Efficient GPU Implementation of Stencils on Unstructured Grids

Bachelor Thesis

André Albert Rösti
May 14, 2020

Advisors: Dr. T. Gysi, Dr. T. Grosser, Prof. T. Hoefler
Department of Computer Science, ETH Zürich

Contents

Abstract

Stencil computations on unstructured grids with a regular Z-dimension are frequently performed in meteorology for atmospheric modeling. While using a GPU to parallelize execution of such stencils promises fast runtimes, supporting the irregular structure in two dimensions complicates their implementation, as neighborship access necessitates indirect memory lookups.

In this thesis, we explore various approaches of storing unstructured grids with one regular dimension and accessing them from the GPU in a stencil computation. We benchmark the performance of three real-world stencils of varying complexity on various unstructured grid implementations, some of which make explicit use of the Z-regularity. In our initial naive approaches, we observe slowdowns of 122% (simple stencil), 52% (medium-complexity stencil), and 5% (complex stencil) compared to executing the same stencils on entirely regular grids. Notably, through a neighborship table compression scheme, we are able to improve those values to 49%, 30% and 5% (no improvement) respectively. Other optimizations are explored. We also detail the foundations of the Nvidia CUDA GPU architecture.

Chapter 1

Introduction

Weather prediction is a challenging task for computers. A large number of factors, drawn from large data sets, interact in ways governed by complex physical equations. In order to solve the governing equations of the atmosphere, the computations are discretized on grids spanning the globe (or some local segment of it). Methods such as the *finite differences* and *finite volume* method solve these equations by performing (simple) calculations on each cell that depend only on a limited neighborhood of that cell. This type of computation, called a *stencil*, can be implemented efficiently on graphics processing units, which provide a massively parallel architecture.

Needless to say, a weather prediction computation for some instant in the future is only useful if the computation terminates before that point in time. Therefore, performance is a critical aspect of any application in weather prediction. The desire for more accurate results, on the other hand, opposes fast runtimes.

More fine-grained grids provide more accurate results but naturally lead to higher data traffic and slower runtimes. So-called *unstructured grids* provide a compromise between globally coarse and globally fine-grained regular grids. Unstructured grids enable higher resolutions in areas of interest while other areas that require less detail can be covered by larger cells. Because of their inherent irregularities, however, unstructured grids add additional overhead. Finding the location of neighboring cells' values especially becomes more involved, requiring additional memory lookups.

The main aim of this thesis is to contrast stencil performance in regular and unstructured grids, as well as exploring some means of optimization. The discrete approximations of the atmosphere's equations are of low arithmetic intensity – the computations performed on each cell are simple. However, the operations performed are very memory-bandwidth hungry, as each cell requires a lot of input data (several neighboring cells). Thus data locality greatly aids performance. The major focus of this report thus lies in evaluating different memory access strategies and schemes for optimizing the neighborship lookups in unstructured grids for three selected stencils.

In the following sections, we first explain the characteristics of grids and stencils in more detail and elaborate on the architecture of GPUs, on which these stencil calculations are performed. We continue by showing our pursued methods of implementing

regular and unstructured grid memory layouts and show different unoptimized and optimized means of accessing grid elements in stencil computations from the GPU. In section 6, we finally present the observed overhead that the use of unstructured grids imposed when calculating identical stencils on identical data.

Chapter 2

Related Work

Solano-Quinde et al. [10] present an algorithm for implementing scientific analyses in unstructured grids on GPUs for more general problems. They identify occupancy and memory access as the main limiting factors of performance. Their paper also explores the implications of using different memory layouts for the unstructured grid representation, arriving at the conclusion that struct-of-array-type layouts (see also section 4.3 in this report) are better suited for GPUs because of coalescing concerns.

In this report, we assume the unstructured grid on top of which to apply a stencil is already given. An overview of the various methods of how such grids can be generated is given in [8]. Section 3.4 of the publication also briefly hints at how an unstructured grid may be stored. This forms the basis of our initial naive storage approach.

A lot of research has been carried out concerning the compressed storage of meshes [9][7]. However, most of those approaches are not applicable to our problem, where decompression has to be virtually free and reordering of values is not permittable. The approach presented for compressing social network graphs in [2] is similar to our compression approach.

Chapter 3

Background

3.1 Grids

A grid partitions (tessellates) some space into a discrete number of cells. In weather/climate modeling, the three-dimensional space of the atmosphere is subdivided by a grid to facilitate finding numerical solutions to equations governing the weather. Each cell may contain one or multiple values (*fields*) such as the temperature or humidity at a location.

One kind of grid is the *regular grid*. In this type of *structured grid*, each cell is of uniform size and has a fixed amount of six direct neighbors (top, bottom, left, right, front, and back). Real-life objects that have the structure of a regular grid are checkerboards (two dimensions) or a Rubik's cube (three dimensions). Because of their regularity, storing such grids in memory is straightforward.

In certain use cases, the use of *unstructured grids* is beneficial to the requirements of the application at hand. In contrast to regular grids, cells need not be of equal size in unstructured grids and may have a varying number of neighbors. This means that the location of a cell's neighbor in memory no longer follows a regular structure. Because neighbor's locations are no longer inherently clear, an unstructured grid requires a neighborship table to describe its structure. Accessing a neighbor requires an indirect memory lookup in order to determine its location in memory.

This thesis is concerned with the performance implications that porting a stencil computation from a regular to an unstructured grid entails. It compares the cost indirect addressing imposes upon several different widely-used stencils.

While an unstructured grid theoretically supports arbitrary neighborship relations, for this thesis, we restrict irregularities to the X-Y-plane and assume even unstructured grids are regular (i.e. always have at most two neighbors) in the Z-dimension. This is a common use case in atmospheric modeling applications. In practice, such as in so-called *icosahedral grids*, most of the neighborships in the X-Y-plane will also most often have some structure to it. We make use of these regularities in the optimizations described in section 4 and 5.

3.1.1 Coordinates and Indices

In our implementations, cells in a grid are identified by *coordinates*, which relate a cell to its real-world position and *indices* which give the storage location of a cell in memory.

Throughout the rest of this report we refer to the size of a grid as the *maximum number of elements* in each dimension, and denote it by the vector

$$d = \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix}.$$

A unique identifier for each cell on the grid is given by its coordinates, denoted in similar fashion by a vector

$$p = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}.$$

The coordinates correspond to a position in Euclidian space. Coordinates are chosen such that each integer coordinate maps *uniquely* to at most one cell. The opposite (i.e. each cell having only one coordinate) is only true in regular grids. In the regular grid, direct neighbors (adjacent/face-touching cells) differ in only one coordinate by an amount of one. In unstructured grids, not every coordinate is necessarily assigned to any cell, and one cell may span multiple coordinates. Therefore, an unstructured cell's neighbor might have coordinates that differ by more than one.

As memory is one-dimensional, a mapping from the three-dimensional coordinates of a cell to its location in memory is required of a grid implementation. We call this location in memory the *index* and denote it with the letter i. The mapping from three-dimensional coordinate space to the one-dimensional memory index defines the memory layout of the grid. In regular grids, this mapping is straightforward, while in unstructured grids it may be arbitrary. How regular and unstructured grids are laid out in memory is detailed in section 4.

3.2 Stencils

Finding approximate numerical solutions to the governing equations of physical processes, as in meteorology, often entails performing certain unchanging calculations on every cell of a grid. The result of this process is again a grid of similar size. For each cell, the calculated output value is dependent only on a bounded small number of neighboring cells' values (*neighborhood*). Such a computation is called *stencil*. Simple stencils may require only the values of the current cell as well as directly adjacent (face-touching) neighbors in order to calculate the output value, while more complex ones could also depend on diagonal neighbors, neighbors-of-neighbors, etc. Yet, some spatial locality is guaranteed.

Halo In stencil computations, special consideration needs to be given to the boundary cells of the grid. Cells at the boundary of a grid have fewer neighbors than inner cells. As such, the output value for a stencil depending on neighbors is unclear for cells who do not have those neighbors. One way to address this issue is to execute the stencil only on the safe inner values of a grid, separated from the boundary by a certain amount of padding. The amount of padding used depends on the size of the neighborhood which a stencil requires for its computation. We call the set of cells residing in the padding around the boundaries of the grid *halo*. Stencil implementations may include a branch instruction that prevents any computations if a cell lies in the halo. Alternatively, for ease of implementation and performance, it can be beneficial to store the halo separately in memory.

3.3 GPU Programming on the Nvidia CUDA Platform

Graphics processing units (GPUs) particularly lend themselves to stencil computations. In this section, we contrast the execution model of classical *central processing units* (CPUs) with that of GPUs. We explain why stencil applications can profit from execution on the GPU and elaborate on the fundamentals of the *Nvidia CUDA* platform for execution on the GPU.

Most classical computer programs run sequentially on the CPU. Computations are performed step-by-step. Such a sequence of operations is called *thread*. Early CPUs were only capable of running one thread at a time. Even today, only a handful of threads (i.e. execution streams) can run truly in parallel on a CPU. Operating on large data sets on the CPU therefore still mostly involves repeated calculations within loops that handle one data point at a time.

Most performance optimizations in CPUs target *latency*; caches, pipelining, branch prediction, and similar techniques aim to reduce the time between issuing a command and storing its result. While CPUs have become much faster since their inception, the improvements to performance have recently slowed due to physical constraints.

Sustained demands for faster runtimes and more complex applications have thus forced rethinking the sequential execution model. Many real-world applications on large data sets consist of largely unvarying computations on many data points. These applications often have few sequential dependencies. Applications in computer graphics, for example, often entail highly monotone computations that are repeated for every pixel displayed on the screen. The resulting value of the pixel at one edge of the screen probably does not require knowledge of the result of a pixel at the other end, but sequential execution still dictates one value being calculated before the other. This observation spurred the development of parallel architectures, such as in GPUs. GPUs overcome the performance limits of sequential computation by providing thousands of hardware units which are able to compute (run threads) independently from one another, at the same time.[11, Chapter 2] This increases the *throughput* of those devices: While a *low-latency* CPU reacts to an issued command fast, it provides only one result for one data point. Meanwhile, a *high throughput* GPU might take a longer time to issue the command, but

it calculates the results for several datapoints "at once".

CPUs, therefore, shine in scenarios where calculations are less straightforward and predictable, while GPUs are more useful wherever monotone computations on large data sets are performed. Another advantage of GPUs is *scalability*: Because the performed parallel computations are completely independent, larger problem sizes can effectively be made more efficient simply by adding more parallel execution capabilities to the hardware, i.e. adding more processors. Despite the name, GPUs are today no longer just used for graphics processing. Such computing is also called general-purpose GPU programming, or GPGPU for short.

The task of applying a stencil to a grid (sections 3.1, 3.2) greatly benefits in terms of performance from execution on highly parallel architectures such as graphics processing units (*GPUs*), as it can easily be parallelized by decomposing the problem domain. Each thread may be responsible for calculating the result of one cell or a small set of cells in the output grid. As data dependencies are limited to a local neighborhood, parallel threads can work on spatially separated data concurrently without risk of data races in most cases. Data races only occur if threads share some dependencies, i.e. when their neighborhood overlaps. In that case, each thread can re-compute its dependencies (called *computation on-the-fly*) or threads may share their results by means of *synchronization*.

In this thesis, we use the *Nvidia CUDA* architecture to implement meteorological stencil computations on the *Nvidia Tesla V100* GPU, and we assess their performance on different types of grids.

3.3.1 SIMT Execution Model

CUDA employs a *Single Instruction, Multiple Thread (SIMT)* execution model. This model can be compared to the *Single Instruction, Multiple Data (SIMD)* model but makes writing programs more straight-forward to programmers experienced in sequential programming.[5, Section 3.1]

In a *SIMD* model, instructions are "wide": They support input operands that are larger than single scalar values. Programmers explicitly issue these *vector instructions* to perform a calculation on a set of values. In contrast, in the SIMT model used by *CUDA*, programmers do not need to explicitly perform operations on multiple data points. Instead, code is written such that it operates on single scalar values. Many instances, i.e. *threads*, of this code are run which differ only in an input *thread index* they receive. In most applications, this input is used to determine which data point the thread operates on. Upon execution, when the same operation is executed in many threads on consecutive data points, they are grouped together automatically to be executed in parallel in SIMD-like fashion.

3.3.2 Software

Code to be executed on the GPU is written in specially-annotated functions called *kernels*. Kernels can be written as regular C code but are compiled by the distinct Nvidia compiler *(nvcc)* to *Parallel Thread Execution (PTX)* machine code which can be

run on the GPU. Inside a kernel, a *thread and block index* is made available. Using a special syntax recognized by *nvcc*, kernels can be launched from the CPU onto the GPU (sometimes simply called *device*, as opposed to *host* which refers to the CPU). The *CUDA* Application Programming Interface (API) provides a device synchronization routine, which is required to synchronize CPU and GPU after the computation of the kernel on the GPU is completed. This enables relatively simple offloading of parallelizable workloads onto the GPU as a coprocessor while continuing to run the rest of the program asynchronously on the CPU. The *CUDA* API furthermore provides routines for memory allocation on the device, memory prefetching (see 3.3.4 unified memory), and setting certain device parameters.

```
// Kernel Definition. This function runs on the GPU.
__global__
void multiply(double fac, double *input, double *output) {
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    output[i] = fac * input[i];
}

// Kernel Invocation. This function runs on the CPU.
int main(int argc, char **argv) {
    double *input, output;
    int N = 1024 * 1024;
    cudaMallocManaged(&input, N * sizeof(double));
    cudaMallocManaged(&output, N * sizeof(double));
    cudaMemset(input, 42.0, N * sizeof(double));
    multiply<<<N/256, 256>>>(3.0, input, output);
    cudaDeviceSynchronize();
    cudaFree(input);
    cudaFree(output;)
}
```

Listing 3.1: Example showing kernel, its launch and CUDA API calls for allocating unified memory

3.3.3 Streaming Multiprocessors

CUDA-capable GPUs are structured as an array of so-called *Streaming Multiprocessors (SMs)*. Each SM contains a multitude of scalar processor cores. These cores include arithmetic, logic, and floating-point units (ALUs, FPUs) and perform the actual computations on the data points. Together, they are used to advance computation in a *warp*. A warp is a set of (typically 32) threads that are executed concurrently on the same multiprocessor. In every cycle, each thread inside a warp uses one of the scalar cores to execute the same instruction (on different data), or it is turned off if a branch

diverged previously. A warp of multiple threads at the same instruction pointer executing the same instruction is therefore similar to executing a SIMD instruction on multiple data points.

Each SM has its own scheduler and operates completely independently of the other SMs. If the threads inside a warp diverge (because of branching instructions), the different execution paths are executed sequentially on the SM. It is therefore paramount to performance that kernels are written in such a way that threads in the same warp follow the same execution path whenever possible. One way to achieve this is to ensure branch conditions involve only the thread index divided by 32 (i.e. the warp size), as this will be the same value for all threads inside a warp.

SMs choose the threads to form a warp from a pool of threads called a *block*. When one warp finishes executing or is stalled (e.g. because of a memory dependency), the next set of threads forming a warp in the block is chosen to be executed. All threads in a block execute on the same SM. Resources such as registers and shared memory are divided among all threads in a block. Threads in the same block can communicate with one another through so-called *shared memory*. Using the `__syncthreads()` command in a kernel synchronizes all threads in the same block.

When a kernel is launched, the programmer may specify how many threads should be part of the same block (*block size*), and how many blocks there should be (*grid size*). These parameters are called the *launch configuration*. It is important to have more blocks than SMs, as otherwise, some SMs will have no work to perform. Furthermore, if a synchronization instruction is used in the kernel, it is beneficial to have multiple blocks per SM, as to occupy the SM when one of the blocks is stalling because it is waiting for synchronization. The number of threads per block (block size) should be a multiple of 32 to ensure warps can be entirely filled with threads. [3, Section 10]

3.3.4 Memories

There are several types of memory address spaces available in kernel code. Figure 3.1 shows these address spaces in the programming model on the left, and how they are implemented in hardware with caches in the Volta architecture on the right.

