Bachelor thesis
Computing Science

Radboud University

Blueprints for out-of-core code generation using CUDA

Author:
Patrick van Beurden
s1003725

First supervisor/assessor:
Prof. dr. Sven-Bodo Scholz
svenbodo.scholz@ru.nl

Second assessor:
Dr. Pieter Koopman
pieter@cs.ru.nl

June 23, 2022

Abstract

Most parallel computing systems today are a combination of a central processing unit (CPU) and one or more graphics processing units (GPUs). Both types of devices maintain their own dynamic RAM (DRAM), but the average amount available to each typically varies significantly. Whereas it is relatively inexpensive to have 64, 128 or even 256 gigabytes of DRAM available to the CPU, a lot of high-end GPUs only have 8 to 16 gigabytes to work with. This discrepancy limits the data complexity of algorithms that can benefit from GPU acceleration.

It is possible to rewrite an algorithm in such a way that the data it processes does not have to fit the GPU memory in one go. However, doing this manually is error prone and complex, so we are looking into ways to automate this process. Hence, this paper is meant as a stepping stone for generating data size agnostic code for map-like operations, such as vector addition, and (3-point) stencil computations using NVIDIA's CUDA.

Contents

Chapter 1

Introduction

GPUs are known for their high memory bandwidth, fast computation speed and relatively low energy cost for high parallelism. These attributes make them especially attractive for performing algorithms that can process large blocks of data in parallel. Not surprisingly, GPUs are increasingly being used for high performance computing and data-intensive scientific applications such as weather forecasting [16], physics simulations [7] and machine learning [25].

In order to take advantage of the large computational power of a GPU, the data required for a given algorithm first needs to be transferred to the device's dedicated memory. Currently, most high-end GPUs only have between eight and sixteen gigabytes of space available, whereas the main memory of a modern parallel computing system can easily be several hundred gigabytes. Unfortunately, this discrepancy limits the problem size of applications that can benefit from GPU acceleration.

The indicated memory problem relates to a similar challenge in the past when the data structures in use for purposes such as numerical linear algebra, scientific visualisation and computer graphics became too large to fit in the main memory of a computer. Consequently, research has been done on the development of algorithms [22, 27] that are designed to achieve high performance on the CPU when their data is stored in external memory. These type of algorithms are often referred to as *out-of-core* algorithms.

Lately, it has become increasingly more common that the memory available on the GPU for immediate computation is insufficient in size to keep the entire data in memory during the execution of an algorithm. As one might expect, the need for executing large problem domains in a distributed fashion on one or multiple GPUs has been a topic of recent research on stencil computations [12, 13, 23, 24]. However, creating a correct and efficient solution is non-trivial, because it usually involves multiple technologies and a lot of domain-specific knowledge.

This research looks at generating out-of-core code on the GPU for two

different types of common algorithms; map-like operations and (3-point) stencil computations. In order to achieve this, CUDA's asynchronous communication model is first used to concurrently process, or *stream*, chunks of data on the GPU when the data does fit. This *in-core* streaming method is then extended to realize out-of-core execution.

The main contribution of this paper are rewriting schematics, or *blueprints*, for in-core streamed and out-of-core code generation. My approach to establishing these blueprints is to first identify two commonly used algorithmic classes. From there, I manually investigate the respective challenges for creating out-of-core code for these algorithms. Finally, I look at the performance one can expect by implementing a simple variant of both algorithmic classes in C++. Ultimately, these blueprints could then function as a starting point for implementing streamed or out-of-core code generation in a domain-specific array language such as SaC [21].

In the second chapter of this report, the background for this research is covered. After that, chapter three presents the in-core specification, and based on that, the in-core streamed and out-of-core blueprints for both map-like operations and 3-point stencils in a C++-like pseudo-language. Next, chapter four evaluates the performance of the aforementioned blueprints when implemented in C++. Finally, chapter five discusses related work and chapter six contains the concluding remarks and possible future work.

Chapter 2

Background

2.1 CUDA

In section 2.1, an introduction to CUDA is given by providing a brief description of CUDA and going over several technical details such as the programming model and the threading model. Figure 2.1 compares the addition of two vectors of a sequential C++ program and a possible CUDA counter part. Although the syntax of the CUDA program may still be unfamiliar at this point, it is further explained in subsections 2.1.1 through 2.1.3.

2.1.1 What is CUDA?

CUDA is a parallel computing platform and programming model that allows programmers to utilise the computational power of GPUs. Its programming model emphasizes two important design goals, the first of which is to provide a low learning curve for programmers familiar with a supported programming language [4]. This is done by extending a standard language (e.g. C++) with a minimalist set of keywords for expressing parallelism [6]. The second goal is to allow for the design of highly scalable code [6] and an example of this is the concept of grids and (thread-)blocks, which will be discussed in more detail in subsection 2.1.3 of this chapter.

Furthermore, it was the first API that allowed for general purpose computing on GPUs (GPGPU), because prior to CUDA's release in 2007, the only alternatives were designed for graphics programming (e.g. openGL). This seems to be one of the reasons why CUDA applications are now commonly used for several different purposes such as medical imaging, computational fluid dynamics and environmental science [20].

Listing 2.1: Sequential

```
1  void add_vector(int N, float *a,
2                  float *b)
3  {
4      for(int i = 0; i < N; i++)
5      {
6          a[i] = a[i] + b[i];
7      }
8  }
9
10 int main()
11 {
12     size_t N = 1000000;
13
14     float *a = new float[N];
15     float *b = new float[N];
16
17     for(int i = 0;i<N;i++)
18     {
19         a[i] = 1.0;
20         b[i] = 1.0;
21     }
22
23     add_vector(N, a, b);
24
25     delete [] a;
26     delete [] b;
27
28     return 0;
29 }
```

Listing 2.2: CUDA program

```
1  __global__
2  void add_vector(int N, float *a,
3                  float *b)
4  {
5      int index = blockIdx.x * blockDim.x
6                  + threadIdx.x;
7      int stride = blockDim.x * gridDim.x;
8
9      for (int i = index;i<N;i+=stride)
10     {
11         a[i] = a[i] + b[i];
12     }
13 }
14
15 int main()
16 {
17     size_t N = 1000000;
18
19     float *a = new float[N];
20     float *b = new float[N];
21
22     for(int i = 0; i < N; i++)
23     {
24         a[i] = 1.0;
25         b[i] = 1.0;
26     }
27
28     float *dev_a = 0;
29     float *dev_b = 0;
30     int block_n = 256;
31     int blocks = (N + block_n - 1)
32                  / block_n;
33
34     cudaMalloc((void**)&dev_a,
35                N*sizeof(float));
36     cudaMalloc((void**)&dev_b,
37                N*sizeof(float));
38
39     cudaMemcpy(dev_a,a,N*sizeof(float),
40                cudaMemcpyHostToDevice);
41     cudaMemcpy(dev_b,b,N*sizeof(float),
42                cudaMemcpyHostToDevice);
43
44     add_vector<<<blocks, block_n>>>(N,
45                                     dev_a, dev_b);
46
47     cudaMemcpy(a,dev_a,N*sizeof(float),
48                cudaMemcpyDeviceToHost);
49
50     delete [] a;
51     delete [] b;
52     cudaFree(dev_a);
53     cudaFree(dev_b);
54
55     return 0;
56 }
```

Figure 2.1: A comparison of adding two vectors in a sequential program and in a (parallel) CUDA program.

2.1.2 Programming model

First of all, CUDA programs are heterogeneous co-processing programs. This means that both a CPU and a GPU are involved in the execution of CUDA code. Each of these components are dedicated to their own specific task. The CPU, which is referred to as the *host*, executes the serial parts of the program and the GPU, which is referred to as the *device*, executes the parallel parts of the program. This kind of co-processing approach gives the best performance for parallel-intensive programs, but also many mostly-

sequential codes [18].

Furthermore, CUDA kernels are special functions that are executed N times in parallel by N different CUDA threads on the device [4]. They are defined using the "__global__" keyword and are called in the same manner as regular C++ functions, except that the amount of threads that will execute the kernel are specified with a special syntax: "$\langle\langle\langle ... \rangle\rangle\rangle$". Within a kernel, the programmer also has access to built-in variables, such as the thread ID of the executing thread.

Finally, the programming model assumes that both the host and the device have their own separate dynamic RAM (DRAM) [4], which are aptly referred to as the *host memory* and the *device memory*. In order for the device to operate on data from the host, the program first has to call for a memory transfer from the host to the device. When the device has finished its computation, the resulting data in turn is often transferred back to the host. For this purpose, CUDA provides the programmer with several runtime functions, of which "cudaMemcpy(...)" is one of them that is commonly used. It has four arguments: the destination buffer, the source buffer, the size of the data that should be transferred and the direction of the memory transfer (e.g. cudaMemcpyHostToDevice), respectively.

2.1.3 Threading model

In order to leverage parallelism, it is important to understand CUDA's threading model. First, when launching a kernel, the programmer has to indicate by how many threads the kernel should be executed. All the executing threads requested at kernel launch are then grouped together in so-called blocks. These blocks can combine their threads together in several different shapes spanning up-to three dimensions, which is illustrated in figure 2.2. One reason why this is useful is that data belonging to a given algorithm usually has up-to three dimensions, so several blocks could then be combined to match the shape of the data, which in turn gives easy and predictable access to any element.

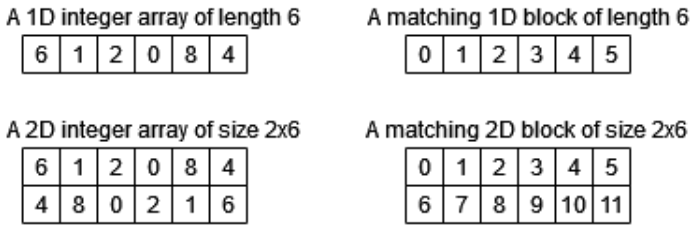

Figure 2.2: An example of two random shapes of input data and matching blocks of threads. The values in the blocks are the thread IDs.

The amount of threads in any of the three block dimensions can be specified

at launch, but the size limit in the x-dimension is 1024. All blocks combined form a so-called *grid*, which can also have up to three dimensions and has a limit of $2^{31} - 1$ blocks in the x-dimension [4]. Each block in a grid executes independently and their order of execution cannot be guaranteed. This means that, currently, the only way of synchronizing blocks is by exiting the kernel. On the other hand, synchronization between threads within a block is possible using provided functions such as "__syncthreads()".

As hinted at earlier, this model allows the programmer to consider a problem as a grid and divide it into smaller blocks of sub-problems that can be solved cooperatively by the threads in that block [4]. To illustrate this better, let us consider a common shape of input (and output) data: a vector of size N. There are many different algorithms that can be performed on this data type. For example, there might be a second input vector of size n that a programmer wants to add to the first vector.

In order to maximise parallelism, it would be ideal to request a thread for each element in the vector. In this case, a matching grid would then have $(n + 128 - 1)/128$ blocks with each block having 128 threads. Each thread in the grid now corresponds to exactly one index of the output vector (except for some threads in the last block if n is not divisible by 128). An appropriate kernel queries the thread ID of the executing thread and then performs the addition for the corresponding index. This can be implemented as shown in figure 2.3

```
1  __global__
2  void add_vector(int n, float *a, float *b)
3  {
4      int index = blockIdx.x * blockDim.x + threadIdx.x;
5      int stride = blockDim.x * gridDim.x;
6      for (int i=index;i < n;i+=stride)
7      {
8          a[i] = a[i] + b[i];
9      }
10 }
```

Figure 2.3: A CUDA kernel code example of vector addition.

First, a variable "index" is created by using the keywords "blockIdx.x", "blockDim.x" and "threadIdx.x" that are provided by CUDA. The first keyword refers to the location of the executing thread's block within the grid, the second one refers to the size of each block and the third refers to the executing thread's position within its own block.

Unfortunately, situations might arise where the executing grid is not large enough, because of a bug or hardware limitations. In this case, some or all threads are responsible for the calculation of more than one index,

with each of their indices separated in distance by the size of the grid. Therefore, a stride, which corresponds to the size of the grid, is calculated and a loop is used to prevent unintended behaviour, which is called a *grid-stride loop*[1]. Figure 2.4 gives a more visual and intuitive example of how the previous kernel calculates the index of the output vector corresponding to the executing thread.

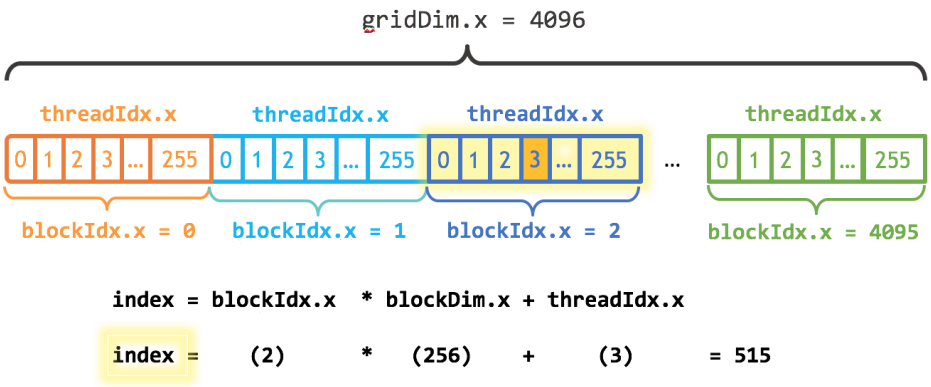

Figure 2.4: In this example, the grid contains 4096 blocks and each block has 256 threads. When the third thread in the third block of this grid executes, its corresponding index in the vector is calculated as shown. See footnote 1 for figure source.

2.2 Asynchronous memory transfers

The standard communication model in CUDA is synchronous, which means the memory transfers in figure 2.1 are also synchronous. Therefore, whenever a memory transfer is initiated, further execution on the *host* is blocked until the data transfer to the *device* has been completed.

Moreover, the device first copies the entire data from the host, then launches a kernel for the requested computation and afterwards it copies the resulting data back to the host. This entire process happens in what is referred to as the *default* (or *null*) stream. In general, a CUDA stream is a sequence of commands that execute in order [4].

Instead of only using the default stream, CUDA also support the notion of multiple streams. These different streams could then potentially be executed concurrently. Consequently, while the commands within one stream happen sequentially, the order in which they happen between streams is not guaranteed. Although this additional concurrency introduces an extra

[1] https://developer.nvidia.com/blog/even-easier-introduction-cuda/

layer of complexity, it also allows for asynchronous interaction between the host and device. For example, an asynchronous alternative for the memory transfers in figure 2.1 can be implemented as shown in figure 2.5

```
1  cudaHostRegister(a, N*sizeof(float), 0);
2  cudaStream_t stream;
3  cudaStreamCreate(&stream);
4
5  cudaMemcpyAsync(dev_a,a,N*sizeof(float),
6                  cudaMemcpyHostToDevice, stream);
7  kernel<<<blocks, block_n, 0, stream>>>(N, dev_a);
8  cudaMemcpyAsync(a,dev_a,N*sizeof(float),
9                  cudaMemcpyDeviceToHost, stream);
10 cudaDeviceSynchronize();
```

Figure 2.5: An example of ansynchronous communication in CUDA.

First, it is required to *pin* the host memory, which can be done with *cudaHostRegister(..)* and is shown on line 1. This pinning page-locks the memory and prevents it from being swapped out [28], which is vital, because asynchronous transfers allow the GPU to directly access the host's main memory [4]. On lines 2-3, a stream that is different from the default stream is declared and created. Next, on lines 5 and 8 a new method is shown, namely, *cudaMemcpyAsync(..)*, which initiates a memory transfer in the specified *stream*. As one might expect, *cudaMemcpyAsync(..)* does not block the host until the memory transfer is finished, unlike *cudaMemcpy(..)*. Lastly, *cudaDeviceSynchronize()* is introduced on line 10 to actually ensure that the host is blocked till all prior commands have finished, so that race conditions are prevented.

In general, switching to asynchronous communication can improve data throughput and overall performance depending on the combination of hardware used, the size of the input data and the ratio between computation and communication [28].

As hinted at earlier, it is possible to leverage these asynchronous transfers for the purpose of creating multiple streams, which each transfer a portion, or *chunk*, of the entire vector. On top of that, one could also divide the kernel computation between the different streams, so that each stream independently (1) transfers data to the device, (2) launches a kernel on this subset of data and then (3) transfers the final result back to the host.

In theory, this leads to transfers and computations within a stream overlapping with the commands in other streams [4]. Overlapping communication and computation can lead to increased performance and can, for example, be used to more efficiently split a stencil computation between multiple GPUs [23, 24].

2.3 Tiling

Using several streams to execute a given algorithm comes with a new challenge. Partitioning the input vector(s) into chunks of data and streaming them through the device will not produce the correct result if one stream is dependent on a value calculated by or assigned to another stream. Therefore, the key challenge is to find a partitioning scheme that does not violate any possible data dependencies.

Tiling is a strategy often used to both improve the data locality and to maximize parallelism of stencil computations [9]. I draw inspiration from this technique to make streaming, and ultimately out-of-core execution, possible for both map-like operations and stencil computations. Note that improving data locality in the context of out-of-core in recent work is also referred to as *temporal blocking* [5, 12, 24].

Several variants of tiling exist and the one preferred for a given stencil can depend on various factors such as dimensionality of the inputs, the processing device (i.e. CPU, GPU or both) and the memory architecture (i.e. shared or distributed). In this subsection, rectangular tiling [19] is introduced in the context of map-like operations, and, overlapped tiling [9, 19], [10, 11, 15] and split tiling [1, 8, 26] are introduced in the context of stencil computations.

2.3.1 Map-like operations

In this paper, a map-like operation is an index-wise function application with one or multiple lists as input. For example, common list operations in functional programming languages such as *map* and *zip* can be considered map-like operations. Another example would be vector addition, since it essentially is index-wise addition of multiple lists (typically two).

