewcontent.cgi?article=3017&context=compsci.

[6] Guy E. Blelloch. *Vector Models for Data-parallel Computing*. MIT Press, Cambridge, MA, USA, 1990.

[7] Guy E. Blelloch and Bruce M. Maggs. Algorithms and theory of computation handbook. chapter Parallel Algorithms. Chapman & Hall/CRC, 2010.

[8] Shawn Brown and Jack Snoeyink. Modestly faster histogram computations on GPUs. In *Innovative Parallel Computing (InPar), 2012*, pages 1–7. IEEE, 2012.

[9] John R. Celis, Dennis Gonzales, Erwinaldgeriko Lagda, and Larry Rutaquio Jr. A comprehensive review for disk scheduling algorithms. *International Journal of Computer Science Issues (IJCSI)*, 11(1), 2014.

[10] Evan R. Cohn. The beta operation: A parallel primitive. Technical report, DTIC Document, 1988. http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA204463.

[11] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973.

[12] NVIDIA Corp. Modern GPU. http://nvlabs.github.io/moderngpu/.

[13] NVIDIA Corp. CUDA C programming guide v5.5, July 2013. http://nvlabs.github.io/moderngpu/.

[14] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[15] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. Fast scan algorithms on graphics processors. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 205–213, New York, NY, USA, 2008. ACM.

[16] Faith E. Fich. The complexity of computation on the parallel random access machine. In John H. Reif, editor, *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[17] Mark Harris. Optimizing parallel reduction in CUDA, 2007. `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf`.

[18] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with cuda. In Hubert Nguyen, editor, *GPU Gems 3*. Addison Wesley, August 2007.

[19] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient gather and scatter operations on graphics processors. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 46:1–46:12, New York, NY, USA, 2007. ACM.

[20] Jared Hoberock and Nathan Bell. Thrust library. `http://thrust.github.io/`.

[21] Daniel Horn. Stream reduction operations for GPGPU applications. *Gpu gems*, 2:573–589, 2005.

[22] David Kanter. Intel's merom unveiled. `http://www.realworldtech.com/merom/`.

[23] David B. Kirk and W. Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[24] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '97, pages 370–379, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[25] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[26] Pedro J. Martín, Luis F. Ayuso, Roberto Torres, and Antonio Gavilanes. Algorithmic strategies for optimizing the parallel reduction primitive in CUDA. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 511–519. IEEE, 2012.

[27] Yossi Matias and Uzi Vishkin. On parallel hashing and integer sorting. *Journal of Algorithms*, 12(4):573–606, 1991.

[28] Duane Merrill. Cub library. `http://nvlabs.github.io/cub/`.

[29] Duane Merrill and Andrew Grimshaw. Parallel scan for stream architectures. *University of Virginia, Department of Computer Science, Charlottesville, VA, USA, Technical Report CS2009-14*, 2009. `http://www.cs.virginia.edu/~dgm4d/papers/ParallelScanForStreamArchitecturesTR.pdf`.

[30] Duane G. Merrill and Andrew S. Grimshaw. Revisiting sorting for GPGPU stream architectures. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 545–546, New York, NY, USA, 2010. ACM.

[31] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[32] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Bart Mesman. High performance predictable histogramming on GPUs: exploring and evaluating algorithm trade-offs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2011.

[33] NVIDIA. Kepler GK110 whitepaper, 2012.

[34] NVIDIA Corporation. *CUDA C Best Practices Guide v5.5*, July 2013. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf`.

[35] Jeff Parkhurst, John Darringer, and Bill Grundmann. From single core to multi-core: preparing for a new exponential. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 67–72. ACM, 2006.

[36] David A. Patterson and John L. Hennessy. *Computer Organization and Design : The Hardware/Software Interface (3rd Edition)*. Morgan Kaufmann, 2007.

[37] Victor Podlozhnyuk. Histogram calculation in CUDA. *NVIDIA Corporation, White Paper*, 2007. `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/histogram64/doc/histogram.pdf`.

[38] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[39] Abhiram G. Ranade, Sandeep N. Bhatt, and S. Lennart Johnsson. The fluent abstract machine. Technical report, DTIC Document, 1987. `http://www.dtic.mil/dtic/tr/fulltext/u2/a327476.pdf`.

[40] Karl Rupp. CPU, GPU and MIC hardware characteristics over time, June 2013. `http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/`.

[41] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.

[42] Thorsten Scheuermann and Justin Hensley. Efficient histogram generation using scattering on GPUs. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 33–37, New York, NY, USA, 2007. ACM.

[43] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[44] Ramtin Shams and R. A. Kennedy. Efficient histogram algorithms for NVIDIA CUDA compatible devices. In *Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS)*, pages 418–422, 2007.