Local Memory

Local memory is visible only to one thread and is almost always implemented using registers. There is a limited number of registers available in the SM. Using a large number of registers in a kernel thus reduces the number of threads that can be launched on the same SM. Registers can be spilled to local memory, but this incurs a cost; registers are on-chip per SM, whereas spilled registers require access to device memory.

Shared Memory

Shared memory is shared between all threads in a block. It resides locally in each SM, close to the functional units. The amount of shared memory is limited per SM

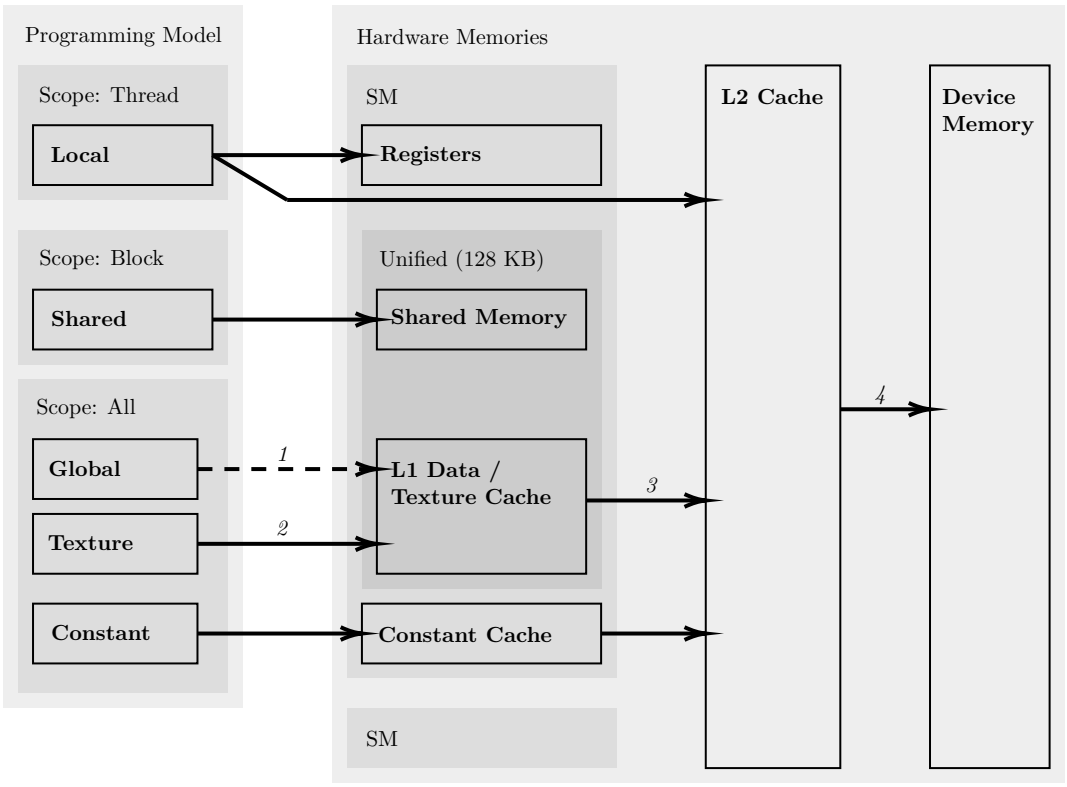

Figure 3.1: Memory Hierarchy of the Nvidia Volta architecture (compute capability 7.0). Global memory accesses are only cached in the unified L1 data/texture cache if they are read-only. Illustration based on information provided in [4, Sections 2.3, 5.3.2, H.6].

and therefore using more shared memory reduces the number of threads that can be launched in parallel. In the Volta architecture, shared memory competes for space with the L1 cache for global and texture memory accesses.

Bank Conflicts When using shared memory, one has to be wary of bank conflicts. Accesses to shared memory of multiple threads can be executed simultaneously as long as they fall into separate so-called *shared memory banks*. In the *Volta* architecture used in this thesis, there are 32 banks. Consecutive 32-bit words are mapped to consecutive banks. To avoid bank conflicts it is thus important that for any two threads in the same warp, shared memory addresses accessed are coprime, i.e. for two memory accesses at addresses i and j, $i \neq j \mod 32$. An exception to this rule is if all threads access the same address, in which case a *broadcast* occurs.

Global Memory

Global memory is accessible by all threads across all blocks. Using the `cudaMalloc()` and `cudaFree()` routines provided by the *CUDA* Runtime API it can be allocated and freed. Note that the pointers returned by `cudaMalloc()` cannot be used in CPU code. Instead, data has to be manually copied from the host to the device and vice versa using the `cudaMemcpy()` function with `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost` parameters, respectively.

Since *CUDA* version 6.0, there also exists *unified memory* which relieves programmers from manually having to copy memory back and forth. Unified memory provides an address space that is accessible from both the host (CPU) and the device (GPU). Copying is done on-demand when data is being accessed. Therefore, when timing exclusively the kernel runtime, switching from managed memory (with explicit memory transfers before and after the kernel run) to unified memory could impact the measured kernel-only run time. The overall runtime does not change, as memory has to be transferred in both cases; the transfer simply moves from the explicit call to `cudaMemcpy` implicitly to the first access to a unified memory address in the kernel.

Synchronizing the memories between host and device can also be made explicit in unified memory using the function `cudaMemPrefetchAsync()`. We use this mechanism in our benchmarks to ensure only the relevant aspects of kernel runtimes are reported, without any distortion by memory transfers that have to take place for any kernel either way.

Additionally to global/unified memory, there also exist *constant* and *texture memory*. These address spaces are also persistent across threads. Constant memory may not be written, but provides better performance when all threads access the same address. Texture memory is similar to global memory, but special routines in the Cuda API are provided for accessing it. Furthermore, it is cached in a way that profits from accesses which are spatially local in 2D.

Coalescing A paramount performance concern for memory-bandwidth-intensive kernels is the pattern of global memory accesses. When all threads in a warp read consecutive 4-byte words, these reads are executed as one larger vector load instruction (the accesses are said to *coalesce*). This enables simultaneous reading of memory for all those threads. On the other hand, if the addresses accessed are sparse, each request has to be serviced in series by the SM. This reduces performance drastically. Therefore a memory layout should be chosen where data needed by different threads at the same computation step is laid out sequentially wherever possible. If this is not doable, a solution can be to intermediately load some data into shared memory with a coalesced access and then accessing the needed values through shared memory fetched by another thread. If all threads require access to the same address, constant memory may also be a remedy for uncoalesced accesses.

Caches

Caching of memory accesses turned out to be another major factor in determining the runtimes of our implemented stencils. In the used Volta architecture, there is an L1 and an L2 cache for global and constant memory accesses. The L1 cache is private to each SM and shares its space with shared memory; using shared memory can therefore also be seen as an explicitly managed cache. Only read-only accesses to global memory are cached in L1, as this cache is per-SM and writes would require inter-SM synchronization. A second, separate L1 cache services constant memory reads and writes. The L2 cache is shared between SMs. It functions as a cache for global reads/writes as well as for local memory spilled from registers and constant accesses missing the L1 constant cache. In [6, Chapter 3], the latency for an L1 hit is reported at 28 cycles, and at about 193 cycles for an L2 hit. A more detailed overview of the memory structure and caches is also given in that paper.

3.3.5 Performance Considerations

To understand the GPU performance of a kernel, it is important to understand the benefits of latency hiding. Instructions that load from (slow) memory or depend on the result of a previous instruction that has not yet been completed can cause so-called *stalls* in both CPUs and GPUs. While on the CPU, many performance optimizations target reducing the number and the effect of these stalls (i.e. reducing latency), GPUs have the capability to *hide* stalls by means of simply switching to a different warp that is not blocked. To analyze the performance of GPU applications we are therefore interested in two main characteristics of the execution:

1. The *occupancy*, which describes how much work is available to the SMs on the GPU. In the ideal case, no SM should ever be idle waiting for a stalled warp. It should execute useful work instead to achieve high throughput. The *achieved occupancy* is defined as the ratio of active warps to the maximum number of warps supported on an SM. Altering the execution configuration (block size) of a kernel

can aid in improving occupancy. Programs with a higher occupancy are better able to hide stalls.

2. The main *reasons for stalling*, which tell us why individual threads executions block. For many kernels (all stencil applications in this thesis) memory dependencies are the main reason causing stalls. For these types of kernels, a close look at the *achieved memory bandwidths* in comparison to the maximum achievable bandwidth of the used hardware often reveals where the deficiencies lie. For other more computationally expensive kernels, stall reasons may include busy instruction pipelines or execution dependencies. Programs that stall less require less occupancy to hide those stalls; knowing the reason for stalls is thus an important guide in improving performance.

Considering these two characteristics gives an overview of what the main limiting factors of a kernel are. Both factors influence each other; when threads never stall, SMs will always have something to execute (provided there are enough threads). Occupancy will thus be high even if the number of issued threads is not much larger than the number of SMs times the number of threads in a warp (32). If occupancy is low, splitting the problem into smaller parts and increasing the number of threads and blocks may help, but only if these threads do not all stall at the same time.

`nvprof` metrics

Nvidia provides a command-line profiling tool called `nvprof`. This tool supports collecting several metrics as kernels are executed. Some of these metrics which are of particular interest for the analyses to follow are:

Occupancy metrics

`achieved_occupancy`	Ratio of active warps to the maximum number of warps supported on an SM, averaged over all SMs. Higher is better.
`issue_slot_utilization`	Ratio of instructions issued on a per-core level to maximum hardware capability. This is more fine-grained than the achieved occupancy, as it captures if only a few threads per warp are active. Higher is better.
`ipc`	Warp-level instructions executed per cycle. If there are many stalls, this ratio drops. Furthermore, this captures how well the SMs are able to pipeline instruction streams of a kernel. As pipelining is not as sophisticated as on CPUs, simple measures such as loop unrolling may yield better numbers here. Higher is better.

Stall reason metrics	
`stall_memory_dependency`	Stall reasons give insight into why threads cannot execute. If most stalls occur due to memory dependencies, comparing `dram_read_throughput` to the maximum value attainable by the device reveals whether the kernel performance is memory-bandwidth-bound. In memory-bound kernels, *coalescing* and *caching* metrics are especially important.
Coalescing metrics	
`gld_efficiency`	In case of bad coalescing, the device performs reads on many values that the kernel does not actually require. This happens if the requested data's addresses do not align with the bounds of a single load instruction. This metric indicates how much of an executed read is actually used by the kernel, and how much of the read is wasted. Higher is better.
`gld_transactions_per_request`	Reports how many memory transactions (32-byte load instructions performed by the SM) actually had to be performed on average per warp-level (32 threads) memory request. Lower is better.
Caching metrics	
`tex_cache_hit_rate`	L1 cache hit rate. Each SM has its own unified L1 cache. The texture cache is mentioned explicitly in the names of some of these metrics because this cache has not been unified with the global and other caches in some previous iterations of the architecture. Higher is better.
`l2_tex_hit_rate`	L2 cache hit rate. This cache is shared among SMs. Higher is better.
`tex_cache_transactions`	Sum of arrows 1 and 2 in figure 3.1. Absolute number of transactions seen at L1 cache.
`l2_read_transactions`	Arrow 3 in figure 3.1. Absolute number of transactions at L2 cache.
`dram_read_transactions`	Arrow 4 in figure 3.1. Absolute number of device memory reads (uncached).
`global_hit_rate`	Arrow 1 in figure 3.1. Hit rate at L1 cache *only* for global memory reads (excludes texture memory).

Verification	
`gst_transactions`	Profiling also provides a simple means of verifying the correctness of the kernel. Checking whether the number of global store transactions is as expected gives an indication whether the kernel is behaving as expected.

One aspect of profiling a CUDA application in specific that also must not go unmentioned is the *just-in-time compilation* of kernels. The PTX instructions stored in CUDA binaries are not low-level machine code for the graphics card. In order to support running the same application on various platforms, these instructions are instead compiled on-the-fly for each host program run by the Nvidia driver. It is expected that the first execution of a kernel takes more time than subsequent executions due to this compilation step. In our benchmarks in section 6, we, therefore, did not include the first run of a kernel.

Chapter 4

Grid Storage Strategies

In this section, we explain different approaches for storing into memory the regular and unstructured grids which will be used by the stencil computations in subsequent sections. We detail considerations that need to be made when choosing memory layouts for grids for the CUDA platform.

As indicated in section 3.1, we will use the two notions of coordinates in Euclidean space and indices (=addresses) in memory in the following to describe how grids are stored. Two aspects of a grid and its storage implementation need to be described:

1. The way the *values* stored inside a cell are laid out in memory, i.e. how coordinates in Euclidean space map to indices in memory.
2. How the *neighborship relations* for a cell are defined, i.e. given a certain cell, how to find the desired neighbor.

4.1 Regular Grids

Both indexing and neighborship relations are easy to determine in regular grids. Thanks to the grid's regularity, coordinates in combination with the dimension of the grid carry enough information to be directly translated to memory locations.

4.1.1 Row-major Indexing

The coordinates can directly be mapped to a memory index by simple arithmetic. The perhaps most popular way to do this is *row-major indexing*. This is the memory layout many programming languages such as *C* use to lay out multi-dimensional arrays in memory. The cells of the grid are indexed as follows: A cell at coordinates p receives the offset

$$\text{index}\,(p_x, p_y, p_z) = p_x + p_y \cdot d_x + p_z \cdot d_x \cdot d_y.$$

Herein, the factors besides the coordinates are called the strides, i.e. the *x-stride* is 1, *y-stride* is d_x and *z-stride* is $d_x \cdot d_y$. Only in a regular grid are the strides constant – this is the advantage of using a regular grid. Stepping through the memory linearly,

this means that the X-coordinate is the fastest-changing and the Z-coordinate is the slowest-changing. Using this scheme, memory locality is good for cells with similar X-coordinates, but not necessarily so for cells with similar Y- or Z-coordinates.

4.1.2 Neighborship Relations

With row-major indexing, in order to access the value of a neighbor of a cell at position p, it suffices to know the coordinates of the cell and the strides of the grid. The indices of the left (right), top (bottom), and front (back) neighbors are simply given by subtracting (adding) the x-stride, y-stride, or z-stride respectively. This gives the same index as subtracting one from (adding one to) the respective X-, Y- or Z-coordinate and then calculating the index as described above. All that is required for accessing a neighbor's value is simple arithmetic for the index and a memory load at the calculated index to receive the cell's stored value.

For example, in memory, the left neighbor of a cell in memory at location i is located at $i-1$ the top neighbor at $i-1 \cdot d_x$ and the back neighbor (Z-dimension) is at $i-1 \cdot d_x \cdot d_y$. From this, it is evident that when using row-major indexing in regular grids, neighbors in the X-dimension (left/right) will have great memory locality. However, as the grid dimension in X- or Y-dimension exceeds beyond what the processor can hold in the cache, accesses to neighbors in Y- or Z-direction become more costly.

4.1.3 Memory Alignment

The Cuda compiler generally ensures that data structures are well-aligned in memory for the target architecture for coalescing accesses. However, when storing a regular grid for later stencil applications, manual alignment calculations can become necessary due to the *halo* (as defined in section 3.2).

Consider some stencil applied to a regular grid. Because of the lack of neighboring values, a kernel will not operate on values at the boundary of the grid, the halo. The first thread actually executing any load/store instructions will be on an inner value. Therefore, if the halo is not a multiple of the vector load instruction size (32), some loads at the beginning and end of each row will not be aligned. The addresses of the halo cells are not loaded and thus wasted in the instruction.

