UNIVERSITY OF COPENHAGEN

Transform Array Operations to Optimize Memory Access for GPGPU

Mads Ynddal
SJT402

March 3, 2019

Abstract

In this thesis, I investigate the transformation and optimization of memory access patterns, on General Purpose Graphics Processing Units (GPGPU), and its effect on Bohrium. Bohrium is a Just-In-Time (JIT) compiler for NumPy, a Python library for array operations. Bohrium generates code on-the-fly for a GPGPU, which can heavily accelerate computations.

The common goal with Bohrium and this thesis, is to investigate, if it is possible to close the performance gap, between handwritten and auto-generated GPGPU code.

I will implement vector-reductions for the GPGPU, which is currently an unsupported operation. Its addition, will enable large performance gains, as these computations are otherwise carried out by the CPU. It will also allow for operations, that would traditionally use too much memory, as temporary arrays no longer need to be allocated.

I will also deploy methods of optimizing the memory access pattern in general, by rearranging computations to better utilize cache, and achieve memory coalescence. In the same regard, I look at auto-tuning kernel parameters, to find the optimal work-group sizes for the GPGPU. But the implemented optimizations, ends up rendering the auto-tuner redundant.

The changes are tried on a selection of real-world benchmarks collected in the Benchpress repository, which consists of several physics simulations and applications.

Although there is no quantified goal, as to how much faster the benchmarks needs to get, the optimizations have proven to be widely successful. Some benchmarks are running at ten times the original speed, while only a single benchmark is seeing reduced performance.

Comparing performance with a handwritten OpenCL-version of a benchmark, we can show, that the gap between handwritten kernels and Bohrium's auto-generated kernels has significantly narrowed, and is almost comparable. The mentioned benchmark originally took 30.5 seconds to finished in Bohrium. At the end of this thesis, it takes 3 seconds, while the handwritten kernel takes 1.7 seconds.

Contents

Chapter 1

Introduction

1.1 Introduction

This chapter will shortly outline some of the specific, technical terms of GPGPUs, array operations, and details about pseudo-code and graphs in this thesis.

Be aware, that the last parts of this chapter contains some indications of future results, as it has been updated, to give an accurate description, of the thesis in its current state. For example information about the auto-tuner and deviation in results, which will be revealed in the following chapters.

1.2 Motivation

The goal of this thesis, is to narrow the performance gap between NumPy code, accelerated with Bohrium's auto-generated kernels, compared to handwritten OpenCL kernels.

A part of this, is to implement vector-reductions in OpenCL for Bohrium, and implement an auto-tuner to improve the general performance. Bohrium already supports reductions on tensors with ranks above 1 (matrices), but if they can be improved upon, it will greatly benefit performance.

One of the core features of Bohrium, is its ability to detect temporary arrays, and avoid allocating them in memory, and enable computations, that would otherwise have exhausted all memory resources. I will go into more details with this in the following sections.

The only reason this is possible, is because the calculation utilize reductions. When vectorizing classic programming, it is often at the expense of memory.

As an example in the N-body benchmark, we have n objects, which interact with each other. To find the force that they apply to each other, we will have to find the distance from any object, to any other object. This effectively makes an n^2 matrix of distances, which we use to calculate a force, and sum each column to get a force vector for every object. But Bohrium never allocates n^2 elements. We can reduce the columns we generate on-the-fly, and only allocate n elements for the output.

This is a pattern that is seen in almost any vectorized work-load, and why it is an important feature to support and improve in Bohrium. Improving the reduction by any amount, will likely make an improvement in the benchmarks.

When the vector-reductions are encountered in the current state of Bohrium, it will be hit be some hard penalties, if the other operations were performed on the GPGPU. The full amount of elements will be allocated in the GPGPU's memory, and then transferred to the CPU, where it will be reduced and possibly moved back to the GPGPU. The transfer in itself is a slow process, and the following CPU reduction will often be slower, than the what the GPGPU is capable of[33].

1.3 Related Work

There are many systems for Python, which try to harness the power of a GPGPU, with the simplicity of a scripting language. But as far as my research goes, no one fills the spot of generic, fully automatic, GPGPU accelerated array operations in Python. The most automatic ones, are domain specific to, for example machine learning, while the more generic, in one way or another, makes you write low-level kernel code.

And even though several of them are compatible with NumPy, none of them are drop-in replacements for it, but rather an addition to NumPy.

1.3.1 Domain Specific

PyTorch and Tensorflow are two of the most popular machine-learning libraries of Python[24][28]. Both of which are integrated with NumPy, and does GPGPU accelerated computations. But there are several limitations to these claims.

Using them on NumPy arrays always require a conversion or reinstantiating it into a tensor. TensorFlow has the `convert_to_tensor` function, while PyTorch has its `Tensor` object constructor, which takes a NumPy array.

The functions available in both libraries do overlap with NumPy, and a lot of the API has similarities, but it is not an exact NumPy replacement, and the features are all angled towards machine learning. This makes them a poor match, for doing general scientific computing.

1.3.2 Manual

PyCuda, PyOpenCL, Numba and CuPy are four different takes, on making array operations, or arbitrary kernels run on a GPGPU from Python[21][22][8][6].

The first two are quickly described. They facilitate a low-level bridge between Python, and the frameworks CUDA and OpenCL respectively. They have support for transferring NumPy arrays to and from the GPGPU, but it is all performed manually, at the request of the Python developer. All kernels are also loaded and compiled from handwritten C/C++ files.

The latter two both have predefined functions, which mimic some of NumPy's functionality, but none of them seem to be fused together – one of Bohrium's strongest points. They also try to make it easier to write kernels, by allowing the developer to write the kernel in a simpler language. For Numba, this language is Python, for CuPy, it is an supposedly unnamed Python-like language[6, User-Defined Kernels][8, Writing CUDA Kernels].

They all also have the same downside: Which ever way you code GPGPU kernels for them, it will be manufacturer specific. Either for Nvidia/CUDA or AMD/OpenCL.

Whether or not they cover a larger set of NumPy's functions, will not be discussed, as this is mostly a matter of domain. Meaning, what functions are necessary, are up to the exact use-case.

1.3.3 Fully Automatic

In this category, we have Bohrium. But I will shortly mention the programming language Futhark as well[30].

Futhark tries to solve much of the same use-case as Bohrium. Futhark has nothing to do with Python, but is a programming language by itself. Where it has a common ground with Bohrium, is its OpenCL and CUDA kernel generation, based on the user's code.

Bohrium separates itself, but requiring no user interaction whatsoever[3, Features]. It can be applied externally, when starting Python, and it will replace and optimize any NumPy function it supports.

Futhark requires all users to learn their language, although with similarities to other functional programming languages. I will later look at some of their research in segmented reductions.

Bohrium also separates itself from all of the above, not by having a JIT-compiler, which Numba also features, but by its lazy evaluation. In Numba, the developer is as much responsible for the performance of the kernel, as writing it in OpenCL/CUDA by themselves.

Bohrium takes the approach of fully-automatic kernel generation, by doing lazy evaluation of the calls to NumPy functionality. When it is required by Python to produce a result, it looks through the backlog of operations, it has not yet evaluated, and produces one or more kernels, to find the desired results.

Bohrium is also platform-agnostic on the outside. The same NumPy code will run on CPUs and GPGPUs from any manufacturer, as long as it is supported by one of the backends: OpenMP, OpenCL or CUDA. They might have different features or performance, but it will not affect the correctness of the execution.

1.4 Terminology

For clarity, I will define some of the terminology, as there are different words for the same concept between manufacturers. And for other parts, to clarify some more context specific terms.

The first term, is General Purpose Graphics Processing Unit, shortened to GPGPU. This specifies a Graphics Card, that is regularly used for 3D graphics, but is also capable of running OpenCL programs, called *kernels*. But in general, the term GPGPU will be interchangeable with any massively parallel OpenCL-capable device.

Through-out the thesis, I will be using terminology from OpenCL and AMD – contrary to CUDA and Nvidia. OpenCL programs are written in a variety of C99, and complied using the GPGPU-manufacturers proprietary compiler – in this thesis, Nvidia's `nvcc`-compiler (see A.1). If you do not know anything about OpenCL or CUDA, that is fine, I will go through the concepts along the line. If you do know about CUDA, then be aware of the differences. The major ones are the naming of the basic features[7]:

CUDA	**OpenCL**
Shared memory	Local memory
Local memory	Private memory
Blocks	Work-groups
Threads	Work-items
Warp	Wavefront

Especially note the different usages of "local memory". The last row is not a specific OpenCL concept, but the one I will be using instead of "warp". The concept of warps/wavefronts are not available in OpenCL 1.2, which Bohrium will be using.

1.5 Bohrium

Most of this thesis will be based around Bohrium. Bohrium is a Just-in-Time (JIT) compiler for accelerating array operations for NumPy[9][3]. The functionality I will optimize through-out the thesis, is the GPGPU kernel generation, which transparently translates the array operations in the user's code from Python+NumPy to GPGPU kernels. The translation promises the same result as the NumPy implementation, while utilizing the available hardware more efficiently.

In this thesis, I will only be implementing the features for OpenCL. The changes will likely benefit CUDA as well, but I chose to implement it just once, to have time to cover a wider area of optimizations.

The key element to the auto-tuner and reduction implementations, is to speed up the user's code with as little user-interaction as possible. This means the changes should never be seen by the user, and make all of its decisions behind-the-scenes.

If necessary, I will allow for some additional actions to take place during installation, if it has a large benefit to the user.

The JIT compiler performs lazy operations, meaning it will collect a series of operations up until when Python needs the result[3, Features]. Only then, will the actual execution be performed. Apart from saving time on results that are never needed, it also adds to the optimization Bohrium can perform.

Fuser

An important step in optimization and generation of GPGPU kernels in Bohrium, is the fuser[40]. One of the limiting factors of any high-performance operation, is accessing memory[44, 2.2 Bandwidth]. By having the fuser combine operations, we can fetch a value from memory just once, perform a series of operations, and place it back in memory. If not optimized, the value might have been fetched and written back several dozens of times, almost grinding the execution to a halt.

This affects the generated kernel. The fuser will try to stuff as many operations as possible into a kernel to save reads and writes.

The fuser is fed almost verbatim what the user requests from NumPy in an internal instruction format. The fuser combines the flat instruction list into a layered structure called `Blocks`, which also encloses operations inside ranks.

In addition to saved time from reading and writing the same value back and forth, Bohrium also supports *array streaming*[41]. The technique enables Bohrium to handle data sizes, where NumPy would exceed the available system memory with temporary arrays. The benchmarks later in this chapter have several examples of this. Two of the benchmarks expands their problem to $O(n^5)$ elements, but only temporarily, as these elements are reduced to $O(n^3)$ elements right after they are used. By not allocating these in global memory, and only keeping them as temporary, local variables, we can simulate having the required memory, while saving time on not reading and writing to global memory for temporary data.

Blocks and LoopB

Bohrium represents a kernel in an internal structure. The structure is called `Block`, and is a wrapper, which can contain either a loop (`LoopB`) or an instruction (`InstrB`). A loop is a collection of `Blocks`, and also carries meta-data about sweeps (see below), the rank of the loop, the size (dimensions in rank), and a number of other internal properties.

Sweeps

One of the concepts of the `Block`, is its `_sweeps` property. The "sweep" concept in Bohrium defines an operation, which performs an action along the given rank, and indicates the rank cannot run unrestricted in parallel. Currently, the only two supported sweeps are prefix-sum and reduce. A large part of this thesis, is to implement support for running these operations in parallel, to gain the best possible utilization of the hardware.

Reduce should be familiar as a summation, if the reduction is performed with an addition operator. But the concept of reduction is more abstract than a summation. The reduction takes a list of values, an associative operator and a neutral element. The operator is applied between all the values in the list. The neutral element can be used to pad or start the computations, and will not affect the final result. For example, the neutral element for addition is 0 and the neutral element for multiplication is 1. The neutral element can be insert anywhere in the list, any amount of times we want, without affecting the result. When we begin to look at implementing the reduction, we will see some more rules we can utilize.

Prefix-sum (also called scan) is very similar to a reduction. In short, a prefix-sum takes a vector, and returns a vector of the same length, but every value of the new vector is the reduction of all preceding values in the input list.

As an example, here is a reduction and prefix-sum with an addition operator:

Input list	1	2	3	4	5	6	7
Reduction							28
Prefix-sum	1	3	6	10	15	21	28

1.6 Memory Access Patterns and Tensors

When programming high-performance applications for the CPU and GPGPU, it is important to be aware of the application's memory access pattern. Most developers will know about row- and column-major data access, but maybe not intimately enough to know the memory layout of rank-n tensors. Generally, the same rules apply to the GPGPU as the CPU, but there are some details, that are not immediately clear.

1.6.1 Tensors, Vectors, Matrices and Arrays

Before we look at access patterns, I will shortly describe the concept of tensors in the context of NumPy and Bohrium.

Tensors are a generalization of vectors and matrices, and is often used interchangeably with `array` or `array_like` in NumPy[13]. But beware, that most operations on NumPy arrays are performed element-wise. For example, multiplying two arrays, will give the result of multiplying the two elements, which have the same index from each array, whereas multiplication of a matrix, traditionally means an actual matrix-matrix multiplication[12, Long answer]. In this thesis, all operations will be element-wise, except for the reductions.

In this thesis, I will generally be using the term *vector* for a rank-1 tensor, the term *matrix* for a rank-2 tensor, and rank-n tensor for everything else. When describing the data structure, I will often use the term *array*

When describing reductions, they will always apply to a certain *axis*. This value follows the NumPy implementation, and is zero-indexed, while the corresponding rank will start from 1. An axis of -1 is defined as the highest axis in NumPy, -2 is the second highest and so on.

The stride of an array, is a number for each rank, which defines how far each value is separated in memory. A stride of 1, means the values are following each other. The following dimension, might have a stride of 10, which means, the values for this rank, is found by iterating 10 spaces ahead, for every value. An illustration can be seen in figure 5.7, as well as more explanations in the same section: 5.3.2.

As NumPy is row-major, the strides for an array are ordered from high to low. With this, I mean a matrix should be traversed through every value of `j`, before iterations `i`, in the matrix access of `M[i,j]`.

1.6.2 Optimal CPU Usage

When programming in C for the CPU, it is expected, that the memory-layout is row-major. Meaning, that the rows of the data are placed in memory, in a structure which keeps the rows in a contiguous line.

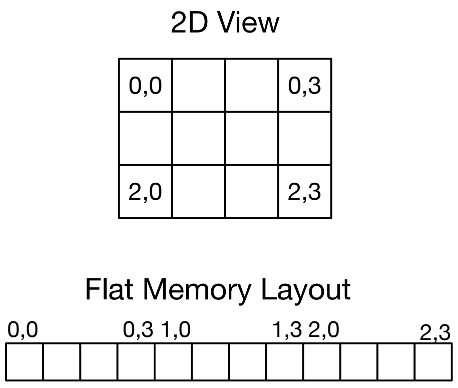

Figure 1.1: Illustration of a matrix and its flat memory representation.

To read the data from memory and into the CPU for the most optimal performance, there are two important properties of the cache: Spatial and temporal. Spatial means, it is recommended to consecutively read data, which is close to the previous value. The first read will request the data at $(0, 0)$. This will make a cache look-up, which on the first try will miss. This makes the CPU fetch the memory containing the data for $(0, 0)$, and update all levels of cache on the way into the CPU. When the next value is read at $(0, 1)$, it will result in a cache hit, and take considerably less time, than the first value. If performed correctly, and without other operations invalidating the cache, we will fully utilize every byte read from memory.

In the opposite case (column-wise), we can get a great performance penalty. This concerns the temporal property. If the data structure is large enough, we can imagine a case, where a read to every column will result in a cache miss. When the first value of every column has been read and used, we have made as many cache misses, as there are columns. Now, if you are in luck, none of the first columns have been invalidated by the later columns in the cache, and all the following calculations are done without a cache miss. This is rarely the case, and most of the time, we will see every iteration of the loop updating the cache, while later invalidating the same cache-lines.

Therefore; always read row-wise, when the data is stored row-major because of the spatial and temporal properties.

This gets slightly more complicated on multi-core processors, as there is another layer of requirements. The different cores require a certain level of cache-coherency. If a value is changed by one core, it has to immediately be reflect in any other cache that might have the value stored. But the precision of this, is on the cache-line's width. This means, that if two processor core interleave their writes, even though they never write to the same exact address, they will keep stalling each other to update the changes. This is known as false sharing[1].

1.6.3 Optimal GPGPU Usage

In OpenCL on GPGPUs, it is also advised to read row-wise, but it is done in a slightly peculiar manner.

The memory system on GPGPUs is slightly more complicated, than for the CPU. It is not just a uniform piece of memory with multiple layers of cache. All modern GPGPUs utilize a memory hierarchy, which the kernel developer has to explicitly use.

Memory Hierarchy

GPGPUs from both AMD and Nvidia feature a special memory hierarchy, with several layers, that are physically separated, and addressed directly. I will not consider integrated graphics cards on the CPU, as these are not the major target for high-performance applications.

On dedicated GPGPUs, its memory will be physically located on the device. The memory will therefore be separate from the host memory, and has to be transferred through the bus they are connected by. This bus is PCI Express 3.0 on currently modern systems. This link is often an order of magnitude slower than the internal memory speed on the GPGPU. Therefore, there is a considerable penalty, when moving a calculation from the host system to the GPGPU.

Any added feature to Bohrium, which can keep data on the GPGPU is preferred. Even if it might not be faster on the GPGPU by itself. Apart from saving time in the short run, it also allows for better scaling of performance.

The PCI Express bandwidth is fixed within a generation of the standard. Using a faster GPGPU or CPU does not speed up the link between the two. The more we can limit the use of the PCI Express bus, the better chance we have at scaling performance, with newer, better hardware and larger simulations.

Contrary to the homogeneous memory system of regular CPU programming, GPGPUs supporting OpenCL, normally have four different types of memory.

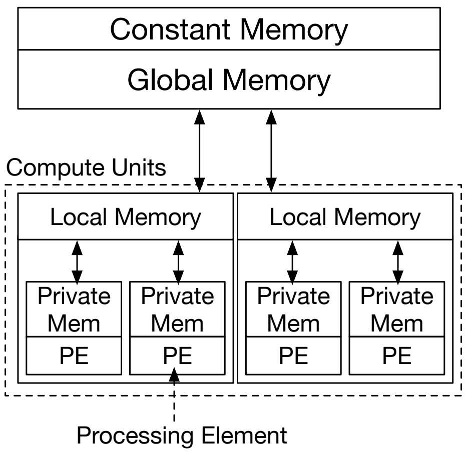

Figure 1.2: Memory hierarchy of generic GPGPU. Processing Elements are the computing devices in a GPGPU. There are always multiple Compute Units, with each a number of Processing Elements[32, Private Memory, illustration by Khronos]

Constant Memory This is what the host-system initializes. It is often not used explicitly, as the compiler will utilize it automatically. This is rather small. On Nvidia's devices, they claim a fixed memory of 64 KB[44, 3.2.5 Constant Memory].

Global Memory This is the slowest and largest layer. When buying a graphics card, this is often the memory referred to by the number of gigabytes. This is the main memory, and is accessible across all of the device. To gain any decent performance, this has to be accessed with *coalesced memory access* – see the following section. Data written to the global memory during runtime is not necessarily accessible to other work-groups. More on that in the work-group section below.

Local Memory This is one of the faster layers. This memory is accessible only within the compute unit it originates. This memory is often used as scratch-pad memory and communication between work-items in the same work-group.

Private Memory This is the default place to store variables and other allocations[32, Private Memory]. The other areas are explicitly annotated, when used. The private memory is only available to the work-item itself.

Depending on manufacturer and model, there will likely be a layer of cache between the global memory and compute units[32, Private Memory, illustration by Khronos].

Work-groups and Work-items

When a kernel is called using OpenCL, it provides a series of parameters along with the function arguments. The arguments contain arrays and constants, that can be used inside the kernel, just like regular C. The other parameters, is a set of specifications for the GPGPU. They define the problem size as a whole number of work-items in each of three dimensions – the dimensions may vary, but is often three. Then there is a whole number, which defines how many work-items that will be placed in each work-group. The problem size has to be evenly divisible by the work-group size.

These work-groups are mapped to the compute units inside the GPGPU. This means, that all work-items in a work-group share the same local memory, and through this, they are able to communicate. The work-items in a work-group, can also synchronize by calling a barrier-function.

There is no specific order of which the work-groups are scheduled to the compute units, and there is no guarantee, that any two work-groups run at the same time, or one before the other. This has the implication, that kernels, which require a full synchronization across the work-groups, need to be split into two kernels, which run after each other. Only one kernel can ever execute at once on the GPGPU, and OpenCL has a queue for the order of kernels. Some GPGPUs support concurrent kernels, if requested by the developer, but I will not go into details with this.

Wavefronts and Warps

Outside the scope of OpenCL 1.2, is the concept of warps and wavefronts from Nvidia and AMD respectively (I will be using the term *wavefront* irrespectively of manufacturer). This is addressed in OpenCL 2.0 and on-wards, but on the hardware I have available, only OpenCL 1.2 is supported (see A.1)[17, Subgroups].

Even though it is not described in OpenCL 1.2, the concept is available, and is often described as an important factor for performance [44, Occupancy].

Work-groups are divided into wavefronts, when the kernel is spawned. Not all wavefronts can be active at once, and are switched between, depending on their state. If a wavefront requests data from global memory, it will be put on hold, while other wavefronts get to run. This is done to hide the latency of global memory, and better utilize the available processing power.

Wavefronts are the smallest scheduleable number of work-items the GPGPUs can work with. For Nvidia, this number is 32, while AMD uses 64[44, Differences Between Host and Device][31, Terminology]. Inside the wavefront, there is no need to call barriers, as all processing is done in complete synchronization. This also means, that any flow-control statements can reduce utilization, if execution diverges inside a wavefront, all work-items in the wavefront will have to wait for the ones that diverged.

Memory Coalescence

The wavefronts have another role, when it comes to accessing global memory. It is possible for any work-item to access any address in global memory, at any time. But, to gain performance, the usage of global memory has to be coordinated.

On GPGPUs there is a concept of *coalesced memory access*, which draws parallels to spatial properties of a cache. The GPGPU is designed to coalesce (merge) memory requests to global memory, when they originate from the same wavefront or half-warp[44, 3.2.1 Coalesced Access to Global Memory][31, 1.5.2 Dataflow in Memory Hierarchy].

If the requested memory address of each work-item are within a contiguous block in global memory, it results in only one memory transaction for data-types of 4 bytes. The memory access also have to align with a certain segment size. On Nvidia's newer cards, these segments are of 128 bytes, if the data size is 32 or 64 bits [44, 3.2.1 Coalesced Access to Global Memory]. It is neither important, that all work-items participate, nor that all work-items are in order.[44, 3.2.1.2 A Sequential but Misaligned Access Pattern]. When these rules are broken, two or more transactions are performed, but it is guaranteed to fetch all requested addresses.

1.7 Pseudo-code

I will be using some slightly special pseudo-code to describe how OpenCL will parallelize the algorithm.

It is based on Python, and thereby does not show any braces or semicolons. Indentation defines code blocks.

Variables `i0`, `i1`, or `i` followed by any other integer, is a loop-iterator. Some loop-iterators are parallelized across work-items. These are instantiated after a `gfor` keyword, meaning *global* for-loop. Data dependencies across iterations of a `gfor`-loop is not allowed, as they can be executed in parallel on different processors of the GPGPU. The keyword `for` defines a regular for-loop, which is performed sequentially in each processor. It is allowed to have data-dependencies here, as it does not leave the multi-processor.

```
gfor i0 = 0..dim(0)
  for i1 = 0..dim(1)
    A[i0, i1] = i0*i1
```

Variables starting going from `A` to `Z` are assumed to be arrays in global memory. They are either read from or written to. Addressing using square brackets and commas, are assumed to translate the index to the memory structure automatically.

The numbering `0..dim(0)` defines the range of numbers from 0, to the value of `dim(0)`. The `dim` should be seen as a built-in function, which returns the size for the numbered rank of the input tensor.

1.8 Reductions

Reduction are inherently serial. Naively, a reduction is often implemented by running a for-loop from the left-most element, applying the operator to the next element, while keeping a temporary variable as accumulator.

This is not an efficient method on a massively parallel system like a GPGPU. Contrary to a `foldl` or `foldr` from functional programming, a reduction makes no guarantee as to which order the operator is applied to the elements. We are allowed to make any partitioning of the input data, and pad it with the neutral element for the given operator. This is why we require an associative operator.

Given an arbitrary associative operator $\odot$, we can reduce a range of numbers as follows:

$$a_0 \odot a_1 \odot a_2 \odot a_3 \odot a_4$$

For use in a reduction, the operator also need to have a neutral element, e. This element can be placed anywhere and as many as needed, without changing the result of the reduction.

$$a_0 \odot a_1 \odot a_2 \odot a_3 \odot e \odot a_4 \odot e$$

Given the associative, we are also allowed to partition the calculations as we like, as long as the sub-results are reduced using the given operator. This is vital, if we want to parallelize the calculations.

$$(a_0 \odot a_1 \odot (a_2 \odot a_3) \odot e) \odot (a_4 \odot e)$$

Although it is not a general requirement, we may also use the commutative property of the operator, if it supports this. Then we are allowed to rearrange any element.

$$(a_1 \odot a_3 \odot a_2 \odot e) \odot (a_0 \odot a_4 \odot e)$$

This includes operators like add, multiply, minimum, and maximum. But it does not work with subtract and division, as these are neither associative, nor commutative.

Reduction is only defined for a vector or rank-1 tensor. When presented with rank-n tensors, there are two options: Reshape the data of the tensor into a rank-1 tensor, and reduce to a scalar. The other option is to perform the reduction along an axis of the tensor. This effectively transforms the tensor to a tensor with one fewer ranks. This can be performed recursively if needed.

1.9 Benchmarks and Graphs

This thesis will center around improving performance. If any changes are to be made, it has to be because of a provable performance increase.

I will in this section quickly write up how benchmarks are performed. Chapter 9, will contain more information, and a deeper look at performance.

I have been taking inspiration from the developers of PyPy, which have a very concise, and nice strategy of optimizing code[23]. PyPy is otherwise not used in this thesis.

Below, I have the four most relevant points for our use-case.

Regression-test Benchpress has provided us with a good range of test, that we can use to measure performance, while Bohrium has its own correctness tests[2]. I will also supply some additional tests, when there is a specific case that might be challenging. Often, performance is not part of a regression-test, but in our case, correctness and performance are equally important to test.

Measure, don't guess Do not guess where the performance issue lie, measure the performance and isolate the cause. This will save time trying to optimize something with a very small importance.

Memory-bound vs. compute-bound PyPy's strategy used I/O bound instead of memory, but in terms of GPGPU's, memory is more relevant.

Determine whether performance is limited by memory speed or by compute speed. By comparing the speed of a test with the theoretical limit, or what can practically be demonstrated, we can see how much performance, we can hope to achieve.

As we will see later, it does not make sense, to assume a higher through-put from global memory, than what we can demonstrate with a memory-copy kernel.

Focus on tight loops Put the effort where it matters the most. As stated by PyPy, find "the 20% of the code where the code is spending 80% of its time". Saving a single instruction in a tight loop, will have a more significant impact, than removing 50 instructions in the initialization.

1.9.1 Benchmarks

To prove that we are making progress, and to verify the changes that are made, I will assemble a set of benchmarks from Benchpress[2].

The benchmarks consist of three embarrassingly parallel problems, three problems that are only possible because of streaming and reduction, and lastly three benchmarks which are of the type: Finite difference.

In general, they are chosen, as they are using functions relevant to the changes we are making.

The embarrassingly parallel problems contain vector-to-scalar reductions, which is one of the core features my thesis will touch. One also contains a very large segmented reduction, which will be relevant at the end of the thesis.

The streaming and reduction benchmarks contain a lot of reductions in inner dimensions. Some of these will be segmented reductions, and it will challenge the fuser and my implementation, as these reductions are also nested, and contains multiple other operations, that will have to be embedded, because of streaming.

The last category, finite difference, are memory bound, but are also known for being highly susceptible to changes in work-group sizes, as this affects memory access, calculations and latency hiding. These are likely to have a high benefit from the auto-tuner and the later changes to access patterns.

Embarrassingly Parallel

These benchmarks are what is called *embarrassingly parallel.* This means, that there is no or little relation between elements during computations. Some of the later benchmarks, will have to synchronize and swap elements at every iteration. These are calculated independently, and summed up, to make the final result.

In this case, they are all heavily compute-bound, as almost no global memory is used.

Monte Carlo Pi A method to approximate π, using random points in 2 dimensions. It's included, as it runs in $O(n)$ time, and using no global memory, and has a large vector-reduction to count the values within the unit circle.

Leibnitz Pi This benchmark also calculates π, but using the Leibnitz formula instead, which doesn't use random numbers, but instead values from 0 to n. Apart from that, it features the same type of calculations as Monte Carlo Pi.

Galton Bean Machine It simulates a classic Bean machine by generating a large amount of random numbers, and summing the results. This has been included because of the large two-dimensional reduction, where the values are not read in from global memory.

Streaming/Reduction

These benchmarks are special, because they would almost be impossible to run using ordinary NumPy. As described earlier, Bohrium is able to detect temporary arrays, and not allocate them in the GPGPU's memory, by the functionality called *array streaming.*

This is a classic challenge, when vectorizing calculations, as algorithm complexity often results in equivalent memory usage[40, p. 3]. If an algorithm like N-body requires n^2

calculations, to find the interaction between all bodies, it translates to a matrix of n^2 in size. Array streaming reduces this to an allocation of just n elements. This is only possible because of GPGPU support for reductions. And this also means, that any improvement to reductions, has an almost universal benefit to calculations run on the GPGPU.

These benchmarks are slightly more memory bound than the embarrassingly parallel problems above. But they also feature some more complicated calculations, which makes them perfect as a balanced benchmark.

N-body This simulates n independent bodies, that form a fictional solar system. It runs multiple iterations, where it calculates the interaction of forces, and applies them to move the bodies accordingly. It has been chosen for its simplicity, while still having multi-dimensional calculations and reductions.

X-ray Sim The code simulates an x-ray source, object and sensor. It has been chosen for its high dimension count, and the reductions on several layers, which would not have been possible without array streaming. The code for this is relatively complex, and has been included to see the benefit of larger code problems in Bohrium.

Magnetic Field Extrapolation The benchmark reconstructs the Sun's magnetic field based on 2-dimensional data, into 3-dimensional data[41]. This too, has been chosen for its high number of dimensions, and it large reduction of 5-dimensional data into 3 dimensions. It also uses data read from global memory in these reductions, differentiating itself from the other benchmarks.

Finite Difference

These benchmark are of the type "finite difference". The algorithms are different, but they all converge toward a state. They run multiple iterations, and at each point, they reduce some parameter to a scalar, which is used as a quotient. This quotient converge towards a value – often zero – and the simulation is halted whenever the desired precision is achieved.

These benchmarks all use a technique called stencils. Stencils try to solve a type of data dependency problem, where an otherwise serial problem is split into blocks. These blocks are still connected computationally, but the stencil-technique borrows values from neighboring blocks. At every iteration, all blocks read from memory, some change is made, and the new state is written to memory for the next iteration. On a CPU, it's possible to make barriers in the code to synchronize loading, calculation and writing as to not cause race-conditions. This is harder on a GPGPU, as we can not synchronize across work-groups. Therefore it will often use two buffers, one for reading the last iteration and one for writing the state of the next iteration.

Heat Equation This uses the Jacobi method to simulate heat spreading out on a 2-dimensional surface. It has been chosen, as it is known to be highly susceptible to kernel sizes. It also contains a reduction which is used to determine when the simulation converges. Benchpress also includes a pure C++ and OpenCL version, which would be good for comparison – see chapter 9.

Shallow Water This benchmark simulates waves on a 2-dimensional surface. It has been chosen because of its more extensive use of stencils, which makes it a little more computational and memory heavy than the heat equation.

Lattice Boltzmann This benchmark simulates the flow of a fluid or gas moving past a cylindrical object. In addition to stencil usage, this have been chosen, as it is the most complicated of the finite difference benchmarks.

1.9.2 Problem Size

The size and iteration parameters for each benchmark has been chosen from two parameters: Memory usage and time. When finding the parameters for a test, I start with an iteration count of 50. This is chosen to offset any initialization process or constant factors, and measure the code, which requires the heavy lifting. Then I increase the problem size either until the execution takes 30 seconds, or until there is no more memory on either the GPGPU or the host system. If the memory limit is met, I increase the iteration count until the target time is satisfied. These requirements are used for the large tests, when finalizing a chapter.

For smaller test, I set a target of 10 seconds instead, and change the iterations to 25 to better meet this limit. This will still show any change in execution time, but they might be less pronounced, and the precision of the result may vary.

1.9.3 Measured Parameters

I measure the time provided by the test itself. For the benchmarks in Benchpress, they have an internal time measurement, which excludes any loading or generation of data.

This time does include compilation of OpenCL kernels. Therefore; I will run several warm-up runs to make sure the kernels have been generated and compiled before measuring the performance.

The choice of this, was to isolate the time for which it takes to run the code in the best possible scenario. One could argue, that it would be more fair to include the compilation, as Bohrium does JIT compilations during run-time. This is true, but it is to be expected, that the compilation is only performed once in the first iteration, even if the user would want to run the code several times, with different data-sets or after altering parts of the code. In these cases, only the changed kernels will be recompiled. The ones that are unchanged, can be reloaded from a cache.

The changes I make to the code are also not expected to significantly alter the compilation time, which is mostly out of my hands.

1.9.4 Graphs

After significant changes to the code, or after implementing something that needs to be benchmarked, there will be some small in-lined graphs. These will contain the most relevant benchmarks. These in-lined graphs will run for a reference-point of 10 second long calculations, determined from the original version of Bohrium. The reference size of the problem, will *not* change during this thesis.

At the end of every chapter, there will be a full-sized graph for better comparison. These will have parameters targeted at 30 seconds, like described earlier.

I have chosen not to display Galton Bean Machine, as it is completely unaffected by every change in this thesis. It can be seen in a full-page graph in appendix B.

Format

All graphs will be bar-charts with the benchmarks along the horizontal axis, and the number of seconds it took to execute along the vertical axis. The bars will be grouped to show the difference between each iteration of the code. So for each benchmark, there will be two or more colored bars, each representing a version of Bohrium.

The colors *are* consistent across every graph. Auto-tuned bars, are stripped. A version of the code, and its auto-tuned results, will be of the same color. Some colors are reused, but never in the same graph.

The vertical axis will start at 0, has a fixed limit, and will be linear along axis. The small graphs will have a limit slightly above 10 seconds, and the full-sized graphs will have a limit slightly above 30 seconds. This is to make visual comparisons easier, and more fair.

As all the graphs will show the time it took to execute, it is desired to have as low a bar as possible.

All bars will be ordered as the oldest version of Bohrium on the left, and the newest version on the right for each benchmark.

On top of the last bar for every benchmark (the newest version) will be a percentage marked. This is the relative change compared to the original version of Bohrium to show the potential performance gain.

Some graphs, will move outside the visible area of the graph. They will be annotated with a time.

Deviation

I have chosen not to include any marking for deviation nor variance in the graphs, as the variation in the results are insignificant. I have measured the standard deviation for all results in the thesis, and none rise above 0.75, and most are below 0.001. As standard deviation is in the same unit as the input values, this means a inaccuracy below 1 second. The worst case of 0.75, as we will see, is an outlier of Magnetic Field Extrapolation, which ends up taking hundreds of seconds – significantly longer than the designated 30 seconds.

The time shown in the graphs, is the mean of three tests, without including the warm-up rounds.

Auto-tuner

I will use the auto-tuner to in a way, map and predict how much performance there is to gain from further improvements. At the end of every chapter, I will run the auto-tuner to see what performance we could see, if the kernels were perfectly balanced in terms of latency hiding and cache usage.

Although this can be a good measure, it does not theoretically bound the possible performance, as it does not take new features into account.

We might not be able to gain the exact performance of the auto-tuner, but if the gap diminishes, we know that we are progressing in the right direction.

If the auto-tuner stops returning any benefits, we can look for other areas, which might have become the next bottleneck.

Chapter 2

Tuning Kernel Parameters

From experimentation, and from other sources, the Bohrium developers have gotten interested in auto-tuning kernel parameters. It can supposedly increase performance drastically, to select the correct kernel parameters.

There could be several outcomes of the auto-tuning. First option, is a new static set of parameters for all kernel, if we can find some, that perform better than the current. Secondly, there are the semi-static, which are pre-selected kernel parameters, based on manual selection, but applied when a certain operation or criteria appears. Thirdly, we have the fully automatic parameter selection, which will create a model based on some prior or on-the-fly benchmarking and testing for the individual machine.

2.1 Related Work

Tuning kernel parameters for GPGPU kernels is a persistent topic in academia[35][34]. Only few have tried to formalize what theoretically makes a good set of kernel parameters[35], whereas the predominant method is still empirical trial-and-error[34].

The approach we will take, is still based on empirical measurements. But whereas others often try to optimize and analyze any arbitrary kernel on a OpenCL/CUDA level, we have the advantage to optimize and control the kernel from Bohrium's built-in vector-machine instruction set – also referred to as "blocks", see 1.5. This heavily limits the variation in memory access patterns that a kernel can feature. And this, in turn, will presumably allow us to make better generalization to the tuning parameters.

The ultimate goal, would be for the auto-tuner to statically find the best kernel parameters, without running the kernel at all. In contrast to some optimization techniques, will run the code several times to analyze the kernel[35, introduction].

2.2 Kernel Parameters

When the kernel has to get executed, Bohrium will have to provide OpenCL or CUDA with a set of parameters. OpenCL and CUDA do things slightly differently, but the result is the same.

The GPGPU is provided with a large amount of data to process, which it will split into smaller chunks. These chunks are called work-groups, and defines for the OpenCL device, how the data will be partitioned. The choice of this partitioning has wide-spanning effects on varies layers of the hardware. For the first part, it will define the amount of possible parallelism across compute units, and secondly the amount of latency-hiding inside the compute units.

Somewhat like a multi-core CPU, a GPGPU has multiple Compute Units, which each process a chunk of the input (see figure 1.2). Like SIMD on CPU-cores, these Compute Units have multiple Processing Elements (PE).

As described in 1.6.3, Nvidia groups 32 PEs into a warp and AMD groups 64 PEs into a wavefront.

While choosing kernel parameters, we will have to take the size of wavefronts into consideration. This means, work-groups sizes which are not divisible by the size of a wavefront can trivially be ignored. Nvidia further recommends to change the work-group sizes in increments of 64 to better utilize registers[44, 3.2.6 Registers].

2.2.1 Valid Parameters

Not every size of a work-group is allowed. Through OpenCL, we can query a device for its limits on work-group sizes. Here is an example from the Nvidia GeForce GTX 970, which is the GPGPU I will be using for this thesis (see appendix A.1).