In order to stream map-like operations, it is necessary to divide the input vector(s) between the streams, so that the outputs corresponding to the resulting chunks of data can be calculated independently. The work of P.S. Rawat et al. [19] shows rectangular tiling to illustrate that this approach would not work for stencil computations. However, it is perfectly suitable for map-like operations.

An example of rectangular tiling for map-like operations is shown in figure 2.6. The dots resemble values in a vector at different time steps t. The arrows show which values in time step 0 are necessary to calculate a value in time step 1. If the calculation of a value in a given tile does not depend on a value in another tile, then that portion of the data can be computed concurrently. Clearly, the rectangular tiles are independent in the case of map-like operations for any number of iterations.

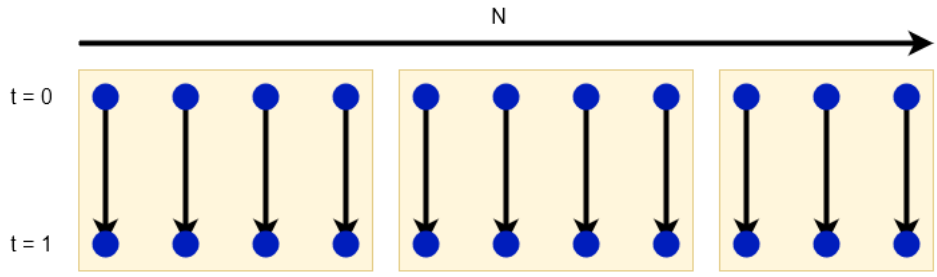

Figure 2.6: An example of rectangular tiling with map-like operations.

2.3.2 Stencil computations

An n-point stencil is a computational pattern that is iteratively applied on a k-dimensional data structure for several time steps. Stencil computations are used for a wide range of scientific and engineering disciplines [17] and have as a result been studied extensively in areas such as high performance computing, compilers and code generation [1, 8, 9, 10, 11, 15, 17, 19, 23, 24, 26].

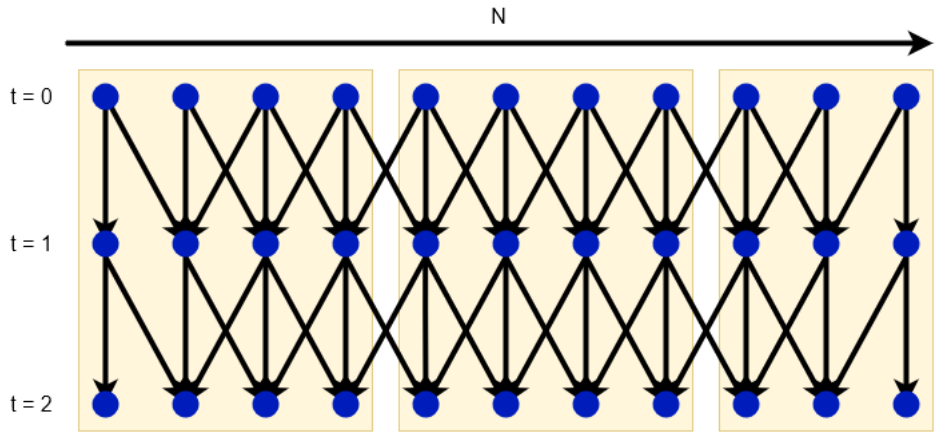

Figure 2.7: An example of rectangular tiling with stencil computations.

Figure 2.7 shows rectangular tiling for a one dimensional 3-point stencil computation. This stencil is the simplest variant of a stencil computation and is the one that this paper will focus on. A value at index i in time step t is calculated from the value at i in time step t - *1* as well as its two neighbouring values at i - *1* and i + *1*. Therefore, these tiles cannot be executed concurrently, because there are many inter-tile dependencies.

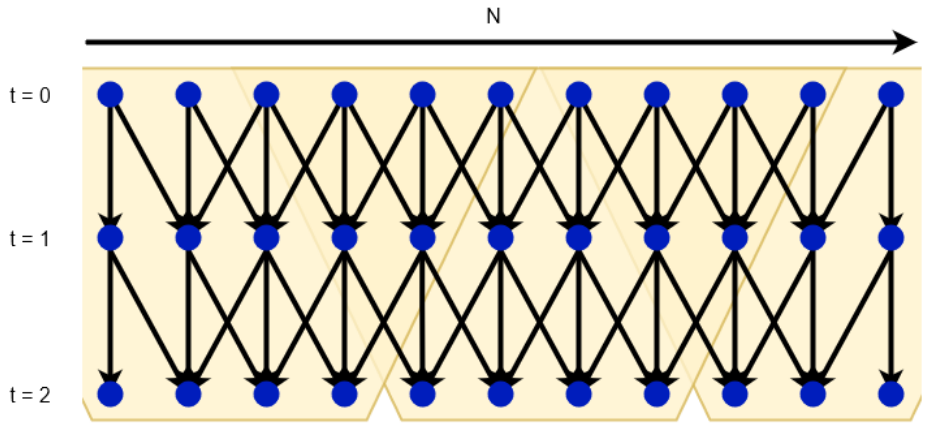

Figure 2.8: An example of overlapped tiling with stencil computations.

A commonly used tiling strategy for a stencil is overlapped tiling, which is shown in figure 2.8. When visualized across the time dimension, these tiles resemble trapezoidal shapes. The core idea behind this approach is that each tile does not only compute some number of values in the final time step, but also all the values that are required from earlier time steps, which results in an overlap between the tiles. Although this introduces redundant computations, it also solves the dependency issues and thus all the tiles can be computed concurrently. These overlapping regions are also referred to as halo regions [11, 19, 26], ghost zones [8, 15] or shadow regions [9].

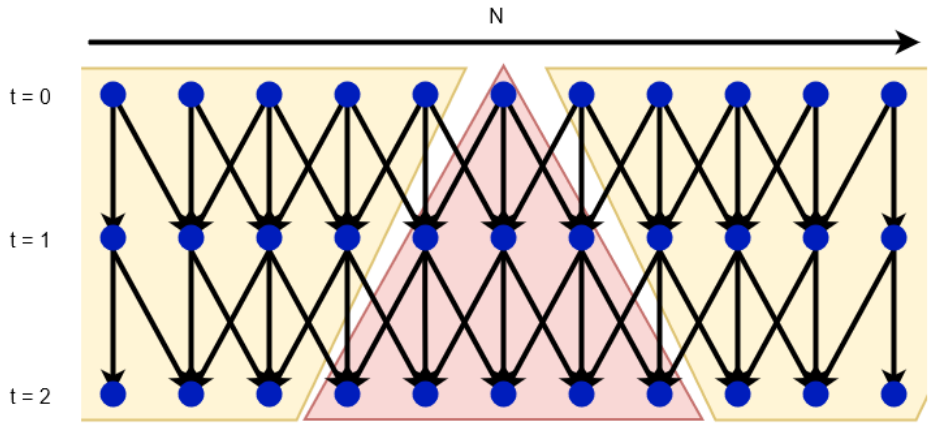

Figure 2.9: An example of split tiling with stencil computations.

An alternative, called split tiling, is shown in figure 2.9. This method does

not require redundant computations, but it does require more communication, because not all tiles are independent. The yellow trapezoidal tiles can be started concurrently, but the red triangle-shaped tile can only be computed when all the yellow tiles are finished (or at the very least the ones directly neighbouring the red tile). Note that the coverage of these tiles do not represent the memory foot-print, because the values that the red tile depends on still have to be in memory.

Furthermore, these tiling strategies are in GPUs usually applied at the thread block level and aim to improve performance by reducing global reads and kernel exits by utilizing the faster local memory shared between the threads in a block [10, 11, 15, 19]. However, the purpose of my research is different, because the goal is to be able to independently calculate much larger chunks of data. In turn, these independent chunks can then be streamed through the GPU to realize out-of-core execution.

Unfortunately, it is not trivial to determine which tiling approach would suit my use-case better and trying both is out of scope. For split tiling, there is no overhead from redundant computations, but the boundaries of each tile will need to be recomputed each loop. This again increases the amount of computation and leads to more control-flow divergence as pointed out by J. Meng, and K. Skadron [15]. However, the authors are considering strategies at the thread block level. Indeed, recomputing boundaries for each block of threads introduces a lot of extra computation. However, I only need to recompute the boundaries once for each stream per time step and this can be done by the host at each kernel launch.

Another concern is pointed out by P.S. Rawat et al.[19], being that split tiling leads to irregular shaped-tiles, which makes it more challenging to achieve high performance on a GPU. This is especially true at the chunk level, because the red triangles will be many orders smaller than the yellow trapezoids. However, the standard split tiling approach can be slightly altered to make the tiles regular, as I will show in chapter three.

Furthermore, the memory footprint of overlapped tiling is clearly larger, which is the main problem this research is trying to tackle. Additionally, while the computational overhead for 1D stencils can be kept relatively small, it becomes quite significant for 3D stencils [19]. Although I only consider a 1D 3-point stencil in my code generation schemes and experiments, it does not hurt to keep in mind how this would scale for possible future work.

Chapter 3

Out-of-core code generation

In this chapter, I present an approach to out-of-core code generation for both map-like operations and 3-point stencil computations, respectively, as they are defined in section 2.3.1 and 2.3.2.

For both algorithmic classes, this is be done by first defining an in-core specification in C++-like pseudo-code. After that, an in-core specification using CUDA streams is presented including rewriting rules and explanations. Based on these streamed blue prints, an out-of-core version is developed.

The in-core specification using CUDA streams serves as a solid checkpoint from both an explanatory and problem-solving perspective. Furthermore, it is also helps with investigating whether streaming a computation through the GPU is interesting regardless of whether the data fits in the memory or not.

3.1 Map-like operations

3.1.1 In-core

In figure 3.1, the in-core specification that I will use as a basis for the out-of-core code generation of map-like operations is shown. The host memory is pinned on lines 1-2 as described in figure 2.5, because this will be a requirement for using CUDA streams later on. Next, two buffers, *dev_a* and *dev_b*, are allocated on the device (lines 4-5) to match the given buffers a and b on the host and, after that, the host memory is transferred to the device (h2d) on lines 7-8. Subsequently, n kernels are launched for a given t time steps (lines 10-14). Although more than one time step does not occur as often as with stencil computations, it is still useful to consider and helpful for performance analysis. At the end, the resulting values in *dev_a* are copied to the host memory (d2h) on line 14.

```
1  a = cudaHostRegister(a, n);
2  b = cudaHostRegister(b, n);
3
4  dev_a  = cudaMalloc(dev_a, n);
5  dev_b = cudaMalloc(dev_b, n);
6
7  dev_a  = cudaMemcpy(dev_a, a, n, h2d);
8  dev_b = cudaMemcpy(dev_b, b, n, h2d);
9
10 for(i = 1; i <= t; i++){
11     dev_a = kernel<<< n >>>(dev_a, dev_b, n);
12 }
13
14 a = cudaMemcpy(a, dev_a, n, d2h);
15
16 kernel(dev_a, dev_b, n)
17 {
18     tid = blockIdx.x * blockDim.x + threadIdx.x;
19     stride = blockDim.x * gridDim.x;
20
21     for(i = tid; tid < n; tid+=stride)
22         dev_a[i] = f (dev_a[i], dev_b[i]);
23 }
```

Figure 3.1: The in-core specification of a map-like operation.

The kernel code is shown on lines 16-23. First, the index of the value in *dev_a* that the executing thread has to calculate is determined and stored in variable *tid*. Next, the stride is calculated just in case the total amount of requested threads is smaller than the size of *dev_a* in the device's memory. Finally, *dev_a[i]* is calculated using the values at i in both *dev_a* and *dev_b* and applying generic function f to them ($i \in [0..n-1]$). This procedure corresponds to the background of grid-stride-loops and map-like operations as described in sections 2.1.3 and 2.3.1, respectively.

3.1.2 In-core streamed

In order to stream a map-like operation through the device, it is necessary to find a way to perform calculations on subsets of the data independently. Recall that a map-like operation is defined as applying a generic function f index-wise to both a value from vector a and vector b.

For example, the operation could be $a[i] = a[i] + b[i]$, where $i \in [0..n-1]$ and $f := (+)$, which corresponds to vector addition. Since you only need the values at i from time step j, where $j \in [0..t]$, to calculate the value i at time step $j + 1$, it is straightforward to create chunks of data that can be calculated independently.

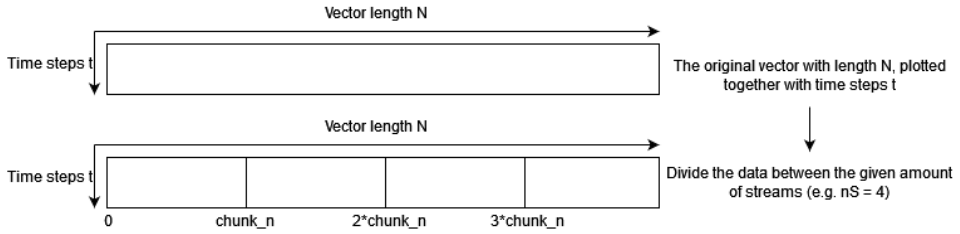

Figure 3.2: A visual representation of dividing a vector in independent chunks for map-like operations.

This process is illustrated in figure 3.2. Clearly, it is possible to calculate $(n + nS - 1)$ / nS and store the result in variable *chunk_n*. Next to representing the chunk size, *chunk_n* can also be used to calculate the offset of each independent chunk in the host and the device buffer. However, do keep in mind that the right-most chunk is not always equal to *chunk_n*, so, a second variable, *lchunk_n*, is necessary, which is equal to $n - nS * chunk\_n$.

```
1  kernel(dev_a, dev_b, n)
2  {
3      tid = blockIdx.x * blockDim.x + threadIdx.x;
4      stride = blockDim.x * gridDim.x;
5
6      for(i = tid; tid < n-1; tid+=stride)
7          dev_a[tid] = f (dev_a[i], dev_b[i]);
8  }
```

$\rightarrow\rightarrow\rightarrow$

```
1  kernel(dev_a, dev_b, offset_l, offset_r)
2  {
3      tid = blockIdx.x * blockDim.x + threadIdx.x + offset_l;
4      stride = blockDim.x * gridDim.x;
5
6      for(i = tid; tid < offset_r; tid+=stride)
7          dev_a[tid] = f (dev_a[i], dev_b[i]);
8  }
```

Figure 3.3: A rewriting schematic for rewriting in-core kernel code to in-core streamed kernel-code. The upper code block shows the original kernel code and the lower code block shows the resulting kernel code.

In order to ensure that each stream can only compute on its assigned data, it is necessary to introduce bounds in the kernel code, which is shown in figure 3.3. The changes are on lines 1, 3 and 6 in the lower code block.

First, two new arguments, *offset_l* and *offset_r* are added to the kernel. The former represents the left bound of the executing threads in the current time step and the latter the right bound. Next, *offset_l* is added to *tid* on line 3 to make sure the threads access the correct region of the vectors. Finally, *offset_r* replaces $n-1$ on line 6 in order to prevent the threads from accessing data that the specific stream should not be operating on.

```
1 dev_a  = cudaMemcpy(dev_a, a, n, h2d);
2 dev_b = cudaMemcpy(dev_b, b, n, h2d);
```

$\rightarrow\rightarrow\rightarrow$

```
1  stream[nS];
2  for(i = 0; i < nS; i++)
3      cudaStreamCreate(stream[i]);
4
5  chunk_n = (n + nS - 1) / nS
6  lchunk_n = n - (nS-1)*chunk_n;
7
8  for(i = 0; i < nS-1; i++){
9    cudaMemcpyAsync(dev_a[i*chunk_n], a[i*chunk_n], chunk_n, h2d,
10                   stream[i]);
11   cudaMemcpyAsync(dev_b[i*chunk_n], b[i*chunk_n], chunk_n, h2d,
12                   stream[i]);
13 }
14 cudaMemcpyAsync(dev_a[(nS-1)*chunk_n], a[(nS-1)*chunk_n],
15                 lchunk_n, h2d, stream[nS-1]);
16 cudaMemcpyAsync(dev_b[(nS-1)*chunk_n], b[(nS-1)*chunk_n],
17                 lchunk_n, h2d, stream[nS-1]);
```

Figure 3.4: A rewriting schematic for rewriting in-core memory transfers to in-core streamed memory transfers (host to device). The upper code block shows the original synchronous memory transfers and the lower code block shows the rewritten code.

Since the map-like operation is now split between streams, each stream should copy their own portion of data. The necessary code transformation regarding the copying of data is shown in figure 3.4. First, the requested *nS* streams have to be explicitly created, which is done on lines 1-3. After that, variables *chunk_n* and *lchunk_n* are calculated as defined earlier on lines 5-6. The formula used ensures that *lchunk_n* is equal to *chunk_n* or smaller in case *nS* does not divide *n*. Finally, asynchronous memory transfers are introduced for each stream on lines 8-17.

The first *nS-2* streams copy the same amount of data (*chunk_n* values), following the exact same offset pattern (at *i*chunk_n*), so these transfers can be initiated in a loop. The last stream also follows the same offset pattern, but instead needs to transfer *lchunk_n* values, so its transfers are declared separately.

```
1 for(i = 1; i <= t; i++){
2     dev_a = kernek<<< n >>>(dev_a, dev_b, n);
3 }
```

$\rightarrow\rightarrow\rightarrow$

```
1 for(i = 1; i <= t; i++){
2   for(j = 0; j < nS-1; j++){
3     kernek<<< chunk_n, stream[j] >>>(dev_a, dev_b, j*chunk_n,
4                                                    (j+1)*chunk_n);
5   }
6   kernek<<< lchunk_n, stream[nS-1] >>>(dev_a, dev_b,
7                                          (nS-1)*chunk_n,
8                                  (nS-1)*chunk_n+lchunk_n);
9 }
```

Figure 3.5: A rewriting schematic for rewriting in-core kernel launches to in-core streamed kernel launches. The upper code block shows the original kernel launches and the lower code block shows the rewritten code up-to and including the synchronization mid-point.