This problem can be aleviated by adding paddings such that not the first *inner* value is 32-byte-aligned instead of the very first element of the array. Given a halo of size $h = \begin{pmatrix} h_x & h_y & h_z \end{pmatrix}^\top$ we want the cell at coordinate h to be aligned, not $\begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^\top$. We thus chose minimal paddings a, b and c and reformulate the index computation such that

$$\text{index}\,(p_x, p_y, p_z) = p_x + a + p_y \cdot (d_x + b) + p_z \cdot (d_x \cdot d_y + c) \tag{4.1}$$

$$\text{index}\,(h_x, p_y, p_z) \equiv 0 \mod 32 \qquad \text{for all } p_y,\, p_z \tag{4.2}$$

Note that the modulo computations are only performed upon grid generation (on the CPU). Once the grid is stored in memory, stencils require only the chosen paddings

a, b, c as additional inputs in order to correctly compute the memory index of a cell at any coordinate (according to equation 4.1)

4.2 Unstructured Grids

In completely unstructured grids, memory indices are in no fixed relation to coordinates. This complicates neighbor accesses. Adjacency information for each cell must be explicitly stored. Accessing a given cell's neighbors requires a lookup of this information. This is contrary to regular grids, where adjacency information is implicitly known through coordinates and grid dimensions (i.e. constant strides).

To facilitate some optimizations and more accurately model the typical unstructured grids in meteorological applications, we restrict the notion of an unstructured grid throughout the remainder of this thesis. Specifically, we pose the following requirements to our unstructured grid implementation and stencils operating on it:

1. **Regular in Z-dimension:** The grid remains regular in the Z-dimension, i.e. the mentioned decoupling of indices and coordinates only occurs in the X-Y-plane. For neighbors in the X-Y-plane, cells with equal X and Y coordinates have the same relative neighborship offsets at all Z-levels. The Z-coordinate retains its meaning and can be used in conjunction with the Z-stride $d_x d_y$ to access neighbors in the Z-dimension. Conversely, the Z-coordinate of a cell can be inferred from the cell's index i, specifically $p_z = i \mod d_x d_y$. Note that knowledge of the absolute Z-coordinate is not required in stencil code, thus costly modulo operations never occur in stencil code.

2. **Stencils operate only on a bounded region, relative to each cell:** A stencil operating on some cell requires only its own cell's fields, its neighbor's fields up to some bounded depth l and constant inputs to calculate its results. Only relative offsets to the current cell's coordinates are accessed. Stencils applied on our unstructured grid do not require knowledge of absolute X and Y coordinates. Given a cell at memory index i, determining its coordinates p_x, p_y is *not* required in stencil (GPU) code. (Of course, the mapping is defined and must be accessible in CPU code in order to access and display the grid values – this access is not required to be efficient, though.)

As with the regular grid, our implementation must define how the *values* (next section) and the *neighborship information* (section 4.2.2) are stored. We describe certain important considerations in the context of a SIMT application for both but focus on the latter.

4.2.1 Memory Layout and Indexing of Cells

The inner values in our unstructured grid implementation may be stored in an arbitrary layout. The *halo* cells must be stored separately from the inner values. In this separate

block of halo cells, the memory layout may also be arbitrary. In the following subsection, we describe the considerations that were made in the context of halo storage.

While completely arbitrary layouts for the values are possible, some patterns of regularity will be present in most real-world use cases. Section 4.2.3 describes the two types of memory layouts (*row-major*, *z-curves*) we simulated when benchmarking our unstructured grid implementation.

Halo

To ensure correct results, stencils may only operate on cells where the required neighbors for the output calculation are present. In the regular case, this is ensured by checking in each thread that the current coordinate lies within the inner part of the grid or within the halo. In regular grids, this is fast: coordinates can be determined from memory indices using only arithmetic. On the other hand, a thread in our unstructured grid implementations can not know the absolute coordinate of a cell it operates on, due to the possibly arbitrary layout of cells and unknown (to the thread) mapping from indices to coordinates. For unstructured implementations, a thread receives only a memory address of some cell, for which it must compute the result. Therefore, another approach must be taken.

One solution works as follows: The required neighborship lookups are performed in every thread. When the desired neighbor is not present, a special value indicates this; the program concludes at this point that the currently being operated-on cell lies in the halo and aborts. This might happen, for example, when a thread operating on the topmost cell asks for the index of the cell above it. Consider a neighborship storage implementation that stores a relative offset of the index for each neighbor; in this type of implementation, an offset of 0 is effectively a pointer to the same address/cell the thread is already operating on. It could, therefore, be used as a special value to indicate the required neighbor is not present.

There are two main disadvantages to this approach: First, even for cells in the halo, a (possibly costly) memory lookup is performed before threads determine no computation can be done. Second, due to the inactive/aborted halo-threads, there will be under-utilized memory loads, similar to uncoalesced accesses. The considerations for coalescing made in section 4.1.3 for regular grids do not apply here, because in an unstructured grid with varying strides no constant padding can be chosen to ensure coalescing accesses.

A better solution is to store the halo and inner values in separate blocks in memory. Threads are then initiated to only operate on the inner value block. We opted to store any cell located in the halo of the stencil in front of any of the inner values in memory. With this approach, we can safely operate threads on the memory block of only inner values and be sure to never encounter a halo cell. If there are several stencils with different-sized halos to be applied to the same unstructured grid, spiralling storage of the halo cells moving from the outwards in can even be employed. The starting address of the first inner value then determines what is considered as inner value by the stencil. Stencils with a smaller halo can operate on a memory block starting at a lower address, including some cells that would lie in another stencil's halo. However, storing the halo

at the beginning, separated from inner values, comes at the cost of a reduced memory locality for inner cells whose neighbors reside in the halo. This cannot be avoided.

Across all of our implementations, we employed the second approach of storing the halo separately, in front of the inner values. As in the benchmarks, only one stencil was applied to each grid at once, we did not use the spiraling scheme; halo cells are simply stored in a row-major fashion, followed by the inner values.

4.2.2 Neighborship Relations

To characterize an unstructured grid, adjacency information about all cells must be explicitly stored in memory. We store this information in what we call *neighborship tables*. For clarity, we will refer to two distinct blocks of memory in the following: The *neighborship block* stores information about the structure of the grid, while the *data block* stores the values of the cells.

Suppose there are m types of *neighborship relations* (e.g. top, bottom, left and right neighbors, $m = 4$), i.e. each cell can have at most m neighbors. For each relation, one array of size $d_x d_y$ is allocated in the neighborship block. Each array functions as the *neighborship table* for one relation (e.g. a top-, bottom-, left- and right-array). Consider a cell whose value is stored in an array at index i in the data block. In each neighborship table, at offset $i \mod d_x d_y$, a pointer to a neighbor of that cell is stored. This pointer is a relative offset from the index i.

Storing relative offsets instead of absolute pointers has the advantage that a thread working on the value at index i does not need to know where the beginning of the data block is; it can simply add the offset to the index. An offset of 0 signifies that the cell at this index does not have such a neighbor.

The described neighborship tables are akin to adjacency lists[1, Chapter 12]. Consider a graph representation of a grid, where nodes represent cells, and edges represent the neighborship of two cells. Given m types of neighborship relationships (e.g. "top", "bottom", "left" and "right"), we create m such graphs linking the respective neighbors (e.g. a graph connecting all nodes with their "upper" neighbor). Each neighborship table is the adjacency list of such a graph, where the cells are referred to by their relative offset from the current cell in memory.

All neighborship tables are stored consecutively in memory. We allocate $\texttt{sizeof(int)} \cdot m \cdot d_x d_y$ bytes in the neighborship block to store the neighborship tables. A pointer to the k-th neighbor of a cell stored at index i can be found in the k-th neighborship table at index $i \mod d_x d_y$. Herein, the modulo $d_x d_y$, i.e. the Z-stride, is required for memory indices i for $Z > 0$ to remove the Z-component from the index.

Putting all of this together, the following equation 4.3 describes how the index (in the data block) of the k-th neighbor in the X-Y-plane of a cell stored at index i (in the data block) is computed. The ordering (i.e. which neighbor is the k-th) can be defined arbitrarily, as long as it remains consistent.

$$\text{neighbor}_k(i) = i + \texttt{neigh}[d_x d_y \cdot k + (i \mod d_x d_y)] \tag{4.3}$$

Herein, the function $\text{neighbor}_k(i)$ gives the index of the k-th neighbor of the input cell, `neigh`$[l]$ refers to the l-th element in the neighborship block (as in C), and $d_x d_y$ is the Z-stride (i.e. dimensions of the X-Y-plane). Note that modulo calculations are expensive operations; our optimized grid access variants (see section 5) thus avoid those by using the three dimensions of the thread index to give an index in the X-Y-plane (which would be the result of the modulo operation) and the Z-coordinate separately. Further note that thanks to the regularity in the Z-dimension, neighbors in the Z-direction can be accessed without an additional lookup by simple addition or subtraction of the Z-stride, $d_x d_y$.

Example: Given an array `values` representing the data block, which stores the values of each cell, an array `neighbors` representing the neighborship block, and assuming the "right" neighborship relation is defined as the 2nd ($k = 2$) in the neighborship block, then the value of the right neighbor of cell i is stored at `values`$[i +$ `neighbors`$[2 \cdot d_x d_y (i \mod d_x d_y)]]$.

The stencils benchmarked in section 6 are defined in terms of regular grids. In our implementations, we limit each cell to have at most 4 directly adjacent neighbors. This is the maximum number of neighbors any cell in a two-dimensional quadrilateral grid can have.

The described neighborship storage approach can be further optimized. In the benchmarks in section 6, we will present four variations for the described neighborship tables, resulting from the combination of two properties:

- *Chasing / non-chasing*
  - *Chasing*: Adjacency information is stored only for direct neighbors. Accessing neighbors-of-neighbors requires pointer chasing.
  - *Non-chasing*: Adjacency information is stored up to some depth l (as required by the stencil). One lookup suffices to determine the index of a neighbors-of-neighbors, but additional memory for the neighborship table is required.
- *Uncompressed / compressed*
  - *Uncompressed*: The neighborship table contains one entry for each cell in the X-Y-plane.
  - *Compressed*: Cells with the same relative neighborship offsets share their neighborship table entries. An additional lookup (akin to a dictionary mapping cells to their neighborship table entries) is required.

Given a grid with four neighborship types ($m = 4$), representing the entire grid's values and its structure, including neighbors-of-neighbors up to depth $l \geq 1$, can be achieved with a memory footprint of

$$\texttt{sizeof(int)} \cdot d_x d_y \cdot 2l(l+1) + \texttt{sizeof(T)} \cdot d_x d_y d_z \quad \text{bytes}$$

Herein, `T` is the data type used for values (`float` or `double`, or a struct for array-of-struct-type storage of multiple fields, see 4.3).

Coalescing of neighborship lookups

Storing neighborship offsets in multiple separate arrays in the described manner is crucial for on-GPU execution. It enables coalescing of the accesses to the neighborship tables. Suppose some stencil computation, in which multiple threads desire to calculate the output value for several cells at consecutive memory addresses. They will all require access to the same neighboring cells at each computation step (as the exact same stencil computation is applied to every cell). Therefore, if, for example, all threads require the top neighbor, consecutive addresses in the top-neighbor table will be requested and accesses will get coalesced. If, however, we had a "classical" adjacency list, where the top, bottom, left and right neighbors were each stored together (intertwined) for every cell, accesses by threads to the top-neighbor would be separated by a stride of four elements (the other neighbors), thus making coalescing impossible and forcing sequential loads. The two approaches to storing multiple neighborships are also referred to as *array-of-structs* and *struct-of-arrays*, and are further discussed in section 4.3, where the analogous problem for storing multiple values (fields) per cell is addressed.

Pointer Chasing

Some stencils will require neighboring cells beyond directly adjacent ones, i.e. neighbors-of-neighbors up to a certain depth l. As described above, an entry in the neighborship table is stored only for each neighborship *relation*. Accessing the neighbor of a neighbor thus requires so-called *pointer chasing*: Two lookups to the neighborship tables in series, where the second lookup location is determined by the result of the first. Due to this dependency, such lookups cannot be performed in parallel.

To avoid the issue of *pointer chasing* it can be beneficial to store all neighbors-of-neighbors up to a certain depth l in additional neighborship tables. This increases the memory footprint but can enable faster access to neighbors-of-neighbors.

In a grid where each cell is limited to four direct neighbors, storing all neighbors and neighbors-of-neighbors up to a depth of l requires $2 \cdot l(l+1)$ pointers per cell in the X-Y-plane.

Compression

In unstructured grids with some regular components, the neighborship tables contain many redundant entries. In a regular subsection of the grid, the relative offsets of the neighbors' indices are identical. We thus consider a compression scheme for the neighborship tables. Even though memory limits are not a constraint for any of the benchmarked stencils, compression is interesting for performance (runtime) reasons. This is due to caching; when many threads access the same neighborship table entries, it is very likely those entries reside in the cache and are available much more quickly. Several cells sharing the same neighborship table entries also leaves more room in the cache for entries for other cells.

Yet, to be beneficial to performance, a compression scheme must be very lightweight. Any computation required for decompression adds to the total runtime and could cancel

out or even outweigh the described caching advantages. The following proposed scheme was able to provide some benefit for large enough grids (see section 6).

Compression phase The compression phase occurs on the CPU, upon grid generation. Consider a completely regular grid stored in row-major order. Except for the halo cells, the relative neighbor index offsets are equal to the X- and Y-strides for every cell. In principle, only one neighborship table entry per neighborship relation would thus suffice to encode the strides. In unstructured grids, multiple of these *patterns* of neighborship relations may occur if there are regular components. Our compression works by grouping together such patterns of neighbor index offsets.

The structure of the grid is described by neighborship offsets $o = (o_1, \ldots, o_m)$ for each cell. Each of the m offsets describes the relative offset of one of the neighborship relations. In the compression phase, we create a new pattern for each unique vector of offsets o encountered. The offsets for each pattern are stored *exactly once* in the neighborship tables. All offsets for one pattern reside at the same pattern index in the neighborship tables. An additional array `pattern` of size $d_x d_y$ is allocated. This array provides a mapping from each cell (cell index) to its corresponding pattern (i.e. the index in the neighborship tables).

Stencil phase During the stencil computation, a thread operating on a cell with index i looks up the pattern of its neighborships by reading $\texttt{pattern}[i \mod d_x d_y]$. (The optimized access strategies seperate the X-Y-component in the index i from the Z-coordinate and as such do not require the modulo computation in their actual implementation.) For any subsequent neighborship lookup, the neighbor pointers are read from the neighborship tables using the previously obtained index from the pattern array. The previously described equation 4.3 for neighborship access is altered to the following:

$$\text{neighbor}_k(i) = i + \underbrace{\texttt{neigh}[d_x d_y \cdot k + \overbrace{\texttt{pattern}[(i \mod d_x d_y)]}^{\text{pattern lookup}}]}_{\text{(hopefully cached) neighbor lookup}} \tag{4.4}$$

Compare this to the uncompressed variant: there, the initial lookup of patterns is skipped, and neighborship tables are accessed at $i \mod d_x d_y$. In the uncompressed case, each entry in the neighborship table is only relevant to one cell, and only accessed by one thread per Z-level. In contrast, if compression is effective, there will be relatively few patterns, and the same few neighborship table entries will be accessed by many threads. We anticipate that those few neighborship table entries will reside in the caches due to their large number of accesses. We note that compression adds one indirect lookup (pointer chasing through the pattern array). For this scheme to be effective at reducing the runtime, the positive effect due to caching of the second lookup must outweigh the cost of the required additional pattern lookup.

It appears to be unavoidable to have the $d_x d_y$-sized pattern array if the complete flexibility of an unstructured grid is to be retained. But, thanks to compression, this

Storage	Compression	Constant Memory	Runtime
row-major	-	-	$814\mu s$
row-major	✓	-	$553\mu s$
row-major	✓	✓	**524μs**
z-curves	-	-	$770\mu s$
z-curves	✓	-	**562μs**
z-curves	✓	✓	$639\mu s$

Table 4.1: Effect of using constant memory applying the *Laplace-of-Laplace* stencil on a $512 \times 512 \times 64$-sized grid with $256 \times 1 \times 1$ threads. Using constant memory gives a slight 5% advantage in the highly regular row-major grids; for the more realistic z-curve scenario, it results in a slowdown of 14%.

should most likely be the only array whose entries are solely relevant to one cell per Z-level; neighborship table accesses should concentrate around a few relevant entries.