Work Item Dimensions	3
Max Work-items Dim. 1	1024
Max Work-items Dim. 2	1024
Max Work-items Dim. 3	64
Max Work-group Size	1024

The "Work Item Dimensions" is the maximum number of dimensions that the OpenCL device supports – or rather how many dimensions OpenCL will help us with. We can always write our own memory strides to access n-dimensional data. Nvidia states, that using the multi-dimensional features of OpenCL has no effect on performance. It is just to easier map the problem to the GPGPU[44, 4.4 Thread and Block Heuristics]

The "Max Work-items Dim." tells us the maximum number of work-items in each dimensions, but this doesn't mean we can have $1024 \times 1024 \times 64$ work-items. The work-group size cannot exceed "Max Work Group Size" – 1024 – work-items under any circumstance. So valid work-group sizes could be: $(1024, 1, 1)$, $(512, 1, 1)$, $(32, 16, 2)$ etc.

But apart from the limitations on dimensions, we are free to choose any work-group size between 1 and 1024 – both included.

Another aspect of work-group size is the use of cache, memory and registers in the hardware. We might find situations, where a kernel cannot run because the work-group size demands more local memory, than what the hardware has available. In that case, the kernel either has to be split up, or the work-group size should shrink down. We will address the issue, if it presents itself.

2.3 Proving Kernel Parameters Matter

The domain of sensible kernel parameters is not especially large, but when the dimensions rise, it becomes more complex.

For 1-dimensional kernels on the Nvidia GeForce GTX 970 (appendix A.1), it would be $1024/32 - 1 = 31$ possible parameters for a warp of 32 work-items, which can easily be search through.

When we proceed to 2-dimensional kernel, it gets more complicated. Then we have to take into account the maximum work-group size of 1024, which is the product of the two dimension sizes.

From preliminary testing, I have found that kernels often benefit from maximizing their size to 1024 work-items. To find these, it is a great help to know the 11 divisors of 1024: $1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024$.

By iterating the list forward as the first dimension value and iterate the list in reverse for the second dimension, we will get the 11 parameters, which results in a work-group size of exactly 1024.

I write up a short Python script to find all combinations we will have to test. I use this script later in the auto-tuner to generate the wanted parameters. The script can be found in the appendix C.1.2.

The script finds 139 unique combinations. It starts by finding all valid parameters in a grid with a spacing of 16. For it to be valid, either the x or y axis has to be at least 32, and the product of x and y has to be no larger than 1024. I then iterate through all divisors for the maximum of each dimension. This will find the parameters, which maximize the work-group size, as only few of them are found in the grid. These include $(1024, 1)$, $(512, 2)$, $(128, 8)$ etc. This will add a few duplicate entries, but they are easily filtered by converting the list to a `set`. Performance for this generator is not a concern, as any time used by the generator, will not be included in the benchmark's time.

The same script is modified to support 3 dimensions, and finds 150 unique combinations. The only change, is to now find the product of all three dimensions, and at least one of these has to be above 32. This script can also be found in the appendix C.1.3.

2.3.1 Testing

Bohrium does, in its current state, not support dynamic work-group sizes, as they are pre-defined in its configuration files. It is possible to control the work-group size Bohrium will use for all kernels, through an environment variable.

When Bohrium is set into verbose-mode, it will provide statistics for each kernel. This includes the kernel hash, the number of times it has been called, and the amount of time the kernel takes to execute. We can use this to tune every kernel individually. The verbose-mode does have an impact on performance, but I have no reason to assume, that it will affect the timing of the kernels.

To gain finer control of the kernel parameters, I make a small patch to Bohrium to hijack the function which loads the kernel parameters. I change it, to look for environment variables in the format of: `BH_OPENCL_(kernel hash)_D(XYZ)`, where `kernel hash` is replaced by the unique hash generated by Bohrium for each kernel, and `(XYZ)` refers to one of the three dimensions.

I also add two places in the Bohrium code, where it will output some meta data about the workload size.

I write a Python script, which will be the auto-tuner. The auto-tuner is completely separate from Bohrium, but latches on from the outside. It will start each benchmark I want to test, and parse the output of Bohrium. The benchmarks are run one at a time. For every benchmark, I try every possible kernel parameter, corresponding to the types of kernels. If the benchmark contains 3-dimensional kernels, it will run more iterations than the ones that only contain 1-dimensional kernels.

It will start by making a few warm-up rounds to let the OpenCL run-time compiler get up to speed. Then I make a dry-run without providing any parameters, which makes it use Bohrium's default parameters, to get the kernel hashes, and a baseline to compare to afterwards.

For every set of kernel parameters, I do three runs to rule out random noise in the testing. I then save the lowest time, as this reflects the best possible performance. We are not concerned with one-offs with a really low execution time. We will manually inspect the results to find general relations between kernel parameters and execution-time. And as described in 1.9.4, the difference between tests are minimal.

2.3.2 Results

I run the tuning on the nine selected benchmarks from the Benchpress package[2]. The auto-tuning is run on the 30 second benchmarks, and takes more than 24 hours for all kernel parameters to be tested.

The nine benchmarks have a wide range of kernels. Some have tensors of rank 5, while others, like Monte Carlo Pi, uses just vectors. As explained earlier, OpenCL can only parallelize up to rank-3 tensors, while the ranks above this, are simulated using for-loops. The Magnetic Field Extrapolation benchmark, for example, contain several kernels with rank-5 tensors.

While the benchmarks run, the auto-tuning script writes down the result of every test. As the auto-tuning takes a very long time to complete, I save the results of everything, so it can be used for inspection later. It is much faster to write down any information that might be useful, than to run the auto-tuner again.

I have written a post-processing script, which loads up all the results, and generates valuable reports about each kernel, as well as a graph of the optimizations, which we will see at the end of this chapter.

Some benchmarks generate a lot of kernels, most of which are small, and unaffected by the auto-tuner. As we want to find a way to predict the best kernel parameters, I want to find the kernels, which benefit the most from the auto-tuner. To find these kernels, I find the standard deviation of the times it took for each parameter to complete each kernel. I then filter out all the kernels, that have a standard deviation below 0.0001, as these were mostly unaffected by the auto-tuner.

I then get a range of interesting kernels from each benchmark. Some of them might have such a low running time, that they did not matter in the total time spent, but they are still interesting for the generalization of good parameters.

I will now take a look at the results from a select few of the benchmarks.

Shallow Water

This benchmark saw a staggering improvement from 38.72 seconds down to 9.71 seconds! Note, that the 30 seconds, that the benchmarks are tuned for, does not take into account the overhead of the profiling tool needed for auto-tuning.

The most influential kernel, is the one shown in the following table:

`3cb0b5a5bd732842`		
X-dim	Y-dim	Time
1	96	3.82
1	160	3.82
1	320	3.82
1	352	3.82
1	416	3.82
128	1	19.5

Table 2.1: Results for auto-tuning Shallow Water. The five best parameters are shown. The bottom parameter, is the Bohrium default. The top row shows the kernel hash for identification.

Shown in the table is only the five best parameters, chosen by the script. Notice, that the time does not differ much between each parameter. This signifies that the exact parameter, does not matter. This gives some leeway for when we are choosing the optimal kernel parameters later.

At the bottom is shown the parameters, which Bohrium would otherwise run with.

It is interesting to note, that the parameters seem to be flipped, compared to what one would expect. There must be something in the kernel, which is transposed, or otherwise is best traversed in the opposite axis.

This phenomenon is seen in every kernel of Shallow Water, which is seen in this truncated table:

`df579a0df7f58600`	X	Y	Time
Best	1	416	1.38
Default	128	1	2.6

`febb5192f3448ed2`	X	Y	Time
Best	1	416	1.38
Default	128	1	2.82

`22b59eadf8ea7bb6`	X	Y	Time
Best	1	32	1.29
Default	128	1	4.8

`c49a849cb8fbba2a`	X	Y	Time
Best	1	976	0.265
Default	128	1	2.08

`b3827a02c86a52d9`	X	Y	Time
Best	1	32	1.29
Default	128	1	5.04

`86a23d5f8ff305a3`	X	Y	Time
Best	1	992	0.264
Default	128	1	1.86

Table 2.2: Results for auto-tuning Shallow Water. The six are part of the best optimized kernels for the benchmark. The top-left cell shows the kernel hash for identification.

The parameter shown for each kernel, was the absolute best, but in the same way as before, the neighboring parameters, had the same or almost exact same time.

Leibnitz Pi

I have chosen to show a one dimensional kernel, which does not use any global memory. The time with default parameters is 13.04 seconds and the best optimized time is 12.89 seconds.

`7408ba119f94d3a8`	X	Time
Best	576	2.69
Default	128	2.74

`2f8e12939fa29264`	X	Time
Best	320	10.2
Default	128	10.3

Table 2.3: Results for auto-tuning Leibnitz Pi. The two kernels are the only ones in the benchmark. The top-left cell shows the kernel hash for identification.

Here, it does not change much what the work-group size is. So we might have to detect what kind of operations are performed, and if any of them are memory-heavy. But what we do see, is that larger work-group sizes are preferred, hinting, that 128 might be too low in general.

Lattice Boltzmann

The last benchmark, I want to show, is one that contains a 3-dimensional kernel. The interesting part here, being the peculiar placement of work-items, which resembles what we saw on Shallow Water.

`67a4815d31d50908`			
X-dim	Y-dim	Z-dim	Time
1	1	64	0.227
1	1	48	0.239
1	1	32	0.251
1	16	48	0.311
1	16	64	0.318
128	1	1	2.96

Table 2.4: Results for auto-tuning Lattice Boltzmann. The five best parameters are shown. The bottom parameter, is the Bohrium default. The top row shows the kernel hash for identification.

Here we see, that the parameters are maximized on the third dimension – note, that the maximum size for the third dimension is 64, while the first two are 1024.

This furthers my suspicion, that something must be transposing the parameters. I have double checked the parameters, that are supplied to OpenCL, and they are correct, but inside the kernel, the dimensions could be used wrong.

2.4 Determine the Best Parameters

From the tests above, we proved the need for optimizing the kernel parameters. In several tests, we get a quite significant speed-up. But what technique should we utilize to replicate this during run-time? We obviously want as simple a method as possible, which provides the smallest difference compared to the most optimal parameters we have found.

It is important, that in the search for optimal parameters, that we do not optimize a single benchmark, but make decisions, that will generalize on new, unseen benchmarks.

We have almost performed an exhaustive search for all valid kernel parameters for a set of benchmarks included in the Benchpress package. The benchmarks provide a real-world look into how kernels can be constructed, so we do not base our assumptions off of artificial test code. In a sense, we can make our tuning fit the benchmarks that are proven examples of code we want to run, instead of fitting the tuner to some arbitrary functions we want to run faster.

Here is an example from the Heat Equation benchmark of one of the most significant kernels. The five best kernel parameters shown here have identical times, but the absolute best was found to be be the first one.

The maximum value for the kernel parameters on the X and Y axis are 1024 for the GPGPU we are using (see appendix A.1). This could signify, that a kernel of this type simply benefits from maximizing the Y parameter. A peculiarity of this, is how it's usually the X axis, which is maximized to improve memory coalescence, but as I have speculated before, there might be something, which inverts the axes.

`7bf70a423c305952`		
X-dim	Y-dim	Time
1	928	1.71
1	688	1.71
1	960	1.71
1	976	1.71
1	1024	1.71
32	4	22.1

Table 2.5: Results for auto-tuning Heat Equation. The five best parameters are shown. The bottom parameter, is the Bohrium default. The top row shows the kernel hash for identification.

To compare, I also included the default parameters for Bohrium, which perform quite a lot worse. Do notice the more than tenfold increase in the times recorded.

When inspecting the blocks for the kernel, it shows linear map-like operations. Meaning, it takes in one element per work-item on the same data/cache line as the rest of the work-group, performs a series of arithmetic operations, and puts it back into global memory.

Kernel `7bf70a423c305952` in Bohrium's Internal Blocks

```
rank: 0, size: 11000, block list:
    rank: 1, size: 11000, news: {a7,a8,}, frees: {a2,a8,},
       temps: {a8,}, block list:
        BH_SUBTRACT
            a8[0:11000:11000,0:11000:1]
            a2[0:11000:11000,0:11000:1]
            a1[1:11001:11002,1:11001:1]
        BH_IDENTITY
            a1[1:11001:11002,1:11001:1]
            a2[0:11000:11000,0:11000:1]
        BH_ABSOLUTE
            a7[0:11000:11000,0:11000:1]
            a8[0:11000:11000,0:11000:1]
```

This is pretty straight forward, and it's not hard to see why maximizing one dimensions would be the better option, as it increase potential parallelism and latency-hiding.

But we cannot just accept, that we maximize the highest dimension, as there is a problem with this technique. The Z axis on the available GPGPU, has a limit of 64 elements. This limits the amount of latency-hiding that is possible. And what happens, when we generate a kernel of four or five dimensions? The hardware does not support to parallelize more than three dimensions.

I take a look at the OpenCL code for the kernel above, to see if I can identify the issue, as it would still be assumed, that X should be maximized. The kernel code is shown below.

Generated Kernel for `7bf70a423c305952` Modified to Show Strides as Constants

```
const uint g0 = get_global_id(0); if (g0 >= 11000) { return; }
   // Prevent overflow
const uint g1 = get_global_id(1); if (g1 >= 11000) { return; }
   // Prevent overflow

{const ulong i0 = g0;
  {const ulong i1 = g1;
    double t0;
    double s1_1; s1_1 = a1[ +i0*11000 +i1];
    t0 = s1_1 - a2[11003 +i0*11002 +i1];
    a2[11003 +i0*11002 +i1] = s1_1;
    a3[ +i0*11000 +i1] = fabs(t0);
  }
}
```

The problem might not be immediately apparent, but it lies in the index calculations.

As I described in section 1.6.3, GPGPUs have a concept called *memory coalesce*. This means, that neighboring work-items must retrieve values right after each other, to gain the full memory throughput of the device. From the index calculation, we can see the reason why the kernel parameters are best, when the highest dimension is maximized.

The stride of the array defines, that values for `i1` are separated by one for every element. As we run with the default parameters, the memory accesses pattern will read values in small groups of four, at 32 different locations, 11000 values apart. This means, that almost no memory coalesce is achieved.

With a bit of luck, the cache might alleviate the performance drop, but it is not a preferable way to arrange memory access.

Quite unintuitively, we want `get_global_id(0)` to be assigned to the inner-most iterator (`i1`) and `get_global_id(1)` to the outer-most (`i0`). This will invert the work-group parameters, and we can easily make the rule, to always maximize the first dimension of the kernel parameters, as we would expect would be preferable.

This easy solution of maximizing the inner-most rank, works best for 2-dimensional kernels, and worse with 3-dimensional kernels. But the problem is, with the current version of Bohrium, it is not possible to parallelize the inner-most dimension, if the kernel has more than three, as Bohrium will start inserting for-loops for the inner dimensions.

I realize, that the auto-tuner mostly just undoes the incorrect index calculation. And we cannot make any conclusion on the effectiveness of the auto-tuner before fixing this issue, as the results will be clouded by the other effect. Therefore; I will put the auto-tuner aside, and revisit it, when I have improved on Bohrium.

2.5 Benchmarks

The benchmarks discussed in this chapter were chosen, because they had some interesting abnormalities. But I want to set a baseline, as for how far the auto-tuner can improve performance.

I had made the oversight, not to measure the "copy-to-device" time, provided by Bohrium. This meant, that all benchmarks with a lot of copying, got unfairly favored in the test – incidentally, this included none of the discussed benchmarks above. As I mentioned earlier, the verbose-mode I enabled to get detailed information about kernels, also has a slight impact on performance.

As it takes more than a full day to re-run the auto-tuner, my solution to both issues, is to extract the optimal kernel parameters from the data I have collected, and run each benchmark once more, but this time, it will record time that can be directly compared to the regular benchmarks.

In the bar chart in figure 2.1, all of the benchmarks are compared to their optimized times.

As we can see, we get some quite extraordinary results. As shown earlier, heat equation is brought down by 90%, while Magnetic Field Extrapolation and Shallow Water see benefits in the 60% range. The 17% and 10% we see for N-Body and X-Ray Sim are also quite impressive. If I had not seen the result of the first three, I would still have thought the boost for N-Body and X-Ray Sim would have been worth it.

These execution-times, should be seen as the unbeatable, optimum of performance, we can get from changing kernel parameters, as we have tried every reasonable setting, and found this to be the best. This of course does not included added features, like vector-reductions, which will change the playing field, and we will have to run the auto-tuner once more, to find the new optimum. But for our work with optimizing the kernel parameters, these are the execution-times we are striving for.

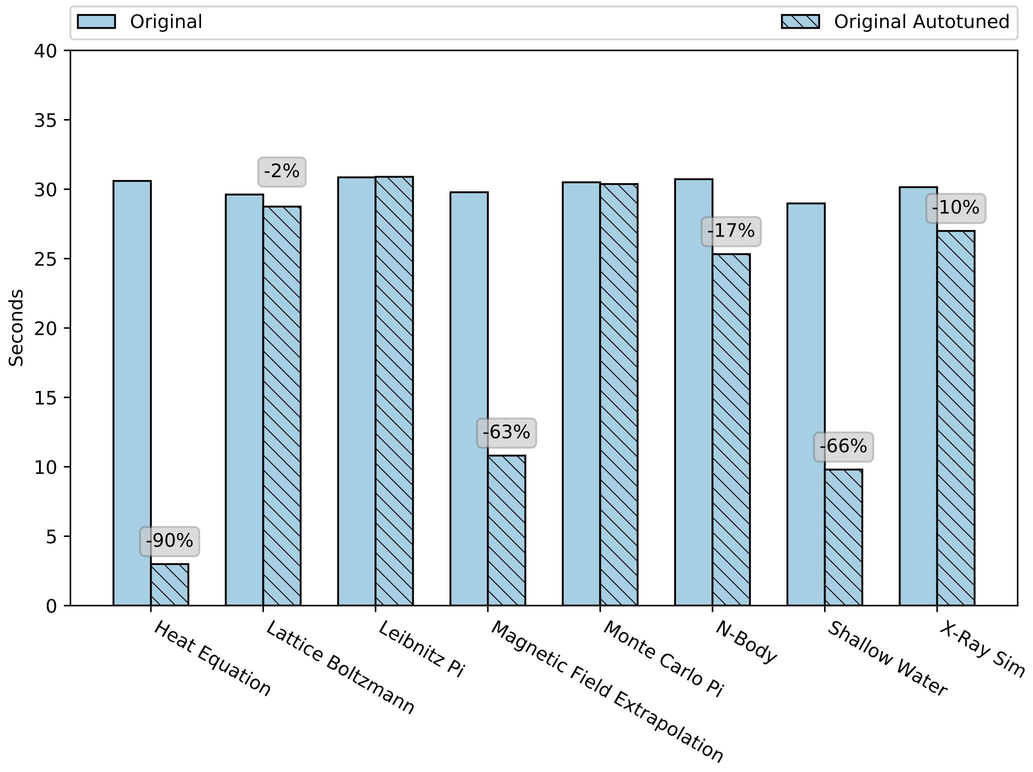

Figure 2.1: Overview of auto-tuned benchmarks. Showing original and latest version of Bohrium as reference.

2.6 Partial Conclusion

The auto-tuner is able to give a staggering performance boost. In the best case, reducing the execution time to a tenth.

Although we find promising results, we have run into an issue, which preexisted in Bohrium. The strides for all memory access are inverted, meaning that all memory access is currently uncoalesced, when the kernels have more than one dimension. This issue inhibits the auto-tuner, from selecting good parameters, when the dimensions rise above 2.

The auto-tuner is set aside for now, even though it only made it into its initial development phase. We have yet to make a system, which will on-the-fly choose the optimal kernel parameters.

I will revisit the auto-tuner, as the issues get sorted out.

Chapter 3

Vector to Scalar Reduction

A reduction is a common parallel primitive, when operating with array operations. Many sequential algorithms can be transformed to parallel array operations, by expanding intermediate results into a second dimension[37]. To get the result, a reduction is often used to transform the intermediate results into the final result. This paradigm might seem abstract, but we just have to focus on the reduction itself for now.

The operation is often used to find the minimum, maximum or summed value of a vector, or along an axis in a matrix. But reduction in not limited to these three operations. Any associative operator can be used to reduce a vector to a scalar. We will later see an advantage of requiring our operators to be commutative as well, even though it is not generally required.

Our aim is to out-perform the current implementation, as it defaults to using the CPU, when a vector-reduction is unavoidable. Apart from the fact that the GPGPU might be faster than the host CPU to reduce a large range of numbers, it also adds time penalties, when data has to be moved between the two.

At the end, we will do some benchmarks on the new version of Bohrium, and the version, which it branched off from. This way we can compare the two, to see if we reached our goal.

We will focus on vector-reductions, as Bohrium currently supports reducing multi-dimensional data, parallelized along an axis.

A late section of this chapter has been moved to appendix D. It describes a way to support vector-reductions, in kernels that are spawned with ranks higher than 1. This will give the fuser more freedom in the future, and caused some surprising performance gains.

3.1 Related Work

A reduction is a thoroughly research subject, which has attracted attention from both Nvidia and AMD to write articles about[38][33].

Both Nvidia and AMD demonstrates, how to write a vector-reduction kernel in CUDA and OpenCL. Both companies agree on the algorithm to use, which focuses on memory through-put with vector-types, reusing work-items, solving memory bank conflicts, and optimizing occupancy of work-items in work-groups. Most of which we will also discuss and demonstrate in this chapter. Reusing work-items and using vector-types are out of the scope of this thesis, as it will have to be supported everywhere in Bohrium to work with reductions.

The final method found in section 3.2.3, will be equivalent to the method of Nvidia and AMD[38][33].

With CUDA, some API calls are available, which simply cannot be replicated in OpenCL. Nvidia's Kepler architecture, includes a feature called "shuffle", which is different from OpenCL's shuffle. OpenCL's shuffle creates a random permutation of elements. CUDA's shuffle, allows for transferring the value of a register to neighboring work-items in a wavefront, without the use of local memory[43]. If the reductions are later ported to CUDA, this feature would be simple to implement, as it directly replaces a read and write from local memory.

3.2 Parallelizing a Reduction in OpenCL

There are two classical ways of parallelizing a reduction. Either divide-and-conquer, or tree reduction[33].

The divide-and-conquer is favorable on a CPU, as it requires very little memory to keep the accumulator. The tree-reduction on the other hand requires scratchpad-memory, to keep the result of every level of the reduction.

The problem with divide-and-conquer, is that, we need more elements, than work-items in a work-group, to have something to divide. Normally in Bohrium, we get the input values from a previous calculation, because of the fuser, which is normally scheduled as one value per work-item. This means, we will need to serially reduce each work-group's sub-result with only a handful of threads, to make a real divide-and-conquer.

The tree-reduction works in parallel all the way down to the last element. But the downside is, that we at every iteration take away half of the potential parallelism – but this is still better than a fully serial reduction.

The best method of reduction, might be to make a hybrid of the two techniques. One where we limit the parallelism as much as possible, but keep the GPGPU cores fully saturated with a divide-and-conquer, and when it's needed, we switch to a tree-reduction.

Communication is only possible inside a work-group. Therefore, we will have to reduce a section of the input data in each work-group, and later combine the sub-results into a single scalar. We will therefore start by focusing on work-group reductions.

Finding the most optimal reduction is likely a problem of memory-access pattern to maximize memory throughput. Meaning, how can we most efficiently retrieve all values, and communicate the result across a work-group? Contrary to a computationally bound kernel, a reduction will require a single ALU operation between every element, which is almost the minimal amount of work one can perform.

I started with a separate implementation in pure OpenCL, following the procedures discussed in the following sections, before integrating it in Bohrium.

3.2.1 Naive Tree-reduction

A parallel reduction can be performed in several ways. The most obvious is shown here:

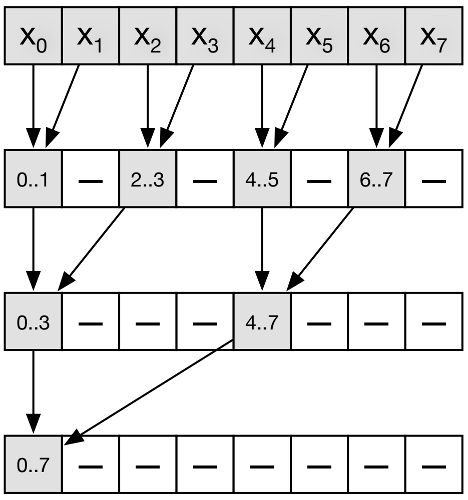

Figure 3.1: Simple tree-reduction. Operator is applied when merging two cells.

Imagine, that we use one work-item for each of the input elements. This way, every second work-item reads in two elements, apply the operator, and replace its own element with the new value. Then at the second iteration, every fourth work-item fetches two values and applies the operator. This repeats until only one value remains.

I have set up a simple benchmark, which tries to reduce the maximum number of 32-bit floats, which can practically fit on my GPGPU – 805,306,368 numbers equivalent to 3 GB of data. The workload size is smaller than the 4GB the GPGPU has available, to avoid a slow section at the end of the GPGPU's memory. This is unique design mistake to this specific GPGPU, and not a general concern[14].

As I try different methods, I will perform the same benchmark. I chose a large amount of numbers, as this increases throughput, reduces importance of constant schedule/allocation factors, and shows a larger separation between methods.

I also verify, that the result is correct. I will go into details with this in section 3.3.5.

I also compare the tree-reduction with a slightly more sophisticated method, where we only use the absolutely required amount of barriers. I do this, by first running a barrier-free tree-reduction inside each wavefront. When each wavefront has one result, the rest of the reduction is performed as tree-reduction with barriers.

Regular	15.88 GB/s
Optimized	34.37 GB/s

This technique does find the right result, but it doesn't perform well on modern GPGPUs at all.

This reason for this, might be because of the way wavefronts are scheduled on the GPGPU. As described in 1.6.3, these are scheduled in groups of 32 or 64 work-items, even if only some of them have work to do. If only 1 of 32 work-items are busy, then the computational performance is limited to 1/32.

Blowing up the example in figure 3.1 to a larger dataset, we will have our reduction spread sparsely all over the work-group. Every wavefront will have a utilization between 50% and less than 1%. Although with the advantage, of not needing any barriers inside each wavefront.

Another issue is memory banks of the local memory. I will not go into great detail, as it is easily solved with the next algorithm. Each work-group accesses local memory through the same 32 memory banks (32 for Nvidia GPGPUs [44, 3.2.2.1 Shared Memory and Memory Banks]). If two work-items use the same bank, it will cause a conflict, and the transactions will be serialized.

Banks are distributed as the address modulo 32. In the tree reduction, we have the unfortunate case, that the same work-items are active in each wavefront. This actually means, that every iteration will cause an n-way conflict, where n is the active wavefronts. For a work-group of 128, n will stay at 4 until the second last iteration.

3.2.2 Compact Tree-reduction

I propose an alternative memory access pattern. Instead of keeping the values spread out over the work-group, could we compact it inside as few wavefronts as possible? That way, the wavefronts which are active, will have a high utilization, while the inactive are quickly disregarded. We also solve the problem of memory banks, as with every iteration, we will half the active wavefronts, and thereby half the conflicts.

The compacting does come at the expense of barriers at every iteration, until we are within a wavefront.

With this method, the utilization will either be 100% or 0% for the active and inactive wavefronts – provided the input length is divisible in the wavefront size. This should increase

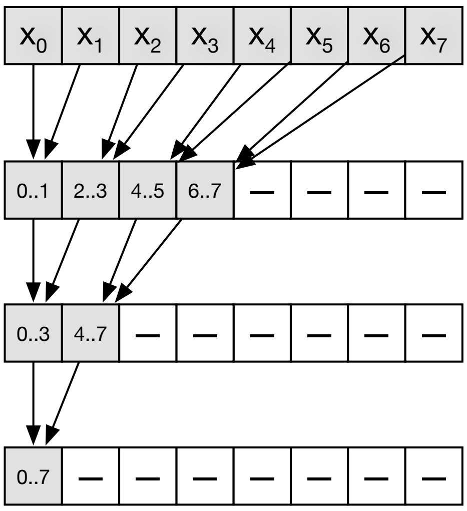

Figure 3.2: Compact tree-reduction. Operator is applied when merging two cells.

performance, as the scheduler can quickly skip the empty wavefronts.

I perform the same test as before with just the reduction method changed. I do the same modification as well, to only use the absolute necessary amount of barriers. This time, the barrier-free reduction has to be done at the end of the reduction, instead of the first part:

Regular	44.75 GB/s
Optimized	66.47 GB/s

This method proves to perform quite a lot better, which means, the additional barriers are less damaging to performance, than less wavefronts and less bank conflicts.

3.2.3 Divide-and-Conquer Hybrid Reduction

We still have a bit of an issue with utilization. Even though each wavefront reaches 100% utilization, it only does so for a single operation before half of the wavefronts are discarded, and we have to call the barrier again. Would it be beneficial to have fewer, but more long-lived wavefronts to reduce the amount of barriers and rescheduling?

This technique draws a parallel to a divide-and-conquer algorithm. If we assume that we have enough parallel wavefronts for latency-hiding, we can make every wavefront reduce several iterations worth of input data, before being discarded, and spend less time churning through wavefronts – see figure 3.3.

The factor here is, how many wavefronts should be combined? This is a balance between several unknown factors of the GPGPU hardware, and can only practically be determined by benchmarking – there are a lot of time factors, that we do not know.

I perform the same test as the previous two methods to see the throughput, and determine which amount of elements to reduce at the same time. Do note, that the 2-element version is almost equivalent to the compacting tree-reduction:

2-element	66.06 GB/s
4-element	84.86 GB/s
8-element	85.20 GB/s

It shows, that 8 elements is the optimal by a small margin, when operating on a very large dataset. As it is quite rare for a reduction at this size, I also try to run the test with just 524,288 elements ($1024 \cdot 512$):

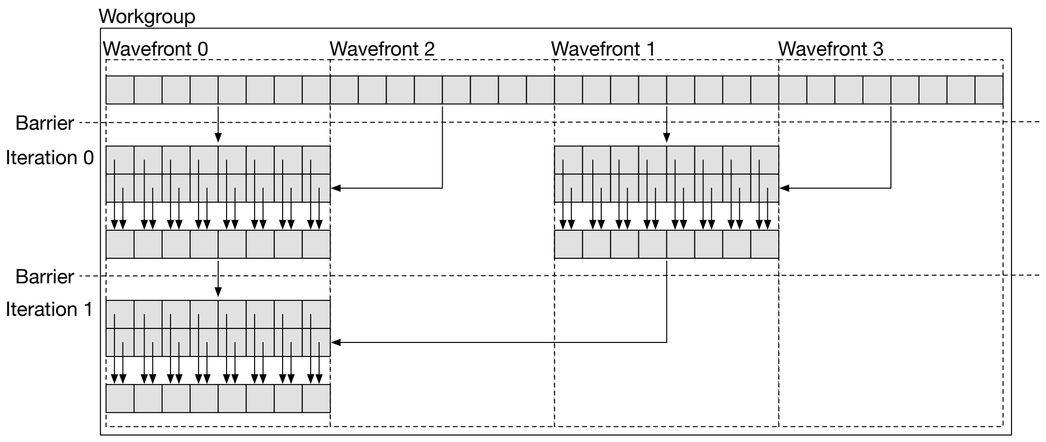

Figure 3.3: 2-element divide-and-conquer inspired reduction. Beware of wavefront numbering, which follows implementation.

2-element	37.43 GB/s
4-element	45.02 GB/s
8-element	44.85 GB/s

Even after several re-runs of the benchmark, the 8-element type always lacks behind by an equally small margin.

I choose to use 4-elements, as this might generally be better, and the difference is neglectable. It also allows for smaller work-group sizes if needed by the auto-tuner.

If the data is not divisible in sections of 4, we will do it for as many iterations as possible, and finalize with a regular 2-element reduction.

When the work-group's data is reduced to the size of a wavefront, we have to use a compacting tree-reduction to get to a scalar for the current work-group.

3.2.4 Completing the Reduction

The OpenCL/GPGPU implementation of the vector-reduction ends up being much more complicated than its traditional CPU counterpart. The reduction need two phases: Pre-reductions, which creates the sub-results, and post-reduction, which combines all the sub-results.

The first phase has two stages. First the rough data-churning of divide-and-conquer, where utilization is maximized. In this stage, only the first quarter of wavefronts in a work-group is active. They take in the data, which were originally read and modified by the three following quarters. They reduce the input by a factor of four, and write it to local memory. This continues recursively until the input data is reduced to the size of a single wavefront.

Then we use compacting tree-reduction, as we do not have to call barriers, when we are within a single wavefront. The scalar from each work-group is placed in global memory. The result we get, is a vector at the length of the number of work-groups. This vector is stored in global memory, to be persistent between enqueued kernels.

One of the limitations of modern GPGPUs, is the fact, that we cannot communicate across work-groups. It is somewhat possible through atomic operations, but it does have a lot of limitations.

What this means for our reduction, is that, we have to find a way to combine the result of every work-group to get a single scalar.

I propose an approach of running a second pass with a reduction kernel, which only runs a single work-group. Thereby, this single work-group will be able to read in all the sub-results from the previous work-group, and give a final result.

The second kernel is similar to the first one, but requires a slightly different strategy, so we cannot just reuse it. Often, we might find, that the number of work-groups is larger than the maximum number of work-group items. We will have to implement a way for each work-item to take care of an unlimited amount of sub-results.

We need to reduce the sub-results to the size of a work-group. I do this by virtually reshaping the sub-results into a matrix of work-group width. This way, I can retrieve one row at a time and reduce it in each work-item's accumulator – see figure B in the appendix.

With a workload at the size of a work-group, we can use the divide-and-conquer approach followed by the compacting tree-reduction. Just like the first kernel. The final result can now be placed in the array provided by Bohrium, to complete the reduction.

The reduction is now complete. A diagram of a full reduction with all optimizations, can be found in appendix B.

3.3 Integrating with Bohrium

A vital part of the project is to implement the reduction in the Bohrium project, so it can get to actual use.

The advantage of Bohrium, is how all the optimizations are performed without any intervention from the end-user of the library. In the same sense, the integration of vector-reductions, should be completely transparent to the end-user.

Currently, Bohrium doesn't support all vector-reductions on the GPGPU. On a standard configuration, any vector-reduction with less than 64,000 elements, will be transferred to the CPU – including instructions immediately preceding it because of the fuser. When the reduction passes 64,000 elements, it gets reshaped into a 2D matrix, so the GPGPU can reduce along an axis. This results in a vector of sub-results, much like the result of our first kernel in the 2-pass technique. These sub-results are transferred to the CPU to finalize. This might introduce extra data copying, which takes time, especially if the result wasn't destined for the CPU anyway, and has to be transferred back.

I will start with a limited case. Bohrium has a concept of "sweeps", which are operations performed an axis of the calculated data. This is currently only reduction or prefix-sum operations. To start out, I will only allow for a single sweep for each kernel, and the sweep has to be a reduction.

3.3.1 Getting Access

I started by making a single `sum` over a vector in Python, as a simple test-setup:

```python
import numpy as np
data_size = 64000
a = np.arange(data_size, dtype=np.float64)
print(np.sum(a))
```

Currently, this would reshape the data to a 32000×2 matrix, which is reduced along the short axis. This produces 32000 sub-results, which are automatically moved to the CPU to be reduced to a scalar. This is done to potentially optimize performance, when large vectors would otherwise have to be reduced on the CPU. The limit of 32000 is configurable in Bohrium. If the limit is set to 0, the functionality is disabled.

I implement a bypass for this configuration, to get Bohrium to generate an OpenCL kernel anyway – even though it would calculate the wrong result. The generated kernel looks like a matrix-reduction along the column axis, although there is only one element per axis. This would generate a sub-result equal to the input, as no operator is applied.

I then add a header file, which contains my implemented reductions. But I can't just call my reduction. I have several properties to extract from the generated kernel: Operator, data-type, neutral element, variable with result from preceding operations and output address in global memory for the result.

I modify the functions, which decides that the reduction can be performed in the outer axis, to also identify the operator (`add`, `multiply`, `min`, `max`, `and`, `or`, and `xor`), which are all associative *and* commutative, as well as the destination for the result to be placed in.

From the operator and result-destination, we can also derive the neutral element and the data-type. We will need this to pad the reduction to a multiple of the work-group size.

The last part is to get the result of the preceding operations – the input-value to feed into the reduction. The challenge here, is finding the variable name itself, but also making it accessible, as it gets declared inside a scope, confined by a set of curly-braces.

```
{
    int a = 0;
}
printf("\%d", a);
```

In the C-code above, the variable `a` will not be defined outside the curly-braces. In this case, it should be simple enough to move the declaration outside the scope of the curly-braces, but it's not that simple in Bohrium.

In the current structure of the code-generator, it is not known what auto-generated name the variables will get before it is too late – when we have already opened the curly-brace and went down a few nested function-calls.

To avoid restructuring too much of the code, I add a hardcoded variable called "`element`" which is defined at the beginning of the kernel, when a vector-reduction is detected. After all operations preceding the reduction has completed, the value which will be reduced, is assigned to the element variable.

I also the destination of the reduction, which is a pointer to global memory, where the output of the reduction should be placed. When a reduction is smaller than or equal to a work-group size, the destination will be the array Bohrium expects the result, and we will skip the second kernel pass. Otherwise, it will contain a temporary array in global memory for each work-group to place its sub-results.

The solution is not perfect, as it has been implemented without changing the internal code blocks, but rather patching on modifications in the code. But it would otherwise require a significant amount of work, as well as coordination with the other Bohrium developers, to implement. This can be changed, if the code proves to be worth keeping.

3.3.2 Bypass Overflow Prevention

The next challenge, is when we meet an input-size, which doesn't evenly divide into the work-groups. Currently, Bohrium has an overflow-guard at the top of every kernel:

```
// The IDs of the threaded blocks:
const uint g0 = get_global_id(0); if (g0 >= 10) { return; } //
    Prevent overflow
const uint g1 = get_global_id(1); if (g1 >= 10) { return; } //
    Prevent overflow
```

In this case, the input data is a maximum of 10×10. Any work-item spawned above this, will return immediately. Until now, that hasn't been a problem.

Reductions are a little different, as for the first time in Bohrium, there is actually communication between the work-items in a work-group. My implementation assumes all work-items in a work-group are available.

To allow for a much simpler implementation of the reduction, I choose to keep all work-items, but assign the redundant ones with the neutral element. This means the extra work-items will not change the result, and will still be able to take part of the reduction. Their presence might very slightly slow down the reduction, but we save time on unnecessary logic in all the other work-groups, which could slow down the reduction over-all.

I do this quite simply by leaving the outer most guard open, and closing it after all calculations are done.

```
// The IDs of the threaded blocks:
const uint g0 = get_global_id(0); if (g0 >= 10) {
const uint g1 = get_global_id(1); if (g1 >= 10) { return; } //
    Prevent overflow
     // Calculations ...
     element = foobar;
}
else{
     element = NEUTRAL;
}
reduce_2pass_preprocess(element, ...);
```

Although demonstrated in the code above, the vector-reduction will for now, not support multi-dimensional kernels. This is both from an implementation standpoint, but also as the fuser currently avoids fusing vector reductions into kernels with more dimensions.

To support it in the reduction, we will have to handle a distribution of work-items, where not all are placed in the first dimension.

A solution to this, can be found in appendix D.