Similarly, instead of one kernel per time step, *nS* kernels per time step are now required. The rewriting scheme for this purpose is shown in figure 3.5. As with the memory transfers, the first *nS-2* streams follow the same boundary pattern and use the same number of threads for their calculations. The left bound is *j*chunk_n* and the right bound is *(j+1)*chunk_n*. Their exact kernel launches are on lines 2-5.

Although on line 2 in the upper code block it is only necessary to specify the number of threads, on lines 3 and 6 in the lower code block the stream that will execute the specific kernel also has to be given. Furthermore, the last stream's kernel is declared separately on lines 6-9, because it uses less threads and the distance between its left and right bound is different from the other streams.

Notably, due to streams being asynchronous, the order of execution of these kernels cannot be guaranteed. It might seem like this could lead to synchronization issues, but it does not, because the data the different streams operate on is fully independent. The only thing that matters is that the time steps of each stream happen chronologically, which they will, because all kernels and transfers *within* a stream are synchronous.

The final necessary adjustment is shown in figure 3.6. Similar to *h2d* memory transfers, each stream copies their assigned data back to the host when it finishes its kernels. This is done by declaring an asynchronous memory transfer for each stream in the same manner as before, but now with the copy direction *d2h*. At the end, on line 8, the host is blocked till all streams are finished executing their tasks. This ensures that the entire map-like operation has finished before giving control back to the host.

```
1 a = cudaMemcpy(dev_a, a, n, d2h);
```

$\rightarrow\rightarrow\rightarrow$

```
1 for(i = 0; i < nS-1; i++){
2   cudaMemcpyAsync(a[i*chunk_n], dev_a[i*chunk_n],
3                   chunk_n, d2h, stream[i]);
4 }
5 cudaMemcpyAsync(a[(nS-1*chunk_n], dev_a[(nS-1)*chunk_n],
6                 lchunk_n, d2h, stream[nS-1]);
7
8 cudaDeviceSynchronize();
```

Figure 3.6: A rewriting schematic for rewriting in-core memory transfers to in-core streamed memory transfers (device to host). The upper code block shows the synchronous memory transfer and the lower code block shows the rewritten asynchronous memory transfers.

3.1.3 Out-of-core

Previously, out-of-core was defined as performing an algorithm on data that is too large to fit into the GPU's memory in one go. An important part of my strategy to accomplish this is to stream chunks of data through the device using CUDA. In section 3.1.2, I show how this can be done for in-core data in combination with map-like operations. Now, in this section I present my strategy for rewriting the in-core streamed version to out-of-core.

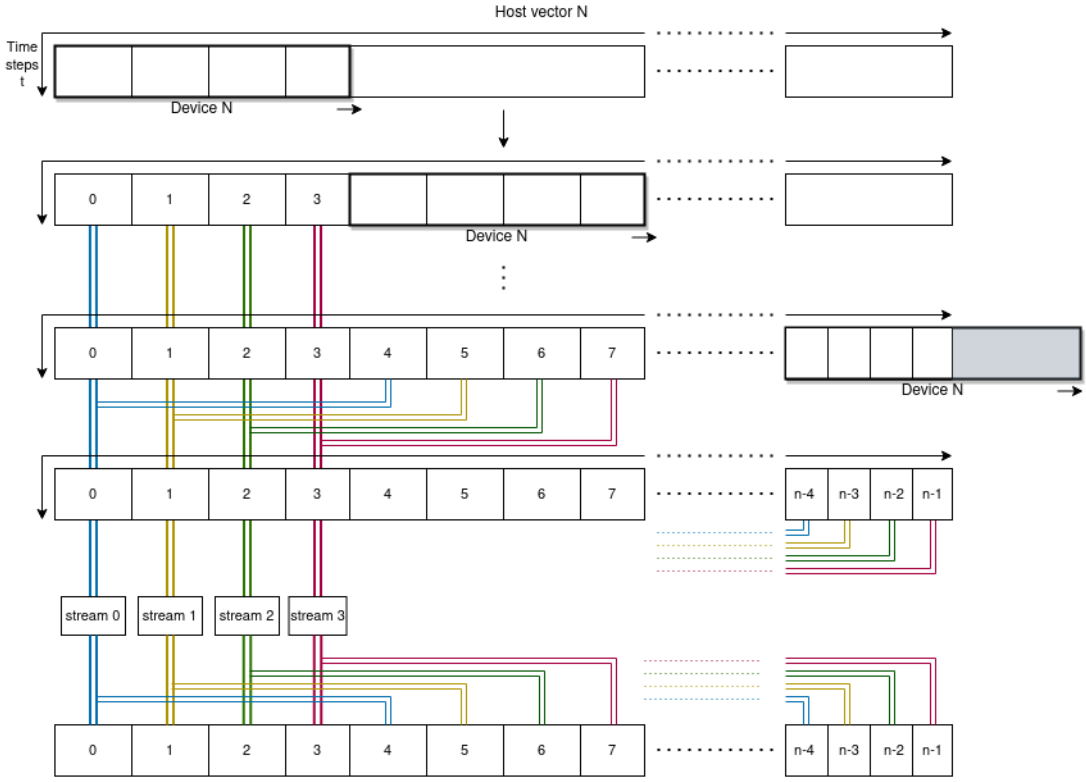

Figure 3.7: A visual representation of the device memory moving as a sliding window across the host memory.

Since it is already shown that the chunks can be calculated independently, the next step is to figure out a procedure for actually streaming the data through the GPU in an out-of-core manner. To accomplish this, I opt for an approach that is based on the idea of using the device's memory as a sliding window that can be moved over the host's memory from left to right, which is depicted in figure 3.7.

The algorithm starts off at index zero and divides the data that currently overlaps with the window in a predetermined number of independent chunks. Subsequently, these chunks can be streamed in similar fashion as in section

3.1.2.

However, after doing so, there is still data left on the host that has not been processed yet. Therefore, while the device starts copying memory and launching kernels, the sliding window is moved forward until it exclusively covers unprocessed data. Next, it is possible to already stream the next portion, which essentially creates several queues of chunks that are streamed concurrently through the GPU, which is illustrated in figure 3.8.

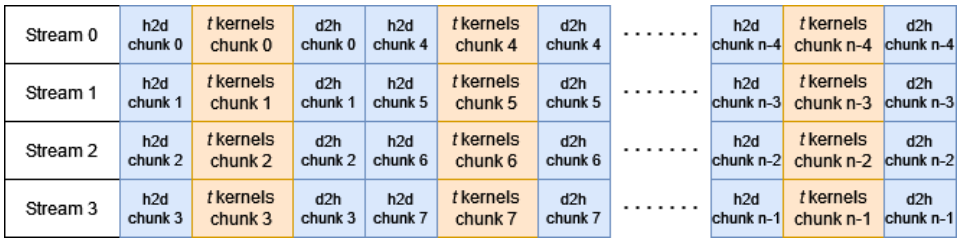

Figure 3.8: A visual representation of the sequence of actions each stream takes based on figure 3.7.

There are two aspects interesting to note here. First of all, it is now necessary to execute t kernels before transferring results back to the host, because any occupied space on the device has to be reused for the next chunk. Clearly, it would be difficult to keep track of how many time steps remain for different chunks and it would also introduce additional unnecessary memory transfer.

Secondly, my method of out-of-core streaming can be seen as a greedy approach, because at any point, I try to fit as much data on the device as possible, without considering how well the available memory divides the data on the host. Consequently, choosing the right values for the different parameters involved is non-trivial, but establishing heuristics is out of scope for this research.

```
1 dev_a  = cudaMalloc(dev_a, n);
2 dev_b = cudaMalloc(dev_b, n);
```

$\rightarrow\rightarrow\rightarrow$

```
1 free_mem = cudaMemGetInfo()
2 dev_n = free_mem * 95 / 100 / dtype_n / 2
3
4 dev_a = cudaMalloc(dev_a, dev_n)
5 dev_b = cudaMalloc(dev_b, dev_n)
```

Figure 3.9: A rewriting schematic that transforms the allocation of device buffers based on n, to allocation based on the available memory on the device.

Since allocating two buffers of size n on the device is no longer possible, the original buffer allocation needs to be changed, which is shown in figure 3.9. First, the amount of free memory on the device in bytes is queried on line 1 in the lower code block. Considering that it is often not possible to use all of the free memory, only 95 percent of the total available space is used, which is shown on line 2. The exact size one should choose here depends on the architecture, so to automate this in future, heuristics should be established.

Furthermore, the new value is divided by *dtype_n*, which refers to the size in bytes of the datatype that is used. Finally, the value is also divided by two, because this operation requires the allocation of two buffers. The result is stored in the variable *dev_n* and then used on lines 4-5 to allocate *dev_a* and *dev_b*.

Now that *dev_n* has been established, most of the rewritten code in section 3.1.2 needs to be altered. The start of this is shown in figure 3.10 and other parts of the new code are spread over multiple figures, which is also indicated with comments on lines 9-10 in the lower code block. Fortunately, the general structure of the code can be reused and thus a lot of the new code will look very familiar.

However, figure 3.10 mostly contains new variables and ideas. First, recall that the core idea is to divide the hosts data up in chunks that do fit on the device and can be independently processed, at the very least, one after each other. Therefore, it is necessary to introduce an approach that is based on the idea of having several rounds of memory transfers and computations.

As mentioned earlier, the number of rounds is calculated in a greedy manner by dividing n by *dev_n* and stored in the variable r, which is shown on line 1 in the lower code block. Next, the calculation of *chunk_n* and *lchunk_n* is slightly altered on lines 2-3 by exchanging n with *dev_n*.

Similar to the possible difference between *chunk_n* and *lchunk_n*, the last round is likely to be smaller than round 0 to n-2. Therefore, the size of the last round and the chunk sizes of the last round are calculated on lines 5-7. Their values are then stored in *lr_n*, *lr_chunk_n* and *lr_lchunk_n*, respectively.

Finally, the actual code for the implementation of these rounds is split in to two separate figures. The code for rounds *0 to n-2* is shown in figure 3.11 and the code for the final round is shown in figure 3.12.

```
chunk_n = (n + nS - 1) / nS
lchunk_n = n - (nS-1)*chunk_n;

for(i = 0; i < nS-1; i++){
  cudaMemcpyAsync(dev_a[i*chunk_n], a[i*chunk_n], chunk_n, h2d,
                  stream[i]);
  cudaMemcpyAsync(dev_b[i*chunk_n], b[i*chunk_n], chunk_n, h2d,
                  stream[i]);
}
cudaMemcpyAsync(dev_a[(nS-1)*chunk_n], a[(nS-1)*chunk_n],
                lchunk_n, h2d, stream[nS-1]);
cudaMemcpyAsync(dev_b[(nS-1)*chunk_n], b[(nS-1)*chunk_n],
                lchunk_n, h2d, stream[nS-1]);

for(i = 1; i <= t; i++){
  for(j = 0; j < nS-1; j++){
    kernel<<< chunk_n, stream[j] >>>(dev_a, dev_b, j*chunk_n,
                                                   (j+1)*chunk_n);
  }
  kernel<<< lchunk_n, stream[nS-1] >>>(dev_a, dev_b,
                                       (nS-1)*chunk_n,
                                (nS-1)*chunk_n+lchunk_n);
}

for(i = 0; i < nS-1; i++){
  cudaMemcpyAsync(a[i*chunk_n], dev_a[i*chunk_n],
                  chunk_n, d2h, stream[i]);
}
cudaMemcpyAsync(a[(nS-1*chunk_n], dev_a[(nS-1)*chunk_n],
                lchunk_n, d2h, stream[nS-1]);
```

$\rightarrow\rightarrow\rightarrow$

```
r = (n + dev_n - 1) / dev_n;
chunk_n = (dev_n + nS - 1) / nS;
lchunk_n = dev_n - (nS-1) * chunk_n;

lr_n = n - (r-1) * dev_n;
lr_chunk_n = (lr_n + nS - 1) / nS;
lr_lchunk_n = lr_n - (nS-1) * lr_chunk_n;

// rounds 0 to n-2, see figure 3.11
// round n-1, see figure  3.12
```

Figure 3.10: A rewriting schematic that transforms the in-core streaming of map-like operations, to out-of-core.

```
1  for (i = 0; i < r-1; i++){
2    for(j = 0; j < nS-1; j++){
3      cudaMemcpyAsync(dev_a[j*chunk_n], a[i*dev_n+j*chunk_n],
4                      chunk_n, h2d, stream[j]);
5      cudaMemcpyAsync(dev_b[j*chunk_n], b[i*dev_n+j*chunk_n],
6                      chunk_n, h2d, stream[j]);
7    }
8    cudaMemcpyAsync(dev_a[(nS-1)*chunk_n],
9                    a[i*dev_n+(nS-1)*chunk_n], lchunk_n, h2d,
10                   stream[nS-1]);
11   cudaMemcpyAsync(dev_b[(nS-1)*chunk_n],
12                   b[i*dev_n+(nS-1)*chunk_n], lchunk_n, h2d,
13                   stream[nS-1]);
14
15   for(k = 1; k <= t; k++){
16     for (j = 0; j < nS-1; j++){
17       kernek<<< chunk_n, stream[j] >>>(dev_a, dev_b, j*chunk_n,
18                                                      (j+1)*chunk_n);
19     }
20     kernek<<< lchunk_n, stream[nS-1] >>> (dev_a, dev_b,
21                                           (nS-1)*chunk_n,
22                                  (nS-1)*chunk_n+lchunk_n);
23   }
24
25   for(j = 0; i < nS-1; j++){
26     cudaMemcpyAsync(a[j*dev_n+j*chunk_n], dev_a[j*chunk_n],
27                     chunk_n, d2h, stream[j]);
28   }
29   cudaMemcpyAsync(a[j*dev_n+(nS-1)*chunk_n],
30                   dev_a[(nS-1)*chunk_n], lchunk_n, d2h,
31                   stream[nS-1]);
32 }
```

Figure 3.11: The rewritten code of rounds 0 to *r - 2* for performing map-like operations out-of-core.

As mentioned earlier, the structure of the code in figure 3.11 is similar to the code in the upper code block of figure 3.10. The *h2d* memory transfers are depicted on lines 2-13, the kernel calls on lines 15-23 and the *d2h* memory transfers on lines 25-31. The access patterns and number of threads are exactly the same for the first *r-1* rounds, so the code can be executed in a loop *r-1* times.

In order to access the correct chunk of data in the host, the indices for *a* and *b* are altered, which is shown on lines 3, 5, 9, 12, 26 and 29. First, the offset for the entire data assigned to the current round is calculated by *i*dev_n*. Next, it is possible to get the correct region of data to each stream *j* by adding *j*chunk_n*. Similar to the streamed version, the memory transfers and kernel calls for the last stream are declared separately, because it operates on *lchunk_n* data instead of *chunk_n*.

```
1  for(j = 0; j < nS-1; j++){
2    cudaMemcpyAsync(dev_a[j*chunk_n],
3                    a[(r-1)*dev_n+j*lr_chunk_n], lr_chunk_n,
4                    h2d, stream[j]);
5    cudaMemcpyAsync(dev_b[j*chunk_n],
6                    b[(r-1)*dev_n+j*lr_chunk_n], lr_chunk_n,
7                    h2d, stream[j]);
8  }
9  cudaMemcpyAsync(dev_a[(nS-1)*chunk_n],
10                 a[(r-1)*dev_n+(nS-1)*lr_chunk_n],
11                 lr_lchunk_n, h2d, stream[nS-1]);
12 cudaMemcpyAsync(dev_b[(nS-1)*chunk_n],
13                 b[(r-1)*dev_n+(nS-1)*lr_chunk_n],
14                 lr_lchunk_n, h2d, stream[nS-1]);
15
16 for(k = 1; k <= t; k++){
17   for(j = 0; j < nS-1; j++) {
18     kernel<<< lr_chunk_n, stream[j] >>>(dev_a, dev_b, j*chunk_n
         ,
19                                         j*chunk_n+lr_chunk_n);
20   }
21   kernel<<< lr_lchunk_n, stream[nS-1] >>>(dev_a, dev_b,
22                                           (nS-1)*chunk_n,
23                                 (nS-1)*chunk_n+lr_lchunk_n);
24 }
25
26 for(j = 0; j < nS-1; j++) {
27   cudaMemcpyAsync(a[(r-1)*dev_n+j*lr_chunk_n],
28                   dev_a[j*chunk_n], lr_chunk_n, d2h, stream[j]);
29 }
30 cudaMemcpyAsync(a[(r-1)*dev_n+(nS-1)*lr_chunk_n],
31                 dev_a[(nS-1)*chunk_n], lr_lchunk_n, d2h,
32                 stream[nS-1]);
```

Figure 3.12: The rewritten code for the kernel calls in round *r - 1* for map-like operations out-of-core.

The final round of the out-of-core map-like operation looks similar to rounds *0* to *n-2*. However, the total round size and most of the chunk sizes are different. Additionally, each occurrence of *i* is replaced with *r-1*, because of the lack of a loop.

First, the *h2d* memory transfers are shown on lines 1-8 in figure 3.12. The access patterns to *dev_a* and *dev_b* remain the same, but all the other instances of *chunk_n* and *lchunk_n* are replaced with *lr_chunk_n* and *lr_lchunk_n*, respectively. Next, the kernel calls are shown on lines 16-24. Although the left bounds remain the same, the right bounds now depend on *lr_chunk_n* and *lr_lchunk_n*. Furthermore, the number of threads are also reduced to their respective sizes for the last round. Finally, the *d2h* memory transfers are shown on lines 26-32, which contain the same changes as for *h2d*.