Using Constant Memory For uncompressed neighborship accesses, coalescing accesses can be easily attained because the access pattern is known beforehand. In the compressed case, the accesses depend on the result of the pattern lookup. Multiple threads often access the same neighborship table entries. This is not optimal for coalescing. For broadcast-type accesses, i.e. when all threads access the same address, constant memory is better suited. Constant memory provides broadcasting capabilities, but its separate L1 cache is smaller in the Volta architecture. We experimented with storing the neighborship tables in constant memory. The pattern array remained in regular global memory, as the coalescing problems do not apply there (every thread accesses its own entry, no interference). While, for the very regular grids with an exact row-major layout, this yielded a small speedup around 5%, it turned out to be much slower for more varied data layouts, such as a *z-order* curves layout (slowdown around 15%). Because the latter is much more realistic in a real-world unstructured grid use case, we opted not to use constant memory in our final implementation of compression. Table 4.1 shows the effects of using constant memory on one exemplary stencil.

4.2.3 Indirect Addressing Overhead

In our benchmarks (see section 6), we compared the performance of stencils on regular vs. unstructured grids. To ensure a fair comparison, it was important to be able to produce and verify the exact same output for both the regular and the unstructured grids. This was only possible by "emulating" an unstructured grid; we took the regular grid as a base, and then also represented it as if it were an unstructured grid, i.e. created a neighborhsip table for it and used indirect addressing in the stencils. This enabled us to capture the overhead of indirect addressing with otherwise unchanging conditions. Note that even though we benchmarked a grid with completely regular neighborships

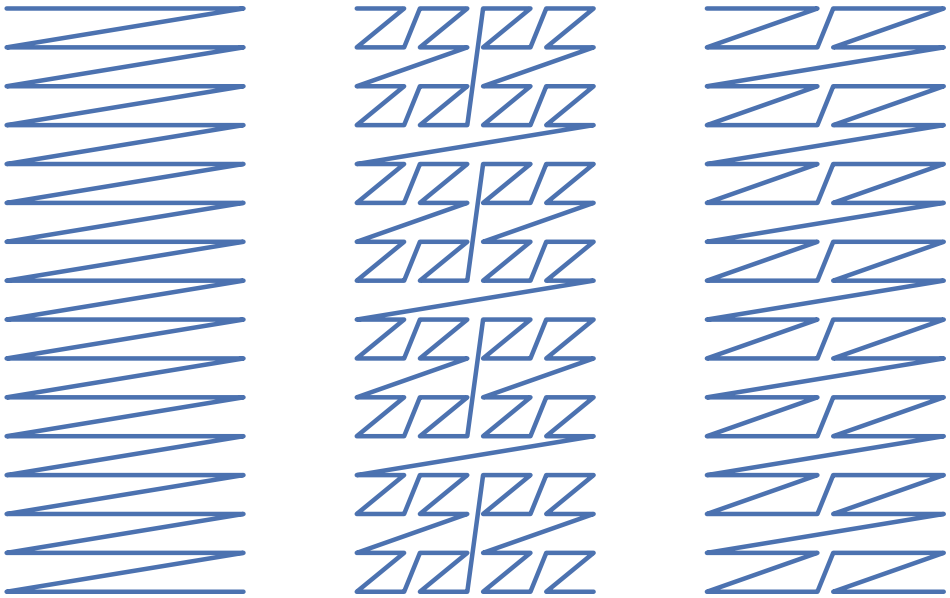

Figure 4.1: The different memory layouts described. The blue lines illustrate the order in which the cells in the X-Y-plane of the grid are stored in memory. The lowest memory index always starts at the top-left corner. Left: Row-major indexing. Middle: Normal z-order curve. Right: Widened z-order curve.

in the unstructured case, this knowledge was not made use of in the optimizations of the unstructured grid – as they stand, the implemented programs would also support completely arbitrary unstructured grids in two dimensions.

In these emulated unstructured grids (essentially regular grids using indirect addressing), we stored the same values as in the regular grid using two layouts: *row-major* (as in the regular grid case) and a variation of Z-order curves (referred to in the remainder of this thesis in short as *z-curves*). Note that the Z in "Z-order curves" refers to the shape described by this memory layout, not to the Z-dimension of our grid. We only use the described memory layouts to store two-dimensional planar slices of our grid.

The memory layout of the emulated unstructured grid using a row-major layout is almost identical to the one for regular grids. The only difference lies in the different handling of the halo values, which is done as described in section 4.2.1.

Z-order curves[12] represent a more realistic unstructured scenario. A variation of this layout might be used to represent a true unstructured grid with some irregularities. The aim of Z-order curves is to give most cells with close X-, Y- and Z-coordinates close

indices. Locality is achieved by *intertwining* the bits of the X- and Y-components of a coordinate, starting with the X-coordinate at the least-significant bit. The middle graph in figure 4.1 shows a Z-order curve. Observe how the blue line representing the order of memory indices seldom makes large jumps, meaning points that are close in physical space remain close in memory. Compared to the row-major layout (left in that figure), this is a big improvement in locality.

Consider a two-dimensional coordinate, wherein both components are given as a bit string of even length n as:

$$p = \begin{pmatrix} p_x & p_y \end{pmatrix}^\top = \left(\left\langle p_x^{(n)} p_x^{(n-1)} \cdots p_x^{(2)} p_x^{(1)} \right\rangle \quad \left\langle p_y^{(n)} p_y^{(n-1)} \cdots p_y^{(2)} p_y^{(1)} \right\rangle \right)^\top$$

The index in Z-order curve ordering is then given by:

$$i = \left\langle p_y^{(n)} p_x^{(n)} p_y^{(n-1)} p_y^{(n-1)} \cdots p_y^{(2)} p_x^{(2)} p_y^{(1)} p_x^{(1)} \right\rangle \tag{4.5}$$

In this thesis, we have implemented a memory layout which uses a modified Z-curve layout in the X-Y-plane. To make use of CUDA's vector instructions, which can consume up to 32 bytes of consecutive memory at once when properly aligned, the Z-curve implemented is a stretched out variant. The last five bits are *not* intertwined. To ensure the indices are dense, the cells are then ordered by the number obtained due to this intertwining and are indexed in a consecutive manner from lowest to highest intertwined number.

Using this, we get repeated lines of 32 cells that share their relative neighborship offsets with other such lines; this allows better-coalesced accesses. Using a width of 32 ensures that coalescing is possible even for one-byte data types; when using four- or eight-byte floating-point types, the width could be reduced. We briefly experimented with smaller stretchings of the Z-curve and observed only minimal differences in runtimes. In the z-curves layout, neighborship offsets are more varied than in a row-major layout, rendering both compression and coalescing less efficient. This is thus a more realistic and useful scenario for modeling a real unstructured grid.

Using a truly arbitrary, randomized memory layout was also considered, but not pursued further after initial tests. In grids with a randomized layout, cache locality was completely destroyed, leading to runtimes around eight times the row-major or z-curve variants. For example, the *Laplace-of-Laplace* stencil described in section 6 completed in around $814\mu s$ for a row-major layout, whereas it took $6999\mu s$ in a completely random layout in otherwise identical conditions. However, such a completely random memory layout is highly unrealistic. Even in real unstructured meteorology applications, large portions of the grid would remain regular. Unstructuredness only occurs in boundary areas, e.g. between areas of different resolutions. Evaluating the performance implications of a completely random layout provide little value to the implementation of meteorological stencils and possible optimizations would be very limited.

Benchmark	Run Time	Load Efficiency
Array of Structs	$8282\mu s$	25.73%
Struct of Arrays	$2546\mu s$	99.39%

Table 4.2: Comparison of run times and global load efficiency (ratio of requested loads to performed loads, where bad coalescing leads to more loads being performed than needed) for the fast waves benchmark with domain size $512 \times 512 \times 64$ and block size of $128 \times 1 \times 2$.

4.3 Representing Multiple Fields

Two of the three benchmarked stencils and most real-world applications perform calculations that require more than one input value per cell in the grid. There are two obvious approaches to storing multiple *fields* for both regular and unstructured grids, *array-of-structs* and *struct-of-arrays*.

Array of Structs In the array-of-structs memory layout, all fields for one cell are stored together, before any values for the next cell. This means that accesses to different fields of the same cell have good memory locality, whereas accessing the same field of different cells requires larger strides.

Struct of Arrays In the struct-of-arrays layout, there are k arrays for k fields. This is conceptually the same as having k different one-field grids. Different fields of the same cell are stored in separate arrays and are thus separated by (at least) the size of one array. This approach requires additional care to keep the indices for cells synchronized across all arrays; deleting or adding a cell requires access to all k arrays.

The struct-of-arrays approach is highly beneficial to most GPU stencil implementations because of coalescing. When a stencil is implemented such that each thread is responsible for the calculation of one output value, all threads will most likely try to access the same field on different cells concurrently. In the struct-of-array layout, these accesses are able to coalesce (if the cells appear consecutively in memory). Because of this, we have implemented all our benchmarks in this fashion, i.e. as multiple one-field grids. To verify the claim that array-of-structs is slower than struct-of-arrays, we have implemented both variants for the *fast waves* benchmark, see table 4.2.

However, in the struct-of-arrays approach, cache locality might suffer from the large strides between the fields of a cell. It might be desirable to also load the other fields of a cell into the cache when one field of that cell is accessed. In this approach, however, only identical fields from other cells are close and thus loaded in the cache. A compromise that seeks to combine both the caching advantages of array-of-structs and the coalescing of struct-of-arrays is the *array-of-structs-of-arrays* approach, which stores one array, which for each warp-sized block of cells contains a struct of warp-sized arrays for each field. We have not implemented this.

Chapter 5

Grid Access Strategies

In a stencil, computation of the output value of one cell requires access to a neighborhood of cells. Accessing a cell's neighbor's value entails determining the coordinates of the desired neighbor and subsequently calculating the memory index of those coordinates. For structured grids, both of those tasks involve only arithmetic (as described in the indexing section). For unstructured grids, neighbor access in the X-Y-plane requires an additional memory lookup. This is not necessary for neighbors in the Z-direction, as the grid is regular in this direction.

In this chapter, we describe how stencils can obtain the index of required neighbors and access the grid in optimized ways.

5.1 Naive Grid Access and Index Variables

5.1.1 Naive

In the *naive* grid access approach, one thread is mapped to each output cell (total of $d_x * d_y * d_z$ threads). The indexing and neighborship calculations (including the memory lookup required for unstructured grids) are (re-)performed each time a cell's value is accessed in a stencil. One inefficiency of this approach is that most stencils require the same neighboring cells multiple times in different parts of their calculations. Even though the structure of the grid does not change, the indexing calculations are redone in the naive approach on every neighbor access.

5.1.2 Index Variables

The *index variables* approach (*idxvar* for short) addresses the issue described for the *naive* approach. There is also one thread per output cell. In this variant, in the first phase of the kernel, all required neighboring cell's indices are determined and stored in variables. These index variables are then used whenever a neighboring cell's value needs to be accessed. This ensures that indexing/neighborship operations (including lookups) are only performed once, even if the same cell is accessed multiple times within one thread. The additional index variables potentially increase the register usage of the

kernel, but they reduce the number of expensive memory lookups into the neighborship table if the same neighbors are re-accessed within the same kernel.

5.2 Optimizations Making Use of the Z-Regularity

5.2.1 Index Variables + Shared Memory

In this approach (*shared* for short), the stencil is implemented such that there is one thread per output cell (as in *naive* and *idxvar*). However, not all threads perform the neighbor index lookups. Instead, for each cell in the X-Y-plane, designated "leader" threads in each block perform the index calculation/lookup of all required neighboring cells at the $Z = 0$ level. The designated threads store the result of that computation/lookup in shared memory. If the Z-index modulo the Z-block-size is zero, a thread is a designated lookup-thread (leader). After the indices of the neighboring cells at the lowest Z-level are determined and stored in shared memory, all threads in the block synchronize. All threads then access shared memory to obtain the required indices. They add the appropriate constant Z-stride to them to obtain the index of the neighbors at their Z-level. Using this approach, the regularity of the grid in the Z-dimension is exploited in order to only perform one global memory lookup per block for the neighborship information. The shared memory lookups are cheaper than lookups in global memory, but synchronization adds some overhead. For this approach to be effective, the block size in the Z dimension needs to be large enough.

Bank Conflicts

As mentioned in section 3.3.4, bank conflicts occur in the Volta architecture when two threads try accessing addresses that are equal modulo 32. Bank conflicts have to be avoided in order to make use of the full performance of shared memory.

A thread in the *shared* access strategy stores or reads multiple shared memory slots (one for each neighbor). Assume we store neighborship information of each cell contiguously in shared memory. If the total number of required neighbors is very unfortunate, for example 8 neighbors per cell (equals an array of 32 bytes), then all threads end up (trying to) access the same memory bank simultaneously in the index lookup phase. This happens because the length of the neighborship array happens to align all top neighbors of cells into one bank, all left neighbors into another bank, etc. In the index lookup phase, all threads then attempt to load their top (left, ...) neighbor pointer at the same time.

In the *shared* access strategy we address this problem by adding padding to the shared memory storage of neighbors. Thus, neighbors are not stored contiguously, but have gaps in between them. The padding is chosen as the smallest value that is coprime with 32. This guarantees the least possible number of bank conflicts in this scenario; pointers to the same type of neighbor are spread out evenly across banks for different cells. In other words, all top neighbors are spread evenly across banks, all left neighbors

Block size	Variant	Runtime
$128 \times 1 \times 8$	Shared memory	**684μs**
	Warp broadcasting	737μs
$32 \times 1 \times 8$	Shared memory	**675μs**
	Warp broadcasting	693μs
$1 \times 1 \times 32$	Shared memory	13654μs
	Warp broadcasting	**12711μs**

Table 5.1: Median runtimes (20 runs) for the Laplace-of-Laplace benchmark (z-curves memory layout, pointer chasing, uncompressed) for three select block sizes. We observe that warp broadcasting is only faster than shared memory in the last block size configuration, which is the slowest overall. We therefore did not further investigate the use of warp broadcasting.

are spread evenly across banks (but may coincide banks with top neighbors, because those are not accessed at the same time instant), etc.

Warp Broadcasting

Loads and stores to shared memory produce some overhead. A more lightweight alternative provided by the Cuda architecture are so-called *warp-level primitives*. Those allow threads that are *within the same warp* to exchange data more efficiently – in a direct register-to-register fashion.

We briefly experimented with a variant of *shared* strategy, using warp broadcasting instead of shared memory. In this variant, all threads in the first lane (first thread of each warp) are the designated leader threads that do the actual neighbor lookup. Other threads in the same warp receive the leader thread's pointer by use of the `__shfl_sync()` method.

While the actual warp broadcasting is more lightweight than shared memory, it is also more limited: only threads within the same warp can exchange data. This sets restrictions on the kernel launch configuration; the threads in a block have to be allocated such that cells across different Z-levels fall within the same warp (otherwise no neighbor pointer sharing can take place). To make use of warp broadcasting, some additional calculations also need to be made (determining the lane ID, masks of active threads). For those reasons, warp broadcasting appeared to be slightly slower than the shared memory access variants in most realistic scenarios. In launch configurations with many threads in the X- or Y-dimension, warp broadcasting even was considerably slower (due to no sharing being possible within warp limits anymore). However, in launch configurations with many threads in the Z-dimension, warp broadcasting slightly outperformed shared memory. These launch configurations are rather slow overall (compared to other block sizes), though, and are therefore not very useful.

See table 5.1 for a comparison of the warp broadcasting approach to using shared

memory on an exemplary benchmark.