3.3.3 Modifying the Header File

The last issue at hand is how we define the data-type of the reduction. All code for the kernels are generally auto-generated. The code for the kernels are almost completely in-lined in Bohrium's C++ code. More complicated functionality is placed in header-files, which is imported into the OpenCL kernel.

My implementation doesn't fit into either of these boxes. My `reduce_opencl.h` file contains three functions and one kernel. All of which contain the data-type of the reduction several places – both in the function declaration and in-lined in the code. This is where OpenCL, which is based on C99, falls short. This scenario is supported in CUDA and OpenCL 2.1 with the support for C++ templates, which is not available on my available GPGPU, and would unnecessarily limit potential users if required.[15, templates]

It is also not possible to define a variable to represent a data-type in C99. Neither would it be practical to write a function for every input type.

As described earlier, I will start by only supporting a single reduction per kernel. This allows me to define the data-type, as well as neutral element and operator, as macros at the beginning of the kernel, and then import the `reduce_opencl.h` file.

```
#define NEUTRAL 0
#define OPERATOR(a,b) (a + b)
#define __DATA_TYPE__ ulong
#include <kernel_dependencies/reduce_opencl.h>
```

From here, we have all the prerequisites we need to start a reduction. These definitions will fill in the blanks in the header file. Here is seen one of the auxiliary functions:

```
inline __DATA_TYPE__ reduce_wave(__DATA_TYPE__ acc, __local
    __DATA_TYPE__ *a, size_t limit){
    size_t lid = get_local_id(0);
    bool running = ((lid%2) == 0);
    for (size_t i=1; i<=limit/2; i<<=1){
        if (running){
            running = (lid%(i<<2) == 0);
            acc = OPERATOR(acc, a[lid+i]);
            a[lid] = acc;
        }
    }
    return acc;
}
```

3.3.4 Allocating Memory and Running 2 Kernels

In the code above, you might have noticed, it references "__local" memory. None of the existing kernels Bohrium can produce, utilizes local memory explicitly – although variables might be implicitly spilled into local memory.

As described earlier, the reduction will in several of the stages communicate through local memory with the other work-items in the work-group.

It is possible to declare usage of local memory directly inside a kernel's source code. But I chose to avoid it, to make it more interoperable with my auto-tuner – otherwise requiring a recompilation at every change. Instead I added some code to Bohrium, which will dynamically allocate the local memory when needed. Below is shown the post-processing setup, as it is slightly more readable:

```
cl::Kernel post_reduction = cl::Kernel(program,
    "reduce_2pass_postprocess");
post_reduction.setArg(i++, reduction_local_size*dtype_size,
    NULL); // Allocate local memory for reduction
```

I chose to implement it directly after the lines executing the regular kernel, using the same information as for when we generate the kernel, as we need the size of the data-type. If we had just performed a vector-reduction, it would also enqueue and execute my post-processing step before going back to the rest of the code.

Contrary to letting it be part of Bohrium's internal representation of blocks and kernels, this is not an optimal implementation. I would favor a more generalized approach going forward, where the internal representation of an operator can require local memory. It should not matter to performance, but rather to the clarity of the code.

Bohrium also does not support a single instruction to generate multiple kernels. It would have been favorable to modify the code-generator to natively support these irregular kernels.

Just like before, this would require a rewrite of some of the data structures and logic, and coordination with the other developers.

I also added, that if the data-size fits inside one work-group, it bypasses the second kernel, as it is no longer required. This is done in Bohrium by simply swapping the destination buffer with the temporary. My benchmarks from the earlier tests, show that it could save about half the time spent, when the data size is this small, as the actual processing is neglectable.

3.3.5 Correctness

Before we can continue, we will have to see, that the reduction also finds the correct results.

Bohrium has a set of tests already, which my implementation runs straight through with success. But I will make a few tests myself, because I want to quantify how much my implementation differs from NumPy. I will go more into detail with testing in chapter 8.

NumPy's Precision From my previous work, I had found that NumPy could find a result several orders magnitude off [47, 4.1 Correctness]. This seems to have changed. In version 1.13 of NumPy, they changed their method of summing numbers, to use a pairwise summation[10]. This has a significantly better numerical stability, and it is similar to our reduction kernel.

In NumPy, summing a vector, used to be a for-loop which ran through all elements, while adding them to an accumulator. This has the side-effect, that a data type of float will not always find the mathematically correct result. It is of course well known, that floating-point numbers have a certain inaccuracy, but this inaccuracy gets accumulated as well.

This is not to say, that NumPy was *wrong*, it is more of an unexpected or undefined behavior. The problem has its roots in how floating-points of a significantly different magnitude gets added together.

Precision of Floats Single-precision floating-point numbers, defined by IEEE 754, has a finite set of bits allocated to the exponent and fraction. These are 1 bit for sign, 8 bits for the exponent and 23 bits for the fraction. To calculate the number it represents, we place each section of bits into this formula:

$$(-1)^{sign} \cdot 2^{exponent-127} \cdot \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

The b_n represents the nth bit in the fraction.

What it effectively means, is that we read the sign to show whether or not the number is negative. The exponent allows us to represent very large and very small numbers, by using an exponential scale. The fraction is added as the decimal places after the exponent to allow for representation of numbers that are not a power of 2.

To illustrate the issue of the accumulator, I will show a few floating point numbers and their representation in figure 3.1.

Number	Sign	Exponent	Fraction
2^4	0	10000011	00000000000000000000000
$2^4 + 1$	0	10000011	00010000000000000000000
$2^{23} + 1$	0	10010110	00000000000000000000001
$2^{23} + 2$	0	10010110	00000000000000000000010
$2^{23} + 2^{22}$	0	10010110	10000000000000000000000
2^{24}	0	10010111	00000000000000000000000
$2^{24} + 1$	0	10010111	00000000000000000000000
$2^{24} + 2$	0	10010111	00000000000000000000001

Table 3.1: Floating Point Numbers, and the Issue of the Accumulator. The number field is in base 10, all other fields are base 2

As we can see in figure 3.1, the small numbers of 2^4 are easily represented, and adding a 1, adds a 1 at the fourth bit from the left. As the exponent rises, the 1 is moved to the right, as it becomes less and less significant to the much larger exponent. When we deal with numbers in the billions of billions, adding 1 to the number, has almost no influence on the number as a whole. This is the power and sacrifice of floating point number. We can deal with very large numbers, but we loose precision.

At 2^{23}, we have have almost run out of bits to represent a 1 in the fraction – we have exactly 23 bits for the fraction. Just for demonstration, I have shown how the fractions represents adding 2^{22} and 2 to 2^{23}.

When we move to 2^{24}, we cross a threshold, and we are no longer able to represent the addition of 1. And this is the issue of the accumulator. We can add 1 from now and to all eternity, and it will never increment the result above 2^{24}. But adding 2 to the number, will work, as it is still within the threshold.

Solution The pairwise summation, that NumPy now uses, uses the probability, that neighboring numbers, will often be of similar magnitude (exponent). We can see this from how 2 is still being represented when added to 2^{24}. If the numbers we are adding have a lot of smaller numbers, we can add those together, and maybe they will together be above the threshold of the accumulator.

Doing this in a tree-like structure, will make it more likely, that the accumulator of each branch is of the same magnitude, and improve the numerical stability.

This is an important detail of the implementation. Floating-point operations do inherently have an amount of inaccuracy. But as we are trying to replace NumPy, it will be good to get close to the same result as it.

Our reduction kernel operates similar to a pairwise reduction. On a CPU, it is actually more costly to do a pairwise reduction, as it will reduce in a tree-like structure instead of linearly apply the operator to the elements. On the GPGPU, we do not have a choice, but to do a type of tree-reduction, which might have been a blessing in disguise.

It is not a perfect solution, but if the numbers are partially or fully sorted, a work-group will be more likely to have numbers of the same magnitude, which means there is a higher chance the rounding error will be less significant.

When every work-group has found their results, the final reduction will equally be more likely to reduce numbers of the same magnitude. This doesn't mean the error is completely mitigated, but it has been reduced.

As the floating-point units from different hardware might not work perfectly the same way, and as we are exploiting the commutative law of our operator to reorder the numbers, we might not get the exact same result. But we can quantify the difference, and justify, if it is within an acceptable threshold.

The post-reduction of our second kernel also doesn't use pairwise reduction for the first part. To reduce the sub-results to a work-group size, we use the linear approach where each thread adds to a single accumulator. If it proves to be an issue, we can could use a small trick to rearrange the sub-results without adding any additional reads or writes. If the pre-processing kernel is aware of the access pattern of the post-processing kernel, we could arrange the sub-results with a stride, which made each work-item in the post-reduction operate on values that were originally neighbors. Then again, if needed, we could add a pairwise reduction to the post-reduction, but this would lead to more usage of local memory.

Results I set up a test to see if we can provoke a rounding error on both our and NumPy part. The test is to sum a vector of increasing length, where the largest is limited by the memory of my GPGPU (see appendix A.1). The other available operators are not as interesting, as they all find the right result. s I run my test with 3 different setups: 64-bit unsigned integer in NumPy as reference, 32-bit float in NumPy and 32-bit float in Bohrium. All tests use addition as operator. The 64-bit unsigned integer has capacity to make the additions with all digits intact.

By looking at the table 3.2, we can see that we actually out-perform NumPy in this test. The column following "NumPy" and "Bohrium" shows the number of correct digits.

Reference	NumPy	Precision	Bohrium	Precision
102378	102378	–	102378	–
1026928540	1.0269285e+9	8	1.0269285e+9	8
147591519778	1.475916e+11	6	1.4759153e+11	7
5428181357550	5.428182e+12	6	5.428181e+12	7
799795513068828	7.997955e+14	7	7.997955e+14	7
36028796884746240	3.6028823e+16	5	3.6028797e+16	7
441352763012546560	4.413528e+17	6	4.4135276e+17	8

Table 3.2: Correctness compared between NumPy and Bohrium. Precision is number of correct decimals.

For the first two rows, they are equal, but then NumPy losses a bit of accuracy, and is in worst case two digits off from Bohrium.

But generally speaking, they are both fairly accurate. This is the accuracy, that one would expect of floating-point operations. If a better precision is needed, there might be other data-types more fitting for the task.

But I will conclude, that the implementation is acceptable, as it passes Bohrium's own tests, and has a fair numerical stability.

3.4 Adding Prefix-sum

I was not originally planning to include a prefix-sum (also called scan) as part of the code. But I realized, I could reuse the whole pre-processing kernel, and most of the post-processing kernel to add this functionality.

I won't go into much detail of the implementation and algorithm for parallelizing prefix-sum, but I will refer the interested, to an earlier project of mine [47].

The implementation has room for improvement performance-wise, but returns the correct result. The most important part, is the fact that it runs on the GPGPU. This allows for my continued work with the fuser to take much fewer considerations, as the lack of prefix-sum could have been in the way. Bohrium has a concept of "sweeps", which currently is prefix-sum and reduce. These both lacked support for a vector input.

As it runs on the GPGPU, it has the advantage over the CPU, that even if it would run slower than the CPU, the data transfer between GPGPU and host, would make up for any time saved. At the current time, the OpenMP backend of Bohrium, which would perform the CPU-scan, does not parallelize the operation, further limiting the CPU's performance.

None of the benchmarks utilize prefix-sum, so its performance, will not be measured. But the operation is incredibly useful, for parallelizing operations. Guy E. Blelloch demonstrated in '93, thirteen useful cases for prefix-sum, many of which are also NumPy operations – for example sorting[36].

The implementation only applies to vector-prefix-sum and not prefix-sum along an axis in a matrix or rank-n tensor.

3.5 Benchmarks

We have now implemented vector-reductions for OpenCL in Bohrium. Some of the benchmarks use this extensively, while others do not contain reductions of this type.

To compare results, and to map our progress, I have run all benchmarks on both the original version of Bohrium, and on the version with vector-reductions. The results can be seen in figure 3.4.

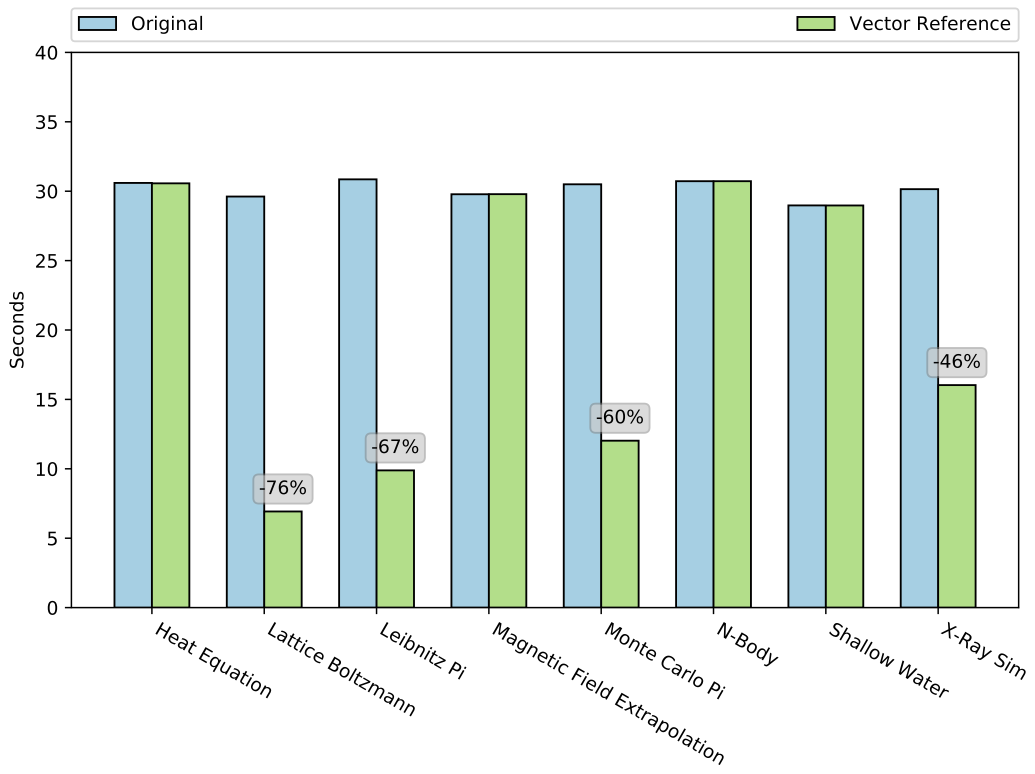

Figure 3.4: Overview of benchmarks. Showing original and latest version of Bohrium as reference.

As we can see, that results are quite impressive. Three out of four benchmarks, which shows benefit from vector-reductions, have performance gains of 60% shorter execution time. And importantly, this comes unconditionally, without hurting other benchmarks, and without any extensive preparations, like the auto-tuner does.

3.5.1 Revisiting the Auto-tuner

We also want to see how the auto-tuner handles the changes, as we need to see, if the vector-reduction excludes some performance gains, the auto-tuner was capable of. If we are lucky, they will stack, to perform even better. The result from the auto-tuner for both the original code, and the new code, can be seen in figure 3.5.

The benchmarks that benefit the most from vector-reductions, are not the same, which benefit from the auto-tuner. This can be seen with Heat Equation and Shallow Water, which sees huge gains from the auto-tuner, but nothing from the vector-reductions. At the same time, Lattice Boltzmann and X-Ray Sim gains the majority of performance from vector-reductions, while the auto-tuner squeezes an additional 9 to 12 percent points out.

We can also conclude, that the auto-tuner saw no drawbacks from the addition of vector-reductions.

3.6 Partial Conclusion

We have investigated the most efficient algorithm for reducing vectors in work-groups, while taking wavefronts, memory banks and barriers into account. We have successfully implemented the first operation in Bohrium, which uses two kernels, to complete an operations, and the first to allocate local memory. All of this, is implemented inside the existing Bohrium code, to become an integrated part of the system.

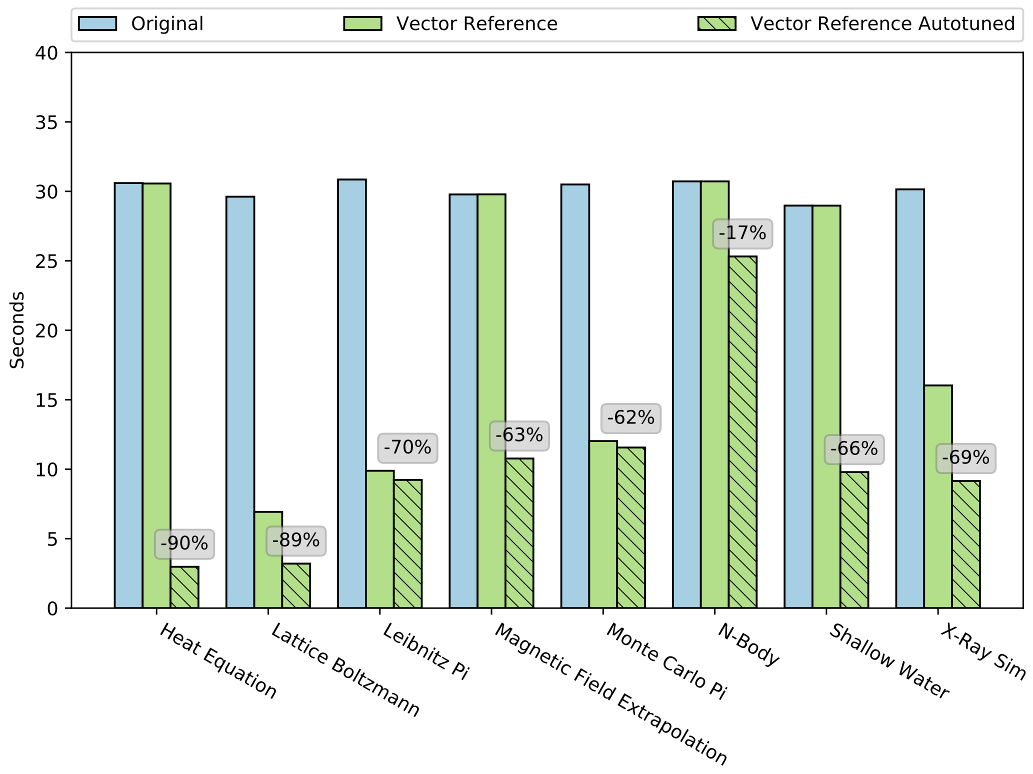

Figure 3.5: Overview of auto-tuned benchmarks. Showing original and latest version of Bohrium as reference.

From the benchmarks and the auto-tuner, we see great benefits from the changes. One benchmark is now running with a 76% decrease in the time measured against the original version of Bohrium. Others are also seeing impressive results of 67% and 60% improvements.

The results from the auto-tuner also proves, that we still have some work to do with the inefficient memory access patterns, as the situation is unchanged.

Chapter 4

Access Patterns

We left the auto-tuner in chapter 2, because the performance gains we saw, seemed to be directly related to a wrongful use of strides in the kernels. In this chapter, I will investigate this further, to see if this proofs to be true, and if there is a way to solve this problem without the overhead of the auto-tuner.

4.1 Related Work

The exact issue is very specific to Bohrium, as it in some ways can be constituted as a bug. I have been unable to find any related work on the subject of access patterns in auto-generated kernels. But there are still sources on the general subject of access patterns.

The "NVIDIA OpenCL Best Practices Guide" and "AMD APP SDK OpenCL User Guide" are very accurate and detailed on how to optimize memory access patterns[44][31].

It all relates to coalesced memory access, which is hardware-specific details. When ordering memory access in the right order, every part of the memory system works in tandem.

4.2 Inspecting the Access Pattern

We have a serious issue surrounding the way Bohrium accesses memory on the GPGPU. Any operation which isn't completely flat, or transposed to be column-wise by the programmer, will have completely uncoalesced memory access. The gains we are seeing from the auto-tuner are clearly the result of the memory stride being corrected, as we can see on the 2-dimensional kernels, which maximize the second dimension – thereby transposing the access pattern.

I inspect some kernels, and find some interesting results regarding the choice of kernel parameters. It seems to be consistent in my speculations surrounding flipped axes.

I set up a simple 4-dimensional array of $50 \times 50 \times 50 \times 50$ in NumPy, and assign 1 to every element. I also force Bohrium to not flatten the kernel. This produces a kernel, with the following index calculation. I've serialized it as regular C-code for demonstration:

```c
for (ulong i0 = 0; i0 < 50; ++i0) {
  for (ulong i1 = 0; i1 < 50; ++i1) {
    for (ulong i2 = 0; i2 < 50; ++i2) {
      for (ulong i3 = 0; i3 < 50; ++i3) {
        a0[+i0*125000 +i1*2500 +i2*50 +i3] = 1;
      }
    }
  }
}
```

In the example above, the variables `i0` through `i3` are the iterators of four nested for-loops, where `i0` is the outer-most for-loop. This is exactly the same in OpenCL, although the three outer-most for-loops would be distributed across different work-groups. The constants inside the square brackets, are the strides, which is taken in each dimension. When we go through our loops, it makes the flat memory structure have "dimensions".

This would have been fine in a single-threaded CPU program, as the CPU would fetch one cache line at a time, and use the whole cache line over the course of the for-loop. This aligns with the temporal and spatial properties of CPU caches.

The problem here is, that on a GPGPU, we actually run a wavefront of simultaneous work-items. To achieve "memory coalescence", which is the most efficient way to perform memory transaction from global memory, the neighboring work-items inside a wavefront, will have to ask for memory addresses adjacent to each other. When coalesced, it will pack a range of memory reads into a single transaction – in addition to other caching gains. This essentially means, that for a rank-n tensor, it is favorable to read/write column-wise in each work-item – reverse of a CPU, which would prefer row-wise. On the GPGPU, a wavefront of column-wise becomes a row in memory, which is why it works.

The CPU also has problems with false-sharing, which means, that it is often preferred to have each CPU thread read as far from each other as possible. For example divide the work-load into contiguous sections, to match the number of CPU-cores. This would make it most likely, that the CPU-cores do not access the same memory page. Analogous, we want something similar for work-groups. We have no reason for work-groups to access the same memory, so an easy way to split them, is by providing a contiguous piece of memory to each.

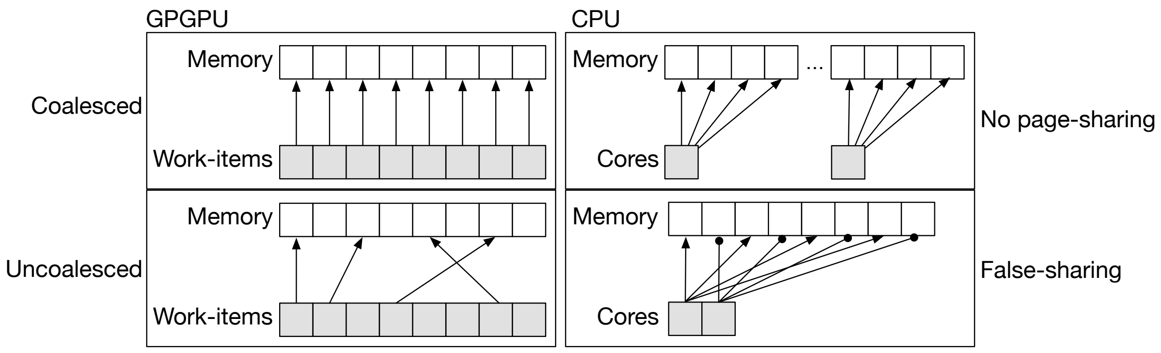

Figure 4.1: Illustration of the access pattern in the currently generated kernels. Note, that a coalesced memory access would have a wavefront grouped along the horizontal axis

Looking back at the code example above, we can see, that a CPU would favor parallelizing the outer-most loop (`i0`), while the GPGPU would favor the inner-most (`i3`). Currently Bohrium parallelizes from the outer-most loop, and up to two dimensions inwards, distributed across work-groups.

If the GPGPU supported an arbitrary number of dimensions, and a homogeneous amount of work-items on each dimension, it would never have been an issue. Then we could simply maximize the inner-most dimension which – if not transposed – has a stride of 1. Strides are shortly explained in the introduction – see 1.6.1.

4.3 Changing the Access Pattern

We can't change the hardware, nor the programmers code, so we have two options: Transpose all memory strides or change our kernel to parallelize on dimensions with lower, and more favorable strides.

Transposing all strides is a valid option, but I want to avoid it, as non-vector sweeps heavily depend on it. Simply transposing all strides, will require significant changes to sweeps and to the fuser, to allow them to be fused together.

It is much easier to manipulate the kernel parameters and kernel generation, as it is just a matter of the ordering of operations.

4.3.1 The Access Pattern Transformation

A generated kernel might look like this, if I force the generator to spawn the kernel as 1-dimensional:

Uncoalesced Kernel

```
const uint g0 = get_global_id(0); if (g0 >= 50) { return; } //
    Prevent overflow

{const ulong i0 = g0;
  for (ulong i1 = 0; i1 < 50; ++i1) {
    for (ulong i2 = 0; i2 < 50; ++i2) {
      for (ulong i3 = 0; i3 < 50; ++i3) {
        a0[+i0*125000 +i1*2500 +i2*50 +i3] = 1;
      }
    }
  }
}
```

Notice, that each thread of the kernel, will write to addresses $125,000$ elements away from each other in global memory. With a work-group size of 128, the GPGPU would also only spawn a single work-group, as the 50 work-items easily fit inside the limit.

I want to change the kernel to reverse the parallelism, to get the threads to parallelize on the inner-most axis. As illustrate below:

Coalesced Kernel

```
const uint g3 = get_global_id(0); if (g3 >= 50) { return; } //
    Prevent overflow

for (ulong i0 = 0; i0 < 50; ++i0) {
  for (ulong i1 = 0; i1 < 50; ++i1) {
    for (ulong i2 = 0; i2 < 50; ++i2) {
      {const ulong i3 = g3;
        a0[+i0*125000 +i1*2500 +i2*50 +i3] = 1;
      }
    }
  }
}
```

This way, the threads will complete a full row of the inner-most axis in no more than four memory transactions – on Nvidia cards, the smallest transactions are performed in half-warps of 16 work-items – instead of up to 50 transactions depending on whether the memory controller might help us out.

4.3.2 Results

I rerun the benchmarks to see if we get any change in performance. As reference, I have the original Bohrium code with default configuration, and the new code, but with the new access pattern disabled, to make sure, that we are still able to select which method to use.

It might be wanted to switch it on/off automatically with an auto-tuner. I also kept it, to make more exact comparisons.

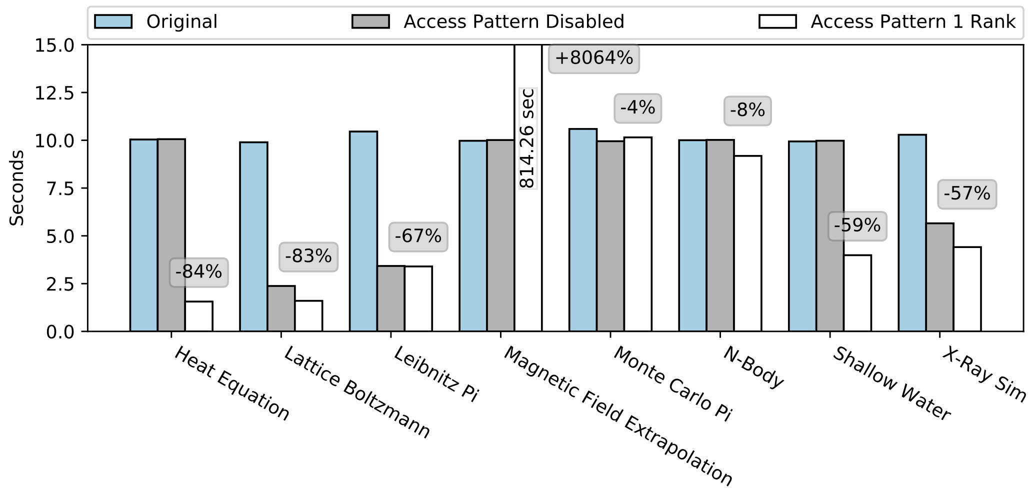

Figure 4.2: Overview of 10-sec benchmarks. Showing the newest version with the access pattern disabled and with the inner rank parallelized.

As we can see in figure 4.2, we do get some decent speed-ups in several tests. Note, that this is the 10-second benchmarks. Heat Equation is down to almost a fifth of its computation time! X-Ray Sim also sees less than half the execution time compared to the original code. And Shallow Water has also shrunk by almost 60%.

But we also see one benchmark having a worse time than before. I will have to inspect this, to find the reason. It is most likely either because it contains an irregular stride, or because it has a very small inner axis, which limits parallelism.

By irregular stride, I mean that some of the dimensions might have been transposed, without rewriting the whole array. Comparing our stride from before of `+i0*125000 +i1*2500 +i2*50 +i3`, a transposed array might look like this: `+i0*125000 +i1 +i2*50 +i3*2500`. We will see if this is a regular occurrence. If it is, we can possibly select which dimension to parallelize, instead of the inner-most. In this case, it would be `i1` instead.

If this isn't enough, we might even look at a different technique. Imagine, a matrix of $1,000,000 \times 2 \times 2 \times 2$ with a regular stride. In that case, a 3-dimensional kernel would provide 8 work-items to parallelize in a work-group, and $1,000,000$ work-groups would be spawned. It would clearly be better to transform some of the iterations from the outer-axis to increase the work-group size. We will see if such a scenario occurs. Otherwise, it might be too rare to optimize for.

4.4 Optimize the New Access Pattern

Magnetic Field Extrapolation performed especially bad with the new modifications. I will have to inspect the kernel to see if I can find the issue.

A single kernel seems to be the culprit, as it takes 99% of the time. The kernel is 5-dimensional, where the inner two axes are reductions, therefore we can only parallelize the 3rd dimension and outwards. We cannot do anything to reductions, as changing their access pattern the way we are doing now, would change the result. We will look at this in section 5, as it is a quite different issue.

But I do seem to find a clear issue. In the original code, the kernel is spawned as 3-dimensional, with the outer three axes being computed across work-groups. This gives a potential parallelism of $100 \times 100 \times 100$ work-items:

The Original Kernel Structure for MFE

```
const uint g0 = get_global_id(0); if (g0 >= 100) { return; }
    // Prevent overflow
const uint g1 = get_global_id(1); if (g1 >= 100) { return; }
    // Prevent overflow
const uint g2 = get_global_id(2); if (g2 >= 100) { return; }
    // Prevent overflow

{const ulong i0 = g0;
  {const ulong i1 = g1;
    {const ulong i2 = g2;
```

With the change I've made, as I have limited the kernel to be 1-dimensional, I will only get the parallelism of the one dimension I choose, which gives a potential parallelism of 100 work-items:

The 1-dimension Kernel Structure for MFE

```
const uint g2 = get_global_id(0); if (g2 >= 100) { return; }
    // Prevent overflow

for (ulong i0 = 0; i0 < 100; ++i0) {
  for (ulong i1 = 0; i1 < 100; ++i1) {
    {const ulong i2 = g2;
```

This is not just within the work-group, but across all work-groups. This means, that a single work-group is performing all the work, as the default work-group size of 128 can contain the 100 work-items. This is clearly not the way to go, when Nvidia informs, that their GPGPUs can have 30.000 or more active treads on a single card from [44, 1.1 Differences Between Host and Device].

The way I choose to solve this, is by adding two more dimensions to the kernel, but to still keep it to act like a 1-dimensional kernel. Meaning, it is currently spawned as 128 work-items per work-group. I want to spawn it as $128 \times 1 \times 1$. This will still act like a 1-dimensional kernel, but it tells OpenCL to distribute the work of the outer two axes to other compute-units/work-groups. Although we still want the majority of the work-items on the inner-most axis, as this is where memory access will be coalesced.

I test if this is a viable solution, by adding the two preceding axes to the one we choose to parallelize over. The rational being, that for an ordered stride, two second lowest strides will follow the lowest.

In figure 4.3, we can see, that it proves to be even more successful. N-Body and X-Ray Sim slimmed down by a significant amount. X-Ray Sim went down yet another 20 percent point from the previous improvements from the 1-dimensional kernels.

Magnetic Field Extrapolation (MFE) is not fixed yet, put we clearly did something right. In the 1-dimensional kernel, the time was around 800 seconds, while it now shrunk to roughly 100 seconds. This is quite an improvement, but not enough to get us to 10 seconds, which we want as a maximum.

If MFE ends up having no gain or a slight slowdown in performance, it might still be worth it, when the other benchmarks see these incredible speed-ups.

4.4.1 Magnetic Field Extrapolation

We still have a peculiar case with Magnetic Field Extrapolation. It surged to more than around 800 seconds in the first test, and now it plummeted to almost 100 seconds. This is a huge jump, but also still more than the time before the new technique. This is definitely

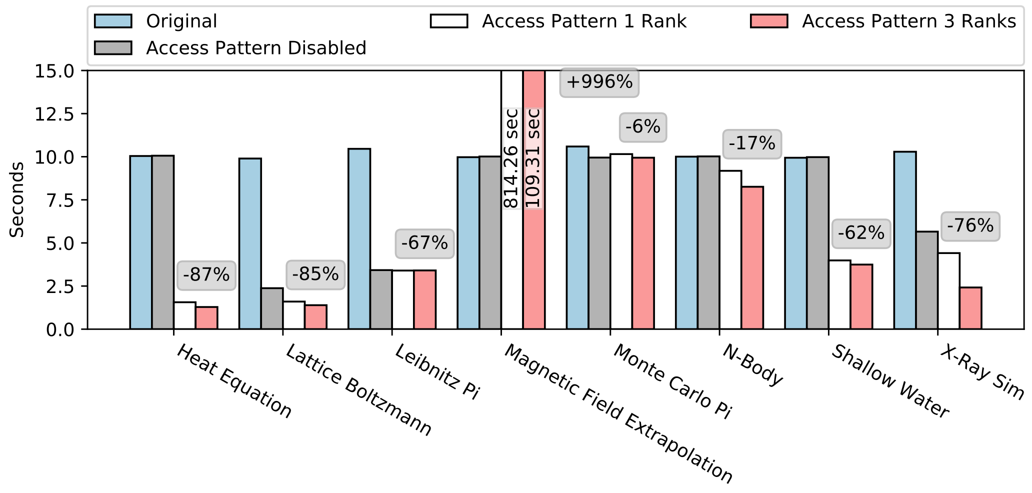

Figure 4.3: Overview of benchmarks. Showing the newest version with the access pattern disabled, and with the inner rank and three inner ranks parallelized.

worth looking into. I would prefer to get the same performance as before, to be fully satisfied with this new approach.

I again inspect the single kernel, which takes the large majority of the time of the benchmarks.

The Full Kernel Structure for MFE With 3 dimensions and 2 Inner Reductions

```
const uint g2 = get_global_id(0); if (g2 >= 100) { return; }
    // Prevent overflow
const uint g1 = get_global_id(1); if (g1 >= 100) { return; }
    // Prevent overflow
const uint g0 = get_global_id(2); if (g0 >= 100) { return; }
    // Prevent overflow

{const ulong i0 = g0;
  {const ulong i1 = g1;
    {const ulong i2 = g2;
      for (ulong i3 = 0; i3 < 100; ++i3) {
        for (ulong i4 = 0; i4 < 100; ++i4) {
```

We have increased the potential parallelism, which definitely was a bottleneck before. But the issue seems to be quite different now. The technique we deployed was meant to optimize cases of reading and writing coalesced memory. This kernel is not one of those cases, because of its reductions on the inner two axes.

The parallelism OpenCL provides us, only fulfills the first three dimensions of the stride, thereafter, two for-loops in each work-item will complete $10,000$ serialized memory reads. This is clearly an issue, but can we easily handle the issue, without affecting our performance gains elsewhere?

As an experiment, I try to transpose the work-group parameters from $128 \times 1 \times 1$ to $1 \times 2 \times 64$. The GPGPU limits the work-items in the third dimension to 64, therefore, I compensate with 2 in the middle, to get the same total size. It's important to have the same total size, as the work-group size can improve other aspects, like latency-hiding.

To my surprise, the latter configuration, performs extraordinarily with a time of 4.72 seconds, compared to 110 seconds for the 10-second benchmark. If we hit 4.72 seconds predictably, it would put the benchmark at more than 50% improvement.

So, how do we rationalize this behavior, and implement it into the new technique?

Nvidia has chosen not to support OpenCL in its profiling tool[29], which means we cannot get the GPGPU to tell us what is going on, if we are not using CUDA. The changes I have made so far, would likely work in CUDA, but I do not find it appropriate, to reimplement everything on another software platform.

I let the issue be, as I have a possible work-around. The most glaring issue is the reductions, which limit our ability to parallelize the inner two dimensions. There reductions are currently uncoalesced. If we coalesce this as well, we could see incredible performance boosts.

4.5 Fitting the Kernel Parameters

While modifying the kernels, I found that a lot of kernels are run on fewer work-items, than 128 (the default kernel size) in the inner dimension. This is clearly a waste, so I want to see what difference it makes, to simply limit the work-groups, to never have more work-items, than needed by the kernel.

Looking at figure 4.4, this doesn't show any difference, but I want to continue with another example. Instead of always running a work-group size of $128 \times 1 \times 1$, how about fitting the work-group size to the three axes we parallelize over? We have seen from the auto-tuning, that many kernels perform better, when the work-group size approaches the maximum (1024), so will it improve our performance to automatically maximize the work-group size?

Imagine a tensor of $1024 \times 32 \times 3$. For this, I would for example spawn work-groups of $3 \times 32 \times 10$ (parameter order is inverted), as this maximizes threads in the coalesced axes, and spill outwards until we have a total size as close to 1024 – the maximum size – as possible. The increased work-group sizes might help with latency-hiding, coalesced memory transactions, and better utilizing the threads in a work-group, as they are not shutdown by the overflow-guard in each kernel.

The method for maximization is shown below. It recursively fills the dimensions, until no more work-items are available.

```
const int x = min(max_x,(int) b[0]);
const int y = min(min(max_y,(int) b[1]), max_size/x);
const int z = min(min(max_z,(int) b[2]), max_size/(x*y))
```

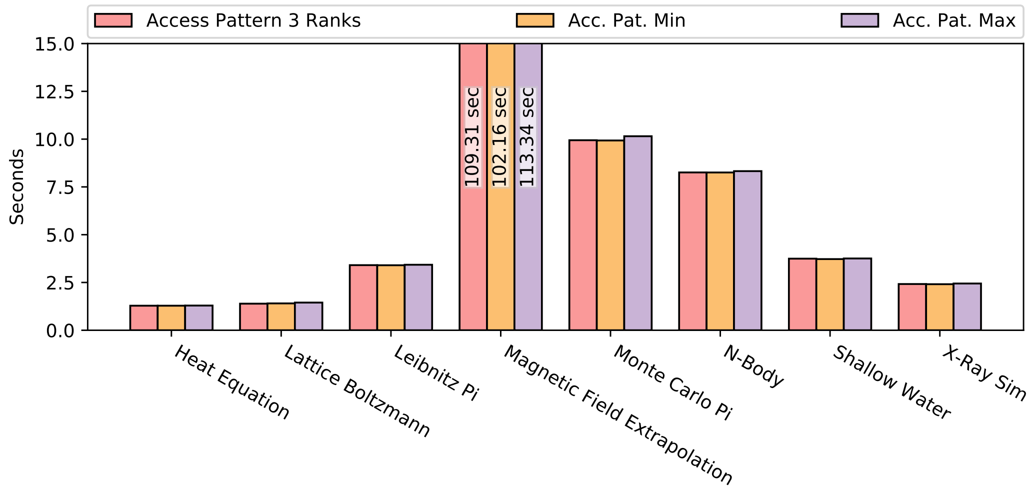

Figure 4.4: Overview of benchmarks. Showing the newest version with maximized and minimized access pattern, and with the three inner ranks parallelized.