3.2 3-point stencil computations

3.2.1 In-core

```
1  in = cudaHostRegister(in, n);
2  out = cudaHostRegister(out, n);
3
4  dev_in  = cudaMalloc(in_dev, n);
5  dev_out = cudaMalloc(out_dev, n);
6
7  dev_in  = cudaMemcpy(dev_in, in, n, h2d);
8  dev_out = cudaMemcpy(dev_out, out, n, h2d);
9
10 for(i = 1; i <= t; i++){
11     dev_out = kernel<<< n >>>(dev_in, dev_out, n);
12     if( i != t )
13         dev_in, dev_out = swap(dev_in, dev_out);
14 }
15
16 out = cudaMemcpy(dev_out, out, n, d2h);
17
18 kernel(dev_in, dev_out, n)
19 {
20     tid = blockIdx.x * blockDim.x + threadIdx.x;
21     stride = blockDim.x * gridDim.x;
22
23     for(i = tid; tid < n-1; tid+=stride)
24         dev_out[i] = f ( dev_in[i-1], dev_in[i],
25                                 dev_in[i+1]);
26 }
```

Figure 3.13: The in core specification of a three-point stencil computation.

In figure 3.13, the in-core specification that I use as a basis for the out-of-core code generation of 3-point stencil computations is shown. The pinning of the host memory (lines 1-2), the memory allocation on the device (lines 4-5) and the memory transfers (lines 7-8, 16) are identical to those in the in-core specification for map-like operations. On the other hand, some variables names, the kernel launches and the kernel code are slightly different.

First of all, the memory pointers *a*, *b*, *dev_a* and *dev_b* are replaced with *in*, *out*, *dev_in* and *dev_out*. I make this decision, because, with stencil computations, it is helpful to make a clear distinction between the in- and output buffer.

Secondly, after each time step, the pointers to *dev_in* and *dev_out* are swapped (line 13), but only if the last time step has not yet been made. The reason for this is that in a given time step i, where $i \in [1..t-1]$, the output buffer becomes the input buffer in time step $i + 1$.

Finally, the actual computation in the kernel code has to be adjusted to

a 3-point stencil. This is shown on lines 24-25, where *dev_out[i]* is calculated using the values at *dev_in[i-1]*, *dev_in[i]* and *dev_in[i+1]* and applying generic function f to them ($i \in [1..n-2]$). This procedure corresponds to the background of a 3-point stencil computation as described in section 2.3.2.

3.2.2 In-core streamed

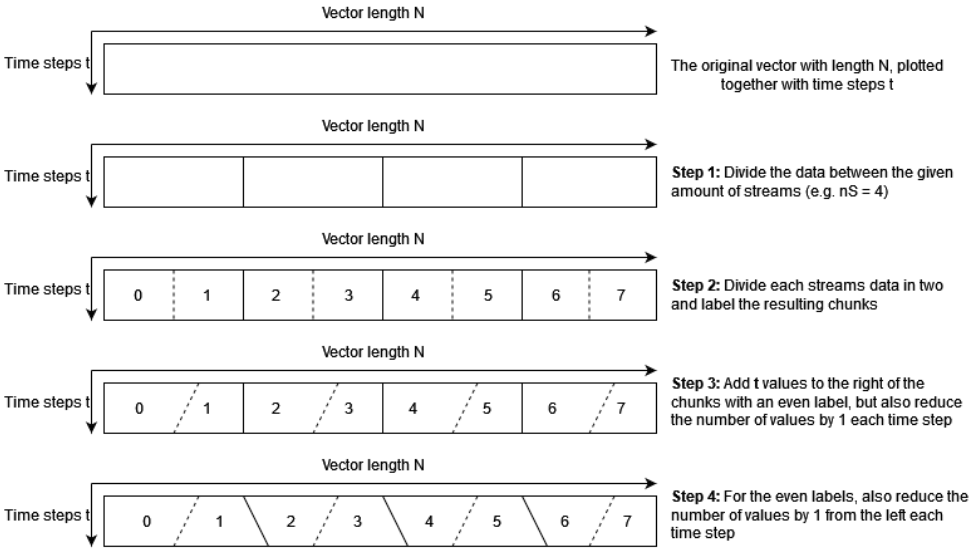

Figure 3.14: A visual representation of dividing a vector in (partially) independent chunks for 3-point stencil computations.

In order to stream a stencil computation through the device, it is necessary to find a way to calculate subsets of the data independently. Unfortunately, this is not as straightforward as with map-like operations, because in order to calculate some value i in time step j, values *i-1*, i and *i+1* in time step *j - 1* have to be calculated first. Clearly, if value *i-1* or *i+1* happens to be part of a different stream, then value i cannot be calculated independently.

In figure 3.14, my strategy for calculating independent trapezoidal-shaped chunks over the time dimension is shown. For convenience, the trapezoids are numerically labeled from left to right starting at 0. Consequently, it is possible to distinguish between two types of trapezoids based on their label; even- and odd trapezoids.

First, step one divides the entire data over the number of streams that were given, which is four in this example. Now, it would be possible to immediately start launching t independent kernels on the streams. However, as hinted at earlier, the streams can only safely read the data they are assigned to, so each time step the number of values that can be calculated,

reduces at the edges.

As one might expect, this would create pyramid-shaped gaps over the time dimension and these will still need to be calculated separately after all streams have finished their kernels. However, these gaps would, most of the time, contain a lot less data then the total data the streams originally started with. Therefore, in order to balance the workload, I divide the data of each stream in half before doing any calculations, as is illustrated in the second step of figure 3.14.

Next, the even chunks are given t extra values at $t = 0$, but the total amount of values is reduced each time step at the edges. This process will create equally sized trapezoidal-shaped chunks as is shown through step three and step four. Note that it is necessary to repeat this process several times if the individual chunk size is smaller than or equal to the number of time steps. However, this scenario will not be considered in the code further on.

Furthermore, these trapezoids cannot be calculated all at once, as pointed out in section 2.3.2. To solve this, each stream first has to calculate an even trapezoid, wait for the other streams to finish and then calculate an odd trapezoid.

```
1 kernel(dev_in, dev_out, n)
2 {
3     tid = blockIdx.x * blockDim.x + threadIdx.x;
4     stride = blockDim.x * gridDim.x;
5
6     for(i = tid; tid < n-1; tid+=stride)
7         dev_out[tid] = f ( dev_in[tid-1], dev_in[tid],
8                            dev_in[tid+1] );
9 }
```

$\rightarrow\rightarrow\rightarrow$

```
1 kernel(dev_in, dev_out, offset_l, offset_r)
2 {
3     tid = blockIdx.x * blockDim.x + threadIdx.x + offset_l;
4     stride = blockDim.x * gridDim.x;
5
6     for(i = tid; tid < offset_r; tid+=stride)
7         dev_out[tid] = f ( dev_in[tid-1], dev_in[tid],
8                            dev_in[tid+1] );
9 }
```

Figure 3.15: A rewriting schematic for rewriting in-core kernel code to in-core streamed kernel-code. The upper code block shows the original kernel code and the lower code block shows the resulting kernel code.

In practice, the trapezoids are computed by introducing boundaries in the

kernel code, which is shown in figure 3.15. The changes are on lines 1, 3 and 6 in the lower code block. First, two new arguments, *offset_l* and *offset_r*, are added to the kernel. The former represents the left bound of the executing threads in the current time step and the latter the right bound. Finally, *offset_l* is added to *tid* on line 3 and *offset_r* replaces $n - 1$ on line 6.

Each time step, these boundaries are reduced or increased on one or both sides by the host. This ensures that each stream only calculates the indices it is responsible for based on the trapezoid it is calculating. However, before diving into more details, it is necessary to understand in which order the trapezoids have to be calculated.

As mentioned before, each stream will calculate two trapezoids; one even and one odd. In general, this means that the first stream calculates trapezoids 0 and 1, the second calculates trapezoids 2 and 3, et cetera. More formally, when given n streams, then stream i, where $i \in [0..n - 1]$, calculates trapezoids *2i* and *2i + 1*.

Moreover, the even trapezoids can immediately be calculated and are fully independent. The non-trivial part is that the odd trapezoids cannot be calculated until the trapezoids they depend on have been calculated. This problem can be solved by introducing a synchronization point. Although there might be several approaches one can take, I have chosen to block the calculation of odd trapezoids until *all* streams have finished calculating their even trapezoid. Not only is this approach simple and concise, it also requires the least synchronization.

All things considered, we can sum up the following workflow for one stream:

1. Copy the data necessary to calculate the even trapezoid to the device.
2. Calculate the even trapezoid.
3. Copy the part of the result that is already final back to host memory.
4. Wait for all other streams to finish calculating their even trapezoid.
5. Copy the remaining data necessary to calculate the odd trapezoid.
6. Calculate the odd trapezoid.
7. Copy the the remainder of the final result to host memory.

```
1  dev_in  = cudaMemcpy(dev_in, in, n, h2d);
2  dev_out = cudaMemcpy(dev_out, out, n, h2d);
```

$\rightarrow\rightarrow\rightarrow$

```
1  stream[nS];
2  for(i = 0; i < nS; i++)
3      cudaStreamCreate(stream[i]);
4
5  chunk_n = (n + 2*nS - 1) / 2*nS;
6  lchunk_n = n - (2*nS-1)*chunk_n;
7
8  for(i = 0; i < nS; i++){
9      cudaMemcpyAsync(dev_in[i*2*chunk_n], in[i*2*chunk_n],
10                     chunk_n + t, h2d, stream[i]);
11     cudaMemcpyAsync(dev_out[i*2*chunk_n], out[i*2*chunk_n],
12                     chunk_n + t, h2d, stream[i]);
13 }
```

Figure 3.16: A rewriting schematic for rewriting in-core memory transfers to in-core streamed memory transfers (host to device). The upper code block shows the original synchronous memory transfers and the lower code block shows the rewritten code.

In order to realise part one of the workflow, it is necessary to rewrite the memory transfers from host to device. This is shown in figure 3.16. First, CUDA streams are created with given amount nS on lines 1-3.

Next, the size of each individual chunk, which is depicted in step 2 of figure 3.14, is calculated and stored in variable *chunk_n*. It is also necessary to keep into account that n may not be divisible by nS. Therefore, the variable *lchunk_n* is introduced to calculate the size of the right-most chunk.

Finally, asynchronous memory transfers are declared on lines 8-12. The number of each stream, i, and *chunk_n* are used to find the correct offset in host memory and device memory. In order to calculate the even trapezoids, each stream has to copy *chunk_n* + t values to the device (see the dotted lines in step 3 in figure 3.14).

```
1  for(i = 1; i <= t; i++){
2      kernek<<< n >>>(dev_in, dev_out, n);
3      if( i != t )
4          dev_in, dev_out = swap(dev_in, dev_out);
5  }
```

$\rightarrow\rightarrow\rightarrow$

```
1  for(i = 1; i <= t; i++){
2    kernek<<< chunk_n+t-i, stream[0] >>>(dev_in, dev_out, 1,
3                                         chunk_n+t-i);
4    for(j = 1; j < nS; j++){
5      kernek<<< chunk_n+t-2*i, stream[j] >>>(dev_in, dev_out,
6                                 j*2*chunk_n+i, (2*j+1)*chunk_n+t-i);
7    }
8    if( i != t )
9      dev_in, dev_out = swap(dev_in, dev_out);
10 }
11
12 cudaMemcpyAsync(out[0], dev_out[0], chunkSize, d2h, stream[0]);
13
14 for(i = 1; i < nS; i++){
15   cudaMemcpyAsync(out[i*2*chunk_n+t], dev_out[i*2*chunk_n+t],
16                   chunkSize-t, d2h, stream[i]);
17 }
18
19 cudaDeviceSynchronize();
```

Figure 3.17: A rewriting schematic for rewriting in-core kernel launches to in-core streamed kernel launches. The upper code block shows the original kernel launches and the lower code block shows the rewritten code up-to and including the synchronization mid-point.

In figure 3.17, a rewriting schematic is shown for rewriting the original kernel code in such a way that it matches steps 2-4 of the outlined workflow.

First, the single kernel call on line 2 in the upper code block has to be replaced by kernel calls for each individual stream. This is shown on lines 1-10 in the lower code block. The kernel call for stream 0 is called separately, because each time step only the right bound decreases. All the other kernels can be started in a for loop, because both boundaries shrink with the same pattern. Stream 0 calculates *chunk_n + t - 1* values in it's first time step and *chunk_n* values in its last. The rest of the streams calculate *chunk_n + t - 2* values in their first time step and *chunk_n - t* in their last. This is coherent with the approach shown in step 3 of figure 3.14.

Next, new code is introduced on lines 12-19 to implement step three and four of the workflow. Notably, only the values calculated in the last time step are final. Since stream 0 calculates more values in it's last time step than the other streams, its memory transfer back to the host is called

separately on line 12. However, the other streams again follow the same pattern, so their memory transfers can be called in a for loop. Their left bounds in the final time step are moved t indices to the right and their right bounds are moved t indices to the left (see figure 3.14, step 3). Therefore, the source and destination offsets in the device and host memory are equal to *i * 2 * chunk_n + t*.

Finally, a synchronization call is made on line 19, which represents the fourth step in the work flow. It is important to note here that, technically, the streams don't actually wait, but the host is blocked till all streams on the device have finished their tasks.

```
1  if(t % 2 == 0)
2      swap(dev_in, dev_out)
3
4  for(i = 0; i < nS-1; i++){
5    cudaMemcpyAsync(dev_in[(2*i+1)*chunk_n+t],
6                    in[(2*i+1)*chunk_n+t],
7                    chunk_n-t, h2d, stream[i]);
8    cudaMemcpyAsync(dev_out[(2*i+1)*chunk_n+t],
9                    out[(2*i+1)*chunk_n+t],
10                   chunk_n-t, h2d, stream[i]);
11 }
12
13 cudaMemcpyAsync(dev_in[(2*(nS-1)+1)*chunk_n+t],
14                 in[(2*(nS-1)+1)*chunk_n+t], lchunk_n-t, h2d,
15                 stream[nS-1]);
16 cudaMemcpyAsync(dev_out[(2*(nS-1)+1)*chunk_n+t],
17                 out[(2*(nS-1)+1)*chunk_n+t], lchunk_n-t, h2d,
18                 stream[nS-1]);
19
20 for(i = 1; i <= t; i++){
21     for(j = 0; j < nS-1; j++){
22         kernel<<< chunk_n-t+2*i, stream[j] >>>(dev_in, dev_out,
23                                                (2*j+1)*chunk_n+t-i
                                                      ,
24                                                (2*j+2)*chunk_n+i);
25     }
26     kernel <<< lchunk_n-t+i, stream[nS-1] >> > (dev_in, dev_out,
27                                                 (2*nS-1)*chunk_n+t-i,n-1)
                                                       ;
28     if( i != t )
29         dev_in, dev_out = swap(dev_in, dev_out);
30 }
```

Figure 3.18: Memory transfers and kernel calls for odd trapezoids, to be added directly after the rewritten code in figure 3.17, but before line 16 in the original in-core specification in figure 3.13.

In figure 3.18, code is shown for steps five and six of the workflow. This code will be generated right after the code in figure 3.17 and just before the code

on line 16 in the original specification. Before continuing the process, the program checks whether *dev_in* actually points to the original input vector (line 1-2). This means that if t is even, *dev_in* and *dev_out* need to be swapped.

After that, the remaining data is copied from the host to the device, which is shown on lines 4-16. Every stream but the last follow the same pattern for finding the correct offset on both the host and the device. The data for the even trapezoid of any stream i starts at *2 * i * chunk_n*, so the data for the odd trapezoid starts at *(2*i + 1)*chunk_n*. However, the first t values were already copied during the process of calculating the even trapezoid, so the offset becomes *(2*i + 1)*chunk_n + t*. Subsequently, the number values that need to be transferred then is *chunk_n - t*.

Similarly, the data transfer executed by the last stream, *nS - 1*, also has offset *(2*i + 1)*. However, it is necessary to keep into account that the size of the last chunk may not be equal to *chunk_n*. Therefore, the number of values that are copied should be *lchunk_n - t*.

Finally, the code for calculating the odd trapezoids is shown on lines 18-28. The process of finding the left- and right bound for the kernel calls is similar to that of the even trapezoids, but now the bounds will be expanded each time step instead of decreased (see figure 3.14, step 3). This means that stream j, where $j \in [0..nS-2]$, starts of with its left bound at *(2*j + 1)*chunk_n + t - 1* and its right bound at *(2*j + 2)*chunk_n + 1*. For the last stream, only the left bound increases and the right bound stays the same, which means the starting offsets are *(2*nS - 1)*chunk_n + t - 1* and *n - 1*, respectively. After t time steps, the last stream will calculate *lchunk_n* values and the others calculate *chunk_n + t* values.

```
1  out = cudaMemcpy(dev_out, out, n, d2h);
```

$\rightarrow\rightarrow\rightarrow$

```
1  for(i = 0; i < nS-1; i++){
2      cudaMemcpyAsync(out[(2*i+1)*chunk_n],
3                      dev_out[(2*i+1)*chunk_n], chunk_n+t, d2h,
4                      stream[i]);
5  }
6  cudaMemcpyAsync(out[(2*(nS-1)+1)*chunk_n],
7                  dev_out[(2*(nS-1)+1)*chunk_n], lchunk_n, d2h,
8                  stream[nS-1]);
9
10 cudaDeviceSynchronize();
```

Figure 3.19: A rewriting schematic for rewriting the final in-core memory transfer to in-core streamed memory transfers. The upper code block shows the synchronous memory transfer and the lower code block shows the rewritten asynchronous memory transfers.

In order to implement the seventh and last step of the work flow, the final memory transfer of the in-core specification needs to be rewritten. This is shown in figure 3.19. At this point, each stream still has to copy back the results that were obtained by calculating the odd trapezoids.

The rewritten code for this is shown on lines 1-8. The offset in device and host memory is equal to *(2*i+1)*chunk _n*, which is the mid point of all the values stream i is responsible for (see the dotted lines in figure 3.14, step 2). The number of values that are copied from the device to the host is equal to *chunk_n* + t for all streams except for the last one, which is equal to *lchunk_n*.