5.2.2 Index Variables + Z-loop

In this approach (*z-loop* for short), there are only $d_x \times d_y$ threads, i.e. only one thread per stack of cells in the X-Y-plane. Each thread calculates the results for all cells with equal X- and Y-coordinates in a loop over all Z. The indices of all required cells at the $z = 0$ level are stored in *index variables* before the start of the loop. The regularity of the grid in Z-direction enables us to update the index variables in each iteration of the loop by simply adding a Z-stride. There is no memory lookup into the neighborship table inside the loop. Thus, even in the unstructured grid case, memory lookups are only necessary once before the loop. This comes at the cost of possibly reduced occupancy, however, as there are fewer threads.

5.2.3 Index Variables + Sliced Z-loop

This variant addresses the issue of low occupancy in the above approach. It is practically identical to it, but splits the Z-loop up in smaller chunks. There are $d_x \times d_y \times \frac{d_z}{m}$ threads, where m is the number of output cells in the Z-direction a single thread should calculate. In the following benchmarks, we used $m = 8$.

Chapter 6

Benchmark Results

In this section, we seek to determine the overhead that indirect addressing imposes on unstructured grids compared to regular grid runtimes, and assess the effectiveness of our optimized approaches. We do this by implementing three real-world stencils, called *Laplace-of-Laplace (laplap)*, *horizontal diffusion (hdiff)*, and *fastwaves*, using our previously described methods for grid access and grid storage (see sections 5 and 4). We use key profiler metrics to better understand what causes the differences in performance.

Real-world stencil applications are applied in a multitude of different scenarios: problem domain sizes, precision requirements and properties of the stencils vary. In combination with the various grid access and grid storage methods we described, this leaves a large number of combinations to be tested. In our benchmarks, we assessed the performance impact of the following properties:

- Input/output conditions
  - Domain size (size of the input and output grids)
  - Required precision (single or double floating-point number precision)
- Stencil properties, such as the *depth and number of neighbor dependencies*, the *number of input and output fields* (number of different values stored in a cell) and the *arithmetic intensity*
- *Kernel launch configuration:* number of threads, blocks, and bytes of shared memory
- Implementation of stencil, i.e. the used *grid access scheme*
- Grid storage implementation properties
  - Memory layout and potential regular patterns in grid structure
  - Memory layout of neighborship storage
    * Depth of stored neighbor pointers (pointer *chasing* vs. *non-chasing*)
    * Compression of neighborship table

6.1 Setup and Benchmarked Stencils

The three benchmarked stencils represent real-world use cases in meteorology. Table 6.1 details the values of the main stencil characteristics (those listed in the second main bullet point above). Note the increasing complexity and number of fields required from the *laplap* to the *fastwaves* stencil. Also note that the *fastwaves* stencil, otherwise the most complex, does not require access to neighbors-of-neighbors.

The calculations performed by the stencils are defined in terms of a regular grid. All three stencils require neighborships as in regular grids (top, left, bottom, right, front, back). To measure the cost of using an unstructured grid, we represent the same regular grid in an unstructured fashion, using two memory layouts, *row-major* and *z-order-curves* (see section 4.2.3). This can be thought of as *emulating* an unstructured grid. Even if the grid structure is entirely regular, we store and access it in the same fashion as we would for a truly unstructured grid in the unstructured benchmarks. Retaining regularity even in the unstructured representation is required to maintain comparability of the results of the unstructured variants with the regular variants. We note that this approach to benchmarking might fail to capture some effects of real unstructured grids with actual irregularities.

For all stencils, we created several variants; for each of the grid access optimizations described in section 5 we implemented a separate Cuda kernel. During the implementation process, the results of each of those variants are verified against an unoptimized reference implementation of the stencil running on the CPU to assure the correctness of the results. In all variants, the implementation of the actual output calculations (as per stencil definition) remains completely unchanged. Only the portions of code responsible for index calculations/lookups (for neighborship access) are swapped out. This is achieved by using *C preprocessor macro definitions* in all places where indexing or neighborship access is required. The macro definitions are then redefined in each variant to either use direct or indirect addressing for the regular and unstructured grids, respectively (with further variations for non-chasing/chasing and compressed/uncompressed neighborship tables).

The stencils were benchmarked on an *Nvidia Tesla V100* GPU. This GPU implements the *Volta* architecture by Nvidia with compute capability 7.0. The reported run times are the median of 20 timed kernel runs. Before the first one of the timed kernel runs, an untimed *warm-up run* is performed (see the last paragraph section 3.3.5 for the reasoning behind this). The GPU is reset using an API call to `cudaDeviceReset()` between each run to flush caches and to free device memory from previous runs. The grid values and neighborship relations (for unstructured grids) are stored in unified global memory and are pre-transferred to the GPU and back to the host (`cudaMemPrefetchAsync()`) before respectively after the kernels are run. Memory transfer from host to device and back is, therefore, *not* part of the reported run times.

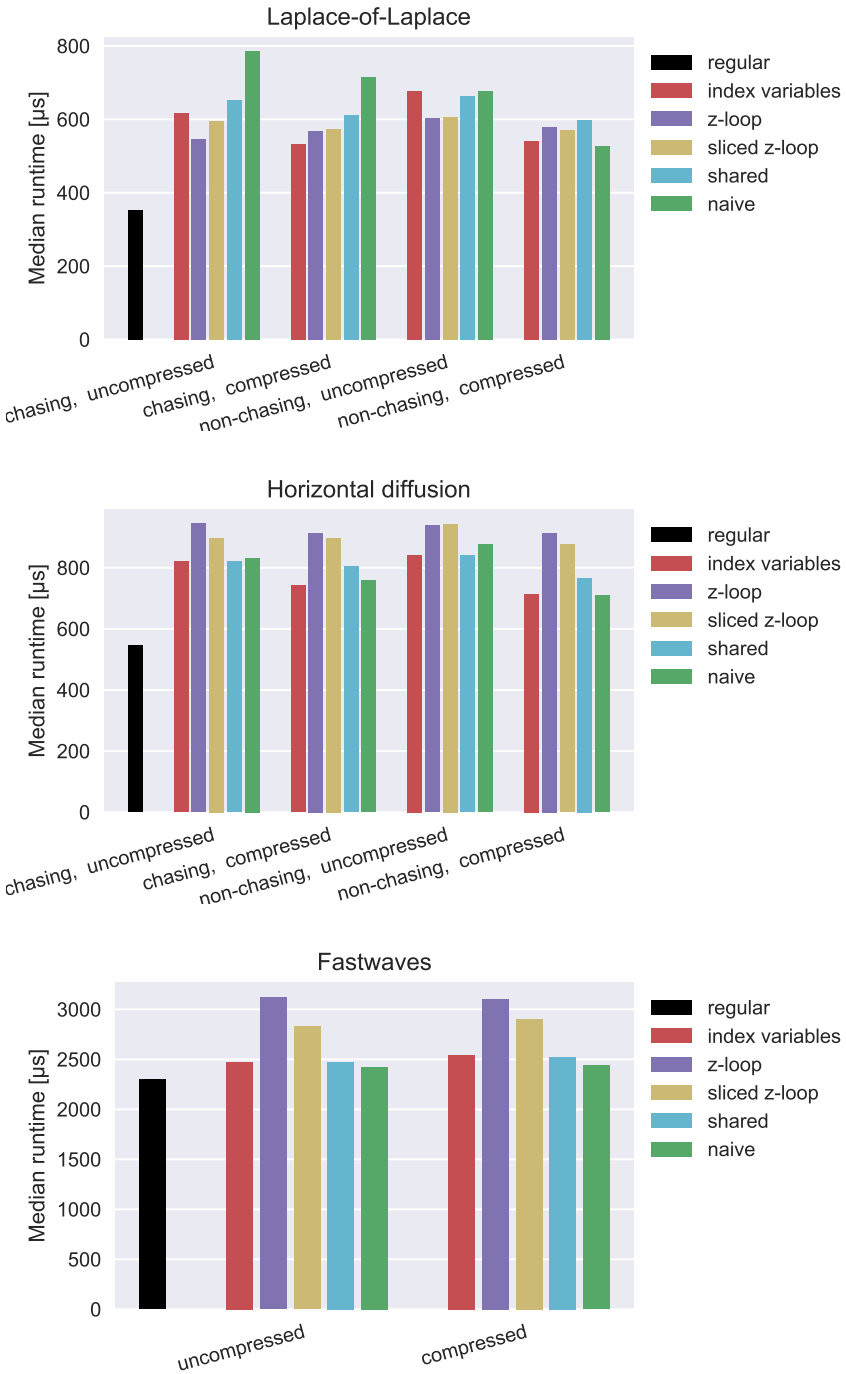

Figure 6.1: Overview of runtimes for all tested stencil implementations. The various grid access strategies implemented are represented as categories (colors of the bars), while the used data structures for the neighborship table (grid storage strategy) are shown as groups of bars on the X-axis. We report the median value across 20 runs, using the fastest respective launch configuration for each bar. The grid size is $512 \times 512 \times 64$ and the *z-curves* memory layout is used for the unstructured grids in this example. The input and output values are double-precision floating-point numbers.

Stencil	Neighborhood			Fields	Arithmetic Intensity
	X	Y	Z		
Laplace-of-Laplace *(laplap)*	(-2, 2)	(-2, 2)	0	1	Add., Mult., Sub.
Horizontal Diffusion *(hdiff)*	(-2, 2)	(-2, 2)	0	2	Add., Mult., Sub., Branches
Fastwaves	(0, 1)	(0, 1)	(-1, 1)	9	Add., Mult., Sub., Div., Branches

Table 6.1: Stencil characteristics. The neighborhood is given as an interval around the coordinates of a cell, i.e. $(-2, 2)$ means neighbors-of-neighbors are required for the stencil output calculation. The stencils are listed by their regular grid runtime in ascending order. Note that the *fastwaves* stencil is the only stencil with dependencies in the Z-dimension, has the highest arithmetic intensity, and accesses the biggest number of fields.

6.2 Overview of Results

Overall, we observed slowdowns around 50%, 30% and 5% compared to fastest regular grid performance for the *laplap*, *hdiff* and *fastwaves* stencils, respectively. Figure 6.1 shows an overview of the overhead for all tested combinations of storage and access strategy on a large grid of size $512 \times 512 \times 64$. Table 6.2 lists the fastest access strategy, the runtimes and the overheads for the three stencils in all possible storage configurations. For different domain sizes, see section 6.6. We chose to display the results for a *z-curve* layout, but results for the *row-major* benchmarks are similar. The fastest launch configuration is plotted for each bar; for a more detailed analysis of what block sizes are beneficial to the different access strategies, see section 6.5. A more in-depth analysis of the different storage and access strategies is given in sections 6.4 and 6.3 respectively.

Laplace-of-Laplace Stencil

Storage	(1)	(2)	Access Strategy	Runtime	Slowdown
regular grid	baseline		index variables	$353\mu s$	-
row-major	✓	✓	index variables	**512μs**	**45**%
row-major	-	✓	index variables	$515\mu s$	46%
row-major	✓	-	sliced z-loop	$582\mu s$	65%
row-major	-	-	sliced z-loop	$608\mu s$	72%
z-curves	-	✓	naive	**528μs**	**50**%
z-curves	✓	✓	index variables	$532\mu s$	51%
z-curves	✓	-	z-loop	$546\mu s$	55%
z-curves	-	-	z-loop	$603\mu s$	71%

Horizontal Diffusion Stencil

Storage	(1)	(2)	Access Strategy	Runtime	Slowdown
regular grid	baseline		index variables	$546\mu s$	-
row-major	-	✓	naive	**683μs**	**25**%
row-major	✓	✓	index variables	$731\mu s$	34%
row-major	✓	-	index variables	$804\mu s$	47%
			shared (tie)	$804\mu s$	47%
row-major	-	-	naive	$845\mu s$	55%
z-curves	-	✓	naive	**710μs**	**30**%
z-curves	✓	✓	index variables	$741\mu s$	36%
z-curves	✓	-	index variables	$820\mu s$	50%
z-curves	-	-	shared	$841\mu s$	54%

Fastwaves Stencil

Storage	(1)	(2)	Access Strategy	Runtime	Slowdown
regular grid	baseline		naive	$2298\mu s$	-
row-major		-	naive	**2400μs**	**4.4**%
row-major		✓	naive	$2433\mu s$	5.9%
z-curves		-	naive	**2426μs**	**5.6**%
z-curves		✓	naive	$2438\mu s$	6.1%

(1) **Pointer Chasing?** (A checkmark means that only directly adjacent neighbors are stored and pointer chasing occurs for neighbor-of-neighbor access. No checkmark means neighbors-of-neighbors are explicitly stored in the neighborship tables.)

(2) **Compressed?** (A checkmark means the neighborship table was compressed by having all cells with identical relative neighbor offsets share their entries. This requires an additional lookup.)

Table 6.2: Fastest access strategy for all grid storage options. *Z-curves* and *row-major* refer to the memory layout of the *values* of the grid, whereas *chasing/non-chasing* and *compressed/uncompressed* are properties of the neighborship table. The runtimes are the median of 20 runs using the fastest respective launch configuration per benchmark. The slowdown is relative to the fastest regular grid implementation. The stencils were executed on a $512 \times 512 \times 64$-sized grid of double-precision floating-point numbers.

Metric	*naive*	*idxvar*
Run time	**2438μs**	2543μs
Global load transactions	126,264,460	124,235,165
L1 transactions	38,077,396	34,593,863
L2 transactions	56,635,611	57,264,861
Device Memory transactions	36,825,619	37,327,310
Executed Instructions Per Cycle	**0.522**	0.497

Table 6.3: Selection of metrics for the *fastwaves* stencil run on a $512 \times 512 \times 64$-sized unstructured grid (*z-curves* memory layout with *double* precision, *compressed* and *chasing* neighborship table) with $32 \times 1 \times 8$ threads, which is the fastest block size for both displayed access variants. The *naive* implementation is faster, even though it redoes the index lookups in the neighborship tables. The total global transaction count shows that the *idxvar* variant does reduce the number of lookups performed. However, the cache transactions (L1 and L2) indicate that the *naive* variant keeps cache contents fresher, which results in more cache hits. This is evidenced by the lower number of actual device memory reads in the naive approach, even though more memory is requested than in the *idxvar* approach.

6.3 Effect of Access Strategy in Stencil Implementation

Before benchmarking, all stencils had to be reimplemented in a way that supports unstructured grids. In this section, we explore the performance implications of the access variants described in section 5, starting from the *naive* variant and moving to the contrived optimizations.

6.3.1 *Naive* and *Idxvar* Access Strategies

The *naive* strategy was the first we implemented. At each neighbor access in the reference regular-grid-stencil, an indirect lookup into the neighborship table is inserted for the unstructured variant. As such, this implementation performs repeated redundant lookups for the same neighborship table entries if the same neighbor is accessed multiple times in one thread. Our initial tests occurred on a $512 \times 512 \times 64$-sized grid (*uncompressed*, *chasing*, *row-major*), and using *naive* access strategy, we observed slowdowns of 107%, 54% and 4.4% (compared to the fastest regular implementations) for the *laplap*, *hdiff* and *fastwaves* stencils, respectively.

We then moved to a straightforward optimization, the *idxvar* variant. This approach tries to eliminate redundant lookups by storing the needed neighborship indices in temporary variables at the start of the kernel execution. In the $512 \times 512 \times 64$ grid (*uncompressed*, *chasing*, *row-major*), this reduced the overheads to 77% and 47% for the *laplap* and *hdiff* stencils.

Comparing the *naive* and *idxvar* strategies to the more optimized variants (discussed below), we observed that caching of neighborship table entries plays a paramount role

in unstructured stencil performance. We discovered that neighborship table lookups become fairly efficient once cached. Due to their simplicity and efficient cache use (at the right block sizes), the *naive* and *idxvar* access strategies are thus generally among the fastest for all three stencils, even though (or because) they do not implement any advanced intricacies.