The result in figure 4.4 is almost indistinguishable, from the earlier results. It is actually a bit underwhelming how little difference it made, when we saw a rather large change when we investigated the auto-tuner.

It's worth noting, that it's also a good sign, that maximizing the work-groups does not have a negative impact. Vector reduction and other reductions moving forward might see a benefit of large work-groups, as this limits the number of sub-results in global memory. Thereby consuming less memory, and more significantly, reducing the work for the post-processing kernel. Although these benefits are not clear in the benchmark above, it does get amplified with the task size.

I revert the changes of both minimizing and maximizing kernel parameters, as they didn't prove useful.

4.6 Benchmarks

We have already looked at the benchmarks for this chapter, as we have seen three benchmarks with increasing levels of parallelism. Although the changes for the last one were reverted.

The benchmarks we have seen earlier in the chapter were the shorter, 10-second benchmarks, used for quick prototyping. The benchmarks in 4.5 are our longer 30-second benchmarks.

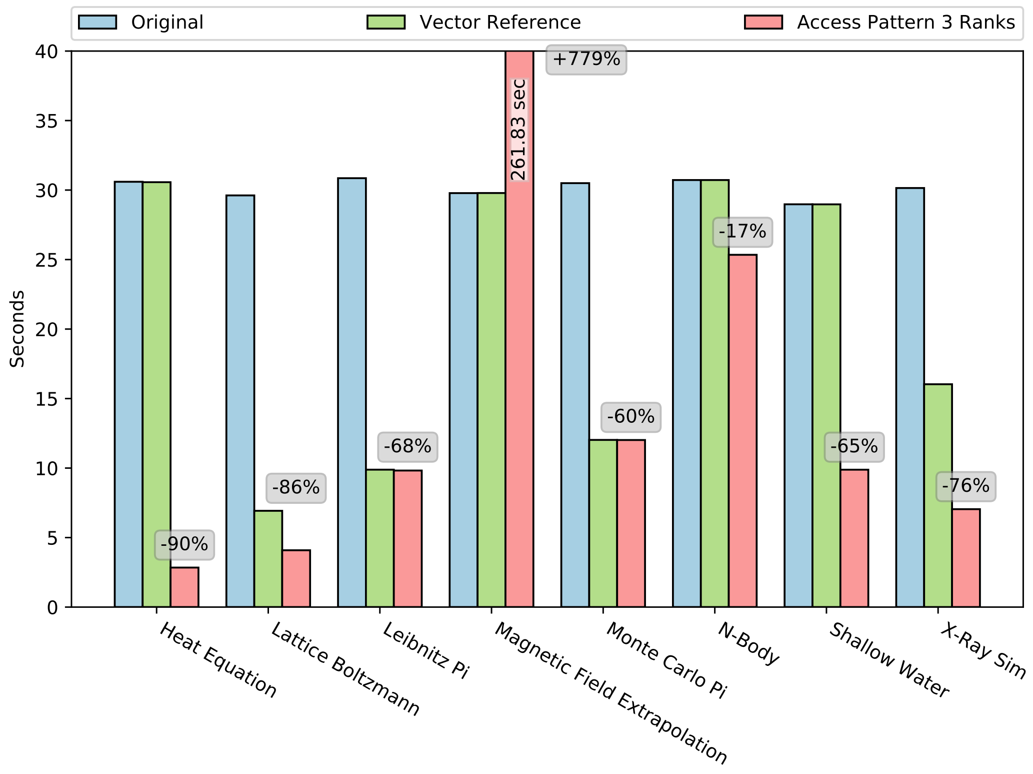

Figure 4.5: Overview of benchmarks. Showing original and latest version of Bohrium as reference.

The version shown, is the one with improved access patterns, with a level of three, as it seemed to be the most optimal, and this version will be improved upon in the next chapter.

4.6.1 Revisiting the Auto-tuner

It might be worth trying the auto-tuner on the benchmarks in the current state. Maybe there is some potential we are overlooking. And now that we have fixed the issues with strides, it might provide more consistent results.

The auto-tuning takes just about 23 hours, for all kernels to try every parameter configuration.

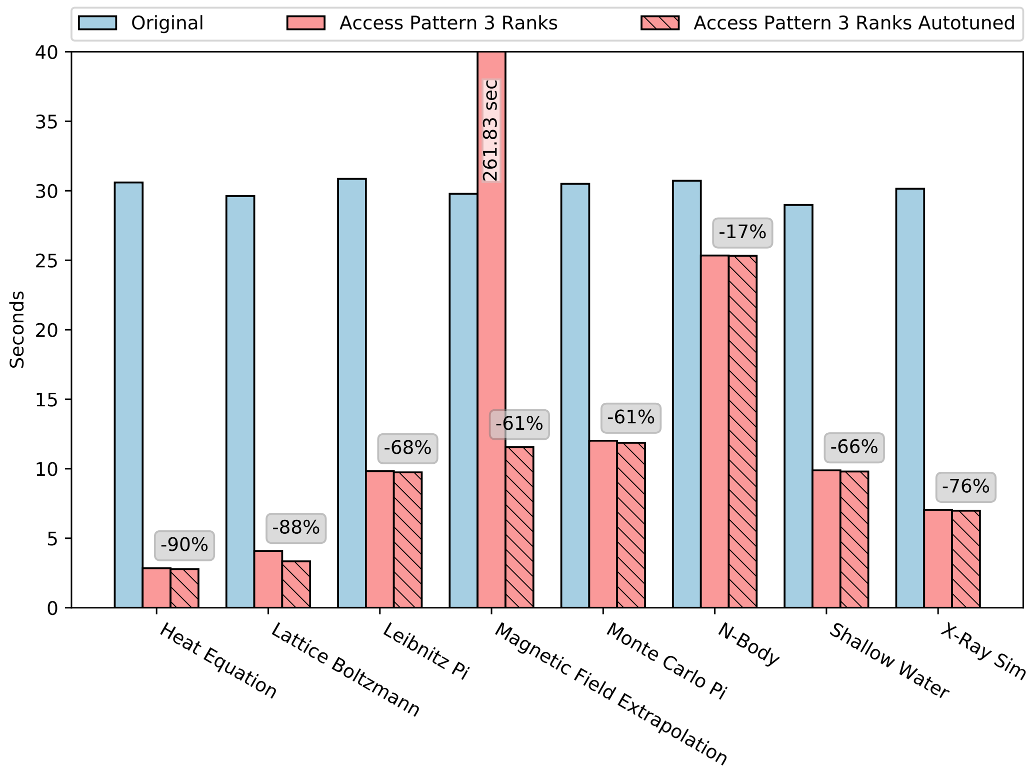

Figure 4.6: Overview of auto-tuned benchmarks. Showing original and latest version of Bohrium as reference.

The Heat Equation and N-Body benchmark had a clear benefit from the auto-tuner from the original version of Bohrium. But after the improved access pattern, it seems the auto-tuner is unable to gain any further performance, than the standard parameters. This is not a bad sign, as we now have rivaling performance to the auto-tuned version. This just shows the changed access pattern was a success.

I also inspect the kernel parameters to verify this. Now, all of the best 2-dimensional kernels have the parameters in the right order, with the X-dimension being the highest, and the following dimension as 1, which is proof, that the changes had the desired effect.

Some kernels for other benchmarks still have some erratic parameters. This includes the infamous Magnetic Field Extrapolation, which has a single kernel, which takes 302 seconds (when profiling tools are enabled). When the kernel parameters are changed from $128 \times 1 \times 1$ to the best tuned parameters of $1 \times 1 \times 32$, we get a execution-time of 10.5 seconds. Adding together with the other kernels, we will get below the -60% mark.

I inspect the kernels with the largest change in execution-time in general. These are incidentally also the ones with the erratic kernel parameters. They all contain reductions on one or more inner axes. This leads me to the conclusion, that I will have to look at reductions independently, as we will have to parallelize these operations better, to gain the desired performance.

4.7 Partial Conclusion

We have now further improved performance by quite a clear margin. Heat Equation benefit the most, with an outstanding 90% improvement, over the previous time, while it did not see any gains from the vector-reductions. Shallow Water also saw a 66% improvement, while Lattice Boltzmann, N-Body and X-Ray Sim saw improvements in the $10 - 20$ percent point range.

The new access patterns gives much more reliable kernel parameters, and seem to have mitigate much of the need for the auto-tuner, as much of the same performance is shown with or without it. This is also reflected in the manual inspection of kernel parameters from N-Body.

The only downside, is the hard performance penalty of the Magnetic Field Extrapolation benchmark. As of the latest benchmark, the execution-time is up by 779%, to a time of 262 seconds, while it initially took 30 seconds. This is something that has to improve.

Chapter 5

Reduction on Rank-n Tensors

Making an effective reduction on a GPGPU, is all about the memory access pattern. Very few operations are performed on each element in the input, so it's mostly a matter of passing the data through to the processor as fast as possible.

Not all reductions are performed the same. When we move above rank-1 tensors (vectors), we will need to use different techniques, to get the most out of our hardware. A bad access pattern when reading and writing on global memory, can easily cripple performance.

This adds complexity to the issue, that we will have to overcome. We will also have to guard the kernels against race-conditions, which can easily happen, if we are not careful with nested/recursive reductions.

5.1 Related Work

This is also one of the more specific issues of Bohrium, although I would have assumed it to be a more general issue. From what I have found, it is a sparsely researched subject.

I have found an article about Futhark, which I will use in the next chapter. It does touch the subject of transposing the data of a reduction to get memory coalesce[42]. But their case is different from ours. They are more keen to transposing the memory, by doing a full memory copy, which could be done implicitly, by making the previous operation do it on write-back. But their approach is clear, they want a single way to do reductions, while I want to adapt the method of reduction to the array, if it provides little to no performance penalty.

The article only discuss transpose for matrices, which is only rank-2 tensors. We will need methods for any rank. It also lightly touches the subject of matrix-reduction, but does not go into details, with how this method would be applied on rank-n tensors.

An article titled "GPU-accelerated real-time stixel computation" also touches the subject of fusing a transpose operation onto the reduction[39]. They seem to be doing this, to transform a segmented reduction into a column-wise reduction. But contrary to our case, they do it for simplicity of their algorithm, while we want to do it for performance. They state, that they accept the performance penalty, as their work-load is small enough to be insignificant.

Our goal is to find a method to, in the best way possible, arrange and apply a reduction in any rank of a tensor. And keep data dependencies within a work-group, for recursive reductions.

5.2 Understanding the Access Pattern

Several properties of the reduction has to be used, to determine the right way to reduce rank-n tensors.

Reductions can be called in Bohrium with some tensor of rank n, and with an axis to perform the reduction along. The result will be a new tensor of rank $n - 1$, where one of the ranks have been removed by the transformation using the given operator.

By definition, a reduction can only collapse one rank at a time. If more ranks are to be reduced, it has to be called recursively for every rank to reduce.

A reduction is performed along an axis of the tensor, provided by the programmer, which complicates our job. Depending on which axis of the tensor, and how the tensor is placed in memory, we have to adapt the reduction strategy, to in the best possible way, coalesce the memory access.

The axis argument points to the nth rank of the tensors (zero-indexed), meaning, a tensor of $4 \times 3 \times 2$, reduced along axis 0, outputs a tensor of 3×2.

We cannot change the axis, nor move the data for the tensor on the device, as the result of the reduction, will change, depending on the axis, and moving the data for the tensor will often take too long, or defeat the purpose of streaming arrays.

The following methods assume an ordered stride, which has the lowest stride in the highest axis (the default for non-transposed arrays in NumPy). Afterwards, we will loosen this assumption, to better generalize the idea of reductions.

5.2.1 Rank-1

Reduction for a vector, is straight forward, as memory will be in one contiguous line, and we can always deploy the same strategy. This is the method implemented in chapter 3.

In the case, a user creates a rank-2 tensor (matrix), and reduces a single column, we will have uncoalesced memory, but we cannot do anything at this moment to change this, as we will not change the layout of memory. But we will have to support this case. With the current implementation, we read in the data column-wise, and reduce it across the work-groups just like a regular vector-reduction. This way is definitely preferable to having a single work-item calculate the whole vector, as it will utilize as many work-items as possible.

5.2.2 Rank-2

For a reduction on a rank-2 tensor, we have two axes of reducing into a vector. Either column-wise or row-wise. Each way requiring a different technique to gain performance.

Axis 1

Reducing along the second axis, we will have a row-wise reduction for every row in the tensor – remember the highest axis has the shortest stride. Here, it would be favorable to deploy a segmented reduction. Each row can be calculated independently, but inside each row, we will have to make sure to not create a race-condition. By using the auxiliary functions of the vector-reduction, we can easily avoid this, and not have to write new code.

There are two ways to parallelize the task. Either each work-item takes a row (uncoalesced), and reduces it by itself, or all work-items cooperate to reduce one row at a time (coalesced).

The first method has the major disadvantage, of being completely uncoalesced. At every iteration, each work-item will load a value from global memory, which is a full row away from each other. See figure 5.1 as illustration. But it is important to note, that we will still get the correct result, even if we do not create coalesced memory access. This means, we can start by implementing this method, and return to the following method for higher throughput.

The second method has the disadvantage, that the work-items will waste time on communication. To start out, I will disregard this penalty. It will only present itself when the

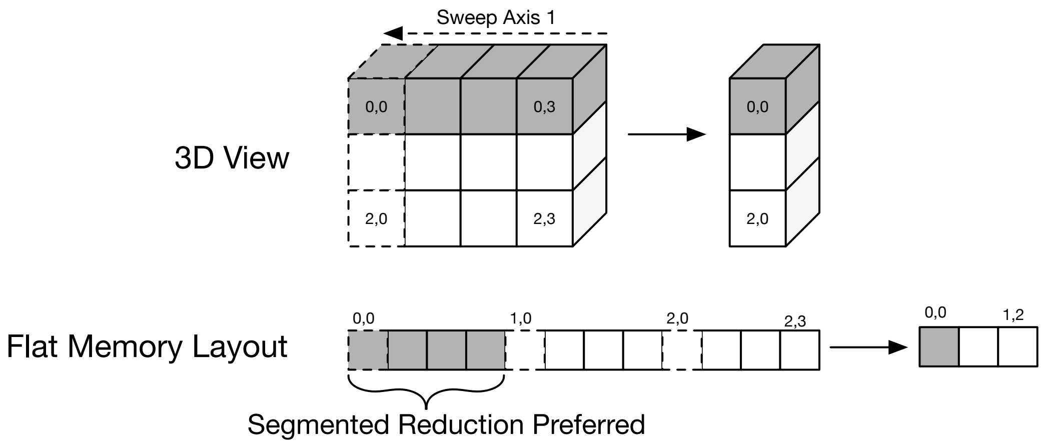

Figure 5.1: Illustrating the access pattern of an axis-1 reduction on a rank-2 tensor. The striped lines illustrate the parallel work-items.

tensor is smaller than the work-group size, or at the last part of a larger reduction, when we have reduced all other elements. But all memory access will be perfectly coalesced, and from our experience in vector-reductions, this seems like a much more important aspect.

As a side note, we also have to think about the potential parallelism for each technique. In the first method, in a tensor of 128 × 128 elements, we can with the default Bohrium kernel parameters, only spawn a single work-group. In 4.3 we saw this heavily bottlenecked the GPGPU. In the second method, we can split the work at every row, if we want, to easily spread across multiple work-groups.

Axis 0

The inner axis is an easier case to implement. Just like the case above, we have two methods to choose from. We can make a work-group reduce one column per work-item, or one column per work-group.

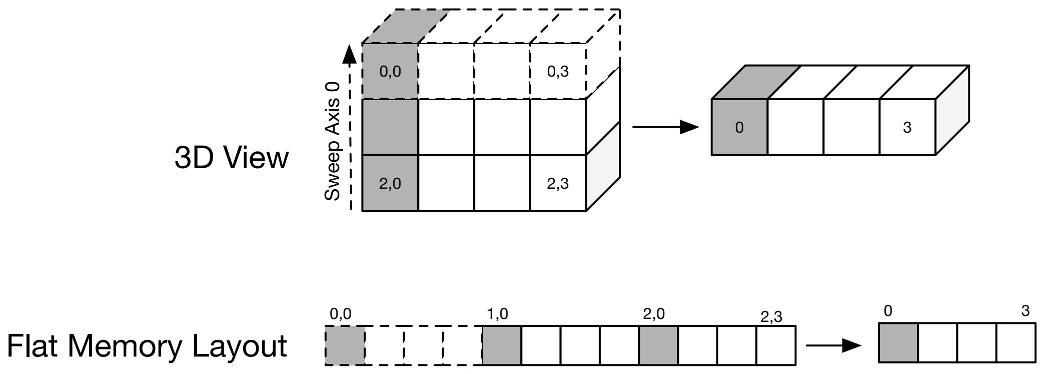

Figure 5.2: Illustrating the access pattern of an axis-0 reduction on a rank-2 tensor. The striped lines illustrate the parallel work-items.

Opposite of the case above, I will choose the method of reducing one column per work-item. This might be counter-intuitive, as I just said it limits parallelism, but it also maximizes memory coalesce. If parallelism becomes an issue, a simple solution, is to limit the work-group size, to let more work-groups get spawned.

5.2.3 Rank-n

Moving up to higher ranks, we will see a pattern in the technique we have to use to reduce.

A rank-3 tensor, can be seen as stack of rank-2 tensors. Likewise, a rank-4 tensor, can be seen as a stack of rank-3 tensors.

When the reductions are in the higher two axes, we can use the techniques described for a rank-2 tensor. Adding a 3rd rank simply create m parallel rank-2 problems, where m is the size of the 3rd rank. Same is true for a rank-4 tensor. It is also m parallel rank-3 problems, where m is the size of the 4th rank. This works recursively for any higher ranks as well.

When reducing lower axes (stride higher than 1), we will only be using a technique equivalent to the column-wise reduction. The data will be spread out by a factor of the stride of the current rank. This does not mean, it is not possible to achieve coalesced memory access, it just limits how large sections we will be able to load coalesced, as illustrated in figure 5.3. The sweep in axis 1, will have its data fragmented, but still coalesced. This is a correlation between wavefront size and the size of the input data.

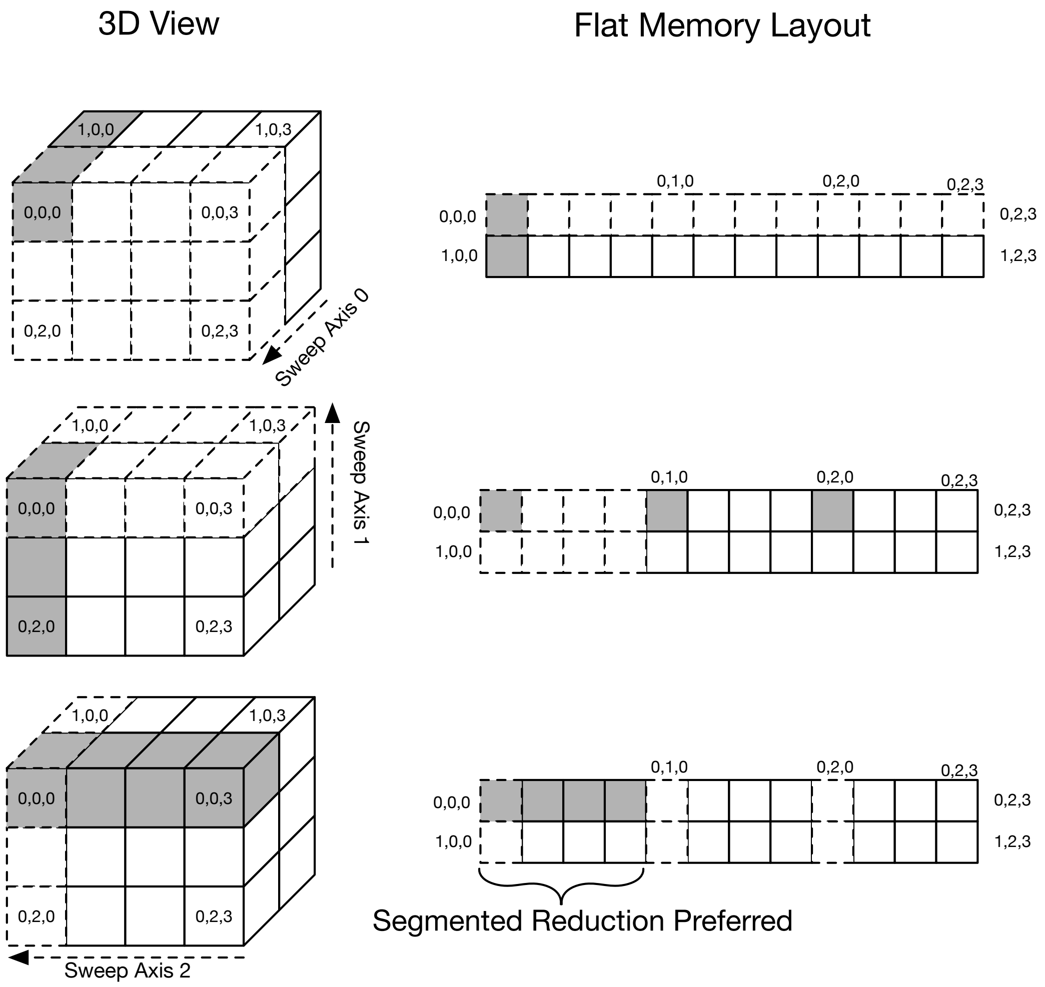

Figure 5.3: Illustrating all possible access patterns of reduction on a rank-3 tensor. The striped lines illustrate the parallel work-items.

We can generalize the rules we have seen for rank-3 tensors to any higher rank tensor, as discussed above, they will be equivalent to a stack of rank-3 tensors. To illustrate, see figure 5.4.

I generalize what we have seen, to the following rules: For reductions on the highest axis, we use the special technique we learned from vector-reductions (see chapter 3), to reduce

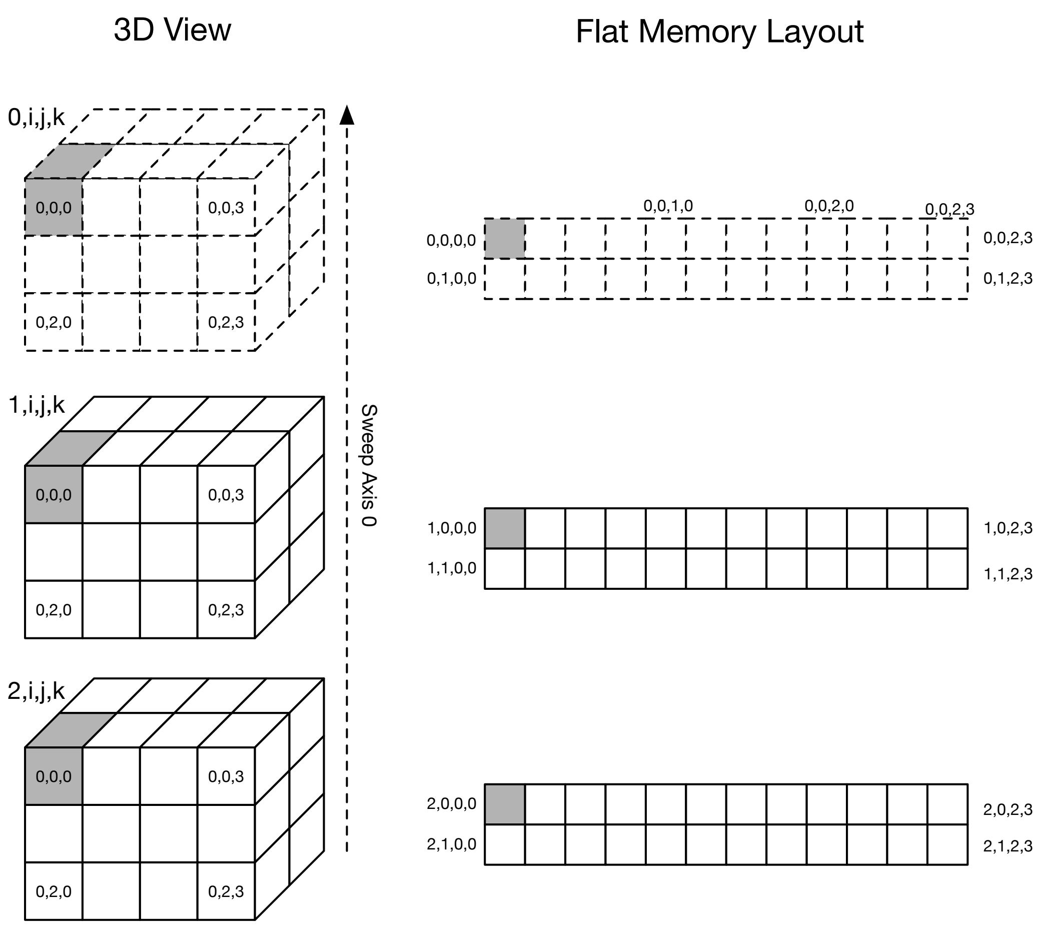

Figure 5.4: Illustrating the access pattern of an axis-0 reduction on a rank-4 tensor. The striped lines illustrate the parallel work-items.

row-wise internally in a work-group. For any other rank, we one the technique of "column-wise" reductions, as this will provide coalesced memory-access. The degree of coalesced memory access, is determined by the input tensor's size.

5.2.4 Reshapeable Data

In NumPy, it is possible to call a reduction on a rank-n tensor, and without defining an axis, getting a scalar result. This implicitly sums each axis recursively (order of axis is insignificant), until a scalar value is returned. At the current state of Bohrium, this will result in n recursive reductions, while it actually might be allowed to run a vector-reduction directly.

When Bohrium gets the instruction to reduce an rank-n tensor, we can instead check to see, if the data is reshapeable and contiguous, and reshape it into a vector, without copying any data, simply by changing the definition of the strides.

It is generally preferred to reshape the data to the lowest possible rank, as this gives us the highest degree of freedom. The number of dimensions in a kernel defines the level of dependency between the data. If we find out, that the reduction might be faster, with a tensor of higher rank, we can easily transform the tensor to have a higher rank – not the other way around.

Sweeps are one of the most limiting factors of Bohrium, because it forces an access pattern upon us, which we can not freely change, but have to make do.

5.3 Nested Reductions

All the discussion above, concerns with a single reduction on any axis. What happens when a kernel wants to reduce two axes recursively? Meaning, reducing once, and use the output of the first reduction as input for another reduction. It's important to remember, that reductions can only take away one rank at a time.

At the current state of Bohrium, we parallelize each rank up until the first rank with a sweep. This ensures, that any work-item has exclusive access to the data it needs.

If we nest several reductions, they will have to be following each other in order to ensure the sub-results stay in the same work-item for the second reduction.

If we did not, we could have a situation where the inner reduction is of axis 0, and the following reduction is axis 1. The axis 0 reduction will create a vector per work-item, where only one value is relevant for the next reduction, while the rest of the data will be distributed across other work-groups. The problem being, that we cannot transfer data between work-groups within a kernel.

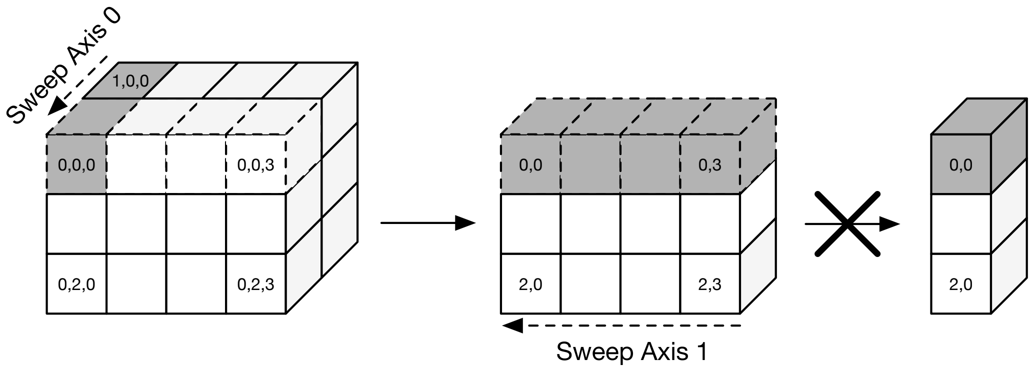

Figure 5.5: Illustrating the conflict of changing reduction axis. The striped lines illustrate the parallel work-items.

Illustrated in 5.5, can we see the issue with reduction ordering. Parallelizing along an axis in the first reduction, might not fit the axis in the following reduction. Here is seen 4 parallel work-items, which we cannot guarantee is within a work-group and can communicate (imagine this is not 4, but $4,000$ work-items). This means, that the result of the first reduction, will not be available to the correct work-items of the second reduction.

5.3.1 Transposing the Reductions

The solution is a small mathematical trick, which helps us organize the reduction. By transposing, we can "rotate" the data so the parallelized work-items align with the two axes.

How it Solves the Issue

Assume a tensor of rank 3, like the one in 5.5. If we generated a kernel for the issue in the illustration, it would lead to a kernel, with the following pseudo-code:

Pseudo-code Showing Conflict in Data Dependencies

```
gfor i0 = 0..dim(2)
  for i1 = 0..dim(1)
    for i2 = 0..dim(0)
      B[i0,i1] += A[i0,i1,i2]

# Conflict!
gfor i0 = 0..dim(1)
  for i1 = 0..dim(2)
    C[i1] += B[i0,i1]
```

After the comment "Conflict!", we cannot proceed, as we cannot assume we have enough spawned work-items for `dim(1)`. Even if we did, we cannot read elements written by other work-items, which will happen, when iterating the inner loop.

If we transposed the tensor, to order the reduction to take the lowest axis, we will have a reduction that works on GPGPUs. See figure 5.6.

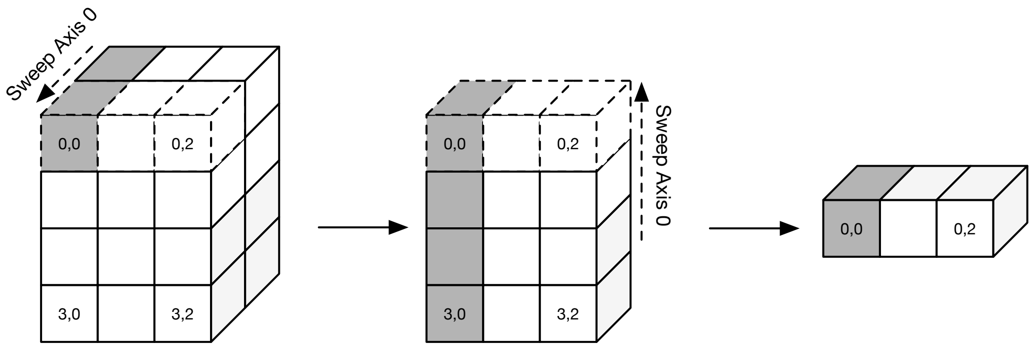

Figure 5.6: Illustrating the tensor transposed to a consistent reduction axis. The striped lines illustrate the parallel work-items.

And we are also able to merge the two reduction into the fewest possible iterations:

Pseudo-code of Transposed Nested Reduction

```
gfor i0 = 0..dim(0)
  for i1 = 0..dim(1)
    for i2 = 0..dim(2)
      B[i0,i1] += A[i0,i1,i2]
    C[i0] += B[i0,i1]
```

Now, the nesting is more obvious in the code, when the ranks are ordered with the inner reduction. It also allows for easier optimizations on the GPGPU, as we can substitute `B[i0, i1]` and `C[i0]` with variables, and either make a single write-back, or just keep the sub-result locally in the work-item without wasting global memory, and the time for writing and reading the array.

Pseudo-code of Single Write-back With Nested Reduction

```
gfor i0 = 0..dim(0)
  sum0 = 0
  for i1 = 0..dim(1)
    sum1 = 0
    for i2 = 0..dim(2)
      sum1 += A[i0,i1,i2]
    sum0 += sum1
  C[i0] = sum0
```

This is how Bohrium accomplishes the streaming/broadcasting arrays. By never allocating memory for the inner ranks – although in this case it reads from global memory – we can "overcommit" memory, if the memory is not used again.

Transposing can be used for any reduction, to make it easier to implement in Bohrium.

5.3.2 How Transposing Works

NumPy and Bohrium already has functionality to transpose tensors, but I will quickly illustrate how it is performed.

The operation is almost free to do, as none of the tensor data is moved. It is accomplished solely by changing the strides of the tensor. This principle is illustrated in 5.7. Notice, that transposing only changes the order of which the indices are placed.

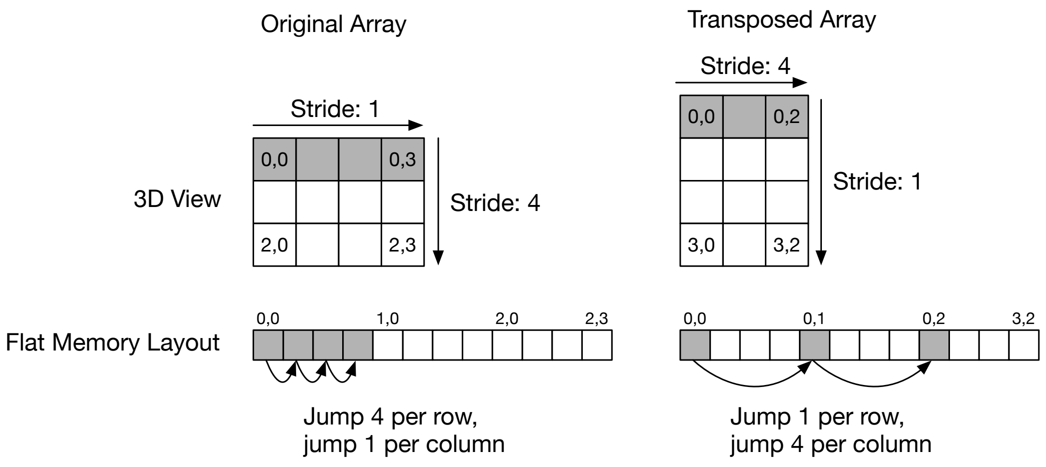

Figure 5.7: Illustration of transposing data. Left shows the original input, right shows the transposed data. The data is untouched, but the strides are changed.

From 5.7, we can say, that the data defines the content, while the stride dictates the shape of it.

The principle in the illustration transfers to any rank, and is not confined to rank-2 tensors. When we have more than a rank-2 tensor, we have the opportunity not just to reverse all indices, but to select which index to place where.

The strides are kept as a list of integers. One stride for each rank. The stride is initially given by the product of the sizes of all following ranks, as it defines how far there is between each value in memory. The first stride is initially 1. Given an array shape of $(6, 2, 4)$, a stride would be given as: $[2 \cdot 4, 4, 1] = [8, 4, 1]$. Transposing the stride, is simply a matter of changing the order of the integers in the stride.

We can transpose any tensor so the reduction axis becomes the inner-most. We can also change the order of reductions, if the user requests a nested reduction. If we prefer the lowest stride as the inner-most or not, we have the freedom to change this by transposing.

5.3.3 Cautions and Notes for Reductions

It's interesting to note, that we do not have to transpose back after the reduction. If we want to reduce axis 1 of a tensor with dimensions $4 \times 3 \times 2$, we can simply transpose the tensor to $3 \times 4 \times 2$ and reduce axis 0. This will output exactly the same 4×2 tensor we would have otherwise.

The transpose technique works, as long as we only need the final result. We cannot guarantee, that results in between nested reductions, are the same, as reducing one axis first, and then the second.

The sub-results are only guaranteed to be correct, if the sub-result contains no transposed axes. Meaning, we can always transpose one axis, and get the right result. If we reduce more than one axis, they will have to precede the place the sub-result is needed. After the usage of the sub-result, we can only continue the reduction in the same kernel, if the reduction axis is naturally aligned as the highest axis.

It is not possible to transpose the tensor while a kernel is running. This interferes with the access pattern, and would cause race-conditions. We saw this earlier in this chapter.

There is no easy solution to this issue, if the reduction has to be performed optimally on a GPGPU, as the access patterns are incompatible. The fallback method is to separate each reduction into different kernels, as this gives the opportunity for synchronization of global memory.

5.4 Implementing the Transposed Reduction

Now, that we have found the theoretical basis for reduction in rank-n tensors, we will proceed to implement it in Bohrium.

I continue the implementation on top of the new access patterns, which fixes the issues with strides in regular operations.

5.4.1 Bohrium's Current State

First, I assess the current state of Bohrium's reductions, to see where to start. I set up a Python script, which instantiate a rank-4 tensor, and reduces each axis independently, using both Bohrium and NumPy, and compare the result.

Below, is a simplified version of the code described above. The sizes are chosen so all axes have a unique size, to control the fuser, and making them identifiable in the generated code.

Summing a Rank-4 Tensor On All Four Axes

```
A = np.arange(5*4*3*2).reshape((5,4,3,2))
np.sum(A, axis=0)
np.sum(A, axis=1)
np.sum(A, axis=2)
np.sum(A, axis=3)
```

When I inspect the kernel, I find that all reductions have been fused into the same kernel, with complete disregard to optimal access patterns, and race-conditions. All data is loaded one element at a time from the input array, and added to the designated output array. It also does not initialize the output array, which means it is adding whatever data was in memory, when the array was allocated.

Thankfully, this is all issues, that we can correct through transposing, modifying the fuser, and injecting the correct identity instructions for the output arrays.

It is not clear, if this is an issue caused by my earlier changes, or if it was present before. I choose to simply fix them regardless of origin.

As mentioned, Bohrium fuses all four reductions together to one big glob of race-conditions. To avoid this, I force Bohrium to break the reductions apart with a print statement in between each reduction.

When compared to NumPy, only the last reduction gives the correct result. This tells me, that Bohrium doesn't recognize axes of reductions, and simply reduces the inner-most axis in the kernel.

To test this theory, I insert a transpose statement, which places the actual axis I want reduced, as the last in the tensor. NumPy supports transposing, where we provide a list of indices, where the indices refer to the original order of the axes. By rearranging the order, we can move an axis to the back of the tensor. The following four reductions are mathematically equivalent to the four above.

Summing a Rank-4 Tensor On All Four Axes Using Transpose

```
A = np.arange(5*4*3*2).reshape((5,4,3,2))
np.sum(A.transpose((1,2,3,0)), axis=3)
np.sum(A.transpose((0,2,3,1)), axis=3)
np.sum(A.transpose((0,1,3,2)), axis=3)
np.sum(A.transpose((0,1,2,3)), axis=3)
```

This gives the correct result on all four reductions, compared to NumPy! So the technique actually works with Bohrium as-is.

This means, I can simply inject this transposing method directly into the NumPy bridge between Bohrium and Python, to immediately gain this functionality, and without any code changed from the user-perspective.