[45] Thomas J. Sheffler. Implementing the multiprefix operation on parallel and vector computers. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, SPAA '93, pages 377–386, New York, NY, USA, 1993. ACM.

[46] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32(2):174–199, June 2000.

[47] Andrey Vladimirov. Arithmetics on intel's sandy bridge and westmere CPUs: not all FLOPS are created equal. *Colfax International*, 2012. `http://research.colfaxinternational.com/file.axd?file=2012/4/Colfax_FLOPS.pdf`.

[48] Nicholas Wilt. *The CUDA handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.

Appendices

Appendix A

Comparison of histogram algorithms

Figures A.1 and A.2 show the performance of our histogram algorithm (called "BankTRISH") compared to that of other existing histogram algorithms for 256 bins.

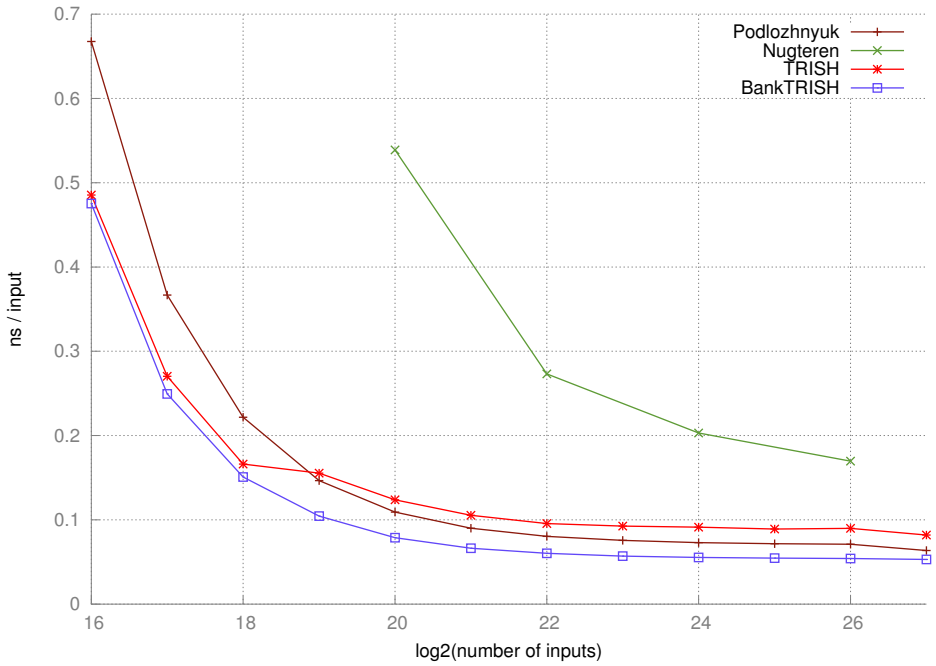


Figure A.1: Comparison of histogram algorithms for 256 bins, random inputs

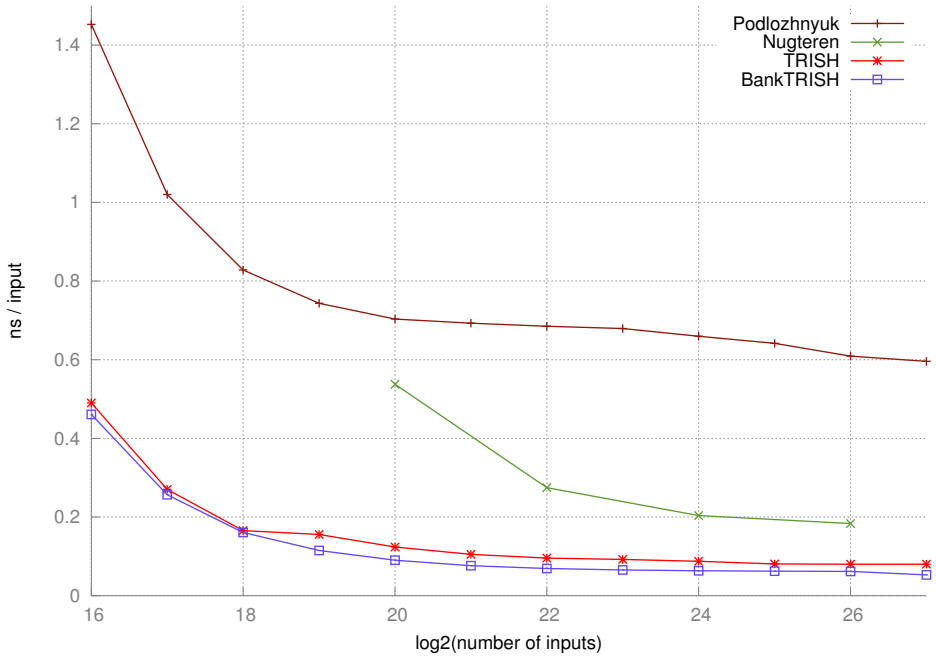


Figure A.2: Comparison of histogram algorithms for 256 bins, degenerate inputs

Appendix B

Adaptation of the sequential algorithm for multireduce

Since the adaptation of the sequential algorithm for the multireduce does not differ in any fundamental way from the equivalent multireduce implementation and promised poor performance, we did not discuss it in detail in the main part. Nevertheless, we still implemented and benchmarked it both for global and for shared memory. The results are shown in Figures B.1 and B.2, respectively.