Surprisingly, for the grid described above, using the *idxvar* strategy with the *fastwaves* increased the overhead to 6.8% (from 4.4% in the *naive* approach). As we later experimented with different grid storage strategies, such as *non-chasing*, *compressed* grids, the *idxvar* strategy also turned out to not always be beneficial.

There are two likely explanations for the occasional advantage of the *naive* variant: First, reading the same memory location a second time becomes almost as cheap as register access after it has been read for the first time, as it will be held in the L1 cache. Reaccessing the same memory location increases its chances of remaining in cache. As such, when a different thread requires the same memory location from the neighborship table, it is more likely to be in cache in the *naive* variant due to its larger number of accesses.

Second, the *naive* approach has better *instruction-level parallelism*. Because the index variables are assigned at the start of an *idxvar* thread, all neighborship table loads must be executed before anything else. Output calculations may only start once *all* required neighborship pointers are loaded. In the *naive* approach, on the other hand, neighborship pointer reads are not gathered at the beginning of the kernel. Instead, they are (re-)loaded at every point they are required in calculations. Thus, after one (or a few) neighbors have been loaded, useful intermediate calculations may already be performed, thus hiding some latency. Table 6.3 details profiler metrics that support these claims.

6.3.2 *Z-loop* and *Z-loop-sliced* Access Strategies

The next obvious step was to try to make use of the Z-regularity of the grid. By loading the neighborship pointers once and reusing them for multiple Z-levels, we hoped for further improvements in runtime.

However, for the *hdiff* and *fastwaves* stencils, the two *z-loop* access strategies perform noticeably slower than the other available access strategies, across all tested grid storage configurations. The same is the case for the *laplap* stencil for *compressed* grids. The one exception is the *laplap* stencil on an *uncompressed* grid: In this case, the *z-loop* access strategy is the fastest.

The hoped-for advantage of the two *z-loop* access strategies is the reduced number of required neighborship table reads. Indeed, the number of reads to device memory is greatly reduced using this access strategy in all stencils (more than halved for *laplap*, 56% for *hdiff*). In the case of the computationally simple *laplap* stencil on an uncompressed grid, this reduced number of reads pays its dividends; the *z-loop* access strategy is the fastest for this specific benchmark.

For other stencils (and the *laplap* stencil in compressed grids), however, using the *z-loop* access strategy is slower than all the alternatives. We assume that this is due

Stencil	Access	achieved_ occupancy	ipc	dram_read_ transactions
laplap	*idxvar*	0.947077	0.812799	11,946,136
	z-loop	0.407226	0.515881	4,715,455
	z-loop-sliced	0.421727	0.638366	4,729,444
hdiff	*idxvar*	0.935448	0.793701	16,606,211
	z-loop	0.363111	0.481847	9,330,226
	z-loop-sliced	0.301747	0.600179	9,354,882

Table 6.4: Relevant metrics for the comparison of the *z-loop* and *idxvar* access strategies, on a $512 \times 512 \times 64$-sized grid (*z-curves* memory layout with *double* precision, *uncompressed* and *chasing* neighborship table) with $64 \times 2 \times 1$ threads. Observe that the *z-loop* strategies effectively reuse neighborship pointers, leading to a lower number of device memory transactions, but suffer from a much lower occupancy and fail to use the full parallel computing capabilities of the GPU.

to the much lower (compared to other access strategies) occupancy of this loop-based access strategy. In the more computationally complex stencils, many latencies in the result computations may occur. To hide those latencies, a large number of threads are required. In the *z-loop* access strategy, sequential processing of all elements with the same X- and Y-coordinates is prescribed by the code – hindering parallel execution on the GPU. Presumably, this is less of an issue in the *laplap* stencil due to its more simplistic result computation and fewer fields, which leads to fewer stalls in need of latency hiding.

In the *z-loop-sliced* variant, we tried to address some of the low-occupancy issues by splitting the loop into smaller, parallelizable chunks. This approach tries to combine the best of both worlds: Reuse of neighborship table reads in a loop and more parallelism thanks to more threads. While the performance does improve in comparison with the *z-loop* access strategy (confirming our theory that occupancy is indeed the bottleneck), it still falls short of the other access strategies due to the added overhead of managing a loop (which requires additional registers and a branch).

Table 6.4 shows the relevant metrics of the described observations for an exemplary benchmark run.

6.3.3 *Shared* Access Strategy

As both the *z-loop* and *z-loop-sliced* variants had issues with occupancy, we explored the use of shared memory to pass neighborship information among cells that share the same neighbor offsets. This allows exploiting the Z-regularity of the grid while maintaining a large number of threads.

The performance of the *shared* variant varies strongly depending on the type of storage of the neighborship tables (i.e. storage strategy) – specifically, whether *compression*

Metric	*idxvar*	*shared*
`tex_ cache_ transactions`	29,962,684	26,589,268
`shared_ load_ transactions`	0	5,666,573
`dram_ read_ transactions`	9,362,526	9,355,089

Table 6.5: Number of transactions for the *idxvar* and *shared* access strategies at various levels of the memory hierarchy for a benchmark of the *hdiff* stencil (grid of size $512 \times 512 \times 64$, *z-curves* memory layout with *double* precision, *uncompressed* and *chasing* neighborship table) at $64 \times 1 \times 8$ threads per block (optimal for both access strategies). Many L1 cache hits (`tex_ cache_ transactions`) in the *idxvar* strategy are simply shifted to equally performant shared memory hits (`shared_ load_ transactions`) in the *shared* strategy. The number of reads encountered at device memory is similar, further indicating that shared memory usage simply serves as an explicit, manually managed cache, taking the same role as the L1 cache in the *idxvar* access strategy.

for the neighborship tables is used or not.

In all scenarios with *uncompressed* neighborship storage, the *shared* access strategy performs almost identically to the *idxvar* approach. A very slight overhead is observed for the *shared* access strategy due to the required thread synchronizations. In this variant, the neighborship pointers loaded into shared memory appear to take the same role as the L1 cache. Thus, the *shared* access strategy can be viewed as maintaining an *explicitly managed cache*. In fact, as mentioned in section 3.3.4, shared memory and the L1 cache share the same physical memory. Table 6.5 further evidences how the L1 cache is simply shifted to shared memory when using this scheme.

As we later discovered in our experiments with different grid storage strategies, grids stored using a *z-curves*, *non-chasing* and *uncompressed* approach stand out in the *shared* access strategy, as this is the only configuration in which the *idxvar* scheme is slightly outperformed. With the aforementioned storage properties, explicit management of the unified L1/shared memory provides a minor benefit. This is probably due to the very large number of neighbor pointers in this storage strategy.

When using compressed neighborship tables, the *shared* access strategy is noticeably slower than the *idxvar* strategy. As there are relatively few neighborship pointers in compressed tables, those remain in *L1* cache in the *idxvar* variant permanently. In the *shared* variant, those same (few) pointers must be explicitly (re-)loaded into shared memory for each thread block, which is less efficient.

6.3.4 Summary

Figure 6.1 gives an overview of the possible storage/access-combinations and their performance. Table 6.2 lists the fastest access strategy for all possible grid storage combinations.

No access strategy clearly dominates in all situations, but the *idxvar* access strategy

is often a good choice. Which implementation is fastest depends on the combination with the grid storage strategy, as well as on the stencil properties. Surprisingly, the *naive* access strategy is competitive. It often performs similarly as the *idxvar* strategy. In situations where a lot of pointer chasing occurs (*chasing* grids and *compressed* grids), using the *idxvar* strategy is advantageous. We observed that the *shared* access strategy behaves very similarly to the *idxvar* strategy. We assume this is due to caching in the *idxvar* strategy having the same effect as the explicitly used shared memory. The *z-loop* and *sliced z-loop* access strategies only gave an advantage in one configuration, namely for the *laplap* stencil in grids stored in uncompressed (both chasing and non-chasing) fashion.

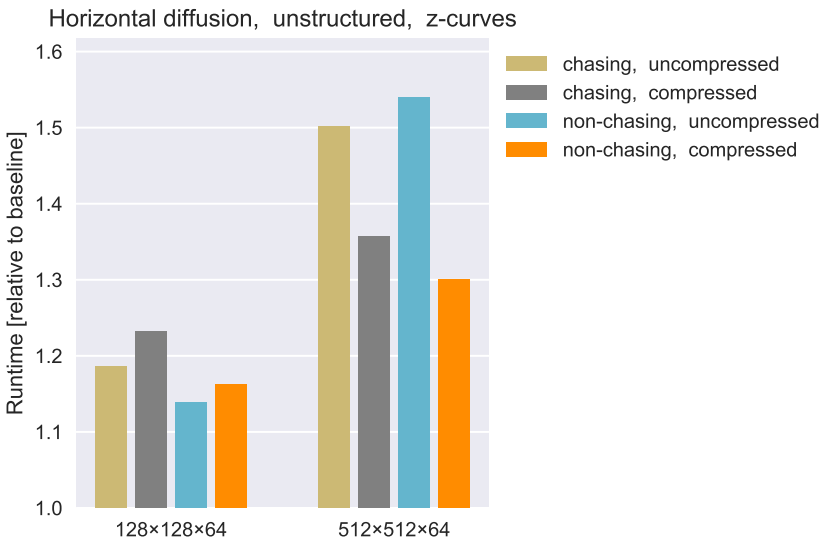

Figure 6.2: Effect of neighborship table properties on runtime depending on problem size. This shows the slowdowns relative to the fastest regular grid implementations for the *hdiff* stencil on a small and a large grid (both *z-curves* memory layout with *double* precision). The respective fastest access variant is plotted for each bar, which is one of *idxvar*, *naive*, and *shared* for the shown benchmarks. This example for the *hdiff* stencil is representative of the other two benchmarks as well, where the slowdowns follow a similar pattern. (Note that in the *fastwaves* stencil, pointer chasing is not applicable as there only direct neighbors are accessed.) Baseline: fastest regular grid implementation.

6.4 Effect of Grid Storage

Having implemented the stencils using the various grid access strategies, we then went on to explore optimization possibilities in the grid storage, namely how the neighborship tables could be stored in a way to enable faster accesses. The different approaches are described in section 4. We evaluated four different neighborship table structures, resulting from the combinations of the following two properties: First, the depth of the neighborship table, i.e. whether only neighbors (*chasing* variant) or also neighbors-of-neighbors (*non-chasing* variant) were stored. Second, whether the neighborship table was *compressed* or *uncompressed*.

6.4.1 Pointer Chasing vs. Neighbor-of-Neighbor Storage

Two of the three benchmarked stencils, *Laplace-of-Laplace* and *horizontal diffusion*, access neighbors beyond the directly face-touching cells (neighbors-of-neighbors). For those stencils, we first assessed the performance when only direct neighbors are explicitly stored. This approach requires *pointer chasing*: To access the neighbor of a neigh-

Size	Chasing?	tex_ cache_ hit_ rate	l2_tex_ hit_ rate	stall_ memory_ dependency	Runtime
$128 \times 128 \times 64$	✓	73%	68%	59%	$41\mu s$
$128 \times 128 \times 64$	-	67%	72%	54%	$40\mu s$
$512 \times 512 \times 64$	✓	68%	684%	74%	$841\mu s$
$512 \times 512 \times 64$	-	63%	37%	60%	$1042\mu s$

Table 6.6: Selected metrics for runs of the *laplap* stencil on unstructured grids of different sizes (all of them in *z-curves* memory layout with *double* precision, *uncompressed* neighborship table), both with and without pointer chasing. The *idxvar* access strategy was used with a fixed launch configuration of $64 \times 4 \times 2$ threads.

bor, two sequential lookups to the neighborship table become necessary. As the second lookup can only be started once the first one has completed, a higher latency occurs in this variant. Therefore, we then also tested variants in which the neighbors-of-neighbors were explicitly stored in memory, reducing the number of required memory lookups for these types of accesses from two to one. We call this *non-chasing* neighborship storage. This approach comes at the cost of a higher memory footprint.

Generally, we observed the following trend: If the grid is small or compressed, pointer chasing becomes an issue, and thus the *non-chasing* variants provide an advantage. In larger *uncompressed* grids, the latency of the two lookups required for neighbors-of-neighbor access can be effectively hidden. This is visible in figure 6.2, which outlines the differences of the storage approaches for the fastest respective variant of the *hdiff* stencil on grids of different sizes.

In all cases, a higher total number of device memory and L2 cache reads was observed for the *non-chasing* variants. This is to be expected, as *non-chasing* variants need to store and read more information from a larger number of different addresses. When pointers to neighbors-of-neighbors are explicitly stored, the L1 cache hit rate lowers by around 6% in both large and small grids. In large grids, the L2 cache hit rate also drops dramatically; for smaller grids, it remains unaffected.

In large *non-chasing* grids, the additional neighbor-of-neighbor entries affect cache locality more negatively than in small grids. This is because larger grids have more neighborship table entries, while the cache size remains constant – consequently, some data has to be evicted from the cache. This does not happen as often in a smaller grid. The variants that perform pointer chasing experience a higher percentage of stalls caused by memory dependencies; neighbor-of-neighbor reads block progress as the second lookup waits for the result of the first one.

We conclude that non-chasing storage is beneficial to small grids, where it is effective in reducing latency, and unfavorable for larger grids due to the caused cache evictions (especially in the L2 cache) increasing the latency of neighbor accesses. Table 6.6 shows

Compressed?	Chasing?	Storage	# Entries	Ratio*	Freq.†
-	✓	both	1,048,576	1	< 1%
-	-	both	3,145,728	1	< 1%
✓	✓	row-major	2054	0.00078	97.7%
✓	✓	z-curves	2435	0.00093	20.3%
✓	-	row-major	4093	0.00156	96.9%
✓	-	z-curves	5299	0.00202	8.1%

*Ratio of number of compressed neighborship table entries to number of uncompressed neighborship table entries. †Frequency of the most common entry in the neighborship table.

Table 6.7: Properties of neighborship table before and after compression for $512 \times 512 \times 64$-sized grids with different memory layouts (*z-curves* and *row-major*), for both *chasing* and *non-chasing* implementations.

an overview of the metrics discussed in this paragraph for an exemplary stencil run.

6.4.2 Effect of Neighborship Table Compression

As most unstructured grids have irregularities only in few places, much of the information in the neighborship tables is redundant. Moving forward, we wanted to investigate whether this redundancy could be made use of to further reduce unstructured stencil runtimes. In section 4.2.2, we described a very simple compression scheme for the neighborship tables motivated by this idea. In this section, we elaborate on its performance implications.

Using *compressed* neighborship tables, we saw a reduction in runtimes for most benchmarks on large $512 \times 512 \times 64$-sized grids. The highest relative speedups were attained for the *laplap* stencil, which ran 23% faster. In the *hdiff* stencil, we achieved a speedup of 21%. In contrast, the *fastwaves* stencil was slightly slowed down by the added overhead of compression and ran 3% slower.

In smaller grids ($128 \times 128 \times 64$ and $64 \times 64 \times 64$), the additional level of indirection introduced by our compression scheme also mostly led to a deterioration in run times. The interplay of grid size and compression can also be seen in figure 6.2.

In the following two subsections we will further elaborate on the efficacy of our compression scheme in terms of (reduced) memory requirements and detail the effects on parallel execution in an attempt to explain the observed runtimes.