This method doesn't use segmented reduction for reducing the axis with a stride of 1, which would be a really good optimization to have, but it does find the correct result, none the less. All other reductions are trivially coalesced, as it uses column-wise reduction. This can be seen in the following code:

Generated Kernel Showing Coalesced Memory Access Pattern

```
{const ulong i0 = g0;
  {const ulong i1 = g1;
    {const ulong i2 = g2;
      float s0; s0 = a0[ +i0*6 +i1*2 +i2];
      s0 = c2;
      for (ulong i3 = 0; i3 < 5; ++i3) {
        s0 += a1[ +i0*6 +i1*2 +i2 +i3*24];
      }
      a0[ +i0*6 +i1*2 +i2] = s0;
    }
  }
}
```

Here, it is seen, that the access pattern becomes coalesced, as the axis with the stride of 1, is placed on `i2`, which is the axis we parallelize across the work-group. From the changes we made with access patterns earlier, we always parallelize the inner-most non-reduction rank. For non-transposed tensors, this is by default the rank with stride 1. If we wanted to parallelize on `i1` and `i0` (currently distributed on other work-groups), it would also align with coalescing the access pattern.

The only concern I have with the access pattern, is the axis we are reducing in this case, is the one furthest apart. This might introduce some type of cache invalidation across work-groups, but that is of concern to the programmer, as they asked for that axis. We cannot do anything at this moment to change this.

5.4.2 Implementation in Bohrium

The transposing can be implemented either in the NumPy bridge, the C++ bridge, or the fuser. I start by implementing it in the NumPy bridge, as we have a higher-level representation of the reductions. Here we are given a list of axes which are reduced recursively. Once it enters Bohrium's C++ bridge, these are broken up into individual reductions, and we will have to deploy more advanced techniques to identity, that they go together, and there is no other operations in between. But with that being said, it is a more complete solution to implement it in the fuser. If the user calls two reductions recursively, but with two separate statements, we cannot identity it in the NumPy bridge, while the needed implementation in the fuser, would catch it regardless.

A normal pattern for programmers is to call two reductions, where the first is in-lined as input to the second like this, or the same split onto two lines:

Challenges for the Fuser

```
# First
np.sum(np.sum(A, axis=2), axis=0)}

# Second
B = np.sum(A, axis=2)
np.sum(B, axis=0)

# Third
B = np.sum(A, axis=2)
A += 1
np.sum(B, axis=0)
```

This can relatively easily be detected, as the fuser can look one instruction ahead, and check for compatibility. It becomes slightly harder, if other operations are placed in between, even though they do not change the relevant values. If so, we will have to do a look-ahead, to make sure the value is not changed in between the calls.

It comes down to cost-benefit – or time-benefit. We want to quickly prototype the technique in NumPy, and if it seems to be working perfectly, we can decide to implement the better solution in the backend. It is only a matter of performance. We will always get the correct result.

I implement the transposing in the NumPy bridge using just 3 lines of code:

Code in `ufuncs.pyx`

```
indices = range(ary.ndim)
ary = ary.transpose(filter(lambda x: not x in axis, indices) +
    axis)
axis = indices[-len(axis):]
```

From the call to NumPy we have `ary`, which defines the input array, and `axis`, which is a list of axes to reduce. The axis list is a list of one element, if one axis is being reduced.

The procedure is to `filter` out the axes that are used for the reduction, and concatenate them to the end of the transpose list. The list will be the order of the axes in the new array. The new array is written back to `ary`, and the `axis` is updated to be the last n values in a range from 0 to the number of axes.

This effectively means, that the axes to reduce has been moved to the highest axes, and the `axis` list contain the axis indices in the new array.

The order of the axes is mathematically insignificant, but after the next chapter, we might find, that one axis is more preferable to have first. For example if the axis with a stride of 1 should be faster or slower than the others. Or maybe depending on the size of each reduced dimension.

5.5 Benchmarks

To determine how the changes has impacted performance, I run the benchmarks again. I expect that we might see slightly lower performance, as the transposing is expected to allow less fusing, because the fuser does not recognize fusing is possible. At the same time, this will weigh against more coalesced access patterns.

Better performance or not, it actually finds the correct result now, compared to before. This is obviously stands before any performance gain.

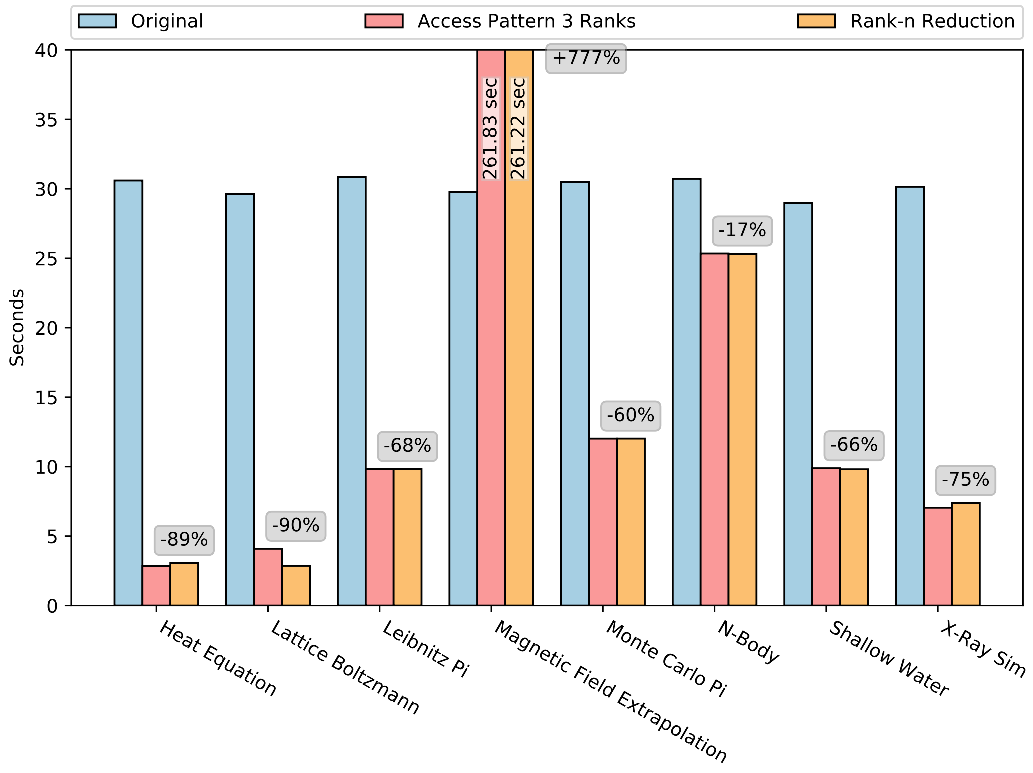

Figure 5.8: Overview of benchmarks. Showing original and latest version of Bohrium as reference.

From the benchmarks in figure 5.8, we can see some slight changes to performance in Lattice Boltzmann and X-Ray Sim, but otherwise indifferent results. Lattice Boltzmann improved relative to the previous chapter by quite a lot. Compared to the original version, it's just 2%.

Magnetic Field Extrapolation is still very far behind, compared to the reference. This is not surprising, as the kernel uses reductions with a stride of 1, and the reductions were already placed last, thereby going unaffected by the changes in this chapter.

5.5.1 Revisiting the Auto-tuner

I have left the auto-tuner three times now. The first time, was because the strides were inverted compared to how the kernels were parallelized. Then I left it, when I had implemented the vector-reduction. Thirdly, after the access pattern was fixed, I left the auto-tuner once again, because the issue seemed to surround inner reductions, which led to this chapter.

Now that the corrections have been implemented, I will revisit the auto-tuner, to see if there is still work left to be done.

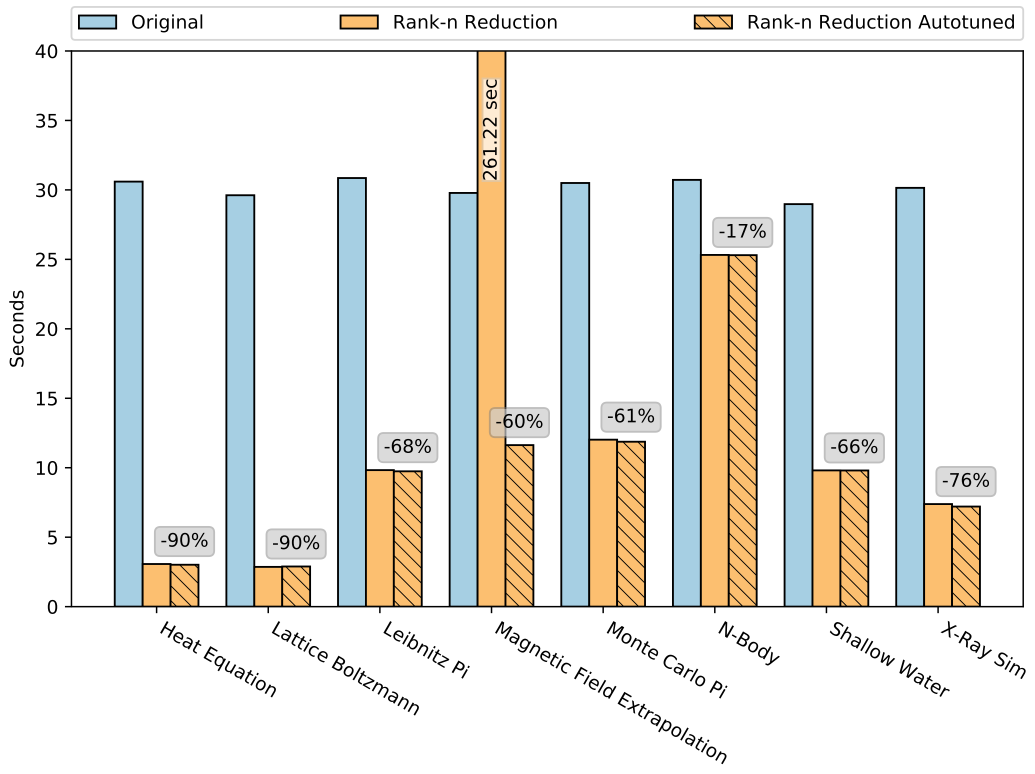

Figure 5.9: Overview of auto-tuned benchmarks. Showing original and latest version of Bohrium as reference.

The results of the auto-tuner is unchanged, which is a good sign, since the changes improves the non-tuned performance, and it did not impair the tuned performance.

The only benchmark, which gets any significant speed-up, is the Magnetic Field Extrapolation. This is something we will have to investigate, since it is not good enough to have a three times as slow performance, as when we started. Especially not, when we can get a 60% improvement using the auto-tuner.

But it looks like, we can or will make the auto-tuner redundant.

5.6 Partial Conclusion

With the changes to reductions, we now get a much more consistent kernel generation, where any axis will by calculated correctly, to find the right result.

The changes has all-in-all been an unconditional success, and we will keep the changes, moving forward.

We still have work to do on Magnetic Field Extrapolation, to have satisfying results across the board.

Chapter 6

Segmented Reduction

From the previous chapter, it would seem, that we need to optimize performance in reductions with a stride of 1. I will try to deploy a technique similar to the vector-reductions. I call this method segmented reduction, as it is a reduction over an axis, where the work-groups are not calculating a single scalar, but a scalar for every n elements, effectively creating a series of independent segments.

The reason for creating segments, instead of independent reductions, is because segmentation fits well with Bohrium, and maybe even with performance, when the axes are short. Apart from the vector-reduction in chapter 3, Bohrium will create a for-loop when reducing. The loop will sum up an axis to a scalar for each work-item. This for-loop is performed in parallel, depending on the stride of the reducing axis, this might be coalesced or not.

The major obstacle, is the limitation, that we do not parallelize reductions. This is also what is causing the uncoalesced access pattern.

6.1 Related Work

The issue of (regular) segmented reduction, row-wise matrix-reduction and irregular segmented reductions, like Thrust's `reduce_by_key`, are only sparsely documented in research[26][42].

For the most part, it is avoided whenever possible, and replaced by either a column-wise reduction, or with functions from the Thrust or cuBLAS libraries [25][39]. But neither Thrust, nor cuBLAS is a direct match. The Thrust-method is to use their implementation for reduction of segments with irregular sizes – likely causing unneeded memory and computational overhead. The cuBLAS method, is to use a matrix-vector multiplication, to create a segmented sum. By multiplying the matrix with a vector of ones, the resulting vector, will be the sum of each row. One obvious downside, is the fixed operator. Apart from that, we might be able to gain better performance, with a more specific implementation.

The only comprehensive research I found on the exact subject, was of the Futhark language[42]. They experiment with different strategies of splitting the work-load across work-items, wavefronts and work-groups depending on segment and work-group sizes. They also experiment with transposing the whole array by rewriting the order in memory to then perform an actual column-wise reduction.

For the small segmented reductions, they use a segmented prefix-sum (scan), and extract the last element of the result, giving the same result as a reduction (see chapter 1.5 and 3.4). They do not specify the code of the prefix-sum, but they hint at Blelloch's methods from 1993[36].

If they used Blelloch's prefix-sum algorithm, it does not seem to be the reason for their performance gain, but they used it to save time on the implementation. The algorithm Blelloch describes, contains two phases: Up-sweep and down-sweep. After the up-sweep phase, the reduction has already been made, and the down-sweep is redundant for the reduction[36].

Even the up-sweep alone is less efficient, than a reduction, as they also conclude in the paper[42].

Through benchmarking, they compare the strategies, and find that for segments larger than the work-group, they can use the same technique as column-wise reductions. On segments smaller than half the work-group size, they favor a segmented prefix-sum. In the extreme case of summing 2^{26} integers, it becomes favorable to transpose the whole array by copying, to use a column-wise reduction, than to use their method for small segments. Note, that if we were to implement this, it would defeat Bohrium's array-streaming capabilities (see chapter 1.5).

The Futhark article, also notes, that very little research has been done in this specific area, with more research targeting irregular segmented reductions[42]. Of the more interesting sources, is one with several types of segmented reductions. It originates from Nvidia, and is found in a GitHub repository called *moderngpu*[27]. I will not go into detail with every method they try, as it is not specific enough for our use-case. But they do demonstrate, that Thrust's `reduce_by_key` is highly inefficient compared to their implementations, by a very large margin.

6.2 Preparation

Before we can implement the segmented reduction, we will have to make some changes to Bohrium. We will first of all have to make sure that the changes we make are compatible with whatever kernel Bohrium might generate.

6.2.1 Separating the Reductions

We cannot assume, that there will only be one reduction in the rank, nor that there isn't any operation preceding or following the reduction, within the same rank. Some of these operations, might even manipulate the data, which is going into the reduction.

I make the observation, that any reduction can be postponed within the rank, as there cannot be any operation using the output within the same rank. This is trivially deduced, as the output will be of rank $n-1$ of the rank compared to where the output was generated. This is seen in the kernel, as only the rank it is nested within, has the correct access pattern to use it.

This helps us a bit, as we can untangle the reductions from other operations, to better arrange the kernel. I propose, that we by default move all reductions to the end of the rank they are within, without considerations, as to what data it depends upon. This allows for much simpler code, as we do not have to make any dependency checks.

When that is in order, we can easily create an auxiliary function, which performs the segmented reduction, and let the existing Bohrium code generate the pre-processing of input data.

6.2.2 Rewriting the Access Pattern

The problem is, that this for-loop with pre-processing still isn't coalesced. It reads in memory in a for-loop, where each value is separated by a stride of 1. This can potentially make each work-item perform a read and write transaction for every iteration of the reduction, where just one value is used, if the values are far enough apart, or if the cache-line gets invalidated. This might also cause problems for the reductions, if they are reading in data, which was written by another work-item in the same work-group. Although this case can be fixed with memory barriers.

What I want to do is "straighten out" the for-loop to run coalesced, and preferably with the same access-pattern as the reductions, so concurrency and race-conditions do not become an issue.

Let us assume, that I have magically coalesced the memory in the for-loop, and we want to start the segmented reduction. Where does the reduction fetch the input from? If the for-loop wrote back all the data to global memory, we would now have to fetch it again, doubling the number of memory transactions. What if we kept it in an array in local memory? Then we would have to make sure we have enough local memory for all the reductions, that are waiting. The GPGPU I have available, has 32 KB of local memory. This is just 8192, 32-bit values at best. If there are more than 64 iterations, with a default work-group size of 128, we have used up all of it. This would very likely happen. Just for our configuration of Magnetic Field Extrapolation, there are 75 values, and they are even 64-bit doubles.

The only option, is to flatten out the access-pattern in the for-loop, and perform the reduction in tandem with the values being read in, processed, and moved into the reduction. This will require just one local variable in each work-item, for each reduction the programmer has called, as it would otherwise, but we will also need some scratchpad memory.

Compared to the memory usage calculations above, this will be orders of magnitude less. Here, we will need 4 or 8 bytes per work-item in the work-group for each reduction the programmer has called. So for example a reduction of an axis with any number of 32-bit floats, and a work-group size of 128, will use 512 bytes, or 1024 bytes for 64-bit doubles. If the programmer calls several reductions inside the same rank, this is simply multiplied. In the worst case of a reduction with 64-bit values, and a work-group of 1024, it will peak at 8192 bytes, which is 25% of the total local memory.

We could theoretically reuse the scratchpad memory, if we can separate the pre-processing of each reduction, but it does not seem to be worth the effort at this moment, as we should have plenty of local memory for several reductions, even on less capable hardware.

To flatten the access pattern, we will have to rewrite which index each work-item will access.

To keep the initial reasoning simple, I limit the code to only support one reduction, as it gets more complicated with nested reductions.

If we cannot make it work with nested reductions, it is easily mitigated. There can only be one rank with stride-1, and by transposing, we are able to place that particular axis at any rank we would like. If we place another axis at the outer-most loop, and the sub-result between reductions isn't needed, the axis, which would have been stride-1, would never be materialized in global memory. It will simply be calculated by keeping a single accumulator within the work-item.

What we want to do, is having the combined 128 work-items take as many elements from the inner dimension, before iterating to the next segment.

With the inner two ranks shaped as 3×2, the access pattern could look like this:

Work-item	0	1	2	3	4	5
Index	0,0	1,0	2,0	0,1	1,1	2,1
Flat index	0,0	0,1	1,0	1,1	2,0	2,1

This is a small example, but when the numbers are in the hundreds, the spacing between elements becomes too large for any caching to help. The problem here being, that work-item 0 and 3, 1 and 4, and 2 and 5 are reading pairs adjacent to each other in memory. By the general rules of coalescing memory access, this will trigger several transactions, although modern hardware might be capable to optimize this exact case. But again, if we move to larger data-sizes, it does become a problem.

What we want to achieve, is the flat index pattern. This way, all access will be in the order, that they are placed in memory. This will cause perfect coalesced access, and dramatically reduce the number of memory transactions.

The observant reader might have noticed, that the indices in the flat access pattern are simply the previous indices transposed. And this is what we want to do.

Because the product of the dimensions are the same, no matter the transposition, we can assure, that we have the right amount of iterations, we just have to rearrange the stride.

6.3 Algorithm

Now that we have defined the frame of the code, we can look at the algorithm of which we will use for the reduction. The use-case is slightly different from the vector-reductions in the chapter 3, but I will skip some of the introduction.

There are multiple ways to reduce segments inside a work-group. We just have to find a balance between complexity and benefit, to maximize performance.

The Futhark article focused on almost any size of segmented reduction – from 2^0 to 2^{26}. I will focus on the ones that seem most likely, from the perspective of the sizes involved in our benchmarks, but also slightly targeting the size of Magnetic Field Extrapolation. These sizes will roughly be in the range of 2^0 and 2^8. If we want a different strategy for larger segments, it is trivial to change method, if needed.

The Futhark article also used a prefix-sum in place of an actual reduction. Although this does work, it is inherently less efficient, as it performs more redundant operations. I will look to implement an adapted version of the vector-reduction from chapter 3, in the hopes of better throughput.

6.3.1 Algorithm: Packing Work-group

The obvious solution, is to pack as many segments as possible in the work-group. There might be a few wasted work-items at the end, if the segment size doesn't equally divide the work-group size. These are moved to the next iteration, meaning, that we might have to run more iterations of the work-group, than the original, uncoalesced method.

Segments Smaller Than Work-Group

In theory, this might be efficient, as we will save a few work-group iterations, because we limit the amount of wasted work-items. In practice, maybe not. A few obstacles present themselves.

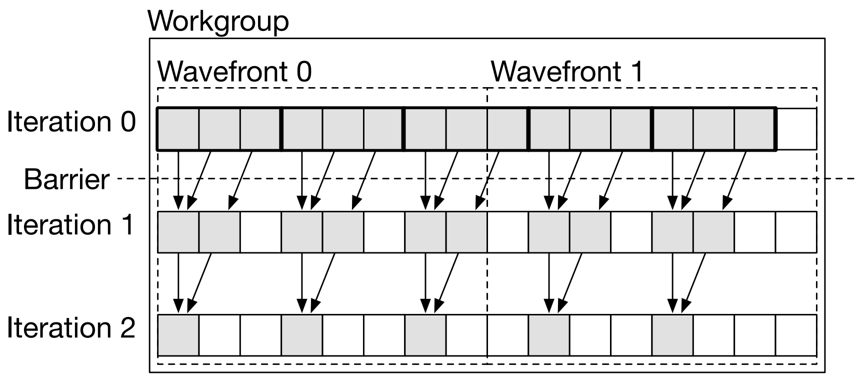

Figure 6.1: Packing algorithm for segmented reductions. The black outlines show segments of 3 elements. Grey boxes indicates usable elements.

Firstly, we will have to add some logic to all work-items, to avoid reading the neighboring data, when they are in an uneven numbered segment size. Any added flow control in these very tight loops, can possibly add to the execution time.

Secondly, some segments will cross wavefronts. Inside a wavefront, everything is synchronous, which means, we can read and write to neighboring work-items, without the risk of race-conditions. These race-conditions can be mitigated, by adding memory barriers, but this is an expensive operation, especially, if it is performed inside a tight loop.

Thirdly, barrier control. If we accept, that we need to call the barrier, we will have to detect when it is needed. We could enable it for every iteration, to be on the safe side, but this will definitely reduce performance – see section 3.2. Alternatively, we will have to detect whenever it is needed. Doing this in run-time will likely not be favorable performance-wise. With a static segment and work-group size, it can be predicted, and be implemented by unrolling the for-loop, or splitting it into a barrier and non-barrier for-loop.

To predict this, we will have to look at the divisors for 32 (wavefront size): $[1, 2, 4, 8, 16, 32]$. If the initial segment size is any of the ones in the list, we will never need a barrier. With any other segment size, and when the total length of the segments is more than 32, we will need at least one barrier.

At every iteration, we divide the segment size in two, and round up, to find the new segment size. Assuming a compacting tree-reduction, if this segment size is not in the list, we need a barrier for that iteration. If an iteration produces a number in the list, no more barriers are needed in the following iterations – for example 15, which will become 8 the following iteration. Other sizes, like 18, will not produce a number in the list before 2: $[18, 9, 5, 3, 2, 1]$

Another observation, is when the segment sizes are between a half to a full work-group size. This creates a full work-group reduction, which can be comparatively inefficient. See figure 6.2

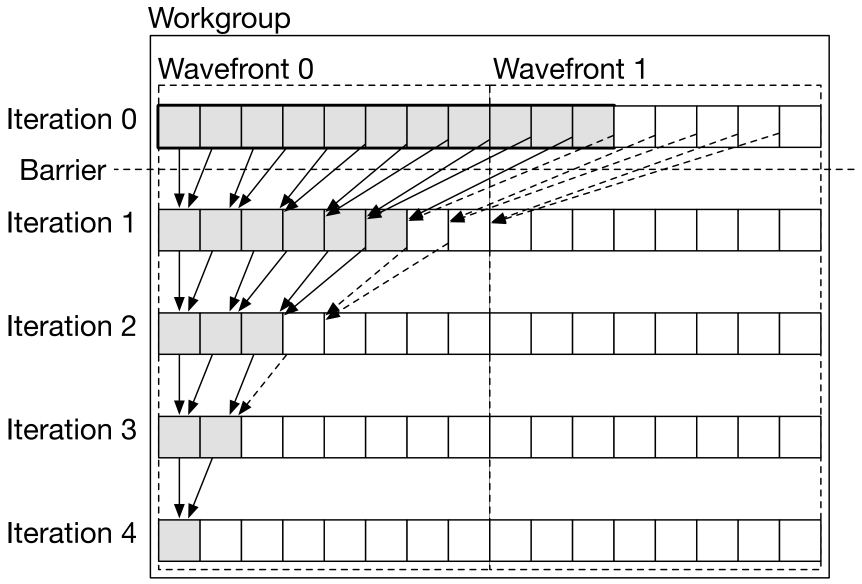

Figure 6.2: Oversize algorithm for segmented reductions. The black outlines show segments. Grey boxes indicates usable elements. Striped lines show padded neutral elements.

In the worst case scenario, just half plus one work-items of the work-group will be active, as we cannot fit in two segments. And additionally, if we can fill the work-group, we will half the amount of occupied wavefronts at each iteration, and end up with an unnecessarily underutilized work-group.

Segments Larger Than Work-Group

We have the opportunity to overload the work-group with the divide-and-conquer method in the beginning of the reduction. It also has the advantage, to utilize the whole work-group, when the segments are really large (hundreds of elements).

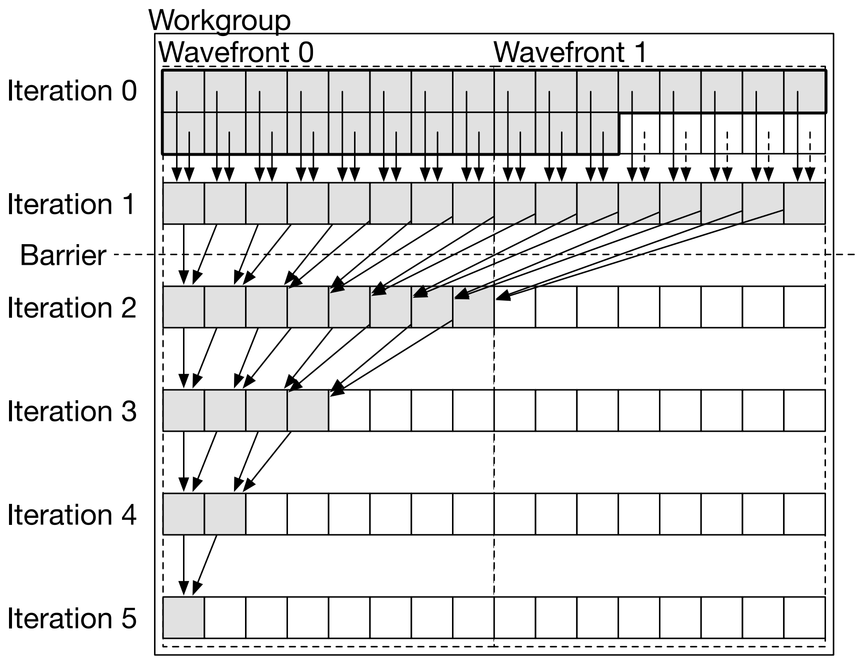

Figure 6.3: Oversize algorithm for segmented reductions. The black outlines show segments. Grey boxes indicates usable elements. Striped lines show padded neutral elements.

The disadvantages are similar to the ones described above. It likely has poor performance for the final iterations of the work-group, as it needs barriers, and the occupancy falls drastically.

This method is only really advantageous, when we have few segments (for example 10-30), which are all very, very large (hundreds or thousands of elements). This can be somewhat mitigated by lowering the work-group size if needed.

6.3.2 Algorithm: Aligning Wavefront

Another solution would be to align all segments with a wavefront, so no segments crosses between two wavefronts. It is obviously inefficient to only place one segment in each reduction, so we will still try to pack multiple segments into each wavefront.

Segments Smaller Than Wavefront

I have come up with two options for placing segments in the wavefronts: Either packing the wavefront, but not crossing the boundaries, or aligning the segment with the start of the wavefront, the middle of the wavefront, the quarters, and so on. Essentially, the last method rounds up every segment size to the nearest power of 2.

The easiest solution for the reduction, is to round up the segment size to the nearest power of two, assuming no segments are larger than 32. This allows us to reuse the auxiliary functions from vector-reductions.

If we look at the tree-reduction in figure 3.1, we can see, that for each iteration, we compute a range of sub-reduction, as if they were an independent segments. The trick is here, to only make the number of iterations we need, to reduced our segments – and not combine all segments to a scalar.

This allows for much simpler code, as we get barrier-free and control-flow-free reductions. But we will have to create some code, which packs and unpacks the segments before and after the reduction. This might be less of a performance issue, as this is placed outside the loop.

The other solution, is to pack as many segments into the wavefront as possible, but never cross a wavefront's boundary. This will enable us to pack more segments into the wavefront, at the expense of complicating the reduction code. But maybe lessening the packing and unpacking before the reduction.

See figure 6.4 for an illustrated comparison.

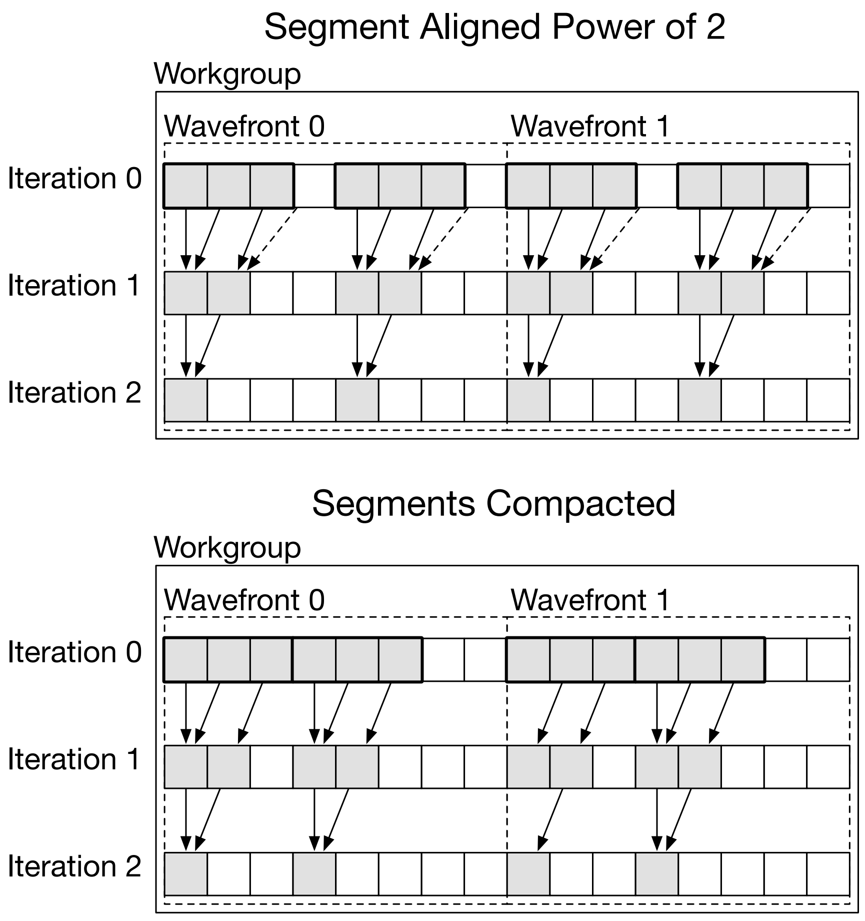

Figure 6.4: Comparing alignment algorithms for segmented reductions. The black outlines show segments. Grey boxes indicates usable elements. Striped lines show padded neutral elements.

To compare the efficiency of the two methods, see figure 6.5. We compare the two alignment strategies, to determine how large the difference between the two is. If the difference is insignificant, we will choose the one with the assumed fastest implementation. I've also included the packing algorithm above in the graph for comparison, even though it becomes highly inefficient at large segment sizes. The numbers inside a wavefront would be the same as the "Compact Alignment", but I've instead calculated the average segments per wavefront, if it went across a full work-group.

When looking at the graph, there is not a major difference between the three in terms of how many segments fit into a work-group. For some very specific sizes, there is a good amount of extra segments, but they are not consistent enough, that it will be worth sacrificing performance on all other accounts. Performance-wise, there might be a lot to gain,

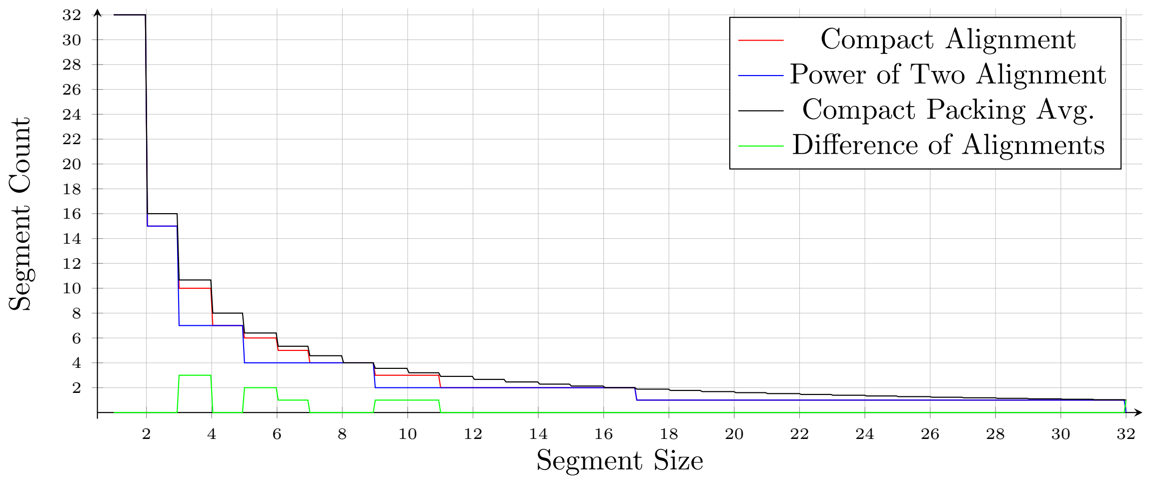

Figure 6.5: Number of segments that can be fit into a wavefront when aligning, with three different methods.

from taking the "Power of Two" alignment, as this heavily simplifies the tight loops in the reduction kernel.

6.3.3 Segments Larger Than Wavefront

The choices might be different, when the segment size overflows that of the wavefront.

I see two choices. The first option, is to load in as much of a segment as possible across the work-group, and run a reduction for the full work-group – see figure 6.3. If the segment size is larger than the work-group size, we can use the divide-and-conquer technique from 3.2.3, until we have a segment at work-group size.

The second option is to fill up each wavefront, and do a divide-and-conquer until the segments are reduced to wavefront size. Then, we do the rest of the reduction inside each wavefront – see figure 6.6.

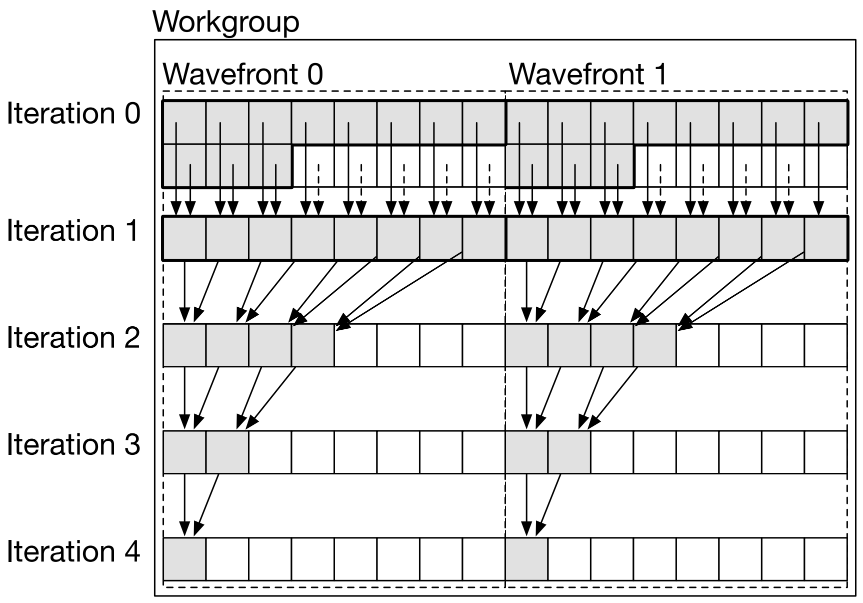

Figure 6.6: Oversize algorithm for segmented reductions. The black outlines show segments. Grey boxes indicates usable elements. Striped lines show padded neutral elements.

The advantage of the first option, is mostly that of reusability. We also might have an advantage of more optimal cache access, as the data will be read in contiguously from start until the end. This should avoid any chance of cache invalidation.

The second methods seems to be more optimal on several accounts.

A major advantage is occupancy. Just like the wavefront method above, all wavefronts will be fully occupied in the initial reduction to get the segment size down. Although, with a work-group of four wavefronts (128 work-items and 32 in a wavefront), the first method will only be fully saturated for a fourth of the time compared to the second method.

In case of a segment size, which is just above half of the work-group size, it will also only be possible to fit in one segment in the first method, while one or more wavefronts will be completely redundant. Imagine a work-group size of 128, which provides four wavefronts. With a segment of 65 elements, the third wavefront will have one value, and the last will have none, and no other reductions can occupy the last wavefront. In the second method, we can trivially skip the last iteration, and save time.

And after the initial period, the first method will have the reduction spread out across the work-group. Then it will have to communicate across the work-group, which requires barriers until the segment size gets below the wavefront size. And at the first two iterations (assuming a work-group size of 128), it will have first two, and then three, completely idle wavefronts. The last four iterations to complete the segment, will be running at just one wavefront out of four.

The second method doesn't have any of these occupancy issues. As said above, each wavefront loads in four times the data, which means they are fully utilized for four times as long. After the initial reduction, the segment size will fit exactly into a wavefront. At this point, all four wavefronts have an independent reduction to complete, and no barrier has been called, nor will it be needed. Then every wavefront performs the last iterations local to each wavefront, and four segments are returned.

The first method has a slight advantage with caching. It always reads data immediately following each other. But the second method doesn't necessarily fail on this point. It still has all 32 work-items in each wavefront read data in a coalesced pattern, and this is the only requirement for memory coalescence. The disadvantage might be, that the wavefront *could* invalidate each other's cache-lines under the right conditions. But as each wavefront only reads the data once, it seems like a rare, or non-existent issue.

It would seem that the best solution is to align with the nearest power of two when inside a wavefront, and with larger segments, keep all the computations inside of the wavefront.

6.4 Implementation

Now that we have established which algorithm to implement, we can start coding.

I will be taking better use of the code generator, than I did in vector-reductions. Back then, I implemented the reduction in a header file. This seemed to be the easiest way, but it has proven to be a slight limitation, when the kernel requires multiple reductions, with different operator and neutral elements. Therefore; I will try to inline the reductions, where they are used. This also makes the code clearer, and changes gets handled better by the compiler – header files are not compiled, if the kernel does not change, but this is mostly an issue for Bohrium developers.

6.4.1 Handling the Overflow Guards

First off, we will have to change the overflow guards. At this time, they are executed in the beginning of the kernel, and lets work-items above the data length return immediately. When we do these reductions, we actually want them to be available.

I tried to figure out a nice way to move the overflow guards from the start of the kernel using if-statements and splitting for-loops, but nothing really fit. Therefore, I had to go for the commonly discouraged `goto` statement, which might have found it's only acceptable use here.