Additionally, since all of these memory transfers are asynchronous, one more synchronization is necessary, which is shown on line 10. This ensures that the final result is correctly copied back to the host and that no unexpected behaviour will occur.

3.2.3 Out-of-core

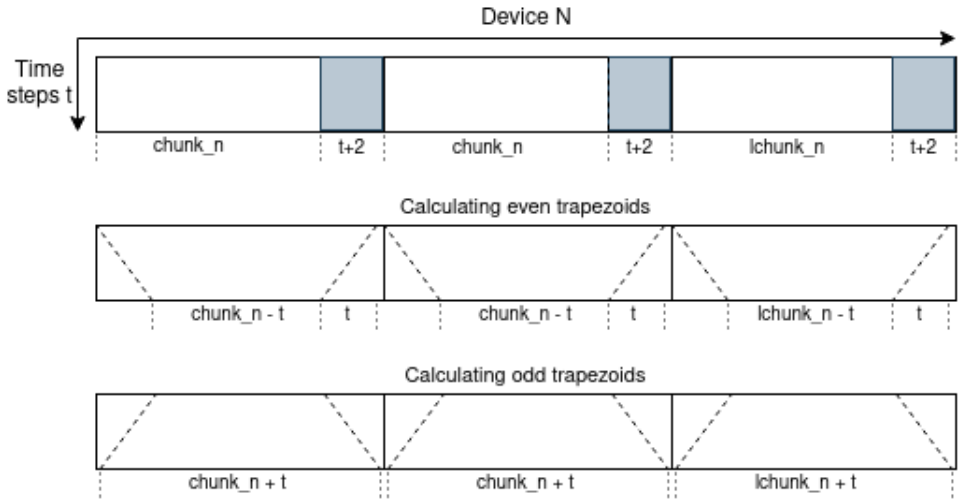

Figure 3.20: A visual representation of dividing the device memory into independent chunks for out-of-core 3-point stencils.

In this section, I present my out-of-core solution for 3-point stencil computations. Many of the core ideas for the out-of-core blueprints of map-like operations also apply here. However, due to the dependencies of the 3-point stencil, which were also discussed in sections 3.2.1 and 2.3.2, the implementation is a lot more challenging.

First of all, the allocation of device memory and the division of data between streams needs to be more carefully considered, which is shown in figure 3.20. Instead of directly dividing the memory, it is first necessary to

reserve space for the number of time steps that will be done. After that, the remainder can be equally divided between the streams. Notably, it is necessary to reserve two extra values on top of the number of time steps (i.e. $t + 2$), because in the last time step of the odd trapezoids *chunk_n+t* values are calculated. Clearly, in case of a 3-point stencil computation, in order to compute *chunk_n* + t values, *chunk_n* + t + 2 are needed.

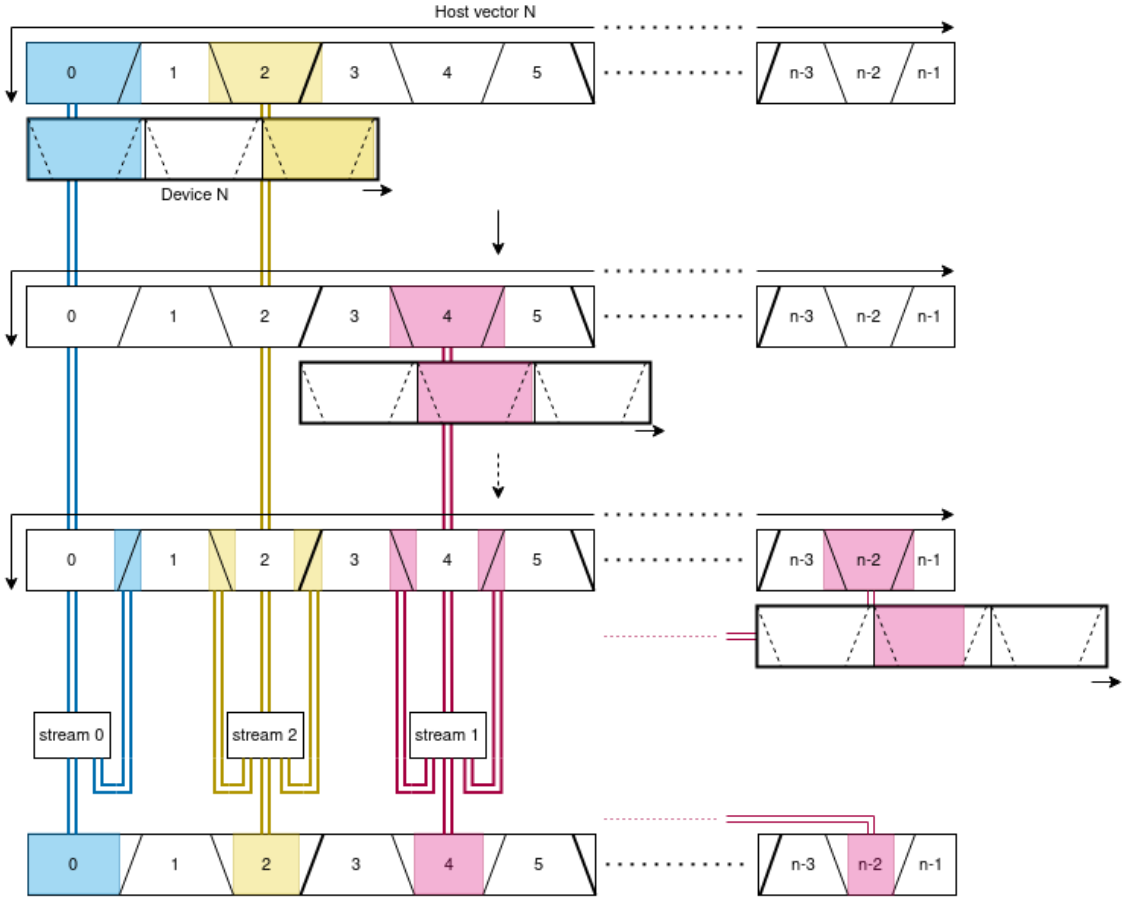

Figure 3.21: A visual representation of the device moving as a sliding window over the host in order to stream the even trapezoids.

Next, figure 3.21 shows the processing of all the even trapezoids in the host vector, based on the division in figure 3.20. Similar to map-like operations, the device memory is used as a sliding window that moves across the host memory. However, it is necessary to more carefully consider which of the chunks that currently overlap with the window will be send to the GPU. Indeed, only the even trapezoids can be processed at this time, so they have to be precisely located. In case of an odd number of streams, the number

of chunks that should be considered in each round alternates.

Furthermore, after executing t kernels on the even chunks, the *d2h* memory transfers also have to be slightly adjusted. Although the final *chunk_n* - t values (see figure 3.20) are still copied back to the host's output buffer as before, the other *2*t* values that are not yet final are needed for the odd trapezoids later on. Before, when the entire host vector fitted in the device memory, these unfinished values could be left in place. However, during out-of-core execution, the space they occupy is required to compute the next chunk. Therefore, two additional *d2h* transfers that copy the *2*t* intermediate values to the host's input buffer have to be in introduced. Note that these intermediate values also have to be transferred to the host's output buffer, so the *d2h* transfer containing final results should transfer *chunk_n* + t values instead of *chunk_n* - t. It is essential that these memory transfers are done correctly, because, otherwise, the final output buffer will not be correct.

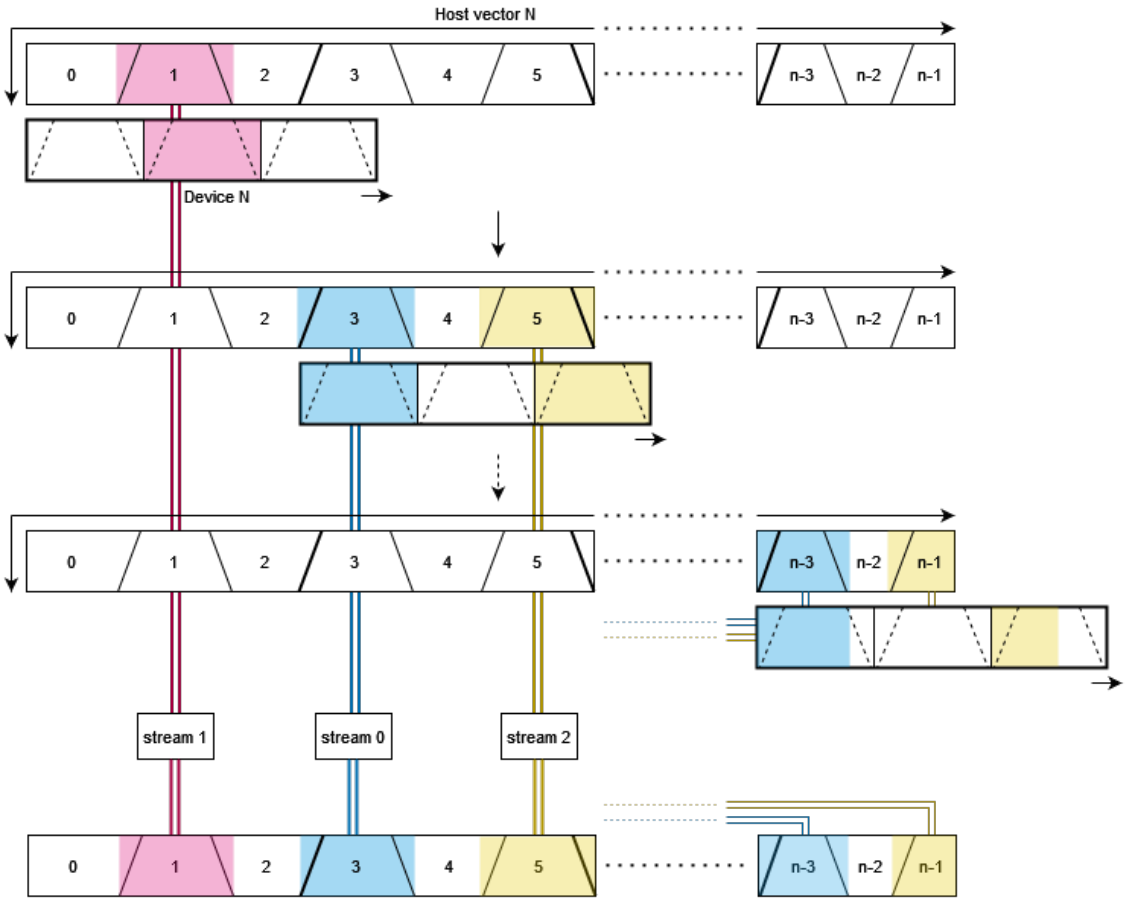

Figure 3.22: A visual representation of the device moving as a sliding window over the host in order to stream the odd trapezoids.

Finally, figure 3.22 shows the processing of all the odd trapezoids in the host vector. In this case, there are less changes required than for the even trapezoids, but it is still important to carefully locate the odd trapezoids on the host. As mentioned before (and shown in figure 3.20), it is also necessary to copy two additional values for each chunk from the host compared to the in-core streamed version. Namely, instead of *chunk_n + t* values, *chunk_n + t + 2* values are copied. When a stream finishes the computation of an odd trapezoid, it only has to copy the final *chunk_n + t* output values once to the host's output buffer.

Although allowing for any number of streams was not much of an issue up until now, doing so for the out-of-core 3-point stencil would result in convoluted code. The reason for this is that the patterns for locating chunks in the host buffer differ between an odd and even number of streams. There-

fore, the transformations that are presented only consider odd amounts of streams.

```
1 dev_in  = cudaMalloc(dev_in , n);
2 dev_out = cudaMalloc(dev_out , n);
```

$\rightarrow\rightarrow\rightarrow$

```
1 free_mem = cudaMemGetInfo()
2 dev_n = free_mem * 95 / 100 / dtype_n / 2
3
4 dev_in = cudaMalloc(dev_in , dev_n)
5 dev_out = cudaMalloc(dev_out , dev_n)
6
7 r_n = dev_n - nS * (t+2);
```

Figure 3.23: A rewriting schematic that transforms the allocation of device buffers based on n, to allocation based on the available memory on the device.

Similar to out-of-core map-like operations, allocating two buffers of size n on the device is no longer possible, so the original buffer allocation on the device needs to be changed, which is shown in figure 3.23.

The necessary transformation is almost completely the same for both types of out-of-core computations, except that part of *dev_n* needs to be reserved for the time steps of each chunk. Therefore, the variable *r_n* is introduced on line 7 in the lower code block. It is calculated by subtracting *nS * (t + 2)* from the available memory on the device (*dev_n*).

Moreover, the out-of-core 3-point stencils have a similar round-based approach as out-of-core map-like operations, but both the code for the calculation of even trapezoids and odd trapezoids need their own r rounds instead. Note that the synchronization shown in the in-core streamed version between the computation of even and odd trapezoids is still required.

On top of that, it is necessary to be more selective of which streams are used in which round, as shown in figure 3.21 and 3.22. Accordingly, when calculating even trapezoids, code should only be executed for any stream j if it has the same parity as current round i, where $j \in [0..nS-1]$ and $i \in [0..r-1]$. On the other hand, when calculating odd trapezoids, code should only be be executed for any stream j if it does *not* have the same parity as current round i. This is achieved by surrounding code for memory transfers and kernels with if-statements as shown in figure 3.24.

```
1 // in case of even trapezoids
2 if((i % 2) == (j % 2))
3     // h2d, kernel or d2h
4
5 // in case of odd trapezoids
6 if((i % 2) != (j % 2))
7     // h2d, kernel or d2h
```

Figure 3.24: An example of using if-statements to select the correct streams j for transfers or kernel computations in a given round i.

The rest of the code transformations that follow after figure 3.24 are largely the same as for the in-core streamed version, but combined with the changes introduced for out-of-core map-like operations and the usage of if-statements as shown in figure 3.24. Therefore, and also to improve readability, all the rewriting schemes for calculating even trapezoids can be looked at in detail in part A.1 of the appendix. Similarly, all the transformations for odd trapezoids are present in part A.2 of the appendix.

Chapter 4

Performance evaluation

In this chapter, a performance analysis of the in-core, in-core streamed, and out-of-core versions is presented for both algorithmic classes.

4.1 Experimental setup

There are two aspects of interest regarding the rewritten code. Namely, the impact in-core streaming has on the overall performance and whether it is possible to achieve the same performance out-of-core compared to when the data fits on the device. In order to answer these questions, the in-core, in-core streamed and out-of-core blueprints are implemented in C++[1].

Furthermore, GFLOP/s and (effective) memory bandwidth are used as a measure of performance. The former refers to the number of one billion floating-point operations per second done by the GPU and gives a good impression of the overall achieved performance. The latter resembles the memory throughput in GB/s. Since both algorithms are limited by how fast memory can be accessed on the device (i.e. memory bound), it is important to see whether the achieved bandwidth comes close to the *theoretical* bandwidth of the device. The code that is used to compute these metrics can be found in appendix A.3.

Kernel	# FLOPs	mem. reads	mem. writes
map-like	1	2	1
3-point	5	3	1

Table 4.1: An overview of the number of floating-point operations, memory reads and memory writes in each kernel execution of the different algorithms.

In case of the map-like operation, a simple vector addition consisting of one floating-point operation (FLOP) is implemented. On the other hand, the

[1] https://gitlab.science.ru.nl/pbeurden/thesis

stencil computation is evaluated with a one dimensional 3-point heat equation consisting of five FLOPs (three multiplications, two additions), without considering convergence. An overview of the numbers relevant for the performance metrics is shown in table 4.1 for both algorithms.

Moreover, the experiments are conducted with individual vector sizes ranging from 20MB to 25GB consisting of single-precision floats and up to 3883 time steps. All combinations of parameters are performed ten times each and the combined run-time of the memory transfers and kernel executions is measured using CUDA events. The exact manner is shown in figure 4.1 and essentially covers the precise moment the first host-to-device transfer is initiated until the final device-to-host transfer has completed.

```
1 // ...
2 cudaEventRecord(start);
3 // first h2d
4 // ...
5 // last d2h
6 cudaDeviceSynchronize()
7 cudaEventRecord(stop);
8 cudaEventSynchronize(stop);
9 cudaEventElapsedTime(&run_time, start, stop);
```

Figure 4.1: Example of how the run-time of each experiment is measured with CUDA events. The event "start" is recorded just before the first h2d memory transfer and the event "stop" is recorded just after the last d2h transfer finishes.

Most of the time, the standard deviation of these runs is between 0.01%-2%, so the mean value is used to calculate the performance metrics. However, there can occasionally be one outlier within a set of ten runs. In that case, the value is removed and the mean of the remaining nine values is used instead.

The experiments are done on a shared cluster that is provided by the university. The cluster in question consists of two separate nodes on one of which a combination of CPU and GPU is used. The details of the hardware and software of the test system are shown in table 4.2. Note that the theoretical peak bandwidth of the GPU, which is based on the memory clock frequency, is equal to 616 GB/s.

Test system	
Hardware	Software
Intel(R) Xeon(R) Silver 4214 12 cores, 2 threads per core @ 2.20GHz min. 1.0 GHz, max 3.2 GHz NVIDIA GeForce RTX 2080 Ti 11GB GDDR6 616 GB/s (peak bandwidth) 13.45 TFLOP/s (max. perf. float) PCI-e 3.0 x16 (15.75 GB/s)	Ubuntu 20.04.4 LTS 5.13.0-28-generic (kernel) NVIDIA driver 510.47.03 CUDA 11.6 gcc 9.4.0

Table 4.2: An overview of the system used for the experiments.