Distribution of Neighborship Table Entries After Compression

Using the compression scheme, cells with the same neighbor *patterns* can share neighborship table entries. Table 6.7 and figure 6.3 show the distribution of unique patterns on a $512 \times 512 \times 64$-sized grid for the two memory layouts benchmarked (*row-major*, *z-curves*). In *row-major* grids, the number of distinct patterns is lowest. In this case,

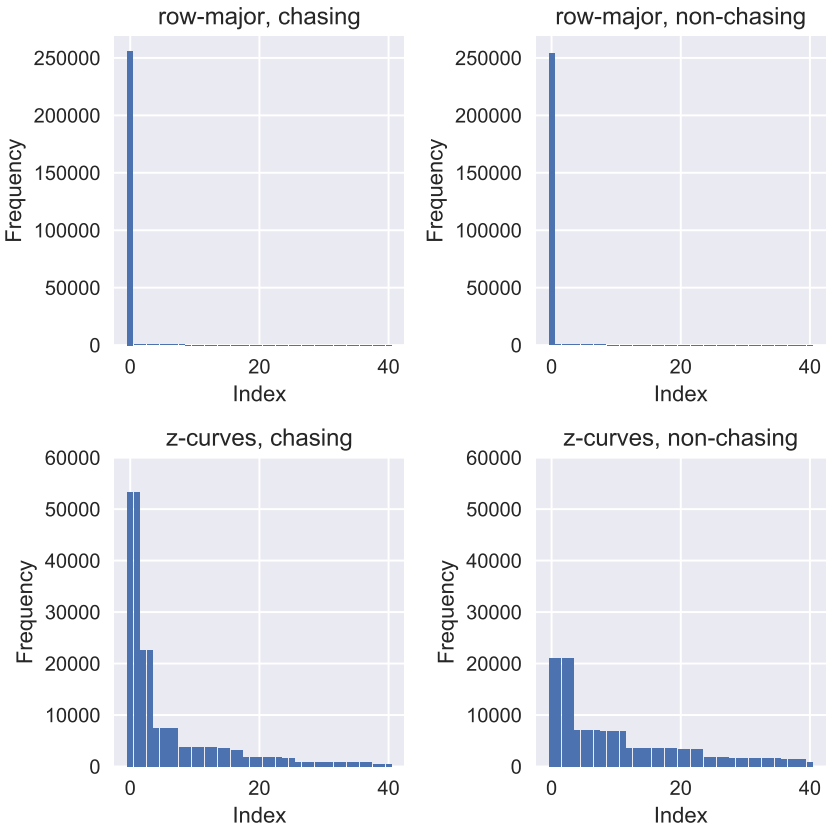

Figure 6.3: Distribution of the first 40 neighborship table entries of a $512 \times 512 \times 64$-sized grid for both benchmarked memory layouts and *chasing* and *non-chasing* implementations. The y-axis shows how many cells share the same relative neighborship offsets. The x-axis is the index in the neighborship table. A sharper peak means more effective compression. As an example, the first two entries in the *chasing* neighborship table of a grid with *z-curves* memory layout are shared by 53340 cells. Compare this to the uncompressed case (not plotted), where each entry is shared across Z-levels between cells with equal X- and Y-coordinates, i.e. 64 times in this example. *Note the different scale in the bottom row!*

Size	Comp.?	tex_ cache_ transactions	l2_tex_ read_ transactions	dram_ read_ transactions	gld_ efficiency	Runtime
$64 \times 64 \times 64$	-	$466,815$	$273,384$	$57,658$	85%	**18μs**
$64 \times 64 \times 64$	✓	$\mathbf{528,064}$	$415,834$	$51,676$	74%	$19\mu s$
$128 \times 128 \times 64$	-	$1,925,279$	$1,468,085$	$508,748$	87%	**51μs**
$128 \times 128 \times 64$	✓	$\mathbf{2,071,128}$	$1,420,441$	$508,208$	75%	$53\mu s$
$512 \times 512 \times 64$	-	$29,980,144$	$22,391,916$	$9,392,553$	91%	$820\mu s$
$512 \times 512 \times 64$	✓	$\mathbf{31,760,437}$	$20,805,522$	$9,402,833$	77%	**741μs**

Table 6.8: Runtimes and relevant metrics comparing executions of *hdiff* stencil on *compressed* and *uncompressed* grids (*z-curves* memory layout with *double* precision, *chasing* neighborship table), implemented using the *idxvar* access strategy. The optimal (fastest) launch configuration block size was chosen for each row, which is different for *uncompressed* and *compressed* grids. Note the higher L1 cache traffic (`tex_ read_ transactions`) and lower global load efficiencies (bad coalescing, `gld_ efficiency`) in all compressed variants.

only the elements in the halo have different neighborship pointers; all others share the top entry. Using a *z-curves* layout, the distribution of neighborship table accesses is flatter. However, there is still a fairly high concentration of several frequently occurring patterns.

Note that a completely random grid would result in a very even distribution. In such a scenario, compression would not be able to reduce the number of entries meaningfully.

Runtime Effects of Compression on Stencil Execution

The motivating idea behind implementing compression for the neighborship tables was an anticipated improvement to the caching of frequently used neighborship table entries. As we observed, neighborship information is distributed favorably and concentrates around a few frequently used entries (see table 6.7 in the previous section). By using compression, we hoped that these entries remain in the L1 cache, and thus reduce the number of expensive reads to the L2 cache (or even device memory) required for neighborship index lookups. Yet, because cells with identical neighborship patterns share neighborship table entries in the compressed scheme, a mapping from cell index to pattern index must be implemented. This mapping introduces an additional lookup, whose latency may offset the advantage of cached neighborship table entries.

Encouragingly, the desired improvements to caching seem to take place. We observed that more frequent use of the L1 cache occurs across all benchmarks using compressed storage (i.e. all memory layouts, all access strategies, and all grid sizes). We measured a larger number of L1 cache transactions and a higher throughput from the L1 cache to the processor. In total, more data was transferred from the L1 cache and less data had to be loaded from the L2 cache and device memory. The measures in table 6.8 confirm

that the anticipated caching of neighborship table entries does indeed happen.

Yet, as noted in the introduction to this section, compression gave an advantage only on large grids and only for the simpler *laplap* and *hdiff* stencils. We suppose that this limited advantage is due to the following several drawbacks caused by the additional indirection added, i.e. the required pattern lookup:

First, the pattern lookup leads to uncoalesced memory accesses. This can not easily be avoided, since the pattern may map neighborship lookups to any address.

Second, the added level of indirection causes latency which cannot easily be hidden. A cell's results may only be computed once the values of its neighbors have been loaded (latency of neighborship lookup). To do this, the neighbors' indices must be obtained, which in turn cannot be done before the pattern lookup has completed in the compressed scheme (latency of pattern lookup). In the case of a grid that stores only directly adjacent neighbors (i.e. *chasing* grid), accessing neighbors-of-neighbors leads to an even longer chain of pointers to be followed.

Third, in small grids, the supposed advantage of compressed grids may take place even without compression, because even uncompressed neighborship tables fit into the caches. Thus, we pay the overhead of compression with no gains.

Lastly, if a complex stencil requires a large number of fields, accesses to the values of these fields can contend the limited L1 cache space. This causes neighborship table entries to be evicted from the cache, nullifying the purported advantage of compression. Employing an explicitly managed cache, as in the *shared* access strategy, may alleviate this issue. Though, in the *fastwaves* stencil, we had no success with this approach.

In conclusion, the added overheads of the described compression scheme diminish its usefulness in several scenarios. Still, in large grids with medium-complexity stencils, compression can give considerable advantages. We reason that this advantage does indeed come from the better L1 cache locality and the reduced L2 and device memory traffic.

6.4.3 Summary

We observed that *pointer chasing* does inhibit performance in all grids with few entries in the neighborship table, specifically small and medium-sized grids, as well as large grids using neighborship table compression. In these cases, explicitly storing neighbors-of-neighbors in the neighborship tables improves runtimes, but not as strongly as one might anticipate. For large grids with an uncompressed neighborship table, performing pointer *chasing* in stencils is faster than *non-chasing* variants (probably due to the large number of neighborship table entries a *non-chasing* variant requires in this scenario).

Experimenting with a simple compression scheme, we determined the following: In large grids, for both the *hdiff* and *laplap* stencils, compressing the neighborship table provides a significant benefit. However, the additional memory lookup required and unavoidable uncoalesced accesses to the neighborship table limit the possible advantage. In the *fastwaves* stencil, the additional overhead of compression is counter-productive and slows execution down. The same is the case for small grids.

6.5 Optimal Block Size

One of the main determining factors for the performance of a kernel is the launch configuration, which includes grid size (number of blocks) and block size (number of threads, described in section 3.3.3). As we implemented and evaluated our grid storage and grid access strategies, we consistently ran our benchmarks across a large range of launch configurations. In the previous sections, we reported best-case runtimes using the respective optimal launch configuration. This optimal configuration varies heavily depending on the implementation, as we will detail in this section.

Cuda provides the option to specify block sizes in three dimensions, providing each thread with an X-, Y- and Z-index. In our benchmarks, we have tested all possible combinations of block sizes in steps of powers of two from 32 up to 512. In order to cover the entire problem domain (which remains constant in size), a decrease in the number of threads is always accompanied by an increase in the number of blocks (grid size) and vice versa. Owing to the way we map thread indices onto memory indices (X thread index maps onto memory index directly), thread sizes in the X-dimension of less than 32 are highly inefficient, as they lead to non-coalescing memory accesses. The X-dimension of the benchmarked block sizes is therefore always at least 32.

6.5.1 Overview

The optimal *total number* of threads depends mostly on the grid access implementation employed. In general, the same patterns can be observed for all three tested stencils and all tested grid memory storage implementations: When using the *naive*, *idxvar* or *shared* grid access strategies, it is best to have a high number of threads in total ($256-512$). The *z-loop* and *z-loop-sliced* thrive on a lower number of threads due to occupancy concerns. An exemplary overview of the runtimes as a function of total block size is given in figure 6.4.

The best *shape* of the block size vector changes with the used grid storage strategy. Using *uncompressed* neighborship tables, it is best for the *naive*, *idxvar* and *shared* access strategies to have 32 or 64 threads in X-dimension and the rest in Z-dimension. Those implementations profit of more Z-threads because of the regularity of the grid in the Z-dimension. The *non-chasing* variants profit even more of additional threads in the Z-dimension. This is because, in non-chasing variants, more neighborships are stored in the table; cache entries are thus evicted faster if many different X- and Y-coordinates are accessed. In *compressed* neighborship tables, on the other hand, additional threads in the Z-dimension are of no use to any of the access strategies – the greatly reduced number of neighborship table entries after compression remains in cache independent of the Z-block-size. See figure 6.5 for a comparison of how changes in the Z-dimension of the block size vector translate to runtime changes. Whether the grid is stored as *row-major* or using *z-order* curves does not strongly impact the required block sizes.

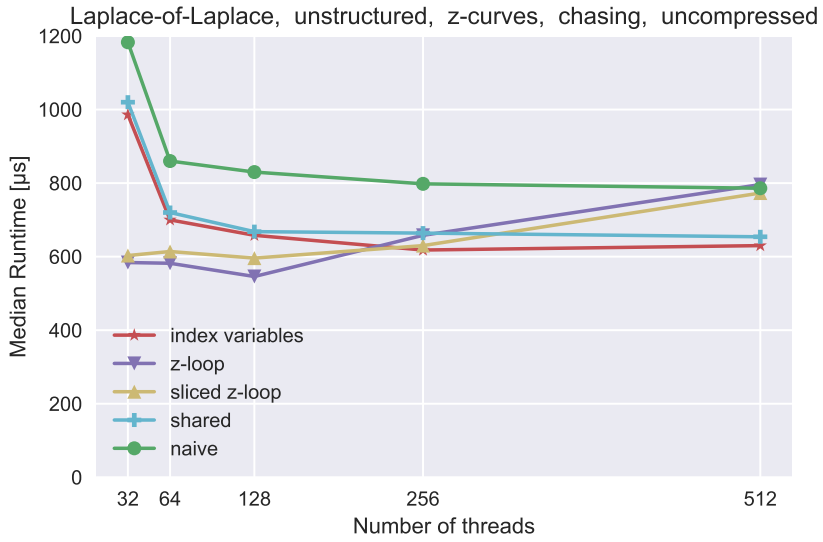

Figure 6.4: Impact of total block size (product of X-, Y- and Z-blocksize) on the runtime for different implementations of the *laplap* stencil on a $512 \times 512 \times 64$-sized grid (*z-curves* memory layout with *double* precision, *chasing* and *uncompressed* neighborship table). The low-occupancy *z-loop* and *z-loop-sliced* access strategies do not profit from a larger number of threads, contrary to the other variants. Results for other stencils and variants are of similar shape.

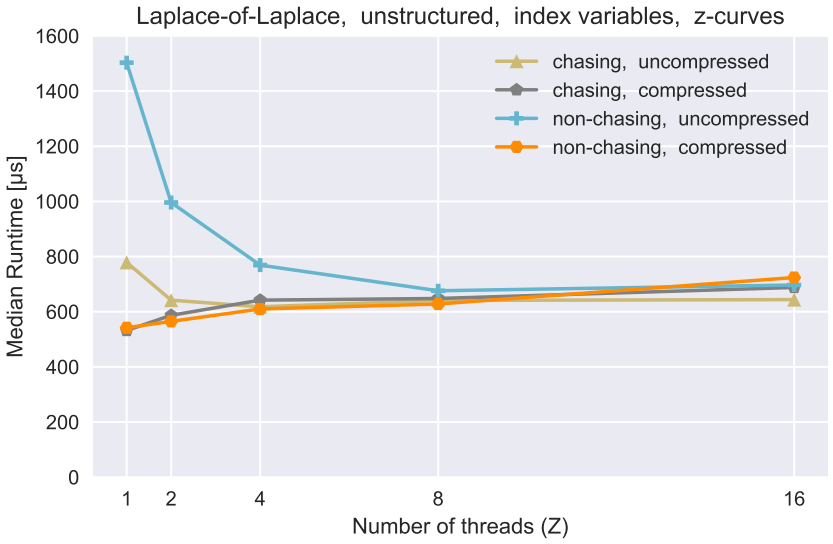

Figure 6.5: Fastest runtimes of all configurations for fixed numbers of threads in the Z-dimension (other dimensions free) for the *laplap* stencil on a $512 \times 512 \times 64$-sized domain (*row-major* memory layout with *double* precision) for *compressed* and *uncompressed* variants, with and without pointer *chasing*. Note how the Z-regularity of the grid makes it advantageous to have a larger number of threads in the Z-dimension, as this leads to caching of neighborship table entries, but only if the neighborship table is *uncompressed*.

Blocksize	tex_cache_hit_rate	l2_tex_hit_rate
$32 \times 1 \times 1$	44.39%	67.69%
$32 \times 1 \times 2$	57.96%	68.12%
$64 \times 1 \times 2$	66.21%	60.36%
$64 \times 1 \times 4$	71.72%	60.81%

Table 6.9: Cache hit rates for different block sizes for the *laplap* stencil, implemented using a *idxvar* grid access strategy on a $512 \times 512 \times 64$-sized domain (*z-curves* memory layout with *double* precision, *chasing* and *uncompressed* neighborship table).

6.5.2 Optimal Block Sizes per Access Strategy

In this section, we further elaborate on the optimal block sizes for each access strategy and aim to explain (using profiler metrics) why certain shapes of thread blocks are beneficial.

Naive, *Idxvar* and *Shared* Grid Access Strategies

In order to always cover the entire problem domain, the number of blocks b in each dimension in the *naive*, *idxvar* and *shared* grid access strategies is chosen as a function on the problem size d and the chosen number of threads t as follows:

$$b = \left(\left\lceil \frac{d_x}{t_x} \right\rceil \quad \left\lceil \frac{d_y}{t_y} \right\rceil \quad \left\lceil \frac{d_z}{t_z} \right\rceil \right)^\top$$

For the *naive*, *idxvar* and *shared* implementations, a high total number of threads per block is beneficial to performance. For grids using an *uncompressed* neighborship table, increasing the number of threads in the Z-dimension especially improves performance.

For the *naive* and *idxvar* strategies (in grids using uncompressed neighborship tables), the advantage of multiple Z-threads can be explained through the regularity of the grid in the Z-dimension. Threads operating on cells with identical X- and Y-coordinates access the neighborship tables at the *same index*. When multiple threads operate on different Z-levels, caching of these shared neighborship table entries becomes very effective. Evidence for this presumed reason for the speedup is given by two facts: One, of all block size combinations tested, the ones with more threads in the Z-dimension perform best. Two, the Nvidia profiler reports higher cache hit rates if the number of threads in Z-dimension is increased. As an example of this for the *idxvar* access strategy, see table 6.9.