All generated kernels have one or more of the statements in the code below. They make sure, that only the required work-items enter the kernel.

```
const uint g1 = get_global_id(0);
if (g1 >= 5) { return; } // Prevent overflow
```

My plan is to replace the return-statement with a `goto`-statement, which brings the work-item to the first reduction, if needed, or otherwise to the end of the kernel.

GPGPUs act very different from multi-core processors. When the work-items within a wavefront diverge onto different paths of a flow-control statement, they cannot execute at the same time. GPGPUs use a paradigm called SIMT (Single Instruction, Multiple Threads), which means that a range of threads – work-items – will share the same program counter and other related registers[45, 2.1.1 SIMT Architecture]. To be compliant with C, the GPGPU disables diverged threads, while inside a flow-control statement. If needed, it will serially execute each block of code, which cannot be performed by the whole wavefront.

What I want to accomplish, is to effectively disable the work-items that are out-of-range of the data, until they reach a reduction. Because of the way divergent flow-control acts on GPGPUs, the work-items that are ahead, will wait for the other to catch up. This happens, because the compiler will look for the earliest place the two diverged code-paths cross again.

To make sure this is also what happens in practice, I compile the code below, which has some fictional work-load in an auxiliary function.

```
if (lid < 5){
  a[lid] = A[gid];
  test_label:
  test_nothing(lid, a, local_size);
} else {
  a[lid] = 0;
  goto test_label;
}
```

I inspect the assembler code, and look for the branching. The code below has been truncated to the most important part.

```
    @%p1 bra         BB0_2;
    bra.uni          BB0_1;
BB0_2:
    cvt.u32.u64      %r3, %rd1;
    ...
    bra.uni          BB0_3;
BB0_1:
    mov.u32          %r2, 0;
    st.volatile.shared.u32   [%rd2], %r2;
BB0_3:
    cvt.rn.f32.u64   %f2, %rd1;
    st.volatile.shared.f32   [%rd2], %f2;
    ret;
```

In the beginning, all work-items are together. After the first instruction, the ones with the register `p1` set to true, will jump to the `BB0_2` sub-routine. When they are done with the sub-routine, they will jump to `BB0_3`, with the special `bra.uni` instruction.

Definition:
`bra.uni` is guaranteed to be non-divergent, meaning that all threads in a warp have identical values for the guard predicate and branch target[18]

This instruction is a hint, that the code can be run on the full wavefront without issues. The threads which didn't branch, executes their sub-routine and rendezvouz with the remaining work-items at `BB0_3`. Remember, that in assembler, the program will flow directly from `BB0_1` to `BB0_3`, without any branching.

I also compare the code to its none-`goto` equivalent:

```
if (lid < 5){
  a[lid] = A[gid];
}
else{
  a[lid] = 0;
}
test_nothing(lid, a, local_size);
```

Thankfully, the resulting assembly is the exact same. Therefore, I choose to continue with this method.

I inject `goto` statements into the overflow guards. If there is no segmented reductions, it skips the computing block, and goes to the bottom of the kernel code. This replaces the earlier changes for vector-reductions, which changed the overflow guard to an if-else statement. Now the else has been replaced by the `goto`.

I needed a way to inject a neutral element to the redundant work-items. I solved this by adding an `if (false)`-statement, which can never be executed by the regular threads. By placing the `goto`-label inside the following code block, only the redundant work-items will reach the bootstraping of neutral elements. The following shows the vector-reductions, but it looks the same for segmented reductions.

```
const uint g0 = get_global_id(0); if (g0 >= 6) { goto
    skip_block; } // Prevent overflow

{const ulong i0 = g0;
  // Large computation
  element = ...;
}
if (false) {
  skip_block: ;
  element = NEUTRAL;
}
reduce_2pass_preprocess(element, a, res);
```

While refactoring the vector-reductions and implementing the ground work for segmented reductions, I improved the internal representation and the ease of detecting these reductions with meta-class, which is carried around with the kernel state. This eases the amount of repeated code, and makes it easy to control when a segmented- or vector-reduction is appropriate.

6.4.2 Writing the Segmented Reduction

Writing the segmented reduction itself, was not nearly the same challenge, as getting everything into place. As we have to take care not to modify anything around the reduction, I have to make 100% sure when it's the right case to inject the segmented reduction, and make sure the redundant work-items are moved to the right places.

First off, to make sure the segmented reduction only is inserted in supported cases, I have to check several parameters. These requirements can be loosened, when I get time to develop support for it. Some special cases might not be worth supporting, if they never appear in real-world kernels. But the kernel has to work no matter what. If we find an unsupported case, it will fall back to the uncoalesced reduction we had before.

Number of Sweeps To make the initial implementation simpler, I only allow for a single reduction in the kernel. Problems arise, when there needs to be multiple `goto`'s. These will have to have unique labels, and we will have to link the redundant work-items to the right order of reductions. It will be supported later, but it requires further handling. When there are more than one reduction in a kernel, they are often nested, which also requires some work to detect when the redudant work-items are completely finished.

Sweep Axis We have to check the sweep axis of the reduction, to know when it's the right time to inject the reduction. If the fuser provides us with a reduction, which is of stride-1, which is not in the inner-most rank in the kernel, we will not handle it as a segmented reduction at the present time. This is again possible to support, but requires a little more handling because of possible data-dependencies, and detection of whether it is actually a nested reduction, where segmentation is not needed.

Valid for Segmented Reduction Of course, we will also have to check that the reduction has a stride of 1 on the reducing axis, as the new feature of transposing reductions, will put any reduction in the inner-most rank.

No Other Instructions At last, I check that no other instructions are inside the rank with the reduction. This is because I completely change the lay-out of the for-loop. As noted above, this can cause data-dependency issues, if the data is not calculated in the same work-item, as the one needing the data for the reduction. This will be supported later in this chapter.

The segmented reduction ends up "stealing" the inner-most rank completely from the regular flow in Bohrium, and replacing it with the alternative access pattern (the flattening described earlier). The code for it, heavily complicates the kernel at first glance, as it adds a solid 45 lines of code. But it should, nevertheless, be better optimized, than using the old kernel.

I put the injected code inside a curly-brace closure, to isolate local variables from the outer ones, just like if it was still a regular for-loop. Then, I added an entry-point for the `goto`-statements. The work-items that enter through the `goto`-statement, will be tagged by a boolean, which is only true, if the work-item came from the `goto`.

To better understand, I have written this pseudo-code for a better overview:

```
# Previous operations...
s0 = 0.00;
{
  bool tmp = False;
  if (false):
    seg_reduce: # Entry-point for goto.
    tmp = True
  bootstrap_local_arrays()
  segment_size = round_up2(size)

  foreach segment_id in number_of_segments: # Iterated 4 at a
     time for work-group size of 128
    acc = 0
    for (i0, i1, i2) in segment:
      # Inject any other operations here
      acc += a0[i0, i1, i2]
      acc = reduce_segment(acc, segment_size)

    if (local_segment_id == 0):
      write_back[segment_id] = acc
    barrier()
    if (tmp):
      return
    s0 = write_back[workitem_id]
}
# Following operations...
```

For the next part, it calculates the segment size (rounded up to a power of two), the local work-item ID inside of the segments, segments per workgroup etc. Then the actual for-loop iterates as many segments, which fit into the work-group, and repeats for as many iterations as needed. As an example, 7 segments of 32, with a work-group size of 128, iterates twice. As the segments does not evenly divide the work-group size, the last iteration is simply skipped.

The loop-header in C, looks like the following. Remember, that it is performed in parallel, while each `workitem_id` is unique to each work-item:

```
for (size_t segment_id = (workitem_id/32); segment_id < 7;
    segment_id += 4)
```

The 32 threads, which had `segment_id` of 3 in the first iteration, are incremented to 7 at the second iterations, meaning, they are not within the bounds, and can continue without the other wavefronts. This is the type of programming unique to OpenCL, CUDA and SIMT, as we often have more threads than needed.

Inside the segment for-loop, we load in the data from global memory. It's always read in via a for-loop, no matter the segment-size – even if just one iteration. But in the same style as above, it loads in chunks of up to a wavefront. If needed, it will read in multiple chunks, while already reducing the elements, as they are read in. This is exactly the same algorithm as vector-reductions did on a work-group scale. This is just done at a wavefront scale.

One interesting thing to note, is how `goto`-statements work in C. If the `goto` passes variable instantiations, seemingly no compiler will stop the code from then using the uninstantiated and undeclared variables, it will just do some undefined behavior. This lead to some interesting bugs, before I realized, that I better reinstantiate all array-subscription variables, so everything is in order. This for-loop also makes a platform, for where other operations can be injected into the kernel. In this closure, all operations are performed the amount of iterations they would otherwise, but all in a coalesced access pattern.

The last part is the actual reduction. The reduction itself is a modification of the work-group reduction from final stage of the vector-reductions. The limiter and offsets are simply adapted to work with wavefront, instead of work-groups.

The first work-item in every segment, then writes its final reduction to a local `write_back` array. At the end of all the reductions, this array contains all reductions for the whole work-group. Before returning to the regular Bohrium code, all work-items synchronize at a barrier, to make sure the `write_back` array contains valid data for every work-item. Then all the redundant work-items skip to the end of the kernel, while the rest read the result from `write_back`, places it in the variable it would normally have ended in, and continues the kernel as normal.

6.4.3 Supporting Other Operations Along Segmented Reductions

So at this point, we have a completely working Bohrium, which will at the opportunity, inject a segmented reduction. But this isn't quite enough. My goal is to fully support the Magnetic Field Extrapolation (MFE), which we saw earlier performed substantially worse after transposing the strides – assumed because of the stride-1 reductions.

MFE loads in some data from an array, which is multiplied onto the input to the reduction.

We already have the platform to support these embedded operations, as the code already rewrites the iterator for the arrays for use in the segmented reduction.

I change the code to simply inject the nested code block into the middle of the segmented reduction, at the place commented in the pseudo-code above as "Inject any other operations here".

This works for quite a lot of cases, but two issues arise: The variable name given for temporary variables, is determined inside the function, which generates the embedded code block. This cannot be accessed from the point in the code, where we generate the segmented reduction. The other issue, does not affect correctness, but it still injects the reduction inside the embedded block, which increments an accumulator on the outside. Thankfully, this is overwritten by the segmented reduction, so it will not change anything.

The second issue is easily fixed, by adding a small statement, which checks, if the embedded block is a segmented reduction. If so, it adds a set of forward slashes, to make the following instruction a comment – meaning the reduction is ignored. I could have skipped it completely, but it is very informative, when inspecting the generated kernel, to see what would normally have happened, and to which variables.

The first issue is a little different. This is a classic boot-strapping issue. We want the variable name, but we cannot determine what it is, until it has been generated.

I solve it, by adding an argument to the function, which writes the code. At the end of the function, we make it return a list of generated variable names, from the list of `bh_view` instances, we provided as argument. `bh_view` is the internal, abstract definition of a variable. By doing it this way, we do not have to guess, if it was transformed to a temporary variable, if it is accessed directly from global memory, or something else in the future.

And very conveniently, these variables are returned as string, ready to be injected in the code generator, for the segmented reduction.

In an indirect way, we have also moved the reduction to the end of the rank, like I wanted earlier. Given by the initialization of the segmented reductions, injecting the next rank, filtering out the reduction in the embedded rank, finding the variable name for the output, and reduce the segments afterwards.

6.4.4 Supporting Multiple Segmented Reductions

In MFE, apart from loading some values from global memory, it also has not just one, but three segmented redutions in the same rank.

For simplicity, we assumed earlier, that there would only be one segmented reduction in a rank. We are now ready to expand this, to any number of reductions.

We have three variables, that should be unique for each reduction: `write_back` array containing the result for each thread, local accumulator variable, and lastly an array for scratchpad memory for each reduction.

I quite simply generate one of each variables for each of the reductions, suffixing them with a number unique to the reduction. I had some minor issues with these numbers, as they would change order from each call to the kernel, and thereby ruining the cache. But this was easily mitigated by sorting, and using a global ID number from inside the relevant `bh_views`.

Comparing to the description above, the code now does the following: Initialization of the each segmented reduction, injecting the next rank, filtering out the reductions in the embedded rank, finding the variable names for the output, and reduce all segments at once afterwards.

There is slight waste in local memory, by having three independent scratchpad memory areas. But I deemed it to have a very low impact on the available memory, as each uses up to $128 \cdot 8$ bytes. Having three, sums up to 3072 bytes, out of 32 KB available.

6.4.5 Supporting Nested Segmented Reductions

The next issue with MFE, is that, not only do we have multiple segmented reductions, which depend on a multiplier from an array. They are also nested inside another reduction.

With our `goto`-method, we have some challenges. I cannot just add a `goto` at the bottom of the segmented reduction, telling it to go to the original entry point, because this will not increment the iterator for the outside reduction. The iterator has also not been initialized, as the `goto` skips the preceding lines of code.

My work-around, is to change the overflow header once more. Instead of initializing the iterators at every nesting, I now define them when we enter the kernel, for all work-items. Then at every nesting, the regular work-items, will assign the existing variables a value, instead of both defining and assigning on the same line. The redundant work-items, do not jump directly into the segmented reduction. Instead, I have created a replica of the kernel structure in a closure, that only the redundant threads can reach.

To better understand, imagine a 3-dimensional kernel, where only the outer loop is parallelized in the kernel. The inner two dimensions contain reductions. The pseudo-code would loosely look like this:

```
if (tmp):
  goto label

gfor i0 = 0..dim(0)
  for i1 = 0..dim(1)
    for i2 = 0..dim(2)
      label:
      seg_reduce()
      if (tmp): return
```

It is clear, that only the first iteration would be performed, after which the redundant (`tmp`) work-items return. My proposed work-around would make the kernel resemble the following pseudo-code:

```
if (tmp):
  for i1 = 0..dim(1)
    for i2 = 0..dim(2)
      goto label
      return_label:
  return

gfor i0 = 0..dim(0)
  for i1 = 0..dim(1)
    for i2 = 0..dim(2)
      label:
      seg_reduce()
      if (tmp):
        goto return_label
```

The redundant work-items will have the shell of the kernel structure, which iterates them through the correct number of iterations, until they are expended.

I implemented this fully in Bohrium, and it was very promising, until I got to testing correctness.

6.4.6 Goto and Barriers

After having implemented everything, I find that the `nvcc` compiler or GPGPU has a bug – or an intentional, but undocumented quirk – that `gotos` will disable any further barriers after they are called. This happened, as I was implementing support for segmented reductions nested in a for-loop. I would use the `goto` to skip the rest of the for-loop, go to a subroutine, which incremented a counter, and return to the for-loop. But it kept getting out of synchronization, even though I used a barrier.

I cut the kernel down to the few vital lines, and the issue disappears, as soon as the `goto` is removed. The reason I didn't catch it earlier, was because the segment-size often divided the wavefront size perfectly, meaning no work-items were using the `goto`. And in other cases, it is simply up to chance, to get a race-condition.

I have not inspected the assembler code, but it might have given the answer. In any case, even with the answer, the method didn't work on a fundamental level, so I had to scrap it. As I showed earlier, I had back-ported the `goto` method to vector-reductions. Even here it caused issues, when the work size got irregular. My conclusion being, that it could not even be used to skip a chunk of code completely. While I do get why the compiler would have issues with jumping in and out of a for-loop, it is worthless, if it doesn't work with the vector-reductions.

My theory is, that the compiler or GPGPU has some kind of anti-deadlock system, that disables a barrier, when it is not certain, that all work-items will reach it. The following is stated in the OpenCL documentation:

Documentation:
If `barrier` is inside a conditional statement, then all work-items must enter the conditional if any work-item enters the conditional statement and executes the `barrier`.[16]

My solution is, to change the overflow-guards to not return or jump, but to set a boolean variable to true, which marks the redundant work-items. Then I will inject guards in the code, to keep them from changing anything, outside of the segmented reduction.

After exiting any rank, I insert a guard, which will either call `return` on the redundant work-items, if we are done with them, or `continue`, if they are nested inside a for-loop.

This solution works flawlessly, although less elegant, than the `goto` method.

6.5 Benchmarks

With the features in the chapter fully implemented, it is time to measure the performance improvements, we have accomplished. First of, I run the benchmarks on the new version of the code. The comparison can be seen in figure 6.7.

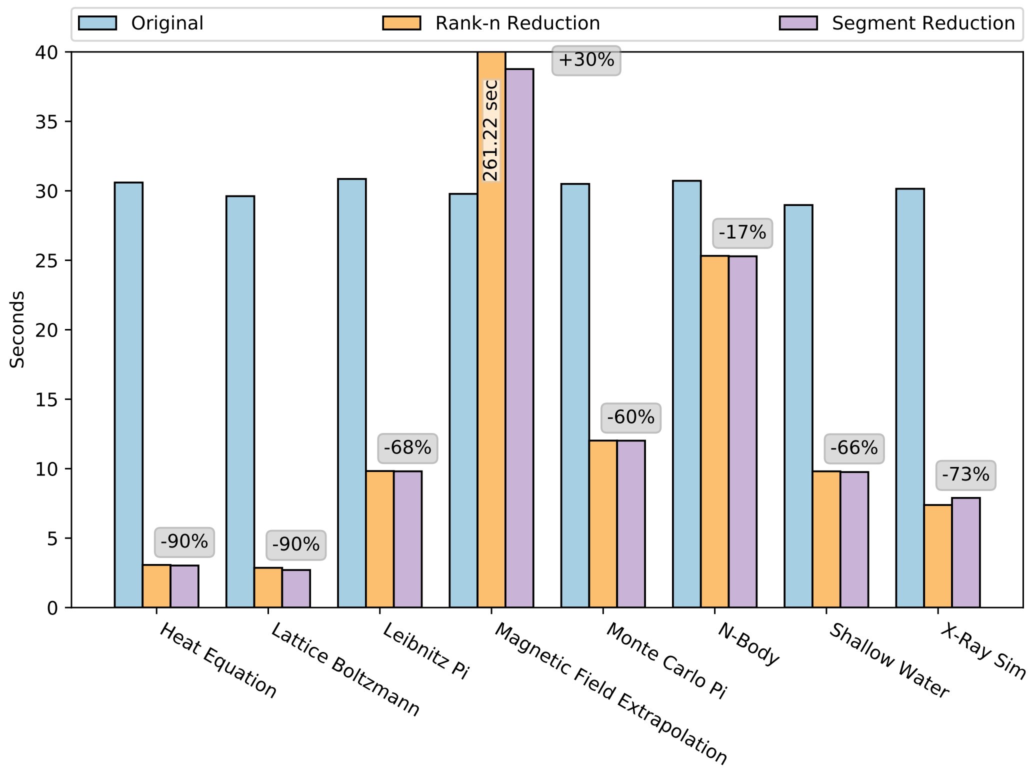

Figure 6.7: Overview of benchmarks. Showing original and latest version of Bohrium as reference.

Seven of the benchmarks have seen considerable improvements through-out the thesis, but most are unaffected by the newest changes. The improvement was targeted at Magnetic Field Extrapolation (MFE), and the improvement is clearly shown.

Impressively, the time for MFE went from more than 260 seconds and down to around 40 seconds, because of the segmented reduction. Sadly, it does not make up for the performance loss, we saw with the change to access patterns.

But the trade-off now seems to be more acceptable. The performance of MFE suffers 30%, but in return, Heat Equation, Leibnitz Pi, Monte Carlo Pi, and X-Ray Sim see staggering improvements after the access pattern has changed. So it is not as easy, as just discarding the changes.

Except for X-Ray Sim, the addition of segmented reductions, does not have any negative impact on the other benchmarks.

6.5.1 Revisiting the Auto-tuner

As in all previous chapters, I revisit the auto-tuner, to see what kind of performance, we could acheive.

Auto-tuner Support

Some of the changes I have made, has excluded support for the auto-tuner, as the auto-tuner wants to change the work-group size, while the segmented reductions allocate local memory, which can only be done using a static work-group size.

Another reason for the limitations, is because the segmented reduction is implemented to assume a flat work-group structure. Although it allows for 3-dimensional kernel parameters, the parameters need to have all work-items in the first dimension. This also heavily limits the number of possible kernel parameters, which we can try.

I made the relatively easy fix for the memory issue. When the auto-tuner is enabled, the allocated memory is just fixed to the maximum work-group size for my card – 1024 elements. This should not have any large effect on the benchmarks.

To not have this enabled permanently, I added a hidden configuration. If the environment variable `BH_OPENCL_AUTOTUNER` is set, only then, will the feature be enabled. This conveniently also enables and disables the output, which the auto-tuner parses.

Results

To see how much room we have for improvement, I ran the auto-tuner for every benchmark, and compared the latest version of the code, with the optimal selection of work-group parameters for each kernel. See figure 6.8.

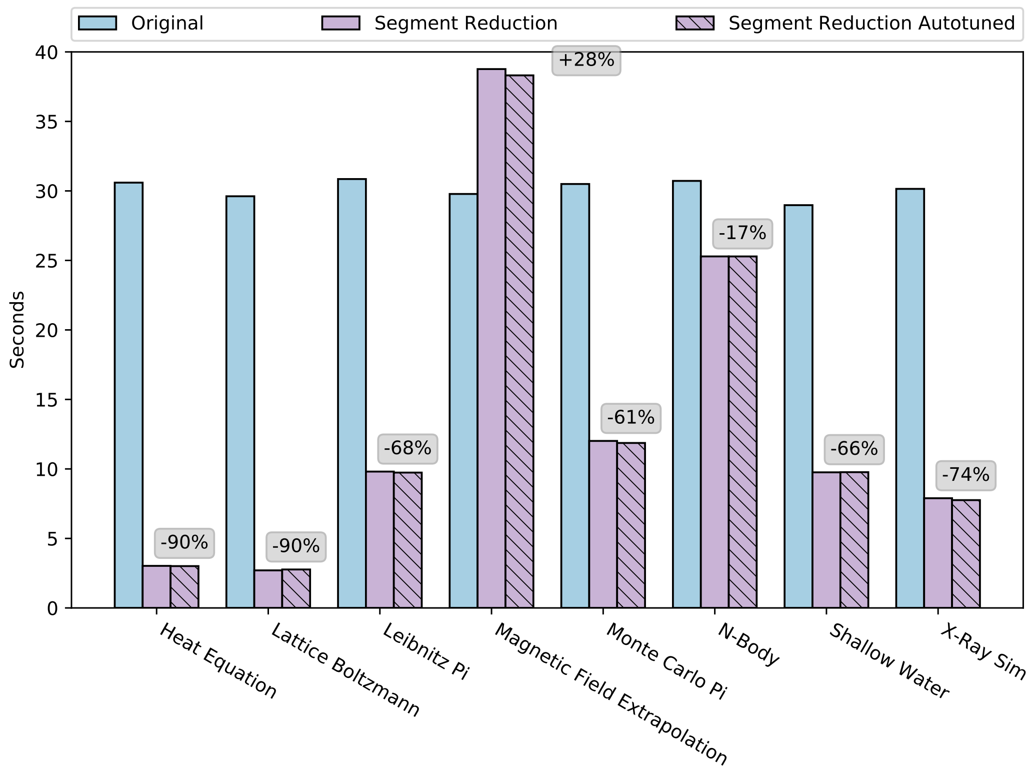

Figure 6.8: Overview of auto-tuned benchmarks. Showing original and latest version of Bohrium as reference.

As we can see from the results, the performance increases from the auto-tuner has stagnated from each added improvement. But it is not because the auto-tuner is being restricted, it is because the changes made to Bohrium, fixes the places where the auto-tuner had an advantage.

As of now, not even the auto-tuner can correct Magnetic Field Extrapolation. This is because of two issues: The kernel parameter that worked best before, is now not possible, and the segmented reduction has significantly changed the access pattern of the kernel.

It is still the single kernel with segmented reductions, which takes the most of the execution-time. When I inspect the best kernel parameter, I find the best parameter to be $384 \times 1 \times 1$. This is 3 times the default work-group size, but it does not have a significant impact.

6.6 Partial Conclusion

The newest changes has mostly not improved performance across the benchmarks. The only real beneficiary, is Magnetic Field Extrapolation (MFE), which was also the target of this chapter.

With the latest changes, MFE is approaching a level of performance, where it could be deemed worth the penalty, of a slower MFE, compared to the massive gains of the other benchmarks.

Depending on future work of Bohrium, it might be beneficial to include the segmented reductions from this chapter. If performance of MFE can be improved in other ways, that might exclude segmented reductions, it will likely not be missed.

Chapter 7

Future Work

Even though, there has been a lot of radical changes to Bohrium, and we achieved much better performance, there are still several thing to look at going forward.

The ultimate goal is to produce OpenCL code, which performs just as well as a handwritten kernel, or if possible, even better. One of the limiting factors, at this moment, is the fuser. It would seem, that is needs to be more aggressive, and more flexible to which operations, that can be fused together.

There are also several methods, that could be used to improve Magnetic Field Extrapolation.

7.1 Fuser

There are still lots of cases, where Bohrium produces more kernels, than what is technically needed. An example of this is, the Heat Equation benchmark, where the handwritten version has a single kernel, while Bohrium generates a handful of kernels for the same part of the algorithm.

Other benchmarks show the symptoms of the fuser needing to be more aggressive, while it does not, at the moment, change the overall execution-time. This can be seen in Monte Carlo Pi, where the bulk is calculated in one kernel, but an extra kernel is spawned, just to divide the single scalar result. This would almost be free to apply, right before writing the result to global memory on the GPGPU.

I have excluded the discussion of vector-types in the OpenCL kernels from this chapter, as I will discuss these in chapter 9.2.2. In the right kernels, these can have a great impact on memory throughput.

7.1.1 Transpose

As I have demonstrated in the NumPy bridge, a great technique to control reductions, and to maximize parallelism, is to transpose the input tensor.

In the future, I want to strip or ignore the shape of all non-sweeps, and simply attribute them a length during fusion – whenever mathematically allowed. The shape is reapplied when returned to NumPy.

When fusing operations, the sweeps should be transposed and flattened, as much as we are mathematically allowed to, to maximize parallelism. Then all other operations are fused onto the sweeps, in the order they are required by data dependencies.

In this step, we can also choose, which axis of a recursive reduction is favorable to have as the inner-most reduction.

The fuser currently has trouble fusing operations together, when they do not exhibit the same size in each rank. Even though a transpose could have legally been applied.

7.1.2 Reshape

The next step is to more aggressively reshape the input data.

The sweeps should be one of the only operations, which dictate the shape of the kernel. If the programmer transposes some of the input data, or makes a slice of a rank-n tensor, we might be forced to keep the shape.

All operations on contiguous data, can be performed in whichever shape is needed, as long as the length is correct, as there are no horizontal data dependencies.

With reshaping, we can also transform a recursive reduction to a regular row-wise reduction. Imagine calling sum twice on a rank-5 tensor. This will result in two separate reductions, which outputs a rank-3 tensor. If these two reductions operate on a contiguous block of data, we can trivially reshape the input data into a rank-4 tensor, where the two reduced axes, has been concatenated.

But complications arises, if any operation comes in between the reductions. In some cases, we might be able to do some tricks, to fuse them together anyway, but otherwise, they will have to stay.

Saving a reduction, can in itself reduce execution-time. But we can also more easily optimize on a kernel with fewer operations, as the kernel will become less convoluted.

7.1.3 Merge

Operations of the same length, can trivially be fused together. These bundles of operations, can be fused onto sweeps, by matching how many of the sweep's ranks are needed to get the same length as the operation.

It might even be favorable, to fuse operations that are not the exact same length. Imagine a rank-3 tensor, with a reduction in the inner-most rank. The kernel is still parallelized across work-groups, for the first two ranks. Here, we might as well merge in operations, if they have the same shape or length.

There are of course complications to this, as the fusing quickly becomes an NP combinatorial optimization problem. But by the end of this thesis, I believe the fuser is where the next performance gains are to be found.

7.2 Optimize Python

In the farther future, it might be Python that bottlenecks performance. If Bohrium succeeds in generating kernels in the quality of handwritten ones, the only difference could end up being Python and C++.

There might be a number of solutions to this. For loops converging towards a value – code which is inside a for-loop, or another Python-level code structures – could be intercepted, and optimized for better hardware acceleration.

We could also look at other implementations of Python, to gain performance. Two common options, are to replace the interpreter with the JIT compiler PyPy, or with the transcompiler Cython.

PyPy would work as a drop-in replacement for Python, and speed-up native Python code. It will not speed-up Bohrium itself, but it might save time in between calls to Bohrium.

Cython could be combined with more advanced techniques of statically generating Bohrium kernels from the Python source code, whenever possible, to avoid JIT compilation overhead. Python is inherently dynamic, which means boundaries for for-loops, and even functions might change out of the blue. But for kernels, that can be statically generated, they can be bundled as compiled binaries. Everything is then transcompiled using Cython into C-code, and then compiled into a native binary – often resulting in a performance boost[20].

7.3 Solution to Magnetic Field Extrapolation

Although Magnetic Field Extrapolation (MFE) never got to achieve the full performance, that we could create with the auto-tuner, all hope is not lost.

7.3.1 Investigating the Access Pattern

The issue is clearly of access pattern. If the kernel performed no memory reads, it would have no reason, to run as slow, as it does.

From the single kernel, that causes issues, we have the following memory access in the inner-most rank of 5. These are nested inside two for-loops for performing reductions, that are called recursively, and three ranks parallelized across work-groups on the outer-most axes. Refer to 4.4.1 for a better illustration of the code's loop-header.

I have omitted the write-back of the reduction, as it is performed fully coalesced outside of the for-loops, just like every other well-functioning kernel does.

Memory Reads of Heaviest Kernel in MFE

```
s8_8 = a8[ +i0*5625 +i3*75 +i4];
t6   = a7[  +i1*421875 +i2*5625 +i3*75 +i4] * s8_8;
t9   = a10[ +i1*421875 +i2*5625 +i3*75 +i4] * s8_8;
t11 = a12[ +i1*421875 +i2*5625 +i3*75 +i4] * s8_8;
```

The problematic part, is the irregularity of the memory strides. A regular access pattern would use every iterator from the beginning to the end, but this is not the case here. `a8` skips `i1` and `i2`, while `a7` and the others skip `i0`.

With kernel parameters of $128 \times 1 \times 1$, we can by the point of view of the work-group, assume `i0` and `i1` are constants. Stripping those parts of the stride calculation, we can try to figure out, why the kernel performs, as it does.

MFE Kernel Stripped of Constant Strides With Param. $128 \times 1 \times 1$

```
s8_8 = a8[ +i3*75 +i4];
t6   = a7[  +i2*5625 +i3*75 +i4] * s8_8;
t9   = a10[ +i2*5625 +i3*75 +i4] * s8_8;
t11 = a12[ +i2*5625 +i3*75 +i4] * s8_8;
```

What we can see here, is that all work-items within the work-group, reads the same value sequentially from `a8`. This is not particularly worrying, as I assume the cache will solve this. The second observation, is that each work-item reads 5625 values sequentially, stopping where the next work-item started.

Before implementing segmented reductions, I ran the MFE benchmark through the auto-tuner – see figure 5.9. This brought down the execution-time by 60%. By a small margin, this is the fastest the code has run.

If we transposed all the kernel parameters, how would these calculation then look? If we transpose the parameters, the best approximation we get, is $1 \times 2 \times 64$, as the GPGPU limits the last parameter to 64. If we assume, that the first two parameters are constant, we will have a work-group with the following access pattern:

MFE Kernel Stripped of Constant Strides With Param. $1 \times 2 \times 64$

```
s8_8 = a8[ +i0*5625 +i3*75 +i4];
t6   = a7[  +i3*75 +i4] * s8_8;
t9   = a10[ +i3*75 +i4] * s8_8;
t11 = a12[ +i3*75 +i4] * s8_8;
```

What we can immediately see, is that the three last reads are identical across every work-item. As I did before, I will assume, that this is handled by the cache, and that in fact, every cache-line being read from global memory, is being fully utilized – although sequentially.

`a8` is now the opposite, as every work-item will read in its own value from it, with no overlap between work-items. But having three out of four arrays to better utilizing memory transactions and cache, is probably the best compromise.

7.3.2 Transposing Kernel Parameters

There are ways we can achieve good performance now, but it might not generally improve performance in every benchmark.

I modified the newest version of the code, to disable segmented reductions entirely, but keeping the controls, that decided when a segmented reduction is appropriate. I used this information, to transpose the kernel parameters, whenever a segmented reduction, would have been injected.

This effectively means, that for the single bad kernel of MFE, and for some miscellaneous kernels in other benchmarks, the segmented reductions will be performed sequentially, but with as large a space between each work-item, as possible. This is for normal kernels, not like the kernel shown above, with the irregular access pattern.

The risk is to parallelize on work-items, which provide a uncoalesced access pattern. Imagine a rank-4 kernel with reductions on the inner two ranks – inspired by MFE:

Fictitious Access Pattern of rank-4 Kernel

```
t0 = a0[ +i0*421875 +i1*5625 +i2*75 +i3];
```

No matter how we distribute the work-items, we either parallelize on `i0`, and each read is 421875 apart, or we do it on `i1`, and each read is 5625 apart. Having them close might help, if it limits memory transactions, by reusing the cache multiple times, but there is no guarantee. The reason it will work for MFE, is because there is no `i0` in the calculation – which is not typical.

The MFE kernel spawns as a rank-3 kernel. The kernel parameters are changed from $128 \times 1 \times 1$ to $1 \times 2 \times 64$. I also changed the rank-2 parameters from 128×1 to 1×128 to not unfairly target the MFE kernel exclusively, by making it a general rule.

The results are quite positive. As seen in figure 7.1, MFE is now, with just a few lines of code, brought down to almost the same execution-time, as the auto-tuner can deliver. The changes also made almost no difference for the other benchmarks, which is very lucky, as making unconditional improvements were not a given.

I do not approve of this solution, as I can not justify, that it will work well, with any workload. It is possible, to leave it as an option, in the Bohrium configurations for now. This would allow the user, to enable it, as an experimental feature.

The only correct way, to solve the problem, of access pattern on a general basis, is to implement, a more effective segmented reduction.

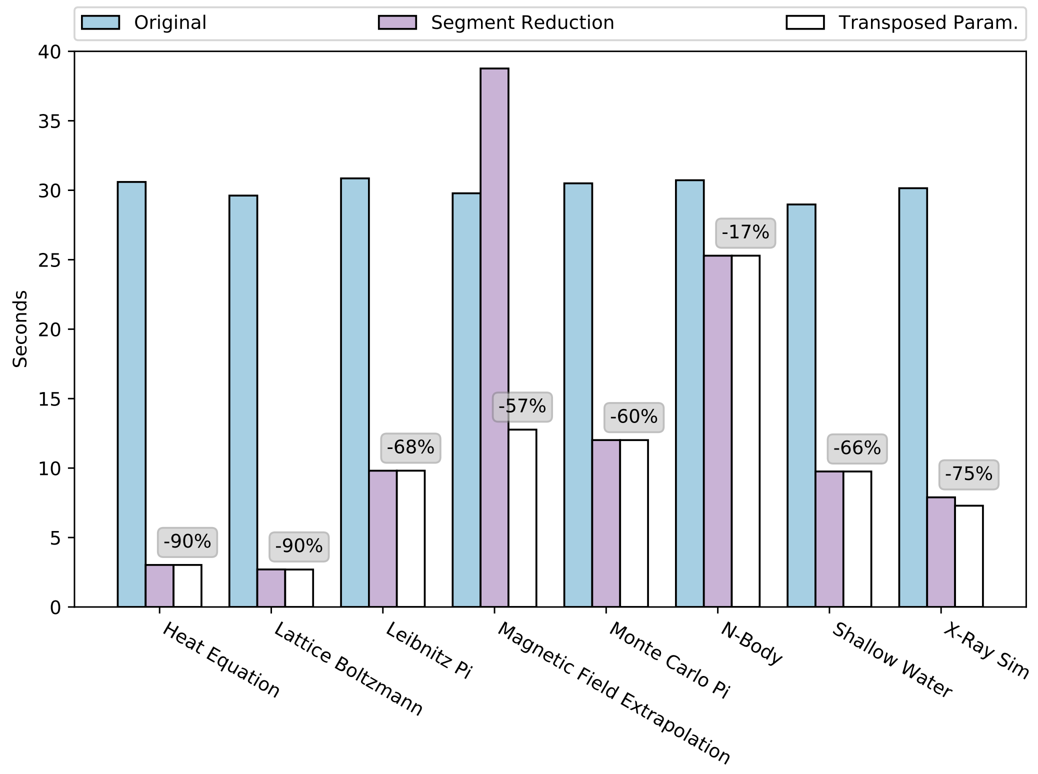

Figure 7.1: Overview of benchmarks. Showing original and latest version of Bohrium as reference.

Chapter 8

Tests and Correctness

As the implementation has been completed, and I know it works in the cases, I have prepared it for, I put it to the test, to make sure it is stable enough for continued development and benchmarking.

As mentioned multiple times, we need to be sure, that the new improvements do not change the results of calculations, as they are otherwise useless.

I did not categorically run Bohrium's test suite after every change, as I probably should have. But I will for this chapter, as well as describe some of most interesting discoveries.

I will not be outlining every single bug, that got uncovered by the tests, as most are uninteresting off-by-one, missing a character in a copy-paste, and so on.

The procedure is straight forward. I will run every test included in Bohrium, as well as my own additions, and fix every issue I meet.

I want to state already, that as of the end of this chapter, Bohrium, including my modifications, passes all Bohrium's tests.

8.1 Removing the Prefix-sum Again

In Bohrium's test suite, there are some extensive tests, which verify the result of Bohrium, compared to NumPy.

Back in section 3.4, I told, that I could very easily implement a vector-prefix-sum, which is the only other type of sweep, apart from reduction.

It did work, and still does, but the tests I made were inadequate. I had made the quick assumption, that just like reduction, the result of the operation, would not be used later within the same rank. But this is only true for reduction, as the output is always of a different shape, than the input.

Within the test suite, there were at least one, but likely more benchmarks, which used the result of the prefix-sum, within the same rank. They are mathematically allowed to, but my rather rough implementation does not support it.

To fix it, I would have to make changes to the fuser, to either disallow this case, and run the prefix-sum sum on the CPU, or by making the fuser split the kernel into two parts – one before and one after the prefix-sum. And of course, only split when needed.

These changes could be extensive, and seemed out of scope of this thesis, so I chose to disable the GPGPU prefix-sum for now, but leave it in for future use.

8.2 Reducing a Single Element

Like described in the section above, I have made the assumption, that a reduction will not be used in the same rank of a kernel. But this is only true in the general cases. If the user requests to reduce a single element, it could make Bohrium trigger a vector-reduction, although it was not really needed. In the particular test, the output of the reduction was then divided by a number, and returned to NumPy. But the reduction had not been made yet, as I always perform the reduction last.

This is the only case, where moving the reduction last is an issue, but thankfully, it has a simple solution. We just copy the input to the output. The reduction of a single value, is simply itself. And as no other handling is needed, we can skip the whole process, without the Bohrium backend running an empty reduction.