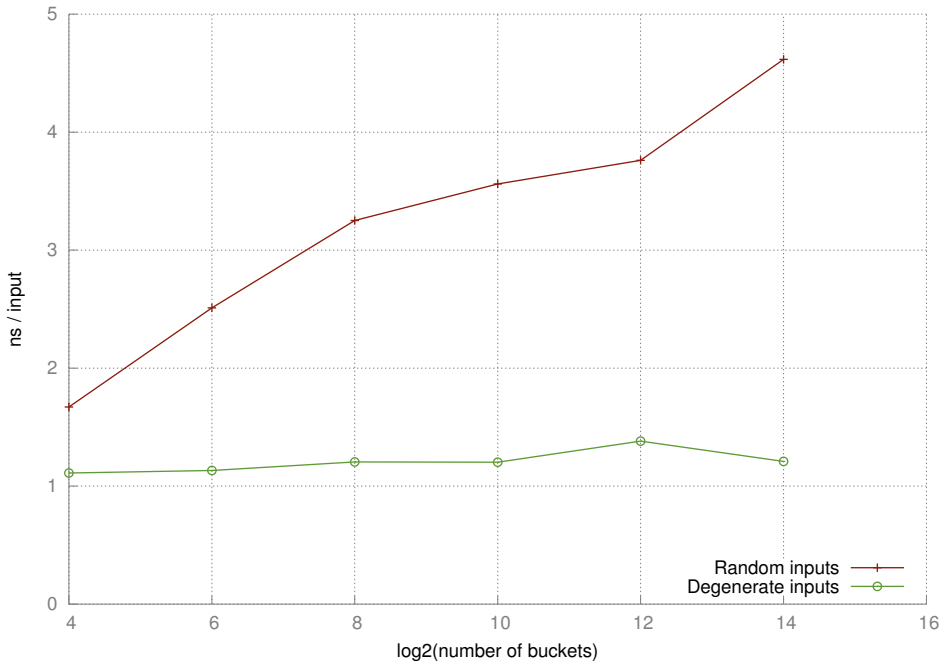


Figure B.1: Runtime of multiscan in global memory

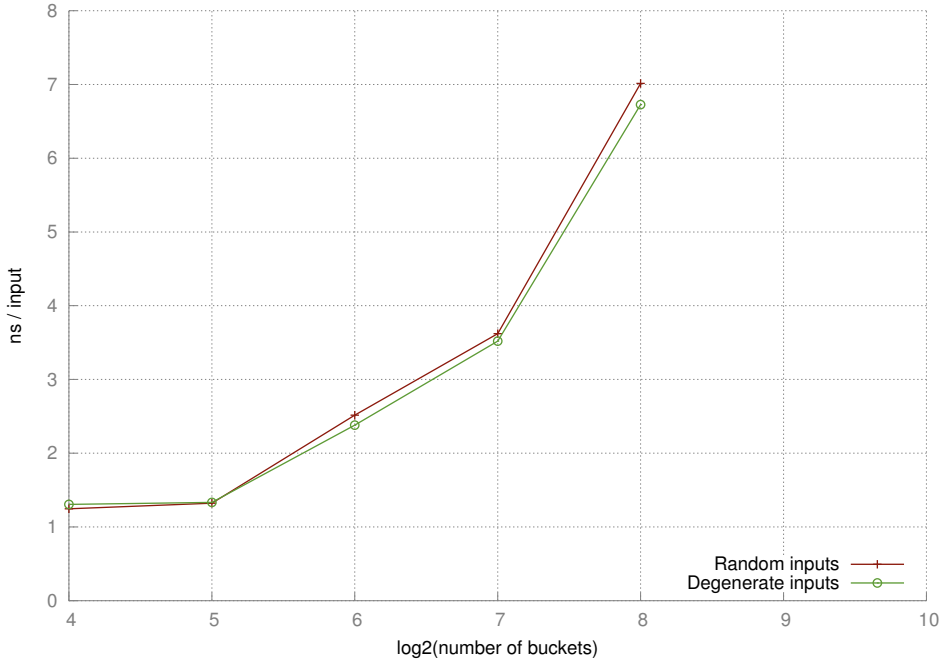


Figure B.2: Runtime of multiscan in shared memory