These effects are strongest when using *non-chasing* grid storage. As this type of storage leads to a larger number of entries in the neighborship table, entries may be evicted from the cache more quickly. More threads in the Z-dimension, which access the same neighborship table entries, prevent this by keeping the relevant neighborship table entries "fresh."

At the opposite end of the spectrum lie the grids stored using *compressed* neighborship tables. These tables are much smaller, and the same few entries are accessed in

Variant	Blocksize	Runtime	achieved_ occupancy	issue_slot_ utilization
idxvar	512	783μs	85%	19%
idxvar	128	844μs	**92**%	18%
idxvar	32	986μs	48%	16%
z-loop	512	796μs	25%	10%
z-loop	128	**546μs**	42%	13%
z-loop	32	584μs	41%	12%

Table 6.10: Runtimes and occupancy metrics of *idxvar* and *z-loop* implementations of the *laplap* stencil on an unstructured grid (size $512 \times 512 \times 64$, *z-curves* memory layout with *double* precision, *chasing* and *uncompressed* neighborship table) for a selection of block sizes. The following block shapes were used: $512 = 256 \times 2 \times 1$, $128 = 64 \times 2 \times 1$ and $32 = 32 \times 1 \times 1$. This table highlights the occupancy issues the *z-loop* variant faces with too large block sizes. Note how the occupancy drops with an increased number of threads, and how the runtime increases with it.

almost all threads. Because of this, neighborship table entries remain in cache no matter what the shape of the block size vector is, and additional threads in the Z-dimension bring no benefit. Cache locality for the accessed values is more important here; depending on the used layout for the values (*z-curves* or *row-major*) this means adding additional threads in X (*z-curves*, spatial locality given), or X and Y (*row-major*, adding another thread in Y gives some spatial locality).

By the design of the *shared* access strategy, a speedup is supposed to be attained with a larger number of Z-threads; if there are more threads in Z-dimension within a block, more sharing of neighborship relations through shared memory can take place. These implementations indeed profit from more threads through shared memory in much the same way as *naive* and *idxvar* variants profit from more threads through the cache in the uncompressed grids.

Z-loop and *Z-loop-sliced* Grid Access Strategies

Contrary to the other stencils, the *z-loop* and *z-loop-sliced* grid access strategies suffer from a high total thread count. As both of these strategies require one thread to perform calculations for multiple cells on different Z-levels, the total number of blocks and threads required to cover the entire grid is smaller. On the largest tested grid size ($512 \times 512 \times 64$), this leads to occupancy issues: As threads stall, there is not enough work that can be scheduled to certain streaming multiprocessors to hide the latency. All threads inside a block are required to be executed on the same SM; having more threads inside a block thus gives fewer blocks that can be scheduled onto stalled SMs. Smaller block sizes, on the other hand, give the scheduler a larger pool of blocks to choose from when trying to hide latency. Compare table 6.10 for an example of how a too large block size negatively

impacts the *z-loop* implementation of the exemplary *laplap* benchmark.

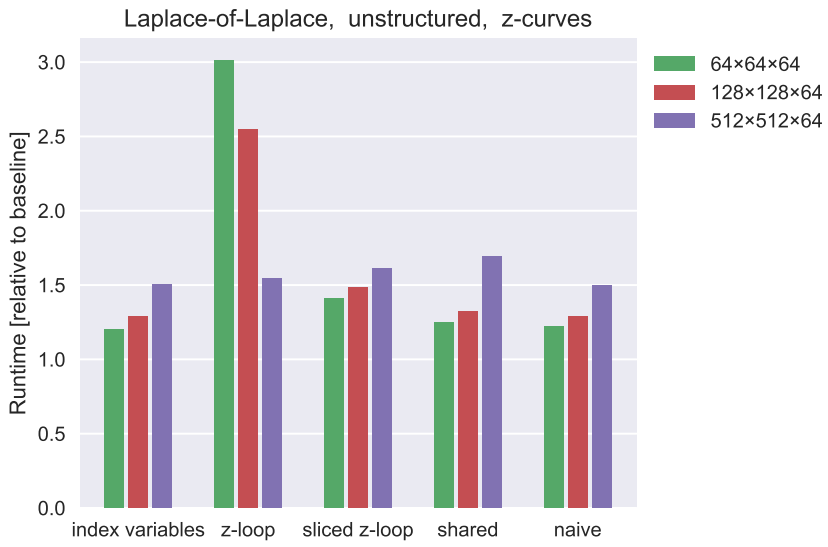

Figure 6.6: Relative overhead compared to fastest regular variant of different implementations of the *laplap* stencil on an unstructured grid with *z-curves* memory layout. For each bar, the fastest neighborship table implementation was plotted (i.e. all combinations of *compressed* and *uncompressed*, *chasing* and *non-chasing* were benchmarked and the fastest setup was chosen). Baseline: fastest regular grid implementation.

6.6 Effect of Problem Domain Size and Precision

In previous sections, we mainly focused our performance analyses on large grids of size $512 \times 512 \times 64$. This section gives some insight into how the performance of unstructured grid stencil computations relates to different input domain sizes. As we noticed, the relative overhead caused by unstructured grids varies strongly depending on the size of the grid. Generally, stencils applied to small unstructured grids are closer to regular grid runtimes than the same stencils applied to large unstructured grids (in relative terms). At smaller sizes, the cache can be used more effectively.

Of special interest are size changes in the Z-dimension only. Changing the Z-size and measuring the accompanying slowdowns reveals how well the different implementations are able to make use of the regularity in the Z-dimension. We investigate this in section 6.6.2.

We also noticed that some grid *storage optimizations* are only effective on grids of certain sizes. This is not addressed in this section, but in section 6.4.

6.6.1 Change in X and Y Domain Size

We have evaluated combinations of grid storage and access variants for the three stencils on grids of X-Y size 64×64 (small grid), 128×128 (medium grid) and 512×512 (large

grid). We observed that relatively seen, increases in the X-Y-domain size affect the unstructured variants more negatively than the regular ones.

For small domains, the runtimes of unstructured variants are generally close to those of their regular counterparts. With increased domain size, there is an increase in the relative slowdown of the unstructured variant with respect to the fastest regular implementation. We assume this is due to cache sizes; while in smaller grids, entire neighborship tables and cell values remain cached, larger grids necessitate larger neighborship tables that cannot fit into the cache as a whole. Figure 6.6 shows this trend for the *laplap* benchmark, but the same characteristics apply to the other stencils as well.

The *z-loop* grid access variant is the only exception to the described characteristics. Its relative slowdown decreases for larger grid sizes. This is probably due to two reasons. First, the z-loop variants already suffer from low occupancy in large grids. In small grids, occupancy is even lower, not making use of the complete parallelism capabilities of the GPU. Second, in small grids, the neighborship table may be in the cache in its entirety after one read. This allows subsequent neighborship table accesses to be practically free. This offsets the supposed optimization of the *z-loop* variant, which aims to reduce the number of reads of neighborship table entries. This optimization only begins to provide a benefit in large grids.

6.6.2 Change in Z domain size

Given a 512×512-sized base grid, we benchmarked our stencils with various Z-sizes ranging from 1 to 64. Analyzing the relative unstructured grid overhead, we observed two main patterns, which can be seen in figure 6.7. Which pattern the overhead follows depends on the chosen grid storage. The observed patterns are as follows:

First, for uncompressed storage, the overhead decreases quickly for the first few Z-size steps, then fluctuates around some constant. This can be most clearly be seen in the *non-chasing* stored grids. Second, when compression is used, the overhead increases in the first few Z-size steps, then also remains roughly constant.

These trends can be explained by two countering effects:

On one hand, the increased number of Z-levels naturally results in an increased total number of cells. In an unstructured grid, some extra effort has to be made to determine the location of each additional cell's neighbors. We call this the *once-per-cell cost*. This effect alone would lead to constant additional relative costs (i.e. linear additional absolute costs) with respect to the regular grid.

On the other hand, as our grids are regular in the Z-dimension, neighborship pointers can be re-used among threads operating on cells with the same X- and Y-coordinates. Let us call a set of cells with identical X- and Y-coordinates a *pillar of cells*. If neighborship pointers are effectively reused, loading them is only a *once-per-pillar* cost. No matter how many Z-levels there are in the grid, the first load of the neighborship pointers remains a constant cost. As Z-size increases, this constant becomes vanishingly small in relation to the overall cost of the regular-grid computation. Thus, considering only the *once-per-pillar* costs, performance should approach regular grid performance asymptotically for increasing Z.

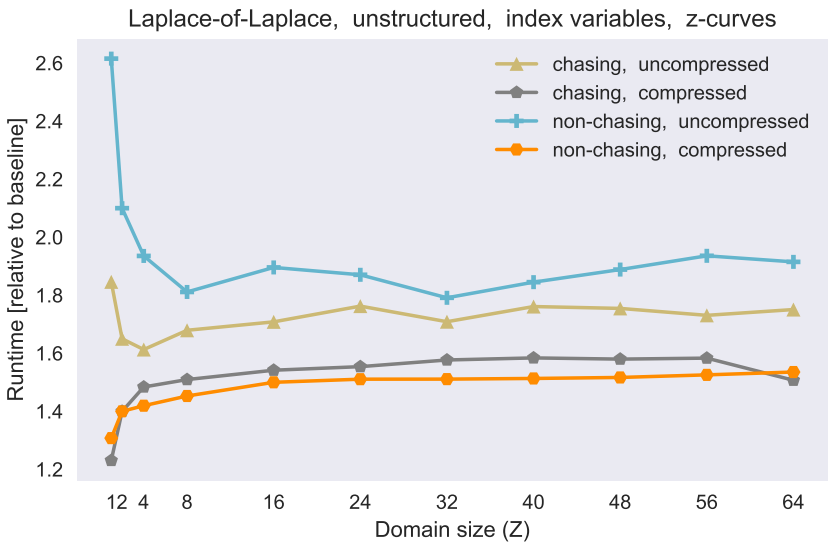

Figure 6.7: Relative slowdowns of the *laplap* stencil, implemented using the *idxvar* access strategy, on unstructured grids of various Z-sizes (*z-curves* memory layout with *double* precision). Baseline: fastest regular grid implementation. The other two stencils show similar characteristics.

The observed relative overheads result from a combination of the two described effects (*once-per-cell* and *once-per-pillar* costs). In the uncompressed variants, the *once-per-pillar costs* appear to dominate; thus, an increase in Z-size leads to better relative runtimes.

Some of our optimized access strategies aim to make explicit use of the *once-per-pillar* nature of neighborship reads (*shared*, *z-loop*, and *z-loop-sliced*). Even the access strategies that do not explicitly exploit Z-regularity benefit from it if neighborship pointers happen to reside in caches. In the best-case scenario, all neighborship pointers are already in the L1 cache (*idxvar*), shared memory (*shared* access strategy), or loaded into registers at the beginning of a loop (*z-loop*) when a thread begins computation.

6.6.3 Effect of Floating-Point Precision

We observed no significant differences in the relative slowdown when using single-precision floating-point data types instead of double-precision data types. As expected, the absolute runtimes decrease with lower precision. This happens in the same proportions for both regular grid and unstructured grid implementations – both are roughly 40% faster than double-precision speeds.

As it appears to not affect the characteristics of our benchmarks, we consistently used double precision data types in all other experiments apart from this section.

Chapter 7

Conclusions

After establishing the basics of general-purpose GPU programming on the Nvidia CUDA platform in section 3, we explored several approaches to storing unstructured grids and accessing them in the context of a stencil application in sections 4 and 5. Concerning grid storage, we addressed the topics of coalescing, pointer chasing (neighbor-of-neighbor access), storage of multiple fields (array-of-structs or struct-of-arrays), and memory layout (row-major and z-curves). For grid access, we explored ways to harness the regularity of the grid in the Z-dimension.

Lastly, in section 6, we assessed the performance of the implemented methods. Across all executed benchmarks, we measured that the main cost of switching to an unstructured grid is roughly a constant time penalty. On a $512 \times 512 \times 64$ grid, the additional time required by the unstructured kernels is between $137\mu s$ and $159\mu s$ for the *hdiff* and *laplap* stencils in the best case, respectively. The additional cost of the fastest unstructured implementation in the computationally more complex fastwaves stencil is lower at $102\mu s$. We suppose this is because it only accesses four neighbors. Likely, the cost paid when switching to indirect addressing is mainly due to the latency of the neighborship table accesses which must occur before any computations can be done and that these measured overheads quantify that latency.

Setting these overheads in relation to the fastest regular implementation runtimes, we get relative slowdowns are in the range of 4% to 71%. The biggest differences in these relative slowdowns are observed due to differences between the stencils. As the overhead is largely constant, simpler stencils with less computation time spent per cell suffer more. For the most simple stencil, Laplace-of-Laplace (*laplap*), relative slowdowns are largest with values between 45% and 71%. The more complex fastwaves stencil, which accesses nine different fields (compared to the one field of the *laplap* stencil) only suffers from slowdowns in the 5%-area. The medium-intensity horizontal diffusion sits in the middle with its $25\% - 55\%$ range.

Table 6.2 summarizes the fastest storage/access-combinations for all stencils and reports their runtimes and overhead. We explored the differences that the employed storage and access strategies make in sections 6.4 and 6.3 respectively. For low- to medium-intensity stencils operating on large grids, we recommend using the *idxvar* access

strategy on a grid stored in *compressed, non-chasing* fashion, as this was the fastest combination across our benchmarks.

We observed that performance differences also depend largely on the problem domain size. Most results presented focussed on grids of size $512 \times 512 \times 64$. In smaller grids, deliberate optimization attempts often proved to be ineffective. Some of our optimizations introduced small, one-time overheads intended to pay off later; this is only effective in larger sized-grids.

The choice of the right number of threads when launching the kernels is especially important. Depending on the implementation, a different number of threads, or a different arrangement of the threads is better. In section 6.5 the effect of the block sizes is evaluated and explained.

Bibliography

[1] Shi-Kuo Chang. *Data Structures and Algorithms*. 2014. ISBN: 9789812383488.

[2] Flavio Chierichetti et al. "On compressing social networks". In: *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2009, pp. 219–228.

[3] NVIDIA Corporation. *CUDA C++ Best Practices Guide (v10.2.89)*. 2019. URL: `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html` (visited on 11/28/2019).

[4] NVIDIA Corporation. *CUDA C++ Programming Guide (v10.2.89)*. 2019. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html` (visited on 11/28/2019).

[5] NVIDIA Corporation. *Parallel Thread Execution ISA Version 6.5 (v10.2.89)*. 2019. URL: `https://docs.nvidia.com/cuda/parallel-thread-execution/index.html` (visited on 11/28/2019).

[6] Zhe Jia et al. "Dissecting the nvidia volta gpu architecture via microbenchmarking". In: *arXiv preprint arXiv:1804.06826* (2018).

[7] Davis King, Jarek Rossignac, and Andrzej Szymczak. "Connectivity compression for irregular quadrilateral meshes". In: *arXiv preprint cs/0005005* (2000).

[8] DJ Mavriplis. "Unstructured grid techniques". In: *Annual Review of Fluid Mechanics* 29.1 (1997), pp. 473–514.

[9] Jarek Rossignac. "Edgebreaker: Connectivity compression for triangle meshes". In: *IEEE transactions on visualization and computer graphics* 5.1 (1999), pp. 47–61.

[10] Lizandro Solano-Quinde et al. "Unstructured grid applications on GPU: performance analysis and improvement". In: *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. 2011, pp. 1–8.

[11] Duane Storti. *CUDA for engineers: an introduction to high-performance parallel computing*. Addison-Wesley Professional, 2016. ISBN: 0-13-417751-7.

[12] Wikipedia contributors. *Z-order curve — Wikipedia, The Free Encyclopedia*. [Online; accessed 2-May-2020]. 2020. URL: `https://en.wikipedia.org/w/index.php?title=Z-order_curve&oldid=952434670`.