8.3 Disable Vector-reductions for Complex Numbers

There are one or more tests, that challenged the reduction of complex numbers. I had been working on it for a while, but there seemed to be some deeper issue, as no amount of neutral elements, nor short or large vectors seemed to provide the correct result. I chose to spend my time elsewhere, and instead made a check, that specifically disables vector-reductions for complex numbers, until I, or the Bohrium developers, take the time to find the cause.

My first priority is correctness, which is the reason for disabling it. The reason for not taking the time to fix it now, is to focus on other more pressing issues, like Magnetic Field Extrapolation.

8.4 Local Memory Limit

Outside of the test-cases, I found an issue with Galton Bean Machine (GBM). The code is formatted in a way, that triggers the segmented reductions, even though it is actually not needed. Segmented reductions are only really useful, when reading from global memory. But GBM does not allocate memory, as the data is temporary.

Running GBM anyway, will still work. But an issue arises, when the data-size increases. The segmented reduction allocates a write-back array for each element, and quickly exceeds the available local memory.

I implemented a quick work-around, that when the number of elements cannot fit in the total size of local memory, segmented reduction is disabled for the kernel. The work-around was meant to stabilize the code, so it would not crash in a case, which was easily avoided. The error still persist in the same way, as when a user exceeds global memory, if the user has multiple reductions that together exceed the limit.

It is not an easy issue to fix, as it will require a type of swapping technique, or splitting the work-load, which is out of scope of this thesis.

I realized later, that this exact issue, was a symptom of a bug in my code, which allocates write-back memory. It is never needed to allocate more write-back memory, than the number of work-items in a work-group.

I kept the work-around above, as having a segment of this size, is out of the intended scope of the chosen algorithm. If segments of this size is needed (and they use global memory), it might be more useful to make inter-work-group sized reductions, like in the vector-reduction.

8.5 Tests for Transposed and Segmented Reduction

During development of chapter 5 and 6, I have been adding hard and tricky test scenarios to a Python script. These include nested reductions, small segments, even and odd sized segments, too large segments and cases, where I wanted to make sure, that the fuser did not do an illegal fusion.

I set up to script to run every test case using NumPy and Bohrium. The result of each is recorded, and compared to the other. First, I test if the array they output are a perfect match, in which case, it was a success. And alternatively, I switch to a test, which checks the difference between the results, in case of rounded off floating point values.

Because I started working with commutative operators in the reduction, it is needed to test for rounding errors, as it is no longer performed identical to NumPy.

There are 23 tests in total, so I will not outline each one, but I will highlight two of the most important ones, that helped me identify errors.

8.5.1 Test Case: Reuse Reduction Inside Rank

I wanted to challenge the code, which generates the write-back and scratch-pad memory for the reductions. Therefore, I added two reductions of the same axis, on the same data, for them to be merged into the same kernel.

This created the issue, that three variables for segmented reductions got redefined, and the compiler couldn't continue.

The solution was to sort the reductions, and remove duplicates, but save them, for the write-back phase. This meant, that the segmented reduction was only calculated once, but the write-back phase included both write-backs, which stored the same variable in global memory, in the two output arrays.

This is certainly a rare case for real-world code, but we need to be sure, that the kernels no matter what, generate valid code.

8.5.2 Test Case: Deep Nesting

To test the extreme cases, I made an array of five dimensions, and performed a reduction on the inner-most axis. The reason why this is a challenge, is because any array up to four dimensions, with a segmented reduction, is covered by the hardware parallelization. What I mean is, that the three outer dimensions are split by OpenCL across work-groups, and the segmented reduction will then not be nested in a for-loop.

The same happens for five dimensional arrays, with reductions on the two inner-most axes. Although this also contains a for-loop for the second reduction, it is a case, that has been tested to work through Magnetic Field Extrapolation.

OpenCL cannot help us, when we move the parallelism to far inwards, as we changed the code to do in chapter 4. The solution Bohrium automatically provides, is to insert for-loops at the outer ranks.

The problem I faced was, my trick with `continue` and `return` guards for redundant items in 6.4.6, proved to be too extensive to implement, as it kept hitting boot-strapping issues – results of a decision made later, affected the code generator now.

My new solution is to wrap any operation outside of a segmented reduction with a `if (redundant)` predicate. These are quickly discarded for kernels, where no redundant work-items are used, as the compiler will perform constant-propagation, and remove the predicate.

I also assume, that the performance impact will be minimal on kernels including segmented reduction. From all the kernels I have inspected, there would be between 0 and 4 checks through-out the kernel.

8.5.3 Numerical Stability

For the vector-reductions, I discussed numerical stability, in reference to NumPy. We found, that NumPy has been updated, to actually get a more correct result, than they used to.

I did not address numerical stability for segmented reductions, as a direct issue, as I have actually followed the same algorithms as vector-reductions. And through-out the tests, the stability has been challenged, as all tests are compared to NumPy. None of the tests show a significant difference in the result. The threshold for the tests are $1e^{-6}$, which is never exceeded.

8.6 Benchmarks

Lastly, I want to run Bohrium through the benchmarks once more. We have made some significant changes to the code, to improve correctness. These changes should not affect the performance by much, but it is important to show the change, especially if the changes causes large degradation of performance.

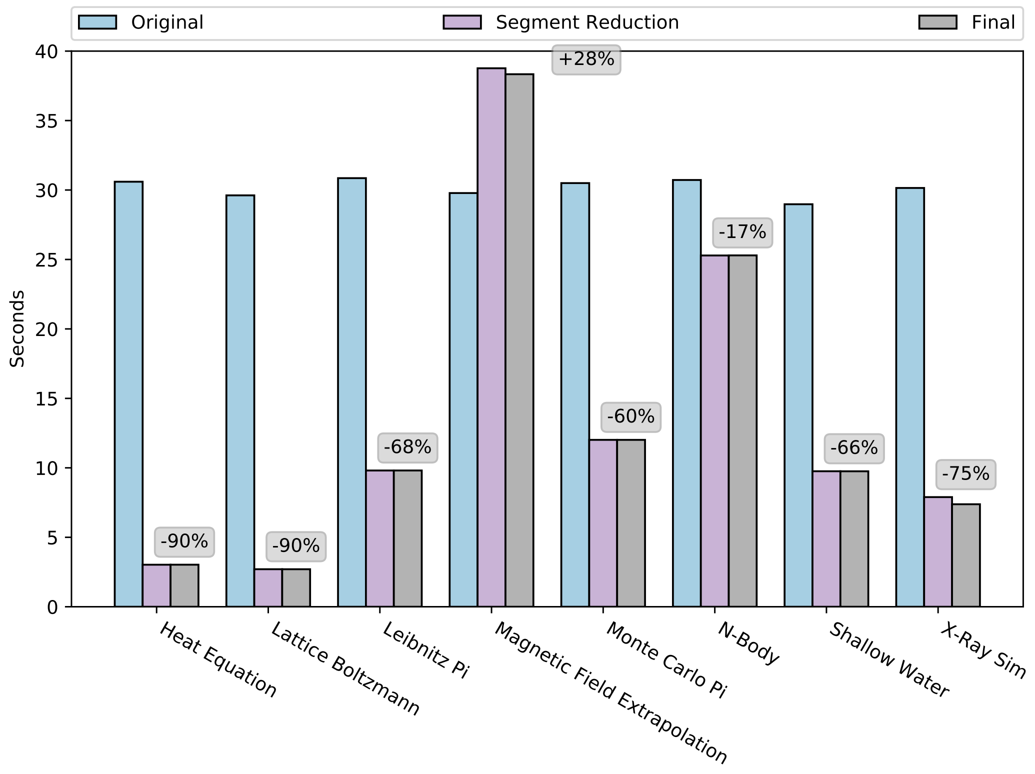

Figure 8.1: Benchmarks after correcting the errors in chapter 8. Showing original and latest version of Bohrium as reference.

In figure 8.1, we can see all the benchmarks, and there is not any large difference at all.

8.7 Partial Conclusion

In addition to the large performance gains, we can now reassure ourselves, that it did not come at the cost of correctness.

After a few changes, we are now able to run all benchmarks and all tests, without any sign of error.

Performance has not been affected noticeably, neither negatively nor positively, which might be attributed to the corrections being of full crashes, rather than of miscalculations.

Chapter 9

Benchmarks

In 1.9, we looked through the methodology and benchmarks used in the thesis. These have been shown through-out the report, and were used to argue for, and show the progress I made.

After all the improvements to Bohrium, I want to take a deeper look into the performance we have achieved, and reflect on the improvements.

9.1 Benchmark Setup

For a matter of completeness, and for later reproduction of the results, I will shortly list the configuration used for my benchmarks.

The hardware configuration can be found in appendix A.1. This is simply the Bohrium and Benchpress configurations.

All Bohrium configurations are set to the default values. For all tests after introducing vector-reductions, the trick to transform vector-reductions into matrix reductions, that can be finished on the CPU, has been disabled (see chapter 3.3). I investigate the fuser later in this chapter, but it has been kept at its default setting through-out the results shown in the thesis.

After chapter 4, where I altered the access patterns, I have changed the default work-group parameters to always have 128 elements in the X dimension, while any other has just 1.

For later reproduction or comparison, I want to include the benchmark parameters. Together with the specifications in appendix A.1, it should be detail enough for anyone to make the same benchmarks as I have.

Benchmark	Seconds	Problem Size	Iterations
Monte Carlo Pi	10	450,000,000	55
Monte Carlo Pi	30	500,000,000	60
Leibnitz Pi	10	480,000,000	25
Leibnitz Pi	30	500,000,000	70
Galton Bean Machine	10	480,000,000	120
Galton Bean Machine	30	500,000,000	350
N-body	10	7,500	25
N-body	30	8,000	68
X-ray Sim	10	35	32
X-ray Sim	30	40	55
Magnetic Field Extrapolation	10	75	26
Magnetic Field Extrapolation	30	80	55
Heat Equation	10	10,000	33
Heat Equation	30	11,000	60
Shallow Water	10	4,750	26
Shallow Water	30	5,600	50
Lattice Boltzmann	10	1,400	32
Lattice Boltzmann	30	1,900	53

The problem size is repeated for every dimension. Meaning; if a benchmark takes 2 dimensional parameters, it is the same number repeated twice.

9.2 Bound Theoretical Performance

When optimizing any task, it might be worth considering, what the theoretical limit of optimization could be. If we converge this limit, the benefit of further improvements, might not be worth the effort.

There are two factors we can bound: Memory bandwidth and processor operations.

9.2.1 Processor Operations

Most of the workloads in the benchmarks are memory bound. Only the few embarrassingly parallel benchmarks, really depend of the processor's ability to churn through instructions, as fast as possible.

In the end, any meaningful kernel, will write out some result to global memory, but for embarrassingly parallel tasks, it will be a proportionally small part of the total time spent.

All of the embarrassingly parallel benchmarks, except for Leibnitz Pi, uses random number generated on the GPGPU. To isolate the performance of Bohrium, and not of the random number generator, I will use Leibnitz Pi to estimate our effective performance.

The 30-second benchmark of Leibnitz Pi, provides an average of 30.97 billion operations per second. This is counted by Bohrium, as simply the number of primitive operations per second, dispatched to the GPGPU, divided by the spend time.

To some degree, this number is only what the theoretical implementation of the algorithm would use, and not necessarily, how many operations the practical implementation will be using. It does not take into account how the GPGPU might calculate its results. For example with the reduction, at the end of the kernel. The number Bohrium provides, does not take into account, which algorithm we are using for the vector-reductions. It also does not take into account, that generating the index-number for each element, might not be a separate operation at all on the GPGPU.

All of the operations inside the kernel are using `long` or `double`. If we assume the number of operations are comparable to double-precision floating-point operations, we will have 25%

of the theoretical limit, which is 122.5 DP GFLOPS on this specific GPGPU (see appendix A.1).

What brings this number down from 100%, could be many factors. One of the most important ones, is still the reduction.

This reduction is performing several reads and writes to local memory. Nvidia states, that memory read or write, takes 4 clock cycles on average for a wavefront, while the latency is 400 to 600 clock cycles[44, 5.2 Memory Instructions]. We are also calling barriers, which stalls some of the computations. At the end, we write the sub-results to global memory, and spawn a second kernel, which yet again reads the data in. This all takes time, which highly affects the throughput. And none of the just mentioned operations, are counted as operations, as they are not part of the algorithm. They are meta-operations, which support the ability to use reductions on a GPGPU.

All of these things skews the numbers, but without them, we would have an even lower performance. Even if we are using some expensive operations for the reduction, our benchmarks show, that it is worth it.

To optimize the performance further, there is probably best chance of gains in the reduction. For the other calculations, we cannot change much. Even if we were to use vectorized types, as Nvidia states, that they do not affect performance, in regards to compute speed[44, Instruction Optimizations]. We could be using some compiler flags to enable faster floating-point operations, at the expense of loosing some checks and accuracy. But it is generally not a good idea, as we cannot assume, what level of accuracy the user needs. The result of the optimized floating-point operation, might even violate both OpenCL and IEEE 754 specifications[4, Optimization Options].

The exact usage of the operations throughput is limited. Apart some very specific use-cases, it is rarely the raw instruction through-put that limits performance. Cases where it matters, are for example password-cracking, where the target hash can be stored in local or private memory, and everything else, is generated and calculated in each individual work-item.

If you are fine-tuning a specific part of the code, it would also be better to isolate that part, when calculating the throughput. The problem with this, is that Bohrium might remove the code entirely, or the generated code might change, when doing this. In this exact case, if we removed the reduction, it would begin to flush all results to global memory, and thereby negatively affect performance.

9.2.2 Memory Bandwidth

One of the most important aspects of performance, as we have seen through-out this thesis, is the optimization of memory access.

Nvidia even has a very clear-cut message to developers, who look for optimizations:

Nvidia Guidelines:
Bandwidth is one of the most important gating factors for performance. Almost all changes to code should be made in the context of how they affect bandwidth.[44, 2.2 Bandwidth]

The Nvidia GeForce GTX 970 used in this thesis, has a theoretical bandwidth of 224GB/s, stated by the documentation. Nvidia does not provide the frequency of the memory clock, but rather a vague “Gbps” in its place. The frequency is around 1750MHz, but is likely hidden, because the hardware moves data four times per clock-cycle, instead of twice, as they have used in the past[44, 2.2.1 Theoretical Bandwidth Calculation]. This value is supposed to be multiplied by the bus-width, to get the theoretical bandwidth.

To find the effective bandwidth, Nvidia recommends, that we find it, by measuring the timing of a specific kernel[44, 2.2.2 Effective Bandwidth Calculation].

Nvidia provides us with the very simple formula below. We simply have to find the total amount of bytes, which a kernel reads and writes from memory, and type them into the formula. Then we run the kernel, and insert the time it took to complete it. The result with be the effective bandwidth.

$$\text{Effective bandwidth} = ((B_{\text{read}} + B_{\text{write}})/10^9)/\text{time}$$

The number is divided by 10^9 to convert the result from bytes to GB/s.

This result is assumed to be way lower, than the theoretical one, as we first of all, will have to do some calculations in between reading and writing. But also because the theoretical limit assumes no waiting for anything, not even a cache-miss or the scheduling of wavefronts.

For demonstration, I choose to use the results from the standalone reduction, I implemented in chapter 3.2.3. These are run as independent vector-reductions in OpenCL. The data is already initialized in global memory, and has to be read in. The code is also split into two kernels, but I will for now focus on the first kernel.

I use a workload size of $1024^2 \cdot 768$ elements of `float32`, equivalent to 3 GB of data. As the work-group size is 256 in this test, it will also write out a sub-results to global memory for every 256 input elements.

$$\text{Effective bandwidth} = ((1024^2 \cdot 768 \cdot (1 + \frac{1}{256}) \cdot 4)/10^9)/0.033342816 = 96.99GB/s$$

The effective bandwidth is found to be $96.99GB/s$. Compared to the theoretical limit, we are at 43.4%, which is clearly far off.

Compared to the previous results in chapter 3.2.3, these numbers do not include the second kernel, neither did the other results contain the improvements found in appendix D.

To improve performance, several changes are necessary. In a previous project, I wrote a similar vector-reduction, which peaks at 158 GB/s on the current software setup[46]. This equals to 70% of the theoretical throughput.

One of the major differences, was the use of vector-types. OpenCL supports data-types like `float4`, which is a group of four 32-bit floats. Normally, a read of a single 32-bit float, will dispatch a read from global memory of 32-bits width. If put it in a for-loop, it will likely dispatch a 32-bit read at every iteration. By switching to `float4`, we tell the compiler to read and write data in 128-bit sections instead – it translates to a single opcode with this width. This obviously reduces the number of the memory transactions, but it also improves on latency, as more reads can be queued up with less waiting[5].

The use of vector-types is not yet supported in Bohrium, which will be needed, before it can be used in the reduction. It can be trivially added in some situations, but if the length of the input data is not divisible by the vector-type's length, it will require some additional code to handle it.

The other major difference, was to use the same work-group to read multiple chunks of data, instead of spawning several work-groups. This is much like the method used for the second kernel of the vector-reduction in this thesis, as is illustrated in appendix B.

With a simple memory-copy, Nvidia can demonstrate an effective throughput of 160 to 180 GB/s using vector-types on a NVIDIA Tesla Kepler K20X, which is a dedicated compute card[5]. The card is not directly comparable to the one in this thesis, but they are of the same Maxwell architecture. The number is just to show, that the optimized vector-reduction, might be at its practical limit, while the theoretical limits seems to be far off. The theoretical limit of the K20X, is 250 GB/s. This puts them at 72% of the theoretical limit – close to our 70%, which included reduction. This is also within range of AMD's case study, which shows a vector-reduction at 77% of the theoretical limit on an AMD GPGPU[33].

To conclude, there is still work to do, to make Bohrium faster. And it might not just benefit the reduction, but also any other memory-intensive kernel. Especially vector-types, are a feature, which should be investigated. It is also worth noting, that the theoretical limit might not be practically achievable, when both Nvidia, AMD and our kernels only move within the $70 - 80\%$ bounds. So I would probably rather look, at the maximum speed of memory-copy, than of the theoretical limit, as it gives a much more usable bound of highest throughput.

9.3 Comparing Performance to Handwritten Kernel

From all these benchmarks and improvements, we make ordinary Python/NumPy code run faster. It is unequivocally easier to write large simulations in NumPy, than writing efficient kernels directly. And any changes and extensions of the application for the kernel, will be weighted by added working time. This also doesn't account for any architectural changes, like changing from CUDA to OpenCL, or maybe in the future, where Bohrium might support multiple GPGPUs or computer clusters.

But if Bohrium is not efficient enough, it could be advocated, that there are still applications, where writing the kernel directly, is the better option.

To measure the true benefit, I have look at the only Benchpress benchmark, which has a handwritten OpenCL kernel available, the Heat Equation benchmark.

Show in figure 9.1, is a comparison of the same problem size, as the 30-second benchmark of Heat Equation.

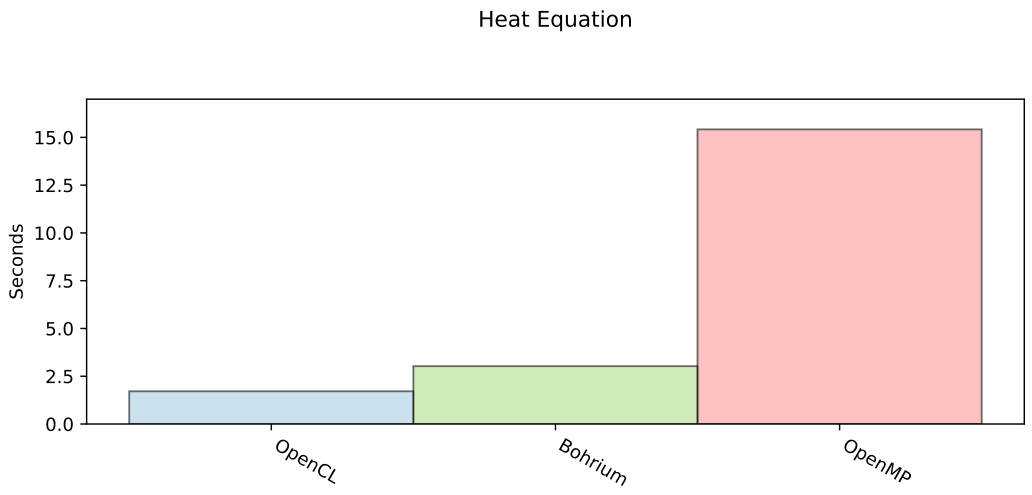

Figure 9.1: Comparison of Different Versions of Heat Equation. The Comparison is Between Handwritten OpenCL, Bohrium with OpenCL, and OpenMP-improved CPU Version.

OpenMP can immediately be disregarded, as it is several factors behind the other two.

As for Bohrium, there can be made a direct comparison, as they are solving the same problem, with the same algorithm, using the same software platform (OpenCL) and the same hardware.

The handwritten OpenCL kernel consists of a single OpenCL kernel with 27 lines of code, and 163 lines of C++ code on the host.

The Bohrium version consists of 53 lines of Python code. Bohrium ends up generating 9 kernels. Of which some of the functionality of the single handwritten kernel, has been split into multiple kernels. 4 out of the 9 kernels are run at each iteration, which means the functionality is likely split into these.

The C++ and Bohrium versions has the difference, that the C++ finds the delta of the convergence by summing with a for-loop on the CPU. The Bohrium version does it on the GPGPU. Maybe Bohrium has an advantage here, as it supports more complex operations, without the programmer having to write, or employ it manually. But in some cases, the C++ developer might know, that the reduction is more efficient on the CPU, for a specific reason.

What we can learn from this, is how the fuser might need to learn some need tricks, to better merge more calculations into the same kernel, and hopefully approach the same conciseness and performance of the handwritten one. And if any operation is advantageous to perform on the CPU, we could find some metrics to make this decision.

Although the competition is not a matter of least amount of code, there is a significant difference. When looking at the C++ code, it has a lot of low-level calls to transfer memory back and forth, between the host and the GPGPU. This is of course handled by Bohrium in the Python code instead, but it does show the choice of complexity over simplicity, being made, when developing for the one or the other.

Having to write simpler and less code in Python, which is easier to maintain, and clearer to the programmer, is obviously an advantage. But if the performance of Bohrium is inadequate, for the programmers application, there is no way around the fact, that a handwritten kernel, must be just as fast or faster, than the generated kernels. Given, that a OpenCL programmer can always write the same kernel as the generated, while Bohrium cannot always generate code, with the same insight as kernel programmer. But Bohrium has the advantage, of easily deploying advanced code, like a reduction, which a random developer, might not put as much effort into.

By the end of this thesis, we have definitely narrowed the gap between generated kernel code, and handwritten version, as the benchmark started at 30 seconds, and ended with approximately 3 seconds. The handwritten version clocks in at 1.7 seconds, which might not be achievable in Bohrium, but at least approachable.

9.4 Best Fuser Setup

Part of the Bohrium configuration, is a type of pipeline of fusers, which each changes the raw instructions from NumPy. They affect the instructions in multiple ways. It can be a simple reordering, change of shape, if it does not affect the result, or any other preparation. Changing the fusers can have great effects on the resulting kernels, and their performance.

At present time, there are four fusers in the standard configuration:

```
greedy, push_reductions_inwards, split_for_threading,
    collapse_redundant_axes
```

The `greedy` and `collapse_redundant_axes` are somewhat required, as the greedy fuser makes the raw instructions form into almost kernel-ready structures. The latter one, removes the occurrence of axes with single element, avoiding a for-loop with a single element.

`split_for_threading` will try to undo some of the work of the greedy fuser, as to try to increase parallelism, by splitting a kernel into two.

`push_reductions_inwards` will not move a reduction, in the way I did with transposing to data-structure. But it will try to reorder the ranks preceding the reduction, if it can improve parallelism.

I want to try three different versions, to see if changing the fuser, has a positive impact on the benchmarks. The three configurations are shown below:

```
greedy, push_reductions_inwards, split_for_threading,
    collapse_redundant_axes
greedy, split_for_threading, collapse_redundant_axes
greedy, collapse_redundant_axes
```

I start by testing the configurations on the version of Bohrium with support for vector-reductions, which is seen in figure 9.2.

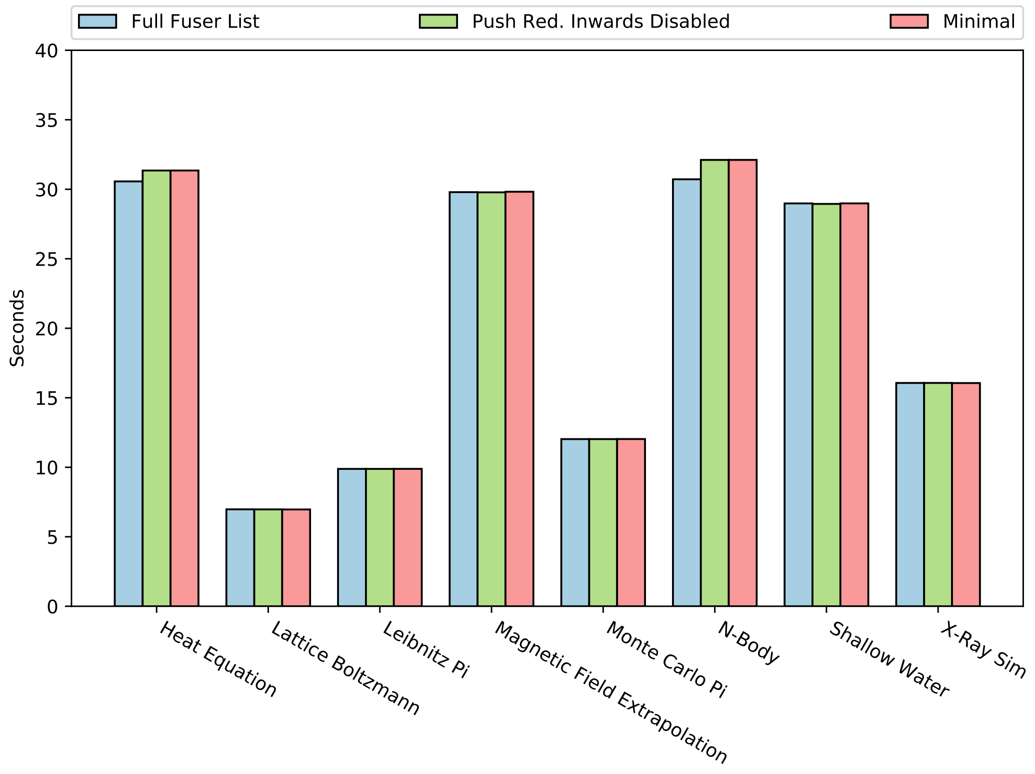

Figure 9.2: All Benchmarks Using the Version Supporting Vector-reductions. Testing Different Fuser Configurations

There is almost no difference in performance across the different configurations. I also test the newest version, which includes segmented reductions – see figure 9.3.

This again provides no dramatic change to performance. If anything, they are best as-is. Therefore, changing the fusers, does not seem to have a valuable impact at the moment.

9.5 Comparing Results on Newer System

I want to see, that the changes I have made, are not only favorable on my system, but will work on other systems as well. I had hoped to find a variety of modern GPGPUs and manufacturers to test on, but I have only managed to find a single system of recent date. I chose not to test on older hardware, than the primary one I have used, as it is already more than four years old.

The new system also features an Nvidia GPGPU primarily meant for gaming: The Nvidia GeForce GTX 2070, which at the time of writing, is just shy of half a year old. The full technical specifications can be found in appendix A.1, which includes processor, memory and so on.

I had hoped to find a GPGPU of the manufacturer AMD, as I have mostly been following the Nvidia programming Guide and NVIDIA OpenCL Best Practices Guide[45][44]. I have

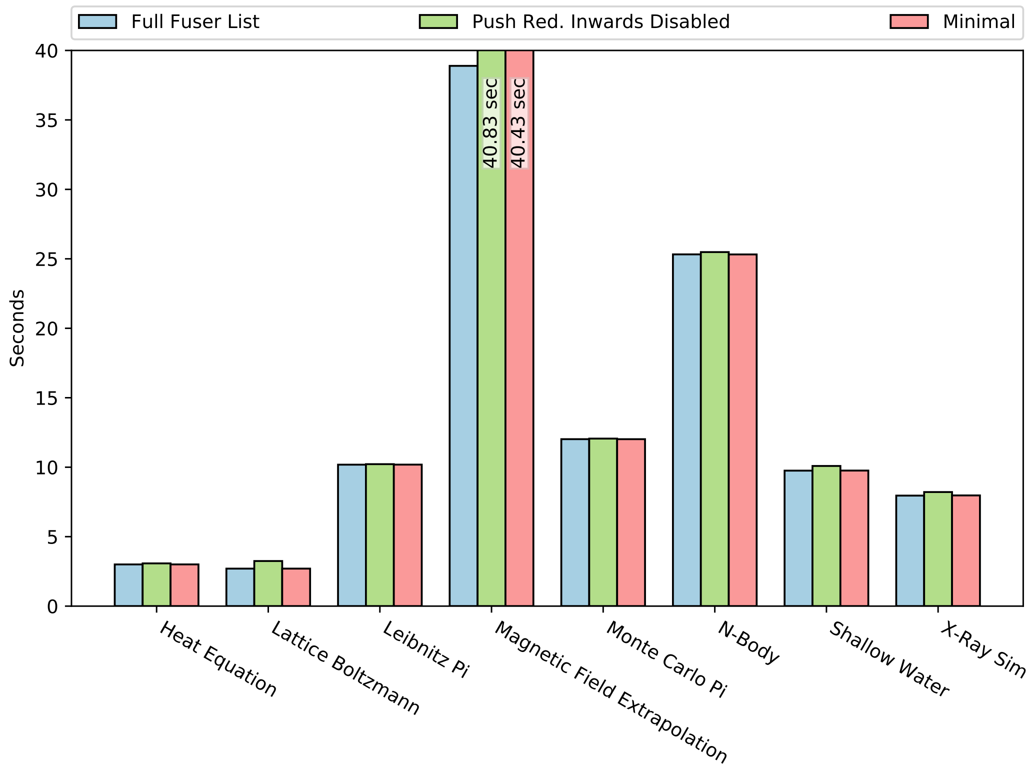

Figure 9.3: All Benchmarks Using the Version Supporting Segmented Reductions. Testing Different Fuser Configurations

at times peak at the AMD documentation[31], to make sure I did not make it too Nvidia-specific. And for the reductions, I even followed a case study by AMD[33]

New GPGPU

Because the new GPGPU is so recent, I had to install a newer version of the Nvidia driver, than the one used for all other previous benchmarks and the primary GPGPU. At the same time I installed a slightly newer version of NumPy and Cython on the system. I will not be running all previous benchmarks and auto-tuning again, but for comparison in this chapter alone, I will rerun the shown benchmarks with an updated system. I do this to rule out, any performance altering changes.

When looking at the results in figure 9.4, I will start by concluding, that we generally do not see a negative impact. It would seem, that the newer architecture, might not be as picky with memory access, either because of better caching, or less restrictions, on what constitutes a coalesced access pattern.

Something has evidently improved, as some benchmarks, like Leibnitz Pi and Monte Carlo Pi, did not perform that bad to begin with. It is hard to tell from these graphs whether it is the much newer CPU or improvements to the GPGPU, that alleviates the issues.

But looking at Heat Equation, Lattice Boltzmann and X-Ray Sim, it is clear, that the features I have implemented are still needed.

Lastly, Magnetic Field Extrapolation has gotten proportionally worse, although still faster than the equivalent benchmark on the primary test GPGPU.

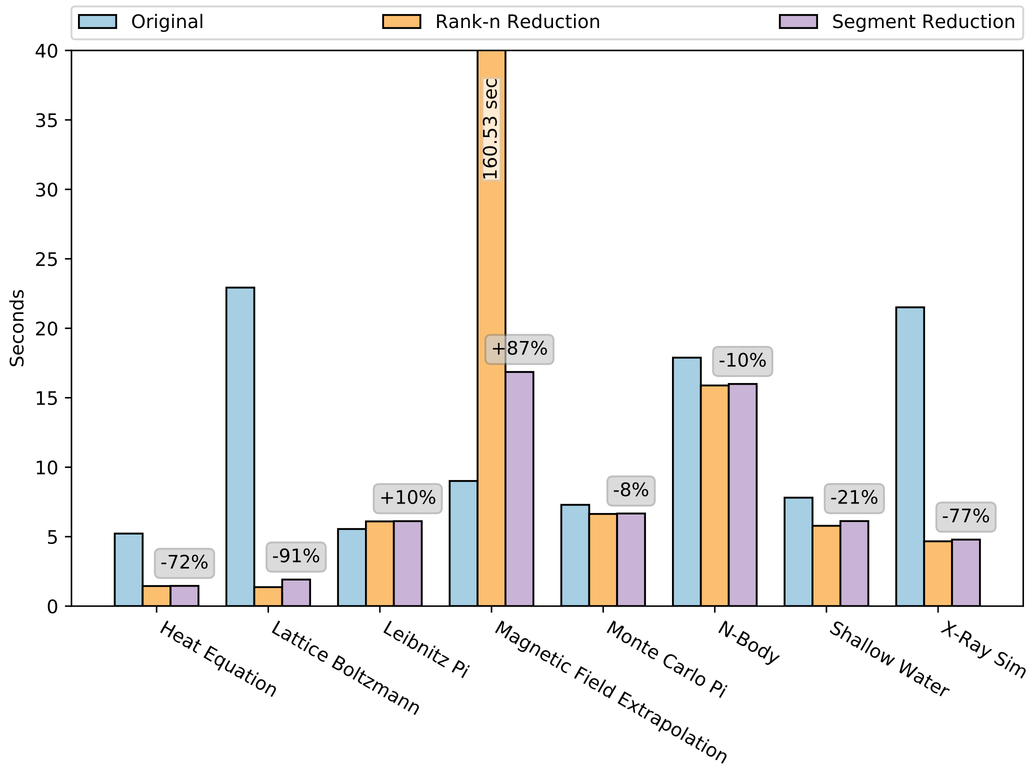

Figure 9.4: Benchmark Suite Using Nvidia GeForce GTX 2070 With Updated Drivers and Software

Primary GPGPU

As described above, the software versions have changed, therefore, I have run all benchmarks once more.

From figure 9.5 we can see that most of the benchmarks are unchanged. All benchmarks are within 2% of the previous NumPy, Cython and Nvidia driver versions.

The only outlier, is Magnetic Field Extrapolation. The final time is the same as before, but the initial time of the original version of Bohrium has reduced quite a bit. It now start around 25 seconds, instead of 30 seconds. This of course gives the impression, that it is much slower now, although it is the same absolute time as before.

Magnetic Field Extrapolation might be interesting to investigate further. I have read through the change logs of NumPy and the Nvidia drivers, but there is nothing signifying a large change for us. There are general improvements to some larger data structures in NumPy, but it is hard to tell, if they affect Bohrium[11].

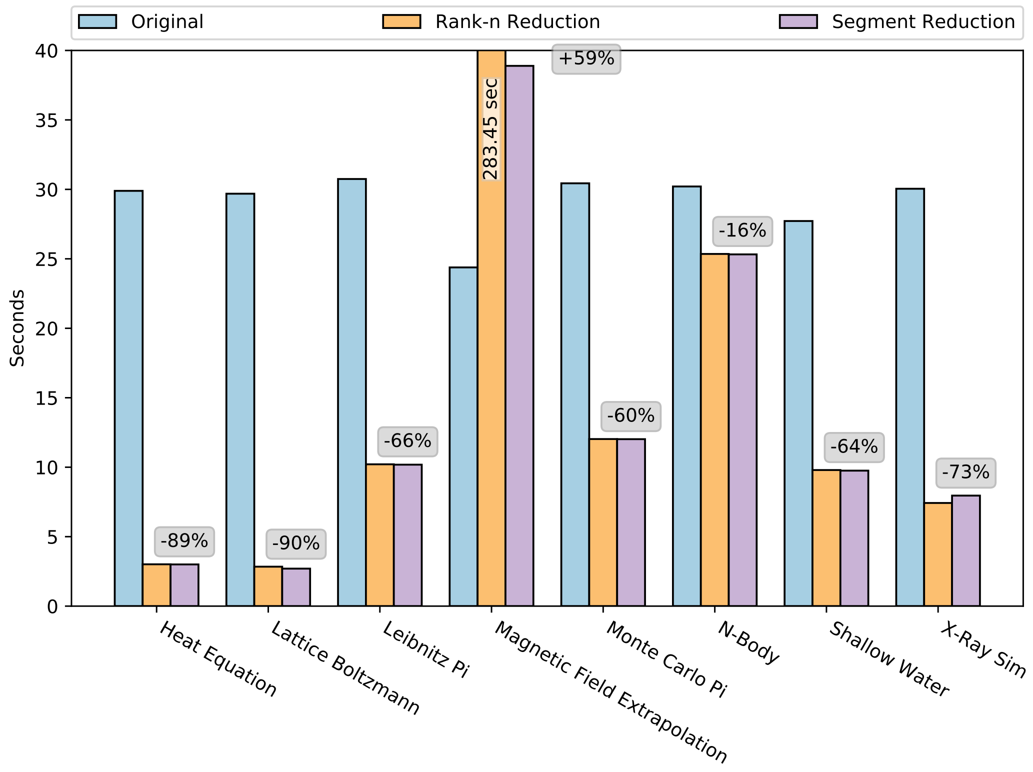

Figure 9.5: Benchmark Suite Using Nvidia GeForce GTX 970 With Updated Drivers and Software

9.6 Auto-tuner vs. Final Results

Lastly, I will look at the results we have been getting from the auto-tuner through-out the thesis, to see if we can draw any conclusions about kernel parameters.

In figure 9.6, I have combined the original version of Bohrium, all results of the auto-tuner, and the newest version of Bohrium.

It shows the progress we have made for each iteration of the code. In the beginning, we saw incredible speed-ups, in benchmarks like Heat Equation. Apart from Magnetic Field Extrapolation, which actually got to run quite fast for a short while, all of the speed-up we saw from the initial auto-tuning, has been matched by the regular code. Especially in the first two iterations, where we also got Lattice Boltzmann, Leibnitz Pi, and Monte Carlo Pi to perform better, than the auto-tuner could deliver.

Now that we have gotten this far, we can conclude, that the auto-tuner is not really needed for Bohrium anymore. I think the reason for this, is because Bohrium only generates kernels of the map-reduce type. There are no matrix-multiplication, sorting, scatter, gather or other more complex operations, that have horizontal data dependencies.

Especially the mentioning of matrix-multiplication, which has a very high relation between performance and kernel size, as it is a classic example of multi-threaded programming. The trick being, a matrix-multiplication can be split up into blocks, and depending on the size of these blocks, it can be tuned to fit the cache size perfectly, giving optimal performance.

But until that day comes, it does not make much sense in Bohrium, to auto-tune every kernel.

Then, we could look at the default parameters. Is the work-group size of 128 the best? It would actually seem so. When we look at figure 9.6, the best parameters at any time (disregarding MFE), does not significantly surpass the default parameters. This means, there can be no other parameter, that would be better, be it a constant parameter across all kernels, or finding the best parameter for each kernel. But this does not mean, that 128 is a magical number, in fact, lots of parameters seem to achieve the same performance.

