On Generating Out-Of-Core GPU Code for Multi-Dimensional Array Operations

Patrick van Beurden
patrick.vanbeurden@ru.nl
Radboud University
Nijmegen, Netherlands

Sven-Bodo Scholz
SvenBodo.Scholz@ru.nl
Radboud University
Nijmegen, Netherlands

ABSTRACT

This paper presents the first results of our experiments for generating CUDA code that streams array operations over the elements of its array arguments from high-level specifications. We look at two classes of memory-bound array operations: map-like operations and iterative stencil computations. We investigate code patterns that stream the arguments of these operations from the host through the GPU and back taking the iterative nature of our experiments into account. We show that this setup does not only enable computations on arrays that are so big that they do not fit into the device memory of a single GPU (hence "out-of-core"), but we also demonstrate that the proposed streaming code outperforms non-streaming code versions even for smaller array sizes. For both application patterns, we observe memory throughputs that are beyond 80% of the hardware capability, irrespective of the problem sizes.

KEYWORDS

code generation, GPUs, CUDA, streaming, memory management, out-of-core computation

ACM Reference Format:
Patrick van Beurden and Sven-Bodo Scholz. 2022. On Generating Out-Of-Core GPU Code for Multi-Dimensional Array Operations. In *Symposium on Implementation and Application of Functional Languages (IFL 2022), August 31–September 02, 2022, Copenhagen, Denmark.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3587216.3587223

1 INTRODUCTION

Functional array programming languages such as Futhark [8], Accelerate [1], Lift [22], or SaC [17] are capable of generating high-performance codes from abstract functional specifications. In their context, it has been shown that code generation from functional descriptions has several advantages over explicit low-level programming:

Application programmers from domains such as financial analytics [15], computational science [16, 26], image processing [27], or machine-learning [28] all benefit massively in term of programming productivity. Without any modifications, the same programs can be compiled to target various different architectures, ranging from shared-memory multi-core systems [4] over distributed memory systems [13] to GPU-accelerated devices [1, 6, 8, 22].

In particular GPU-accelerated devices have recently inspired a lot of work as they are ideally suited for inherently memory-bound algorithms which are common-place in data-intensive array applications. Even in the context of GPUs alone, the benefits of code generation have been widely demonstrated. Non-trivial code adaptations that adjust and optimise the code for any given particular device have shown massive gains over individual, fixed-code versions [6, 23]. These transformations mainly focus on algorithmic restructurings, trying to adjust which intermediate structures to materialise in memory or how to map a given sequence of computations onto thread-spaces. Other work has looked at memory orchestrations, trying to make device-specific and application-specific choices to optimise the overall throughput [25].

However, to our knowledge, almost all of this work assumes that all data of an array computation resides on the GPU device once a kernel starts to execute. In case the size of argument arrays exceeds the memory capability of the device, fully automated code generation for the GPU so far has not been possible. To overcome this size restriction, index-based specifications need to be re-formulated in a streaming form and the streaming of data and GPU kernel invocations needs to be orchestrated accordingly. While map-like computations, in principle, can be easily segmented and transformed into a streaming form, more sophisticated codes that require non-local element accesses are more challenging.

This paper tackles the challenge of generating streaming codes from array operations on arrays that cannot reside in their entirety in the device memory. We identify two code patterns which are (i) found in many practical applications and (ii) can be transformed into streaming codes that loop over data transfers and kernel invocations. Our contributions are:

- The identification of two algorithmic patterns that can be transformed into streaming code: map-like computations and iterative stencil computations;
- CUDA code templates for streaming such algorithms that are parameterised by the number of streaming chunks. These allow the memory requirements of any given algorithm to be adjusted to a suitable streaming granularity;
- A systematic performance evaluation demonstrating the effectiveness of the proposed streaming codes.

In Section 2, we give a brief introduction to SaC and its code generator for GPUs as we use these as starting point for our investigations. Section 3 provides the necessary background on streaming support in CUDA. In Section 4, we discuss the two code patterns that we investigate. Our suggested streaming templates for implementing these patterns are presented in Section 5, followed by a

IFL 2022, August 31–September 02, 2022, Copenhagen, Denmark

ACM ISBN 978-1-4503-9831-2/22/08.
https://doi.org/10.1145/3587216.3587223

performance evaluation in Section 6. Section 7 puts our results into perspective with related work before Section 8 concludes.

2 SAC

2.1 The language

SaC is an array language with N-dimensional arrays at its core. Its design is based around a central data-parallel language construct called *tensor comprehension*.[1] It specifies a result array in terms of mappings from indices to expressions for arbitrarily partitioned index spaces. For example, the addition of two arrays `a` and `b` can be specified as

```
1 { iv -> a[iv] + b[iv] }
```

Here, the shape (sizes in all dimensions) of the result is implicitly determined by the shapes of `a` and `b`.

The rotation of a vector `v` by `n >= 0` elements can be expressed by a slightly more elaborate tensor comprehension:

```
1 { [i] -> v[shape(v)-n+i] | [i] < [n];
2   [i] -> v[i-n] | [n] <= [i] < shape(v) }
```

Line 1 here defines how to compute the first `n` elements, while line 2 defines all remaining elements of the result.