4.2 Map-like operations

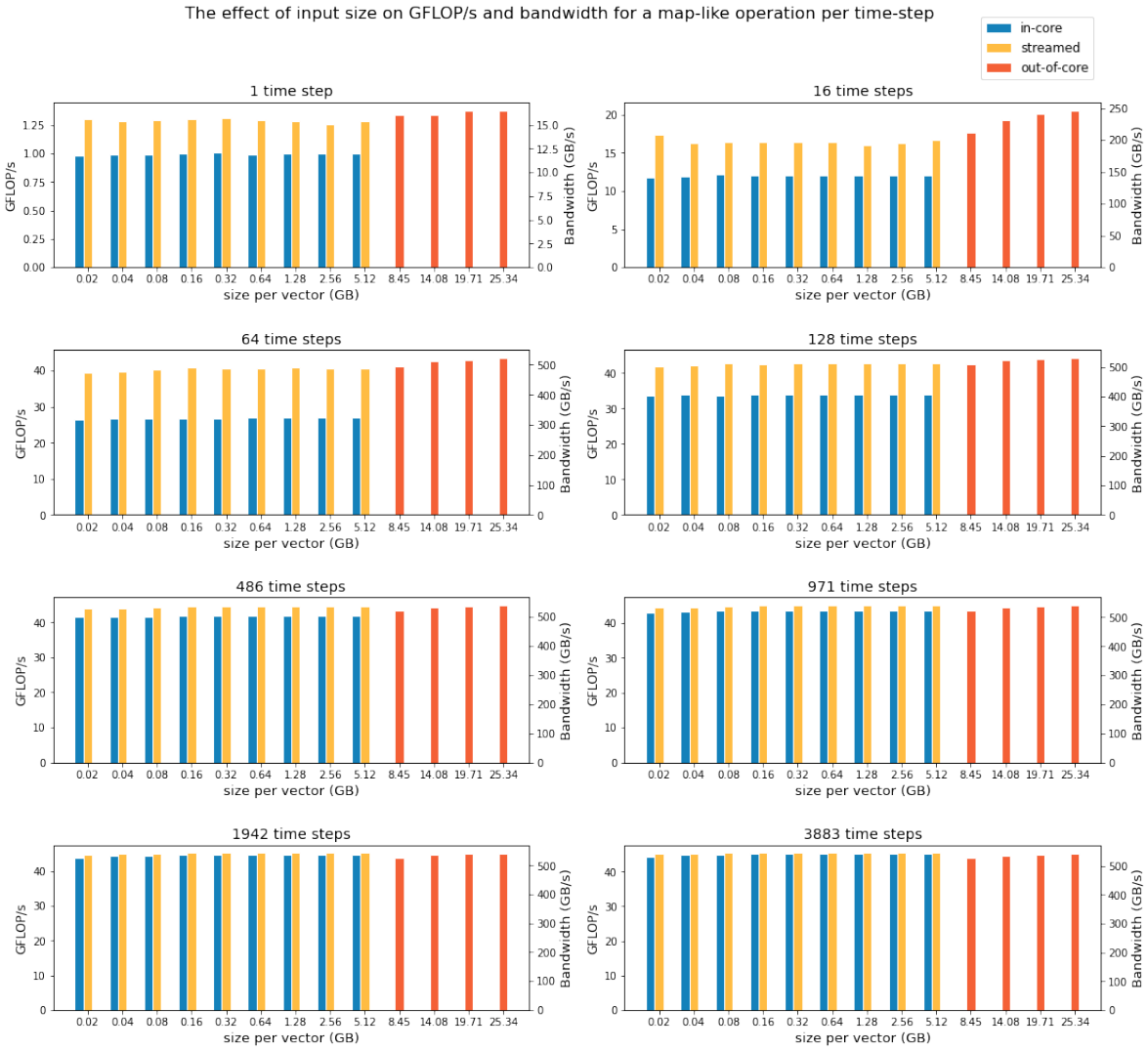

Figure 4.2: The effect of input size on the performance of an in-core, streamed and out-of-core map-like operation at all different time steps.

In figure 4.2, the effect of input size on the performance in terms of GFLOP/s and bandwidth is shown. The left y-axis shows the GFLOP/s and the right y-axis shows the bandwidth. The generally low GFLOP/s and bandwidth at 1 and 16 time steps indicates that this procedure is, initially, latency bound. However, as the number of time steps go up, the GFLOP/s and bandwidth increase to roughly 45 and 515, respectively. Clearly, the effec-

tive bandwidth gets close to the theoretical peak bandwidth, which indeed shows that, overall, the computation is memory bound.

There are several interesting things that stand out. First of all, the streamed and out-of-core version do not only perform significantly better than the in-core version for up to 128 time steps, they reach the peak performance faster. Even at 3883 time steps, the in-core version has not completely caught up yet with the streamed version in terms of performance. However, do note that intermediate results are not copied out between iterations, which is uncommon for map-like operations. Secondly, it seems to be the case that input size has no significant effect on the performance.

Input size	Time steps	Streams	GFLOP/s (tot.)	Bandwidth (tot.)	GFLOP/s (kernel)	Bandwidth (kernel)
1280M	1	-	1.01	12.16	**45.65**	**547.87**
1280M	1	8	**1.32**	**15.92**	44.76	537.15
1280M	3883	-	45.23	542.85	**45.75**	**549.07**
1280M	3883	2	**45.48**	**545.83**	45.7	548.48

Table 4.3: The performance of two data points for both in-core and streamed map-like operations.

Table 4.3 shows two data points for the vector addition of 1280 million elements, where the in-core version is indicated with a dash in the *Streams* column. Both the overall performance, measured as shown in figure 4.1, and the performance of the kernel(s) exclusively, which was measured with nvprof, are visible. At one time step, the performance of the in-core kernel beats the combined performance of the streamed kernels and both are already equal to the approximate peak overall performance shown in figure 4.2. However, the overall performance of the streamed version is **1.3x** higher than the in-core version. Based on the kernel performance, it seems to be the case that this gain can be attributed to the overlapping of communication and computations. At 3883 time steps, the streamed version still performs slightly better, but the difference has become insignificant. This decrease can probably be explained by the fact that the compute-to-communication ratio increases over time, which is consistent with the observation that the performance increase, in general, seems to come from latency hiding.

During the experiments, different numbers of streams were chosen, of which the results are shown in figure 4.3 and 4.4. Note that the reason for these values being slightly different compared to figure 4.3 is that they are not from the same data set, because, initially, nvprof was not used for profiling and, also, the load on the university cluster differs.

All variants perform better than the in-core version for one time step and 1280 million elements, but choosing correctly does seem to be beneficial. The worst performing number of streams is 128, with an approximate

bandwidth of 13.5 GB/s. On the other hand, the best performing number of streams has a bandwidth of approximately 15.3 GB/s, which is, roughly, a **13%** difference.

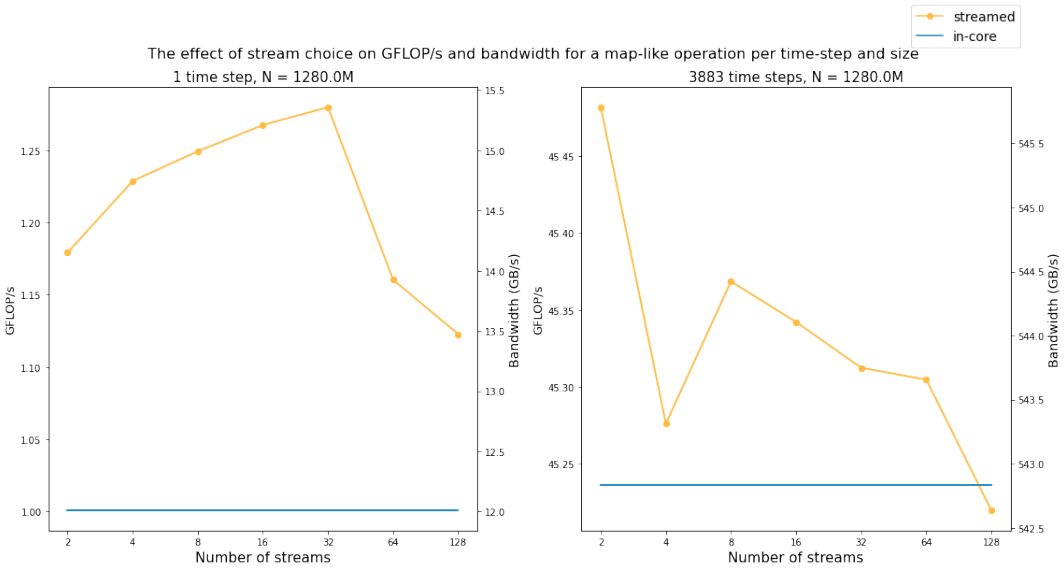

Figure 4.3: The performance of vector addition with 1280 million elements measured with different numbers of streams at 1 and 3883 time steps.

This performance difference between the choice of number of streams remains at 3883 time steps, but it, clearly, is a lot less significant. The discrepancy between the two extremes is now only **0.6%**.

Furthermore, the plots for out-of-core vector addition of 6336 million elements, shown in figure 4.4, tell a similar story. The difference between the extremes at one time step and 3883 time steps in this case is approximately **14.5%** and **0.55%**, respectively.

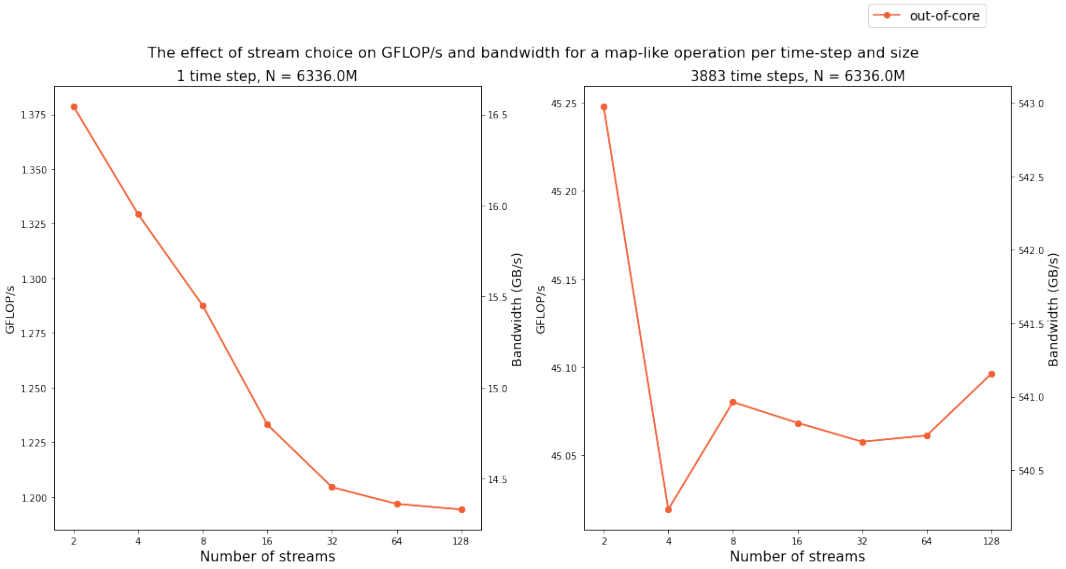

Figure 4.4: The performance of vector addition with 6338 million elements measured with different numbers of streams at 1 and 3883 time steps.

4.3 3-point stencil computations

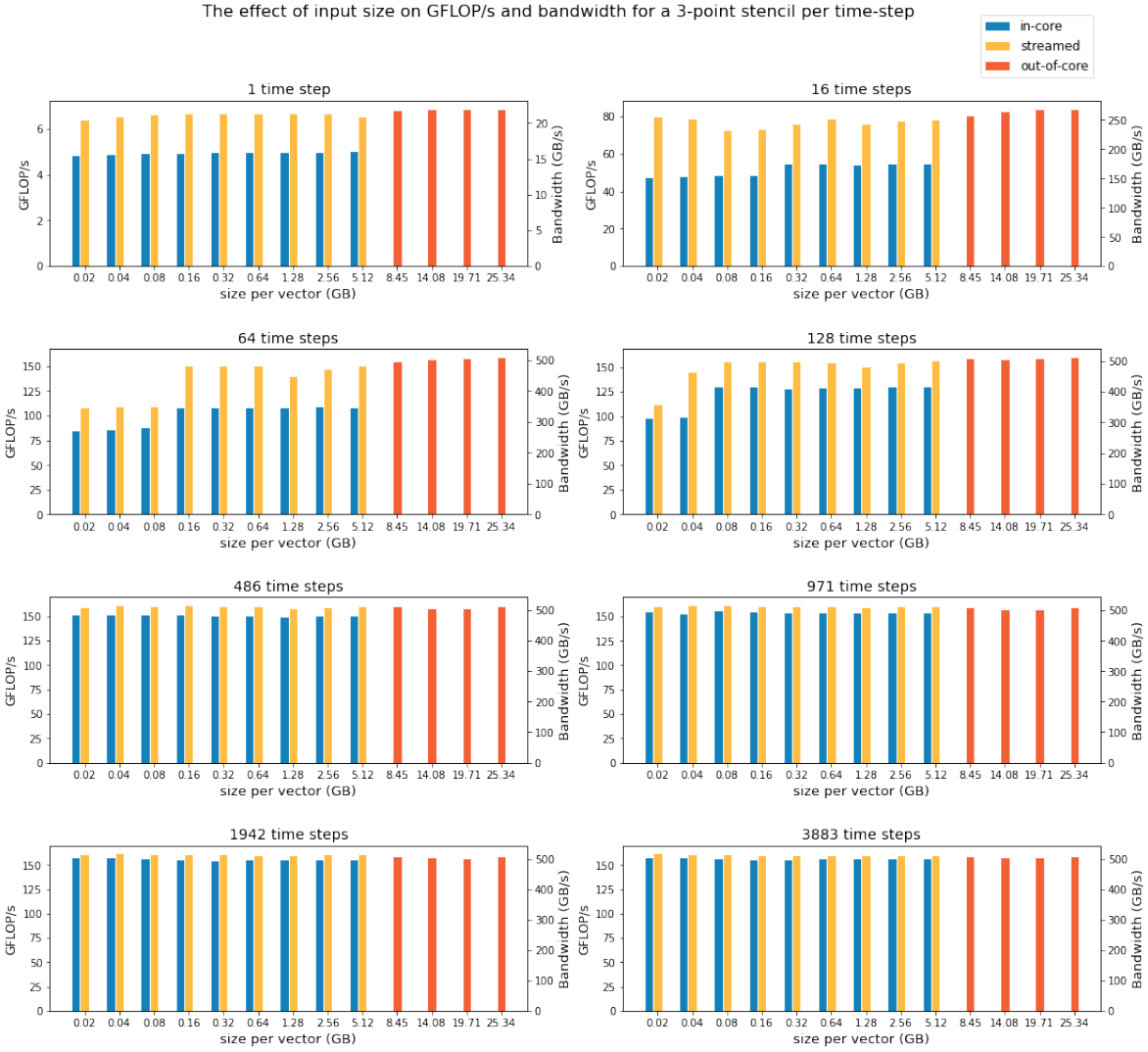

Figure 4.5: The effect of input size on the performance of an in-core, streamed and out-of-core 3-point stencil computation at all different time steps.

The effect of input size on the performance of the 3-point stencil computation in terms of GFLOP/s and bandwidth is shown in figure 4.5. The results are similar to the ones for map-like operations, except that the GFLOP/s are significantly higher, but this can be explained by the fact that the stencil does five FLOPs instead of only one. One thing that differs is that at 64

time steps, 20 MB, 40MB and 80MB perform worse than all other sizes. However, this seems to be the case for both in-core and in-core streamed.

Input size	Time steps	Streams	GFLOP/s (tot.)	Bandwidth (tot.)	GFLOP/s (kernel)	Bandwidth (kernel)
1280M	1	-	5.03	16.09	**165.79**	**530.55**
1280M	1	16	**6.72**	**21.53**	151	483.32
1280M	3883	-	159.01	508.83	**160.25**	**512.81**
1280M	3883	16	**160.49**	**513.58**	155.33	497.06

Table 4.4: The performance of two data points for both in-core and streamed 3-point stencil computations.

In table 4.4, two data points are shown for the 3-point stencil on a vector of 1280 million elements, where the in-core version is again indicated with a dash in the *Streams* column. These results were measured exactly the same as for map-like operations and also show similar improvements. The performance of sixteen streams is **1.33x** higher than that of the in-core version at one time step.

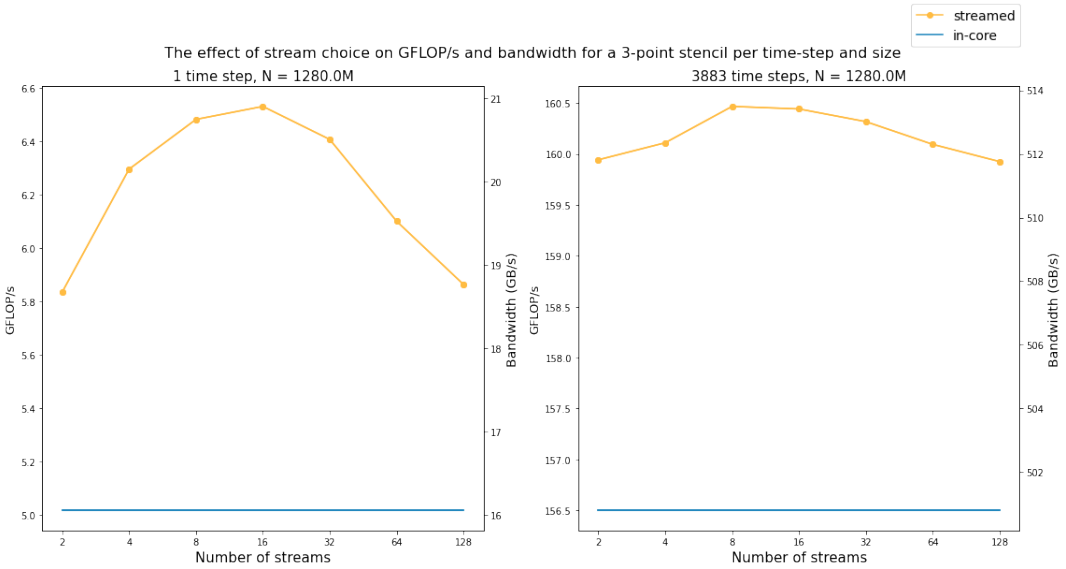

Figure 4.6: The performance of a 3-point stencil with 1280 million elements measured with different numbers of streams at 1 and 3883 time steps.