Just to finish up, there are parameters for some kernel that are better than 128. But a lot of the ones I have inspected, show so little work, that it does not matter whether it takes $1e^{-5}$ or $1e^{-3}$ seconds, when the kernel is run just once or a hundred times. What could improve these situations, is likely to fuse them into other kernels, to save time on synchronization and delays of spawning new kernels.

9.7 Partial Conclusion

We have now looked at all the benchmarks and measurements of the thesis. I have shown some of the parameters, that we could manipulate in the benchmarks and Bohrium to alter performance.

I have looked at the theoretical bounds of the performance we have achieved. This has lead to discussion about several ways to further improve performance, as we have not hit neither the practical, nor theoretical, limit.

I have tested Bohrium, and the new improvements, on another computer, with a much more modern GPGPU. We have seen, that the performance increases, that we have achieved, are not just good for my machine, but likely a general improvement.

We have looked at the performance we can achieve, and reflected on how the changes we have made, has made the auto-tuner redundant to us for the moment.

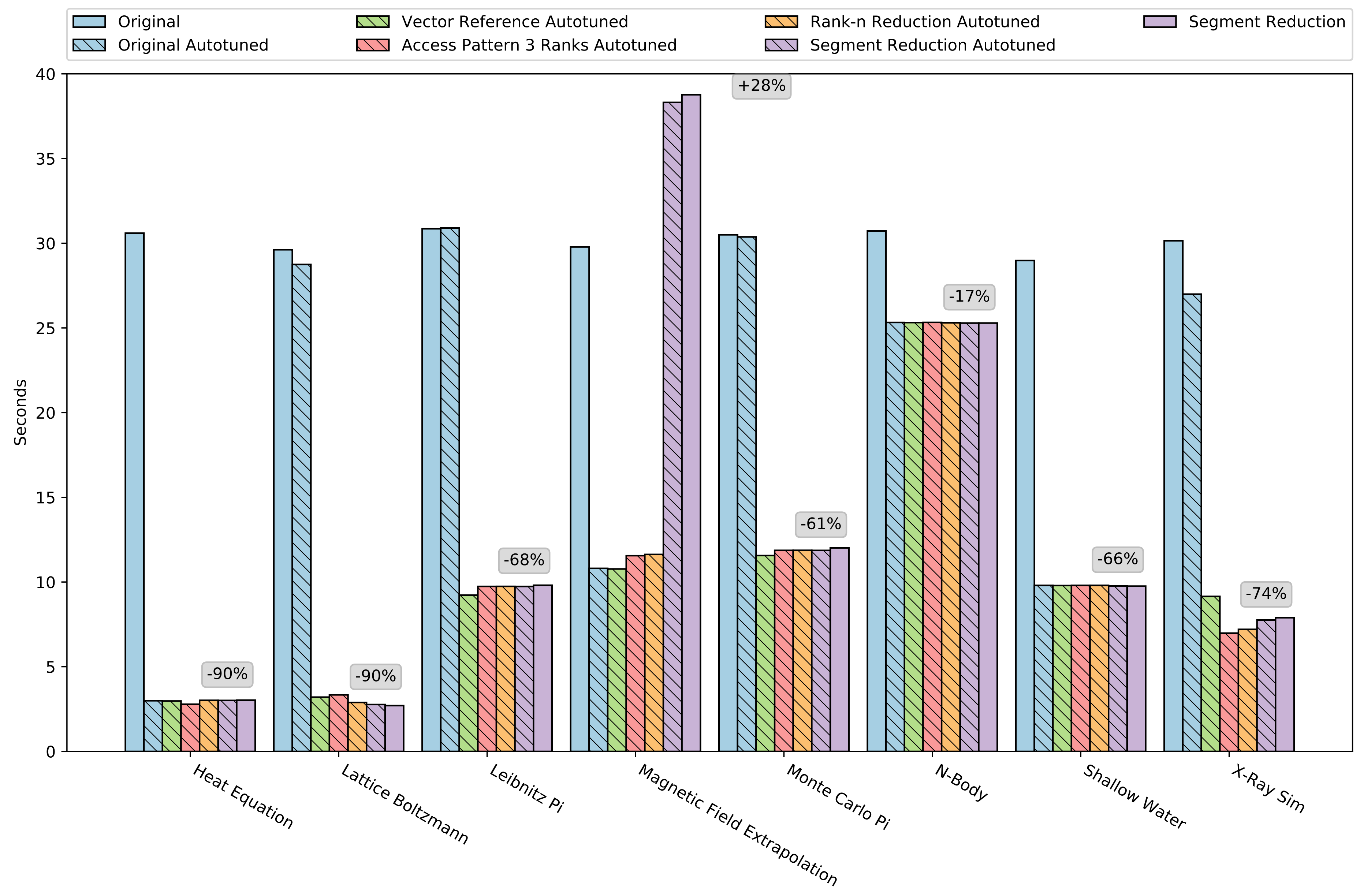

Figure 9.6: Overview of auto-tuned benchmarks. Showing original and all auto-tuned versions of Bohrium as reference.

Chapter 10

Conclusion

10.1 Conclusion

I started looking at the auto-tuner, as was originally desired for large performance increases. It was planned, that the process should be automated, and made quick enough to find the best kernel parameters on-the-fly. But it did not come to that, as instead, it shed light upon the underlying issues, which made it advantageous. With these issues getting fixed one by one, the auto-tuner got made redundant.

The second topic of this thesis, was vector-reductions. I implemented a method of reducing vector on a GPGPU, where the solution is not trivial. It required extensive knowledge of the hardware, to make it perform well and correctly. But it all paid off. From this functionality alone, we sped up benchmarks, by unforeseen magnitudes. This was also a turn for Bohrium, as this was the first operation to not conform with the regular access patterns of supported operations.

As the auto-tuner had revealed some issues with the generated kernels, I moved on to fix these, to get a consistent result from the auto-tuner. This led to the discovery of access patterns going against the rules of coalesced memory access, which is preferred by the GPGPU hardware. After correcting these patterns, we saw staggering improvements in several benchmarks. The parameters of the auto-tuner also now made more sense, as they became less erratic. But the changes also came, with its drawbacks. Although the new pattern is more correct in the theoretical sense, one benchmark seems to have benefited from the wrong access pattern.

After changing the access pattern, I identified, that the benchmark, was now performing a reduction, in an uncoalesced pattern. Which came to light, as two wrongs did make a right, when it came to access patterns. As both the kernel parameters and the reduction were transposed compared to the optimal, they became right in their own sense. With the changed access pattern, just the reduction is left transposed.

To get this sorted out, I implemented a special type of reductions. The new segmented reductions, take over the regular access pattern of reductions in inner ranks, and make their access pattern become coalesced. The new feature did work, as it brought down the execution-time down to bearable levels. But it still came with a penalty, compared to the original state of Bohrium.

Before concluding the thesis, I wanted to proof, that the changes made, has not broken anything in Bohrium. I did this by running my code through Bohrium's own tests for correctness, and additionally implement my own. After correcting a few mistakes, I finally consider the code stable, and ready for other developers to take over.

All in all, this thesis has introduced incredible improvements to Bohrium. Most of the selected benchmarks see performance increases in the range of $60 - 90\%$ shorter execution-time. The improvements I have made in this thesis, has even made these increase come, without the cost of an auto-tuner. The speed-up appear because of better control of the kernel, and better understanding of the hardware.

The improvements even carry over to other hardware, which signifies, that this is not a over-fitted optimization of the hardware I used, but a general improvement.

It is always hard to make unconditional optimizations, especially, when we are not in control of the input. But the balance between performance gains and losses, could be seen as acceptable, if the selection of benchmarks, are representative of typical workloads.

In the end, the changes has positively affected Bohrium, by lowering the execution-time. And in the big picture, this also means, that we have narrowed the gap between developing native OpenCL code, and regular Python code. The more this gap closes, the less appealing it is, to spend the time developing OpenCL kernels, when almost the same performance, can be achieved in a higher-level language.

10.2 Reflection

The exact implementations, I have produced in this thesis, are very narrowly dedicated to Bohrium, and would not just plug into any OpenCL project. In the sense, that the code is centered around Bohrium's internal representation, and how each component communicates and operates. This does not include the vector-reduction implementation, which only requires including the header file, and calling the functions to be used in any OpenCL kernel.

But on a theoretical level, all the changes are widely usable.

Changing the access pattern, from the most obvious way to parallelize, to the way that maximizes coalesced memory access, is important. And this thesis demonstrates, that there are large performance gains to get from even small changes to the code and parameters. The changes, does not even need to modify that many lines of code. It is a lot harder to come up with the solution, than to actually go through with the implementation.

Even though it did not change performance as much, one of the important observations I made, were that of transposing reductions. Applying this transposition to every reduction, gives an incredible advantage, when generating or handwriting kernels. The rather simple technique, simplifies kernel generation and kernel fusion. By always reducing the inner-most axis, it becomes a lot simpler to handle memory dependencies, and it allows to fuse reductions together, which would otherwise be incompatible. And most importantly, it also allows to place the reductions as deep as possible, to not sacrifice the potential parallelism of a kernel.

Anyone working with reduction of tensors, would likely benefit from this technique. And even though this implementation only applies to Bohrium, it is a very common operation, which has a very simple solution. It can also be used to place nested reductions in any favorable order, meaning, we can in some cases, fully avoid uncoalesced memory access, without complicating the generated kernel.

These findings and solutions will apply anywhere, and could easily be implemented in other kernel generating software, as well as handwritten kernels. In fact, the paradigm of these kernels, all follow the same map-reduce type, which is very common in parallel programming.

Chapter 11

Bibliography

[1] Avoiding and identifying false sharing among threads. https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads. [Online; accessed on 11th of January 2019].

[2] Benchpress. https://benchpress.readthedocs.io. [Online; accessed on 14th of February 2019].

[3] Bohrium: Welcome! https://bohrium.readthedocs.io. [Online; accessed on 11th of January 2019].

[4] clbuildprogram. https://www.khronos.org/registry/OpenCL/sdk/2.0/docs/man/xhtml/clBuildProgram.html. [Online; accessed on 19th of January 2019].

[5] Cuda pro tip: Increase performance with vectorized memory access. https://devblogs.nvidia.com/cuda-pro-tip-increase-performance-with-vectorized-memory-access/. [Online; accessed on 20th of February 2019].

[6] Cupy. https://docs-cupy.chainer.org/en/latest/overview.html. [Online; accessed on 14th of February 2019].

[7] Difference between cuda and opencl 2010. https://streamhpc.com/blog/2010-04-22/difference-between-cuda-and-opencl/. [Online; accessed on 14th of February 2019].

[8] Numba. http://numba.pydata.org. [Online; accessed on 14th of February 2019].

[9] Numpy. http://www.numpy.org. [Online; accessed on 11th of January 2019].

[10] Numpy 1.13.0 release notes. https://docs.scipy.org/doc/numpy-1.13.0/release.html#better-numerical-stability-for-sum-in-some-cases. [Online; accessed on 4th of February 2019].

[11] Numpy 1.16.0 release notes. https://github.com/numpy/numpy/blob/master/doc/release/1.16.0-notes.rst. [Online; accessed on 4th of February 2019].

[12] Numpy usage of matrix. https://docs.scipy.org/doc/numpy-1.15.0/user/numpy-for-matlab-users.html?highlight=tensor. [Online; accessed on 11th of January 2019].

[13] Numpy usage of tensor. https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.transpose.html?highlight=tensor. [Online; accessed on 11th of January 2019].

[14] Nvidia discloses full memory structure and limitations of gtx 970. https://www.pcper.com/reviews/Graphics-Cards/NVIDIA-Discloses-Full-Memory-Structure-and-Limitations-GTX-970. [Online; accessed on 20th of February 2019].

[15] The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/. [Online; accessed on 14th of February 2019].

[16] Opencl: Barrier. https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/barrier.html. [Online; accessed on 14th of February 2019].

[17] The opencl extension specification. https://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_Ext.html#cl_khr_subgroups. [Online; accessed on 13th of January 2019].

[18] Parallel thread execution isa version 6.3. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#syntax. [Online; accessed on 14th of February 2019].

[19] Pci passthrough via ovmf. https://wiki.archlinux.org/index.php/PCI_passthrough_via_OVMF#CPU_pinning. [Online; accessed on 4th of February 2019].

[20] The performance of python, cython and c on a vector. https://notes-on-cython.readthedocs.io/en/latest/std_dev.html. [Online; accessed on 15th of February 2019].

[21] Pycuda. https://mathema.tician.de/software/pycuda/. [Online; accessed on 14th of February 2019].

[22] Pyopencl. https://mathema.tician.de/software/pyopencl/. [Online; accessed on 14th of February 2019].

[23] Pypy: Performance. optimization strategy. https://pypy.org/performance.html#optimization-strategy. [Online; accessed on 14th of January 2019].

[24] Pytorch. https://pytorch.org. [Online; accessed on 14th of February 2019].

[25] Reduce matrix rows with cuda. https://stackoverflow.com/a/17864583/1796585. [Online; accessed on 22th of February 2019].

[26] Reductions. https://thrust.github.io/doc/group__reductions.html#gad5623f203f9b3fdcab72481c3913f0e0. [Online; accessed on 22th of February 2019].

[27] Reductions. https://moderngpu.github.io/segreduce.html. [Online; accessed on 22th of February 2019].

[28] Tensorflow. https://www.tensorflow.org. [Online; accessed on 14th of February 2019].

[29] Visualising opencl timelines with nvvp. http://uob-hpc.github.io/2015/05/27/nvvp-import-opencl.html. [Online; accessed on 14th of February 2019].

[30] What is futhark? https://futhark-lang.org/index.html. [Online; accessed on 21th of February 2019].

[31] AMD. Amd app sdk opencl user guide, 2015. Rev. 1.0.

[32] Ravishekhar Banger and Koushik Bhattacharyya. *OpenCL Programming by Example.* Packt Publishing, 2013.

[33] Bryan Catanzaro. Opencl optimization case study: Simple reductions. `https://web.archive.org/web/20170704004945/https://developer.amd.com/resources/articles-whitepapers/opencl-optimization-case-study-simple-reductions/`. [Online; accessed on 3rd of February 2019].

[34] Chris Cummins, Pavlos Petoumenos, Michel Steuwer, and Hugh Leather. Autotuning opencl workgroup size for stencil patterns, 2016.

[35] Thanh Tuan Dao and Jaejin Lee. An auto-tuner for opencl work-group size on gpus, 2018.

[36] Guy E. Blelloch. Prefix sums and their applications. 03 1997.

[37] Mark Harris. Mapping computational concepts to gpus. `https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter31.html`. [Online; accessed on 22th of February 2019].

[38] Mark Harris. Optimizing parallel reduction in cuda. `https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`. [Online; accessed on 22th of February 2019].

[39] Daniel Hernández Juárez, Antonio Espinosa, David Vázquez, Antonio Manuel López Peña, and Juan Carlos Moure. Gpu-accelerated real-time stixel computation. *CoRR*, abs/1610.04124, 2016.

[40] M. R. B. Kristensen, S. A. F. Lund, T. Blum, and J. Avery. Fusion of parallel array operations. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 71–85, Sep. 2016.

[41] Mads R. B. Kristensen, James Avery, Troels Blum, Simon Andreas Frimann Lund, and Brian Vinter. *Battling Memory Requirements of Array Programming Through Streaming*, pages 451–469. 2016. Exported from https://app.dimensions.ai on 2019/01/14.

[42] Rasmus Wriedt Larsen and Troels Henriksen. Strategies for regular segmented reductions on gpu. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, pages 42–52. ACM, 2017.

[43] Justin Luitjens. Faster parallel reductions on kepler. `https://devblogs.nvidia.com/faster-parallel-reductions-kepler/`. [Online; accessed on 22th of February 2019].

[44] Nvidia. *NVIDIA OpenCL Best Practices Guide.* 2009. August 10, 2009 revision.

[45] Nvidia. *OpenCL Programming Guide for the CUDA Architecture.* 2012. Version 4.2.

[46] Mads Ynddal. *GPU-accelerated Reduction and Scan Operations in Bohrium.* 2017.

[47] Mads Ynddal and Mark Jacobi. *GPU-accelerated Scan in OpenCL.* 2017.

Appendices

Appendix A

Hardware

A.1 Specifications

A.1.1 Main System Specifications

The main system is set up in a special way to support an easier work-flow and version control. The details below are of the physical test system. Inside this system, is run a virtual machine through QEMU. The virtual machine has full hardware virtualization, while the GPGPU is forwarded using IOMMU. It has its CPU cores pinned to the physical cores to avoid invalidating cache. All memory is pre-allocated, to avoid any delays. The guest system's performance is verified by running the same benchmark on the host and on the guest. The performance was indistinguishable. Details on how the configuration is made, can be found at ArchLinux's website[19].

Any effects the virtualization has had on the results of the benchmarks, must have been a factor across all benchmarks, and is improbable to have caused any positive side-effect to my results.

The use of a virtual machine, allows me to control version of drivers and software, and freeze the state of the machine, to always be able to revert any changes.

CPU	Intel i5-4690K
CPU Cores	4
CPU Base Frequency	4.4GHz (overclocked)
CPU Turbo Frequency	4.4GHz (overclocked)
RAM	16GB
RAM Frequency	1600MHz
Operating System	Ubuntu Server 18.04

Software Specifications

Nvidia Driver	396.54
Cuda Version	9.1.85
NumPy Version	1.15.3
Cython Version	0.26.1

GeForce GTX 970 Engine Specifications

CUDA Cores	1664
Base Clock (MHz)	1050
Boost Clock (MHz)	1178

GeForce GTX 970 Memory Specifications

Memory Clock	7.0 Gbps
Standard Memory Config	4 GB
Memory Interface	GDDR5
Memory Interface Width	256-bit
Memory Bandwidth	224 GB/s

A.1.2 Secondary System Specifications

CPU	Intel i5-7600
CPU Cores	4
CPU Base Frequency	3.5GHz
CPU Turbo Frequency	4.2GHz
RAM	16GB
RAM Frequency	2133 MHz
Operating System	Ubuntu Server 18.04

Software Specifications

Nvidia Driver	415.27
Cuda Version	9.1.85
NumPy Version	1.16.1
Cython Version	0.29.4

GeForce GTX 2070 Engine Specifications

CUDA Cores	2304
Base Clock (MHz)	1410
Boost Clock (MHz)	1620

GeForce GTX 2070 Memory Specifications

Memory Clock	14 Gbps
Standard Memory Config	8 GB
Memory Interface	GDDR5
Memory Interface Width	256-bit
Memory Bandwidth	448 GB/s

Appendix B

Full-sized Figures

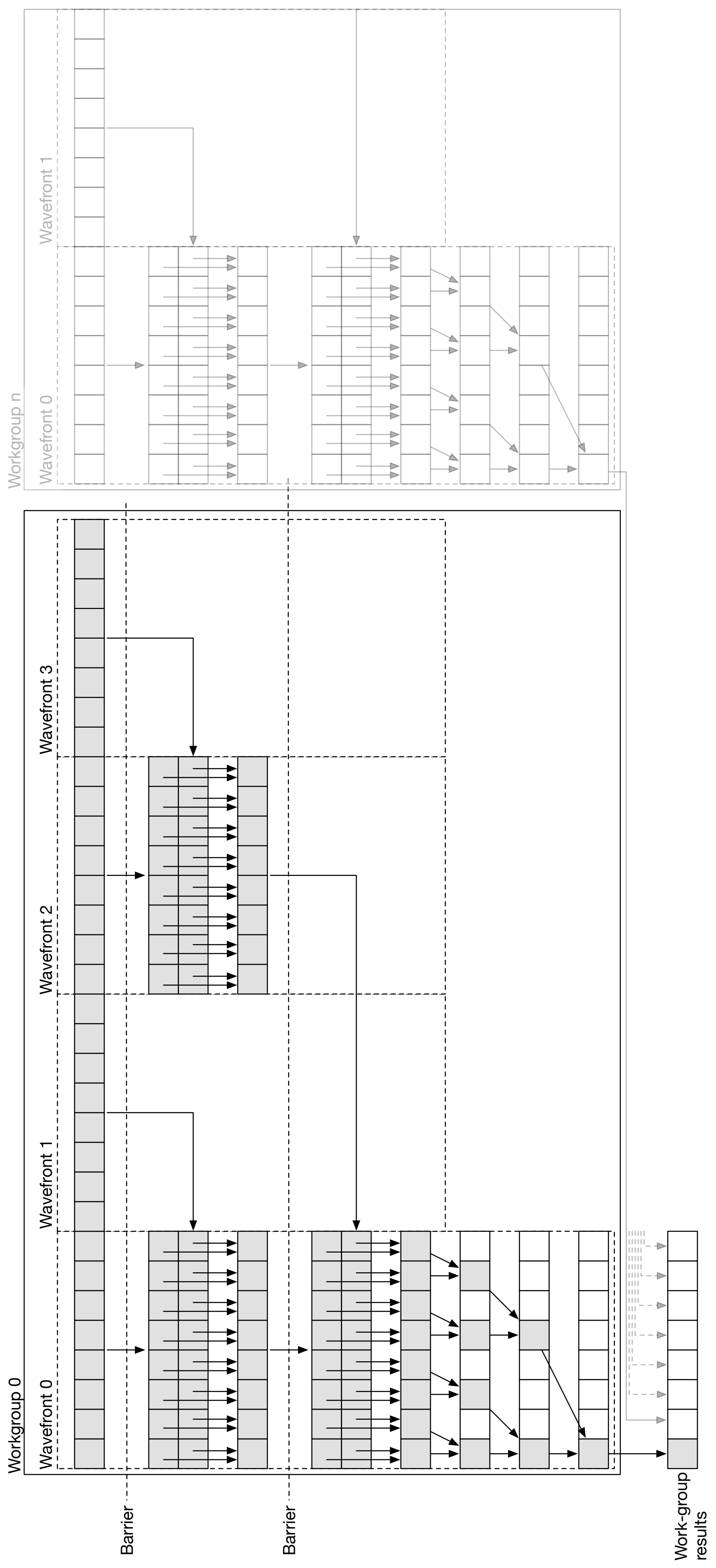

Figure B.1: First vector-reduction kernel

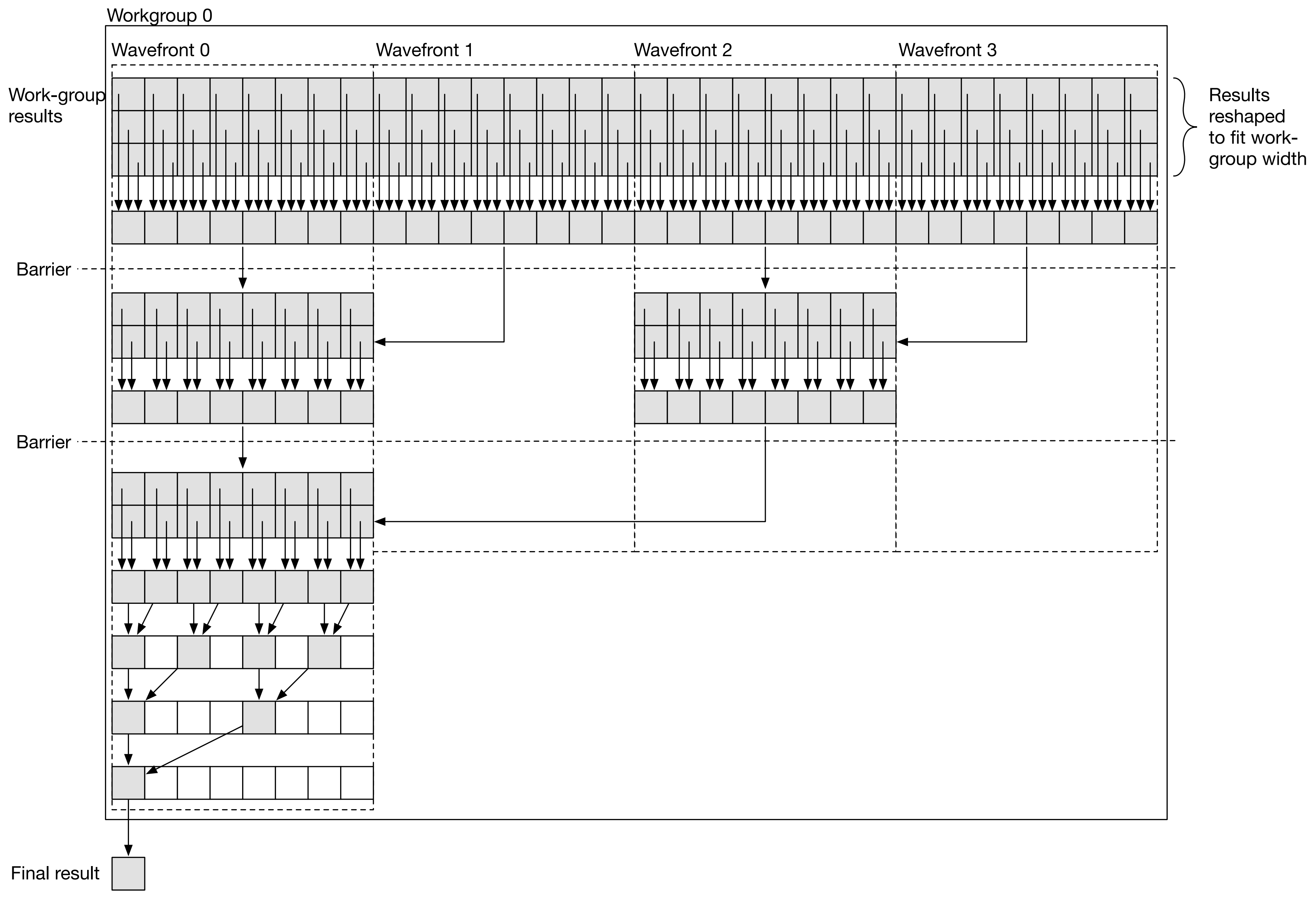

Figure B.2: Second vector-reduction kernel

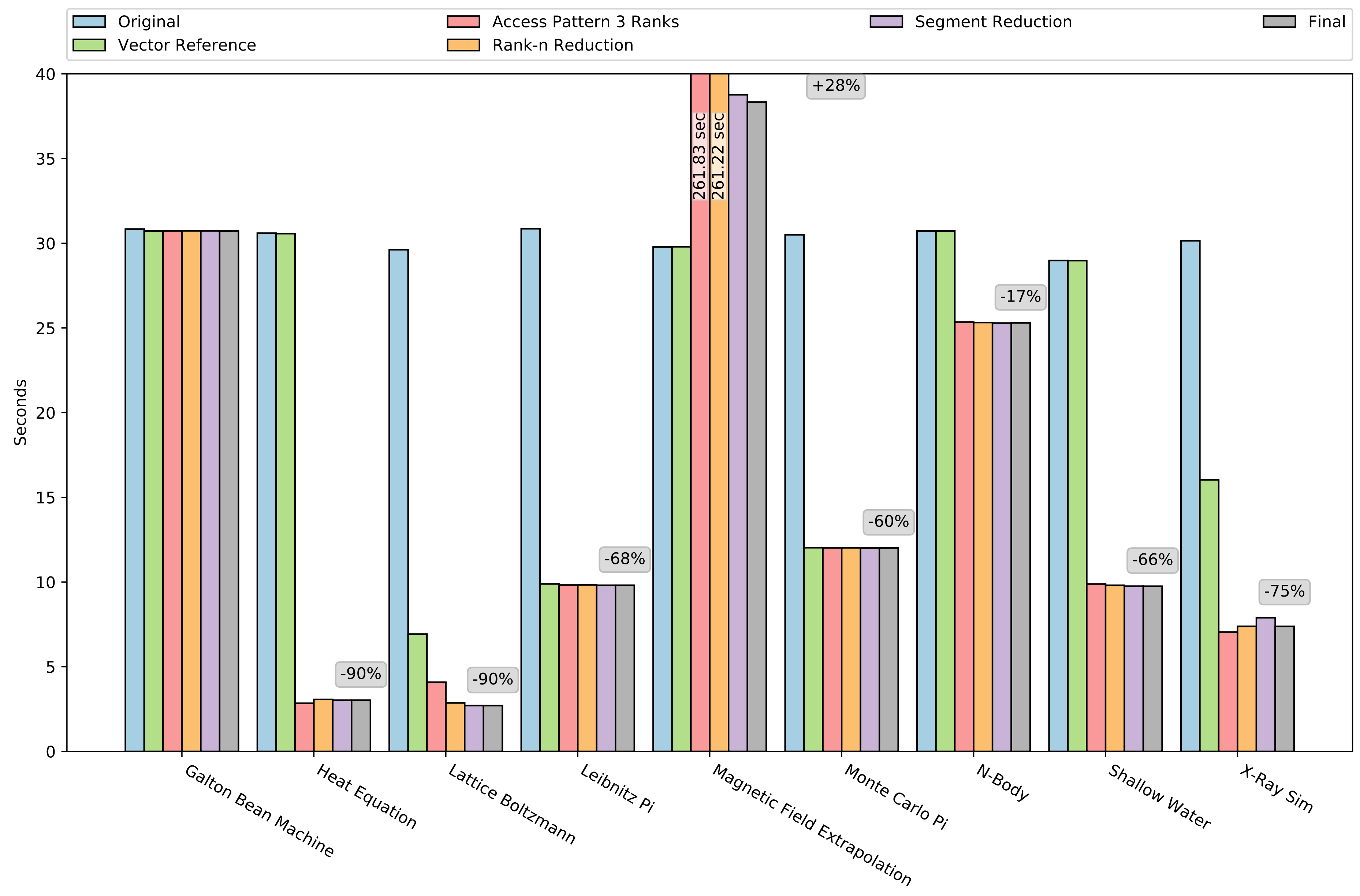

Figure B.3: Combined figure of all benchmark results

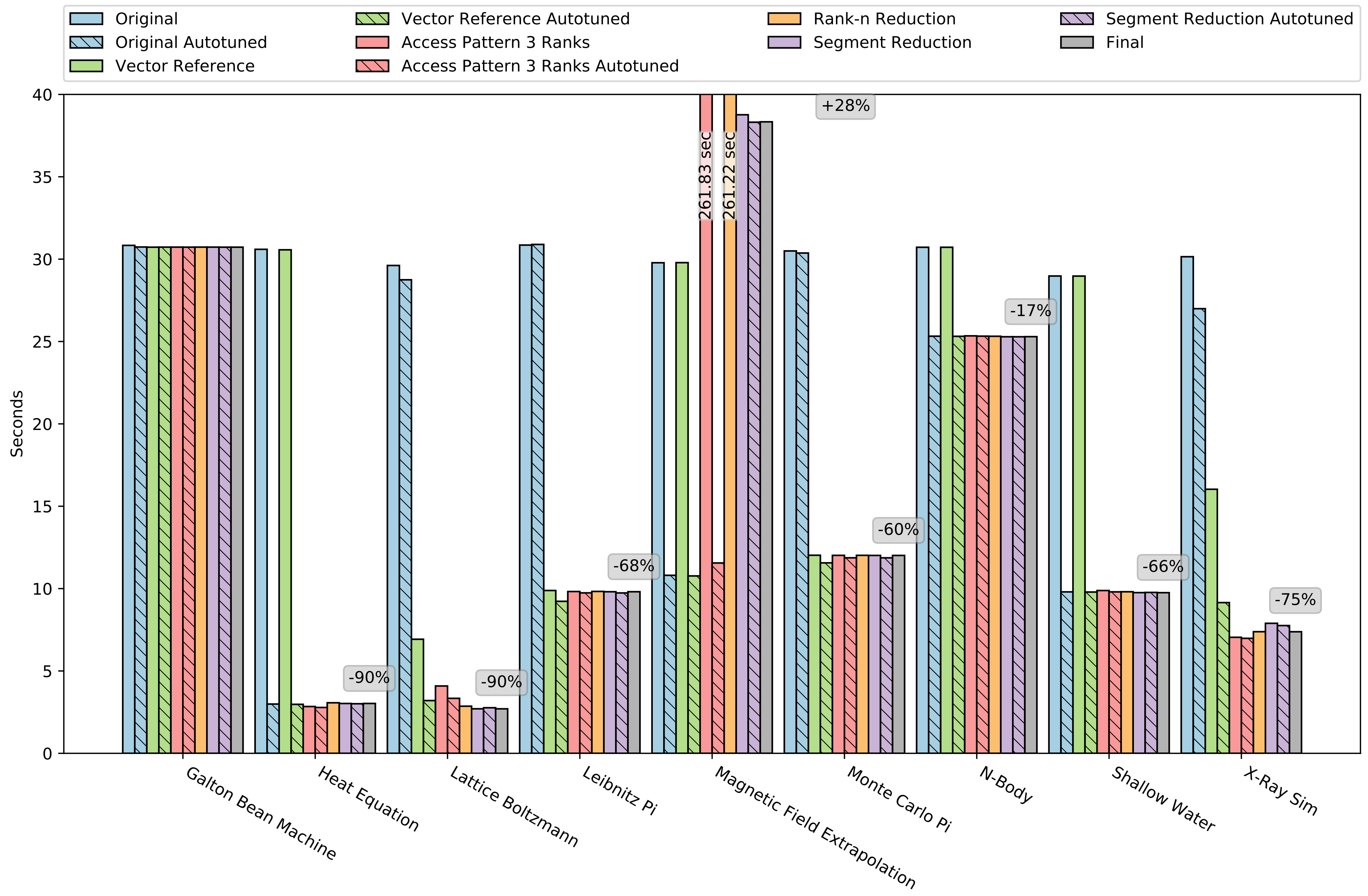

Figure B.4: Combined figure of all auto-tuner results

Appendix C

Code

C.1 Auto-tuner Parameter Generators

C.1.1 Divisor Generator

The following code generates all divisors for the input number. This is useful for maximizing kernel parameters.

The code is found on StackOverflow: `https://stackoverflow.com/questions/171765/what-is-the-best-way-to-get-all-the-divisors-of-a-number`

```
def divisorGen(n):
    large_divisors = []
    for i in xrange(1, int(math.sqrt(n) + 1)):
        if n % i == 0:
            yield i
            if i*i != n:
                large_divisors.append(n / i)
    for divisor in reversed(large_divisors):
        yield divisor
```

C.1.2 2D Parameters

The following code is used in the auto-tuner to generate the test parameters for 2-dimensional kernels.

```python
def split_grid_2d(x_max=1024, y_max=1024):
    step = 16

    for x in range(0,x_max+step, step):
        for y in range(0,y_max+step, step):
            x = max(x,1)
            y = max(y,1)
            if x<32 and y<32:
                continue
            if x*y > 1024:
                continue
            yield (x,y)

    x_divs = divisorGen(x_max)
    y_divs = divisorGen(y_max)
    for x,y in zip(x_divs, reversed(list(y_divs))):
        if x<32 and y<32:
            continue
        yield (x,y)
```

C.1.3 3D Parameters

The following code is used in the auto-tuner to generate the test parameters for 3-dimensional kernels.

```python
def split_grid_3d(x_max=1024, y_max=1024, z_max=64):
    step = 16
    half_step = 16

    for x in range(0,x_max+step, step):
        for y in range(0,y_max+step, step):
            for z in range(0,z_max+step, half_step):
                x = max(x,1)
                y = max(y,1)
                z = max(z,1)
                if x<32 and y<32 and z<32:
                    continue
                if (x*y*z) > 1024:
                    continue
                yield (x,y,z)
```

Appendix D

Supporting Rank-n Kernels

A case we also have to ensure, is making a vector-reduction on the outer-most axis, in a kernel which is spawned through `clEnqueueNDRangeKernel` with a 2- or 3-D work-group size.

The kernel I have been developing, has been focusing on a flat, rank-1 work-group.

There are three ways to solve this. One, disallow multi-dimensional kernels. Two, use only work-items with IDs in the first dimension. Three, flatten the multi-dimensional IDs to emulate a rank-1 work-group.

None of the methods are bound to provide an increase in performance over a rank-1 kernel. The first option will force us to use the CPU again for these kernels, which would be a step back. The second option could work, given enough work-items in the first dimension, although there will be cases with very few work-items in each dimension. That leaves us with the third option, which is most likely to get us close to the rank-1 performance.

To flatten the dimensions, we have to recalculate the local ID, global ID, group ID, and local size. The calculations are simple enough, but I quickly find, that it slows down the kernel, even when running a rank-1 kernel.

I set up macros in the kernel to handle which definition we use to find the flattened IDs. With a rank-1 kernel defaulting to the `get_local_id(0)` and so on directly:

```
#ifdef KERNEL_1D
#define flat_global_id get_global_id(0)
#define flat_local_id get_local_id(0)
#define flat_group_id get_group_id(0)
#define flat_local_size get_local_size(0)
#endif

#ifdef KERNEL_2D
#define flat_global_id (get_global_id(0) + get_global_size(0)
    * get_global_id(1))
#define flat_local_id (get_local_id(0) + get_local_size(0) *
    get_local_id(1))
#define flat_group_id (get_group_id(0) + get_num_groups(0) *
    get_group_id(1))
#define flat_local_size (get_local_size(0) * get_local_size(1))
#endif

#ifdef KERNEL_3D
// ...
```

This is the most efficient way, I can come up with, but it requires a definition, to be injected into the kernel, which is generated in Bohrium.

Here is a benchmark with a rank-1 kernel, where I have defined it to use either KERNEL_1D or KERNEL_3D. For a rank-1 kernel, it will calculate the exact same IDs, but it will have to perform a few more multiplications and additions, if we choose KERNEL_3D.

Type	**1D GB/s**	**3D GB/s**
Naive	15.63	13.13
Naive Optimized	32.81	24.09
Compacting	42.46	27.61
Compacting Optimized	60.64	35.65
Wave Elimination	60.07	36.08
Wave Elim. 4-element	75.52	41.52

This is definitely not a good result for the 3D kernels. The performance might even be crippled further, when the work-groups are spawned for more than rank-1.

Trying the same benchmark, but actually using 2D and 3D work-groups, show similar results to the rank-1 work-group emulating it.

The local size is calculated through this line:

```
(get_local_size(0) * get_local_size(1) * get_local_size(2))
```

The difference between keeping the line, and hardcoding it to a static value, and providing it through a __const kernel argument, is surprising:

Type	**GB/s**
Dynamic	47.96
Argument	55.38
Hardcoded	60.34

It seems to be rooted in the compiler's ability to optimize the code – most likely the for-loops. I find, that hardcoding the call to get_local_size, has a tremendous impact on performance, especially on the first versions of the kernels:

Type	**Hardcoded**	**Dynamic**
Naive	55.10 GB/s	16.01 GB/s
Naive Optimized	55.19 GB/s	42.35 GB/s
Compacting	55.83 GB/s	33.10 GB/s
Compacting Optimized	62.72 GB/s	61.17 GB/s
Wave Elimination	64.67 GB/s	60.24 GB/s
Wave Elim. 4-element	78.45 GB/s	76.40 GB/s

Thankfully, it doesn't change the outcome of my decisions according to which algorithm was fastest.

The hardcoded version, can be achieved, either be a macro at the top of the file, or as a literal in-lined value. The dynamic version has been tested with get_local_size, and by providing it as a __const argument to the kernel.

At Bohrium's current state, it would be fine to hardcode them, through the auto-generator, but this doesn't fit well with the auto-tuner, as work-group sizes are likely to change. Therefore I have later added a switch, that chooses the appropriate method, when the auto-tuner is enabled.