One of the key features of SaC is that almost all generic array operations can be defined in terms of tensor comprehensions. In fact, the entire standard library of SaC is defined through these, providing operations such as *take*, *drop*, *rotate*, etc.

As a consequence of this design, program optimisation as well as code generation for various architectures can focus on tensor comprehensions as the key language construct. For more details on these aspects see [17]. Within the context of this paper, this means we can focus on code patterns that feature one tensor comprehension but may contain further tensor comprehensions at the inside of it. For more details on SaC see [18].

2.2 Code generation for CUDA

CUDA code generation from SaC is based on an identification of suitable tensor comprehensions which are translated into a sequence of host-to-device memory transfers for all arguments, a kernel call for each part of the partition of the index space and a subsequent device-to-host transfer for the result of the tensor comprehension. Figure 1 shows this translation for the simple array addition example from above. The CUDA function `cudaMemcpy` is a synchronous

```
1 a = { iv -> a[iv] + b[iv] };
```

⇓

(a) SaC code

```
1 cudaMemcpy (dev_a, a, n, h2d);
2 cudaMemcpy (dev_b, b, n, h2d);
3 dev_a = kernel<<< n >>> (dev_a, dev_b, n);
4 cudaMemcpy (a, dev_a, n, d2h);
```

(b) Generated CUDA code

Figure 1: Code generation for tensor comprehensions.

memory transfer between host and device: once initiated, both, the host machine and the GPU device, wait until the transfer has been completed before continuing their executions. The first argument denotes the destination, the second the source, and the third argument defined the number of elements to be transferred. The direction of the transfer is indicated by the last argument to that function, `h2d` indicating a transfer from the host to the device and `d2h` indicating the opposite direction. The notion `kernel<<< n >>>` here indicates a kernel call using `n` independent threads, all of which obtain the arguments provided in brackets.

Notice here that we simplified the presented code, leaving out several details such as error handling, the kernel code itself as well as the exact setup of the kernel parameters. For more detailed descriptions see [5, 6].

After the basic code generation, a set of optimisations tries to eliminate memory transfers as much as possible. These optimisations identify and eliminate redundant and complementary transfers between host and device. This does not only include optimisations of static sequences of tensor comprehensions but also optimisations between subsequent instances of sequential loops that contain a GPU-suitable tensor comprehension.

```
1 for (i=0; i<k; i++) {
2     a = { iv -> a[iv] + b[iv] };
3 }
```

(a) SaC code

⇓

```
1 cudaMemcpy (dev_a, a, n, h2d);
2 cudaMemcpy (dev_b, b, n, h2d);
3 for (i=0; i<k; i++) {
4     dev_a = kernel<<< n >>> (dev_a, dev_b, n);
5 }
6 cudaMemcpy (a, dev_a, n, d2h);
```

(b) Generated and optimised CUDA code

Figure 2: Effect of memory transfer optimisations.

An example of the effect of these optimisations is shown in Figure 2. Here we look at an embedding of the simple addition into a sequential for loop. As we can see, all memory transfers have been lifted out of the sequential loop avoiding the need for any memory transfer between host and device during the loop execution. For more details on the memory transfer optimisations see [5, 6].

3 STREAMING IN CUDA

GPUs do not only provide thousands of compute cores but they also come with memory-transfer facilities that can operate independently of the cores. Despite this high level of possible parallelism, most GPU applications are rather synchronous as they follow a sequential model of host-GPU interaction similar to the one described in the previous section: They start with a synchronous transfer from host to device, meaning that the host waits until all data has been transferred before proceeding. Thereafter, the actual GPU computation (kernel call) is being triggered before the host initiates a synchronous transfer of results back to the host memory.

The advantage of this approach is that race conditions between host and GPU, as well as between memory transfers and kernel

[1] Internally, all tensor comprehensions are mapped to a more explicit data-parallel construct named *with-loop* (see [19] for details).

executions can be easily avoided. Even though such synchronous transfers typically are about a factor two slower than non-cached host memory reads from the CPU and about a factor 20 slower than non-cached device memory reads on the GPU, their impact on the overall run-time in most cases can be tolerated due to memory-management optimisations such as those shown in Figure 2, which move the transfers out of performance-critical parts of applications.

However, as we are looking into application situations where we can no longer fit all data on the GPU, we can no longer avoid memory transfers within run-time critical sections. Consequently, we can no longer afford to sacrifice opportunities for overlapping communications and computations.

To enable such parallelism without sacrificing all timing guarantees, CUDA supports the notion of *streams*. Each stream takes an arbitrary sequence of memory transfers and kernel executions and it guarantees a sequential execution within the stream. The default programming model constitutes the special case where there is just a single stream, the *default* (or *null*) stream.