Furthermore, the performance of the kernel(s) alone is at one time step already near peak performance, which likely again indicates that the performance increase is due to the overlapping of computation and communication.

Similarly to map-like operations, the performance impact of the choice

of streams is measured for the 3-point stencil. In figure 4.6, the measurements for an input size of 1280 million at one time step and 3883 time steps is shown. Clearly, the choice matters more when the computation-to-communication ratio is low. At one time step, the performance difference between the extremes is roughly **11.49%** and at 3883 time steps **0.31%**. For the out-of-core 3-point stencil of 6336 million elements (figure 4.7) a similar observation can be made, where the difference between of the extremes at one time step and 3883 time steps is approximately **15.9%** and **0.43%**, respectively.

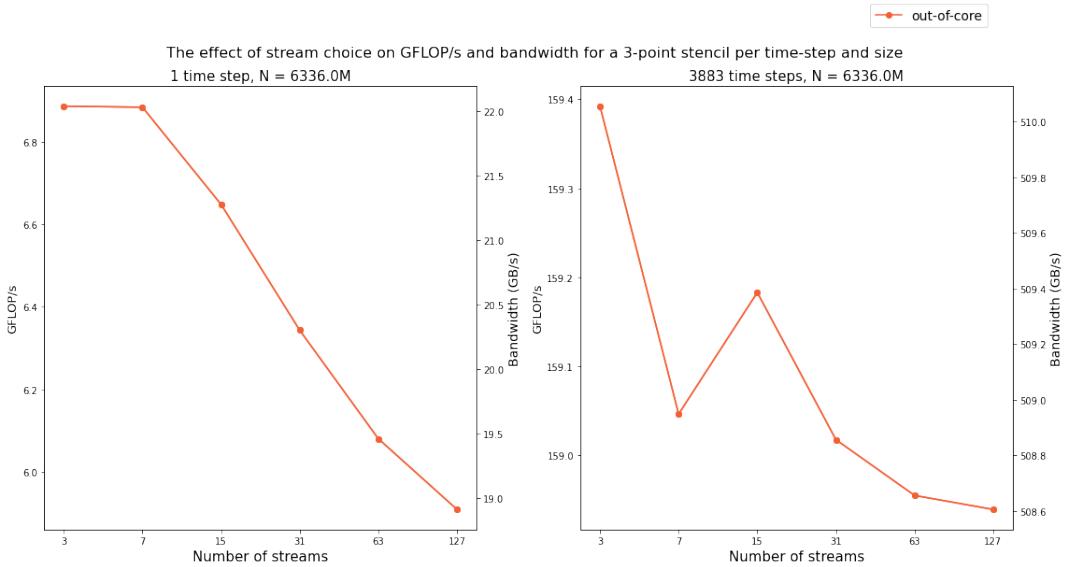

Figure 4.7: The performance of a 3-point stencil with 6338 million elements measured with different numbers of streams at 1 and 3883 time steps.

Chapter 5

Related Work

5.1 Asynchronous memory transfers

The differences in performance between CUDA's communication models in the context of code generation has been studied before. In [28], several extensions for the SaC compiler are introduced to, amongst other things, allow for code generation of asynchronous memory transfers. The authors find that the choice of communication model becomes more critical the lower the compute-to-communication ratio gets.

I have similar findings regarding the performance difference between non-streamed and streamed. Namely, the performance increase of streaming is larger, and the choice of number of streams also has more impact, when the compute-to-communication ratio is low.

5.2 Optimizing stencil computations for the GPU

Lots of research has been done on optimizing stencil computations on the GPU, for example, [23] and [24] look at stencil computations split between multiple GPUs. Multiple CPU threads are used to reduce the overhead from kernel launches and to further close the gaps between communication and computation.

In [23], multiple CUDA streams are used to overlap the computation and communication of halo regions between the neighbouring GPUs and the much larger computation of the non-halo region on the GPU. [24] builds on this by letting the CPU take part in the computations, which led to a reduced solution time on two different GPU clusters. However, the authors of [23] and [24] do not stream chunks of data on the individual GPUs and are not directly targeting general out-of-core execution.

5.3 Code generation for streaming arrays

Research on the automation of streaming arrays through the GPU exists. Both [3] and [14] look at extensions for the embedded array language Accelerate [2], which allow the programmer to explicitly choose to stream a limited set of operations through the GPU. [14] is most closely related to my paper, because it specifically considers so-called regular arrays, which are arrays that do not contain other arrays as elements. For example, scalars, vectors and matrices [14]. The authors use a similar chunk-based approach to scheduling the streams, but communication and computation are not overlapped.

Furthermore, their benchmarks [14] show a more significant impact from the chunk size on performance, for example, a dot product operation reaches optimal performance at approximately sixteen million elements per chunk. On the other hand, I find that eight streams give optimal performance for a vector addition of five million elements, which is a chunk size of approximately six-hundred-fifty thousand. This difference may be explained by the fact that I try to overlap as much communication and computation as possible, because the authors of [14] also suggest that, in general, there could be a significant improvement in performance by doing so.

5.4 Out-of-core stencil computations on the GPU

More recently, a few studies related to out-of-core stencil computations on the GPU have been published. For instance, [12] looks at stencil computations on input data larger than the device memory with a single GPU. The authors split the problem domain in several sub-domains and use ghost zones and temporal blocking combined with memory-saving optimizations and communication overlap to achieve higher performance.

The sub-domains are computed sequentially and ghost zones include some redundant computations, whereas I calculate several sub-domains concurrently by streaming them through the GPU and this is done without redundant computations. [12] also mentions that the performance falls when the problem sizes increases, which is not the case in this study and may be partially explained by the lack of redundant computations. However, do note that the authors of [12] considered 3D stencils, so a direct comparison cannot be made.

Moreover, [13] follows up on [12] by extending the out-of-core implementation to multiple GPUs. In [13], MPI is used to split the problem domain in sub-domains to be handled by separate GPUs, which each then utilise the approach in [12]. Therefore, this method does calculate several sub-domains concurrently, but only between the GPUs, not within an individual GPU. The authors mention in future work that it would be promising to intro-

duce their algorithm into domain specific languages, but do not yet provide rewriting schematics or code generation examples.

Finally, [5] introduces a library (HHRT) that functions as a wrapper around MPI and CUDA to assists the programmer with the non-trivial memory swapping process. On top of that, multiple MPI processes can now be assigned to one device to further reduce the cost of memory swaps. However, memory consumption is increased due to the usage of dedicated swap buffers on the host and CUDA's asynchronous communication model is not utilised.

Chapter 6

Conclusions

This paper looks at out-of-core code generation for map-like operations and stencil computations. In particular, I present blueprints for rewriting the aforementioned operations into their out-of-core version and analyse the performance with hand-coded implementations in C++.

Both rewriting schemes leverage CUDA streams to essentially establish concurrent kernels and concurrent memory transfers. Comparisons are made between in-core implementations and their streamed and out-of-core counterparts according to the suggested blueprints. The latter show an approximate **1.3-1.33x** increase in performance over the in-core version, independently of the input length of the data the algorithms operate on. Moreover, this improvement is maintained for approximately sixty-four time steps, after which the performance gain starts decreasing. Eventually, at 3883 time steps, the in-core version of both algorithmic classes has nearly caught up.

Based on the results, it seems that the performance increase is mostly, if not completely, achieved by overlapping the computation and communication. Since the performance gain starts decreasing after approximately sixty-four time steps, the potential improvement seems to be related to the compute-to-communication ratio of the involved operation.

Furthermore, the number of streams used to stream an algorithm, or execute it out-of-core, has a relevant impact on the performance at a lower number of time steps. On the other hand, this choice of number of streams starts mattering less when the compute-to-communication ratio decreases.

In summary, the overall drive of this research was to find out whether it possible to actually generate code that can process arrays that are larger than the memory that is available on the GPU. Surprisingly, I find that top performance can not only be achieved for both map-like operations, but also for stencil computations. Furthermore, this research shows that in some cases it is even desirable to stream these algorithms, regardless of whether in-core execution is possible. In particular, in scenarios where the overall computation time spent on the device is relatively small, improved

performance can be achieved.

6.1 Future work

In the future, it would be interesting to perform a more extensive performance analysis in order to establish heuristics. Furthermore, the presented rewriting schematics are not yet very generic, so it would be great to, for example, add support for k-dimensional n-point stencils. Finally, implementing the presented blueprints for out-of-core code generation in a compiler is out of scope for this research, so that could also be an interesting next step.

Bibliography

[1] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2012.

[2] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, page 3–14, New York, NY, USA, 2011. Association for Computing Machinery.

[3] Robert Clifton-Everest, Trevor L. McDonell, Manuel M. T. Chakravarty, and Gabriele Keller. Streaming irregular arrays. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell 2017, page 174–185, New York, NY, USA, 2017. Association for Computing Machinery.

[4] NVIDIA Corporation. *CUDA toolkit documentation*, 2021. `https://docs.nvidia.com/cuda/index.html`.

[5] Toshio Endo and Guanghao Jin. Software technologies coping with memory hierarchy of gpgpu clusters for stencil computations. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 132–139, 2014.

[6] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with cuda. *IEEE Micro*, 28(4):13–27, 2008.

[7] Simon L. Grimm and Joachim G. Stadel. THE GENGA CODE: gravitational encounters in N-body simulations with GPU acceleration. *The Astrophysical Journal*, 796(1):23, oct 2014.

[8] Tobias Grosser, Albert Cohen, Paul H. J. Kelly, J. Ramanujam, P. Sadayappan, and Sven Verdoolaege. Split tiling for gpus: Automatic par-

allelization using trapezoidal tiles. GPGPU-6, page 24–31, New York, NY, USA, 2013. Association for Computing Machinery.

[9] Jia Guo, Ganesh Bikshandi, Basilio B Fraguela, and David Padua. Writing productive stencil codes with overlapped tiling. *Concurrency and Computation: Practice and Experience*, 21(1):25–39, 2009.

[10] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. High performance stencil code generation with lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 100–112, New York, NY, USA, 2018. Association for Computing Machinery.

[11] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, page 311–320, New York, NY, USA, 2012. Association for Computing Machinery.

[12] Guanghao Jin, Toshio Endo, and Satoshi Matsuoka. A multi-level optimization method for stencil computation on the domain that is bigger than memory capacity of gpu. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 1080–1087, 2013.

[13] Guanghao Jin, Toshio Endo, and Satoshi Matsuoka. A parallel optimization method for stencil computation on the domain that is bigger than memory capacity of gpus. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8, 2013.

[14] Frederik M. Madsen, Robert Clifton-Everest, Manuel M. T. Chakravarty, and Gabriele Keller. Functional array streams. In *Proceedings of the 4th ACM SIGPLAN Workshop on Functional High-Performance Computing*, FHPC 2015, page 23–34, New York, NY, USA, 2015. Association for Computing Machinery.

[15] Jiayuan Meng and Kevin Skadron. A performance study for iterative stencil loops on gpus with ghost zone optimizations. *International Journal of Parallel Programming*, 39(1):115–142, 2011.

[16] Jarno Mielikainen, Bormin Huang, Hung-Lung Allen Huang, and Mitchell D. Goldberg. Gpu acceleration of the updated goddard shortwave radiation scheme in the weather research and forecasting (wrf) model. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 5(2):555–562, 2012.

[17] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2010.

[18] John Nickolls and William J. Dally. The gpu computing era. *IEEE Micro*, 30(2):56–69, 2010.

[19] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, Mahesh Ravishankar, Vinod Grover, Atanas Rountev, Louis-Noël Pouchet, and P. Sadayappan. Domain-specific optimization and generation of high-performance gpu code for stencil computations. *Proceedings of the IEEE*, 106(11):1902–1920, 2018.

[20] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[21] Sven-Bodo Scholz. Single assignment c: efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[22] Claudio Silva, Yi-Jen Chiang, Jihad El-Sana, and Peter Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. *IEEE Visualization Course Notes*, 2002.

[23] Mohammed Sourouri, Tor Gillberg, Scott B Baden, and Xing Cai. Effective multi-gpu communication using multiple cuda streams and threads. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 981–986, 2014.

[24] Mohammed Sourouri, Johannes Langguth, Filippo Spiga, Scott B. Baden, and Xing Cai. Cpu+gpu programming of stencil computations for resource-efficient use of gpu clusters. In *2015 IEEE 18th International Conference on Computational Science and Engineering*, pages 17–26, 2015.

[25] Guanglu Sun, Xuhang Li, Xiangyu Hou, and Fei Lang. Gpu-accelerated support vector machines for traffic classification. *International Journal of Performability Engineering*, 14(5):1088, 2018.

[26] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. SPAA '11, page 117–128, New York, NY, USA, 2011. Association for Computing Machinery.

[27] Sivan Toledo. *A Survey of Out-Of-Core Algorithms in Numerical Linear Algebra*, volume 50 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1999.

[28] Hans-Nikolai Vießmann and Sven-Bodo Scholz. Effective host-gpu memory management through code generation. In *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages*, IFL 2020, page 138–149, New York, NY, USA, 2020. Association for Computing Machinery.

Appendix A

Appendix

A.1 Even trapezoids

This part of the appendix contains all the code transformations for the calculation of even trapezoids in out-of-core 3-point stencil computations.

```
chunk_n = (n + 2*nS - 1) / 2*nS;
lchunk_n = n - (2*nS-1)*chunk_n;

for(i = 0; i < nS; i++){
    cudaMemcpyAsync(dev_in[i*2*chunk_n], in[i*2*chunk_n],
                    chunk_n + t, h2d, stream[i]);
    cudaMemcpyAsync(dev_out[i*2*chunk_n], out[i*2*chunk_n],
                    chunk_n + t, h2d, stream[i]);
}

for(i = 1; i <= t; i++){
  kernek<<< chunk_n+t-i, stream[0] >>>(dev_in, dev_out, 1,
                                       chunk_n+t-i);
  for(j = 1; j < nS; j++){
    kernek<<< chunk_n+t-2*i, stream[j] >>>(dev_in, dev_out,
                              j*2*chunk_n+i, (2*j+1)*chunk_n+t-i);
  }
  if( i != t )
    dev_in, dev_out = swap(dev_in, dev_out);
}

cudaMemcpyAsync(out[0], dev_out[0], chunkSize, d2h, stream[0]);

for(i = 1; i < nS; i++){
  cudaMemcpyAsync(out[i*2*chunk_n+t], dev_out[i*2*chunk_n+t],
                  chunkSize-t, d2h, stream[i]);
}
```

$\rightarrow\rightarrow\rightarrow$

```
r = (n + r_n - 1) / r_n;
chunk_n = (r_n + nS - 1) / nS;
lchunk_n = r_n - (nS-1) * chunk_n;

lr_n = n - (r-1) * r_n;
lr_chunk_n = (lr_n + nS - 1) / nS;
lr_lchunk_n = lr_n - (nS-1) * lr_chunk_n;

// round 0, see figure A.2
// rounds 1 to n-2, see figure A.5
// round n-1, see figure A.8
```

Figure A.1: A rewriting schematic that transforms the in-core streaming of even trapezoids, to out-of-core.

```
1  cudaMemcpyAsync(dev_in[0], in[0], chunk_n+t, h2d, stream[0]);
2  cudaMemcpyAsync(dev_out[0], out[0], chunk_n+t, h2d, stream[0]);
3
4  for(j = 1; j < nS-1; j++){
5    if(0 == j % 2){
6      cudaMemcpyAsync(dev_in[j*(chunk_n+t+2)], in[j*chunk_n],
7                      chunk_n+t, h2d, stream[j]);
8      cudaMemcpyAsync(dev_out[j*(chunk_n+t+2)], out[j*chunk_n],
9                      chunk_n+t, h2d, stream[j]);
10   }
11 }
12 if(0 == ((nS-1) % 2)){
13   cudaMemcpyAsync(dev_in[(nS-1)*(chunk_n+t+2)],
14                   in[(nS-1)*chunk_n], lchunk_n+t, h2d,
15                   stream[nS-1]);
16   cudaMemcpyAsync(dev_out[(nS-1)*(chunk_n+t+2)],
17                   out[(nS-1)*chunk_n], lchunk_n+t, h2d,
18                   stream[nS-1]);
19 }
20 // kernels, see figure A.3
21 // d2h, see figure A.4
```

Figure A.2: The rewritten code for the h2d memory transfers of round 0 for calculating even trapezoids out-of-core.

```
1  for(k = 1; k <= t; k++){
2    kernel<<< chunk_n+t-k, stream[0] >>>(dev_in, dev_out, 1,
3                                         chunk_n+t-k);
4
5    for(j = 1; j < nS-1; j++){
6      if(0 == j % 2)
7        kernel<<< chunk_n+t-2*k, stream[j] >>>(dev_in, dev_out,
8                                               j*(chunk_n+t+2)+k,
9                                           (j+1)*(chunk_n+t+2)-2-k);
10     }
11
12   if(0 == ((nS-1) % 2)){
13     kernel<<< lchunk_n+t-2*k, stream[nS-1] >>>(dev_in, dev_out,
14                                                (nS-1)*(chunk_n+t+2)+k,
15                                  (nS-1)*(chunk_n+t+2)+lchunk_n+t-k);
16   }
17   if( k < t)
18     swap(dev_in, dev_out);
19 }
```

Figure A.3: The rewritten code for the kernel calls in round 0 for calculating even trapezoids out-of-core.

```
1  cudaMemcpyAsync(in[chunk_n-1], dev_in[chunk_n-1], t, d2h,
2                  stream[0]);
3  cudaMemcpyAsync(out[0], dev_out[0], (chunk_n+t), d2h, stream[0])
       ;
4
5  for(j = 1; j < nS-1; j++){
6    if(0 == j % 2){
7      cudaMemcpyAsync(in[j*chunk_n+1], dev_in[j*(chunk_n+t+2)+1],
8                      t, d2h, stream[j]);
9      cudaMemcpyAsync(in[(j+1)*chunk_n-1],
10                     dev_in[(j+1)*(chunk_n+t+2)-t-3], t, d2h,
11                     stream[j]);
12
13     cudaMemcpyAsync(out[j*chunk_n],
14                     dev_out[j*(chunk_n+t+2)], chunk_n+t, d2h,
15                     stream[j]);
16   }
17 }
18
19 if(0 == ((nS-1) % 2)){
20   cudaMemcpyAsync(in[(nS-1)*chunk_n+1],
21                   dev_in[(nS-1)*(chunk_n+t+2)+1], t, d2h,
22                   stream[nS-1]);
23   cudaMemcpyAsync(in[(nS-1)*chunk_n+lchunk_n-1],
24                   dev_in[(nS-1)*(chunk_n+t+2)+lchunk_n-1], t,
25                   d2h, stream[nS-1]);
26
27   cudaMemcpyAsync(out[(nS-1)*chunk_n],
28                   dev_out[(nS-1)*(chunk_n+t+2)], lchunk_n+t,
29                   d2h, stream[nS-1]);
30 }
```

Figure A.4: The rewritten code for the d2h memory transfers in round 0 for calculating even trapezoids out-of-core.

```
1  for (i = 1; i < r - 1; i++){
2    for(j = 0; j < nS-1; j++){
3      if(i % 2 == j % 2){
4        cudaMemcpyAsync(dev_in[j*(chunk_n+t+2)],
5                        in[i*r_n + j*chunk_n], chunk_n+t,
6                        h2d, stream[j]);
7        cudaMemcpyAsync(dev_out[j*(chunk_n+t+2)],
8                        out[i*r_n + j*chunk_n], chunk_n+t,
9                        h2d, stream[j]);
10     }
11   }
12   if(i % 2 == ((nS-1) % 2)){
13     cudaMemcpyAsync(dev_in[(nS-1)*(chunk_n+t+2)],
14                     in[i*r_n+(nS-1)*chunk_n], lchunk_n+t,
15                     h2d, stream[nS-1]);
16     cudaMemcpyAsync(dev_out[(nS-1)*(chunk_n+t+2)],
17                     out[i*r_n+(nS-1)*chunk_n], lchunk_n+t,
18                     h2d, stream[nS-1]);
19   }
20
21   // kernels, see figure A.6
22   // d2h, see figure A.7
23 }
```

Figure A.5: The rewritten code for the h2d memory transfers of rounds 1 to n-2 for calculating even trapezoids out-of-core.

```
1  for(k = 1; k <= t; k++){
2    for(j = 0; j < nS-1; j++){
3      if(i % 2 == j % 2){
4        kernel<<< chunk_n+t-2*k, stream[j] >>>(dev_in, dev_out,
5                                               j*(chunk_n+t+2)+k,
6                                          (j+1)*(chunk_n+t+2)-2-k);
7      }
8    }
9    if(i % 2 == ((nS-1) % 2)){
10     kernel<<< lchunk_n+t-2*k, stream[nS-1] >>>(dev_in, dev_out,
11                                                (nS-1)*(chunk_n+t+2)+k,
12                                  (nS-1)*(chunk_n+t+2)+lchunk_n+t-k);
13   }
14
15   if( k < t)
16     swap(dev_in, dev_out);
17 }
```

Figure A.6: The rewritten code for the kernel calls in rounds 1 to n-2 for calculating even trapezoids out-of-core.

```
1  for(j = 0; j < nS-1; j++){
2    if(i % 2 == j % 2){
3      cudaMemcpyAsync(in[i*r_n+j*chunk_n+1],
4                      dev_in[j*(chunk_n+t+2)+1], t, d2h,
5                      stream[j]);
6      cudaMemcpyAsync(in[i*r_n+(j+1)*chunk_n-1],
7                      dev_in[(j+1)*(chunk_n+t+2)-t-3], t, d2h,
8                      stream[j]);
9
10     cudaMemcpyAsync(out[i*rounds_n+j*chunk_n],
11                     dev_out[j*(chunk_n+t+2)], chunk_n+t, d2h,
12                     stream[j]);
13   }
14 }
15 if(i % 2 == ((nS-1) % 2)){
16   cudaMemcpyAsync(in[i*r_n+(nS-1)*chunk_n+1],
17                   dev_in[(nS-1)*(chunk_n+t+2)+1], t, d2h,
18                   stream[nS-1]);
19   cudaMemcpyAsync(in[i*r_n+(nS-1)*chunk_n+lchunk_n-1],
20                   dev_in[(nS-1)*(chunk_n+t+2)+lchunk_n-1], t,
21                   d2h, stream[nS-1]);
22
23   cudaMemcpyAsync(out[i*r_n + (nS-1)*chunk_n],
24                   dev_out[(nS-1)*(chunk_n+t+2)], lchunk_n+t,
25                   d2h, stream[nS-1]);
26 }
```

Figure A.7: The rewritten code for the d2h memory transfers in rounds 1 to n-2 for calculating even trapezoids out-of-core.

```
1  for(j = 0; j < nS-1; j++){
2    if((r-1) % 2 == j % 2){
3      cudaMemcpyAsync(dev_in[j*(chunk_n+t+2)],
4                      in[(r-1)*r_n + j*lr_chunk_n],
5                      lr_chunk_n+t, h2d, stream[j]);
6      cudaMemcpyAsync(dev_out[j*(chunk_n+t+2)],
7                      out[(r-1)*r_elems + j*lr_chunk_n],
8                      lr_chunk_n+t, h2d, stream[j]);
9    }
10 }
11 if((r-1) % 2 == ((nS-1) % 2)){
12   cudaMemcpyAsync(dev_in[(nS-1)*(chunk_n+t+2)],
13                   in[(r-1)*r_n+(nS-1)*lr_chunk_n],
14                   lr_lchunk_n, h2d, stream[nS-1]);
15   cudaMemcpyAsync(dev_out[(nS-1)*(chunk_n+t+2)],
16                   out[(r-1)*r_n+(nS-1)*lr_chunk_n],
17                   lr_lchunk_n, h2d, stream[nS-1]);
18 }
19
20 // kernels, see figure A.9
21 // d2h, see figure A.10
```

Figure A.8: The rewritten code for the h2d memory transfers of round n-1 for calculating even trapezoids out-of-core.

```
1  for(k = 1; k <= t; k++){
2    for(j = 0; j < nS-1; j++){
3      if((r-1) % 2 == j % 2){
4        kernel<<< lr_chunk_n+t-2*k, stream[j] >>>(dev_in, dev_out,
5                                                  j*(chunk_n+t+2)+k,
6                                        j*(chunk_n+t+2)+lr_chunk_n+t-k)
                                             ;
7      }
8    }
9    if((r-1) % 2 == ((nS-1) % 2)){
10     kernel<<<lr_lchunk_n-k, stream[nS-1] >>>(dev_in, dev_out,
11                                                (nS-1)*(chunk_n+t+2)+k
                                                        ,
12                                     (nS-1)*(chunk_n+t+2)+lr_lchunk_n-1)
                                           ;
13   }
14
15   if( k < t)
16     swap(dev_in, dev_out);
17 }
```

Figure A.9: The rewritten code for the kernel calls in round n-1 for calculating even trapezoids out-of-core.

```
1  for(j = 0; j < nS-1; j++){
2    if((r-1) % 2 == j % 2){
3      cudaMemcpyAsync(in[(r-1)*r_n+j*lr_chunk_n+1],
4                      dev_in[j*(chunk_n+t+2)+1], t, d2h,
5                      stream[j]);
6      cudaMemcpyAsync(in[(r-1)*r_n+(j+1)*lr_chunk_n-1],
7                      dev_in[j*(chunk_n+t+2)+lr_chunk_n-1], t,
8                      d2h, stream[j]);
9
10     cudaMemcpyAsync(out[(r-1)*r_n+j*lr_chunk_n],
11                     dev_out[j*(chunk_n+t+2)], lr_chunk_n+t,
12                     d2h, stream[j]);
13   }
14 }
15 if((r-1) % 2 == ((nS-1) % 2)){
16   cudaMemcpyAsync(in[(r-1)*r_n+(nS-1)*lr_chunk_n+1],
17                   dev_in[(nS-1)*(chunk_n+t+2)+1], t, d2h,
18                   stream[nS-1]);
19
20   cudaMemcpyAsync(out[(r-1)*r_n + (nS-1)*lr_chunk_n],
21                   dev_out[(nS-1)*(chunk_n+t+2)], lr_lchunk_n-1,
22                   d2h, stream[nS-1]);
23 }
```

Figure A.10: The rewritten code for the d2h memory transfers in round n-1 for calculating even trapezoids out-of-core.

A.2 Odd trapezoids

This part of the appendix contains all the code transformations for the calculation of odd trapezoids in out-of-core 3-point stencil computations.

```
1  for(i = 0; i < nS-1; i++){
2    cudaMemcpyAsync(dev_in[(2*i+1)*chunk_n], in[(2*i+1)*chunk_n],
3                    chunk_n-t, h2d, stream[i]);
4    cudaMemcpyAsync(dev_out[(2*i+1)*chunk_n], out[(2*i+1)*chunk_n
       ],
5                    chunk_n-t, h2d, stream[i]);
6  }
7  cudaMemcpyAsync(dev_in[(2*(nS-1)+1)*chunk_n],
8                  in[(2*(nS-1)+1)*chunk_n], lchunk_n-t, h2d,
9                  stream[nS-1]);
10 cudaMemcpyAsync(dev_out[(2*(nS-1)+1)*chunk_n],
11                 out[(2*(nS-1)+1)*chunk_n], lchunk_n-t, h2d,
12                 stream[nS-1]);
13
14 for(i = 1; i <= t; i++){
15     for(j = 0; j < nS-1; j++){
16         kernel<<< chunk_n-t+2*i, stream[j] >>>(dev_in, dev_out,
17                                                (2*j+1)*chunk_n+t-i
                                                     ,
18                                                (2*j+2)*chunk_n+i);
19     }
20     kernel <<< lchunk_n-t+i, stream[nS-1] >> > (dev_in, dev_out,
21                                                 (2*nS-1)*chunk_n+t-i,n-1)
                                                      ;
22     if( i != t )
23         dev_in, dev_out = swap(dev_in, dev_out);
24 }
25
26 for(i = 0; i < nS-1; i++){
27     cudaMemcpyAsync(out[(2*i+1)*chunk_n],
28                     dev_out[(2*i+1)*chunk_n], chunk_n+t, d2h,
29                     stream[i]);
30 }
31 cudaMemcpyAsync(out[(2*(nS-1)+1)*chunk_n],
32                 dev_out[(2*(nS-1)+1)*chunk_n], lchunk_n, d2h,
33                 stream[nS-1]);
```

$\rightarrow\rightarrow\rightarrow$

```
1  // rounds 0 to n-2, see figure A.12
2  // round n-1, see figure A.15
```

Figure A.11: A rewriting schematic that transforms the in-core streaming of odd trapezoids.

```
1  for (i = 0; i < r - 1; i++){
2    for(j = 0; j < nS-1; j++){
3      if(i % 2 != j % 2){
4        cudaMemcpyAsync(dev_in[j*(chunk_n+t+2)],
5                        in[i*r_n+j*chunk_n-1], chunk_n+t+2,
6                        h2d, stream[j]);
7        cudaMemcpyAsync(dev_out[j*(chunk_n+t+2)],
8                        out[i*r_n+j*chunk_n-1], chunk_n+t+2,
9                        h2d, stream[j]);
10     }
11   }
12   if(i % 2 != ((nS-1) % 2)){
13     cudaMemcpyAsync(dev_in[(nS-1)*(chunk_n+t+2)],
14                     in[i*r_n+(nS-1)*chunk_n-1], lchunk_n+t+2,
15                     h2d, stream[nS-1]);
16     cudaMemcpyAsync(dev_out[(nS-1)*(chunk_n+t+2)],
17                     out[i*r_n+(nS-1)*chunk_n-1], lchunk_n+t+2,
18                     h2d, stream[nS-1]);
19   }
20
21   if(t % 2 == 0)
22       swap(dev_in, dev_out)
23   // kernels, see figure A.13
24   // d2h, see figure A.14
25 }
```

Figure A.12: The rewritten code for the h2d memory transfers of rounds 0 to n-2 for calculating odd trapezoids out-of-core.

```
1  for(k = 1; k <= t; k++){
2    for(j = 0; j < nS-1; j++){
3      if(i % 2 != j % 2){
4        kernel<<< chunk_n-t+2*k, stream[j] >>>(dev_in, dev_out,
5                                               j*(chunk_n+t+2)+t-k,
6                                          (j+1)*(chunk_n+t+2)-t-1+k);
7      }
8    }
9    if(i % 2 != ((nS-1) % 2)){
10     kernel<<< lchunk_n-t+2*k, stream[nS-1] >>>(dev_in, dev_out,
11                                                (nS-1)*(chunk_n+t+2)+t-k,
12                                    (nS-1)*(chunk_n+t+2)+1+lchunk_n+k);
13   }
14
15   if( k < t)
16     swap(dev_in, dev_out);
17 }
```

Figure A.13: The rewritten code for the kernel calls in rounds 0 to n-2 for calculating odd trapezoids out-of-core.

```
for(j = 0; j < nS-1; j++){
  if(i % 2 != j % 2){
    cudaMemcpyAsync(out[i*r_n+j*chunk_n],
                    dev_out[j*(chunk_n+t+2)+1], chunk_n+t, d2h,
                    stream[j]);
  }
}
if(i % 2 != ((nS-1) % 2)){
  cudaMemcpyAsync(out[i*r_n + (nS-1)*chunk_n],
                  dev_out[(nS-1)*(chunk_n+t+2)+1], lchunk_n+t,
                  d2h, stream[nS-1]);
}
```

Figure A.14: The rewritten code for the d2h memory transfers in rounds 0 to n-2 for calculating odd trapezoids out-of-core.

```
for(j = 0; j < nS-1; j++){
  if((r-1) % 2 != j % 2){
    cudaMemcpyAsync(dev_in[j*(chunk_n+t+2)],
                    in[(r-1)*r_n+j*lr_chunk_n-1],
                    lr_chunk_n+t+2, h2d, stream[j]);
    cudaMemcpyAsync(dev_out[j*(chunk_n+t+2)],
                    out[(r-1)*r_n+j*lr_chunk_n-1],
                    lr_chunk_n+t+2, h2d, stream[j]);
  }
}
if((r-1) % 2 != ((nS-1) % 2)){
  cudaMemcpyAsync(dev_in[(nS-1)*(chunk_n+t+2)],
                  in[(r-1)*r_n+(nS-1)*lr_chunk_n-1],
                  lr_lchunk_n+1, h2d, stream[nS-1]);
  cudaMemcpyAsync(dev_out[(nS-1)*(chunk_n+t+2)],
                  out[(r-1)*r_n+(nS-1)*lr_chunk_n-1],
                  lr_lchunk_n+1, h2d, stream[nS-1]);
}

if(t % 2 == 0)
    swap(dev_in, dev_out)
// kernels, see figure A.16
// d2h, see figure A.17
```

Figure A.15: The rewritten code for the h2d memory transfers of round n-1 for calculating odd trapezoids out-of-core.

```
1  for(k = 1; k <= t; k++){
2    for(j = 0; j < nS-1; j++){
3      if((r-1) % 2 != j % 2){
4        kernel<<< lr_chunk_n-t+2*k, stream[j] >>>(dev_in, dev_out,
5                                                  j*(chunk_n+t+2)+t+1-k,
6                                                j*(chunk_n+t+2)+1+lr_chunk_n+
                                                     k);
7      }
8    }
9    if((rounds-1) % 2 != ((nS-1) % 2)){
10     kernel<<<lr_lchunk_n-t+k, stream[nS-1]>>>(dev_in, dev_out,
11                                               (nS-1)*(chunk_n+t+2)+t+1-k,
12                                  (nS-1)*(chunk_n+t+2)+lr_lchunk_n);
13   }
14
15   if( k < t)
16     swap(dev_in, dev_out);
17 }
```

Figure A.16: The rewritten code for the kernel calls in round n-1 for calculating even trapezoids out-of-core.

```
1  for(j = 0; j < nS-1; j++){
2    if((r-1) % 2 != j % 2){
3      cudaMemcpyAsync(out[(r-1)*r_n+j*lr_chunk_n],
4                      dev_out[j*(chunk_n+t+2)+1], lr_chunk_n+t,
5                      d2h, stream[j]);
6    }
7  }
8  if((r-1) % 2 != ((nS-1) % 2)){
9    cudaMemcpyAsync(out[(r-1)*r_n + (nS-1)*lr_chunk_n],
10                   dev_out[(nS-1)*(chunk_n+t+2)+1], lr_lchunk_n,
11                   d2h, stream[nS-1]);
12 }
```

Figure A.17: The rewritten code for the d2h memory transfers in round n-1 for calculating even trapezoids out-of-core.

A.3 Code for performance metrics

```
def calculate_gflops(self, N, flops, t, ms):
    """
    inputs:
        N: nr of elements
        flops: floating point operations
        t: time steps
        ms: runtime in milliseconds
    return:
        Returns GFLOP/s
    """
    return t*flops*N/((ms/1e3)*1e9)
```

Figure A.18: The python function used to calculate GFLOP/s of the experiment measurements.

```
def calculate_eb(self, N, r, w, b, t, ms):
    """
    inputs:
        N: nr of elements
        r: reads
        w: writes
        b: byte size
        t: timesteps
        ms: runtime in milliseconds
    return:
        Returns the effective bandwidth in GB/s
    """
    return t*(r+w)*b*N/((ms/1e3)*1e9)
```

Figure A.19: The python function used to calculate (effective) bandwidth of the experiment measurements.