By creating multiple such streams, the programmer can create opportunities for overlapping operations from one stream with operations from any other stream. That way, ample opportunity for overlapping communication and computation can be created without sacrificing a guaranteed order within each stream. The key operations for utilising streams are `cudaStreamCreate` for creating new streams, `cudaMemcpyAsync` for triggering an asynchronous memory transfer in the specified stream, a stream argument in CUDA kernel calls, and `cudaDeviceSynchronize` as a global synchronisation means. Their slightly simplified signatures look like this:

```
\\ creating a stream:
cudaStreamCreate (stream)

\\ asynchronous memory transfer:
cudaMemcpyAsync (mem_to, mem_from, amount, dir, stream);

\\ kernel execution:
kernel<<<num_threads, stream>>>(...);

\\ sync across all streams:
cudaDeviceSynchronize();
```

Figure 3 shows how these facilities can be used to implement our addition example using two independent streams on the GPU. For simplicity, we assume that we are dealing with a total of `2048` array elements within both arrays. First, in lines 1-2, we create two streams. Thereafter, we have two blocks of code. The first block (lines 4-7) inserts four operations into `stream1`, the second block (lines 9-15) inserts four operations into `stream2`. While the first block deals with the first `1024` elements, the second block deals with the remaining `1024` elements. Both blocks consist of two transfers from host to device, a kernel execution, and a transfer of the result data back to the host. The transfers only relate to the needed portions of the data, and the kernel calls obtain a fourth parameter indicating the offset of the data to be operated upon. Finally, we have a device synchronisation that ensures that both streams have completed before continuing with the host code.

```
1  cudaStreamCreate (stream1);
2  cudaStreamCreate (stream2);
3
4  cudaMemcpyAsync (dev_a[0], host_a[0], 1024, h2d, stream1);
5  cudaMemcpyAsync (dev_b[0], host_b[0], 1024, h2d, stream1);
6  kernel<<< 1024, stream1>>> (dev_a, dev_b, 1024, 0);
7  cudaMemcpyAsync (host_a[0], dev_a[0], 1024, d2h, stream1);
8
9  cudaMemcpyAsync (dev_a[1024], host_a[1024],
10                  1024, h2d, stream2);
11 cudaMemcpyAsync (dev_b[1024], host_b[1024],
12                  1024, h2d, stream2);
13 kernel<<< 1024, stream2>>>(dev_a, dev_b, 1024, 1024);
14 cudaMemcpyAsync (host_a[1024], dev_a[1024],
15                  1024, d2h, stream2);
16
17 cudaDeviceSynchronize();
```

Figure 3: Element-wise addition on two streams.

When comparing this code to the one from Figure 1, we can see that the main difference is that in the streamed version computations on the first part of the arrays can start after only half of the data has been transferred. Thereafter, these computations can be overlapped with the loading of the second half of the data. Likewise, the first half of the results in `dev_a` can be transferred back while the second half is still being computed. This effect of overlapping computation and communication is called *latency hiding* as it allows the latency of the memory transfer to be hidden behind some computation on another part of the data. More details on how to leverage streams for overlapping communication and computation in CUDA can be found in the literature [12, 20].

Note here, that this version of our addition still assumes that all data at some point resides on the GPU. So it does not tackle the problem of *out-of-core* computations on the GPU. Furthermore, this kind of code transformation only works because we can guarantee that both kernels do not need to access any data handled by the corresponding other stream.

4 SUITABLE CODE PATTERNS

When looking at the streaming-based latency hiding technique we explain in the previous section we can see a possible avenue towards out-of-core computations on the GPU: All we need to do is to come up with a way to re-use parts of the device memory when it comes to loading later parts of the argument arrays.

While this sounds trivial in principle, it comes with several challenges. First of all, we need to make sure that a streaming approach is possible at all. The key prerequisite for streaming is the ability to statically guarantee that all data accesses lie within the partially fetched data only, i.e., we need to have some form of guaranteed access locality.

Looking at indirect indexing techniques such as a tensor comprehension of the form

```
{ iv -> a[b[iv]] }
```

clearly show that this is not always possible. In this example, there is no way of predicting the accesses within the array `a` unless we have some very strong static guarantees regarding the values within the array `b`. Without these, any attempt to stream this kind of

computation would require at least the entire array a to be resident on the GPU.

Even in some of the cases where access ranges can be statically determined, it can be rather challenging to partition all arguments in ways that can be compiler generated. As an example, consider matrix multiply:

```
{ [i,j] -> sum (a[i,.] * b[.,j]) };
```

Here, we would have to chunk array a row-wise, while having to chunk the array b column-wise.

Looking at Gauss-Jordan elimination, things get even more involved:

```
 for (i = 0; i < n; i++) {
    a[[i]] = a[[i]] / a[[i,i]];
    a = { [j] -> a[j]       | i == j;
          [j] -> a[j] - a[i] * a[j,i] };
}
```

Here we see that for iteration i of the surrounding sequential loop, we always need two rows in order to compute a row j of the result: the current row (j) of a and the i^{th} row of a as well.

In the context of this paper, we restrict ourselves to a set of simpler code patterns, where we can build on the idea of slicing all argument arrays along the first dimension. Examples like matrix multiply or Gauss-Jordan elimination are left as possible future work.

4.1 Map-like computations

The most obvious group of tensor comprehensions suitable for the kind of streaming we propose in the previous section are computations where all array accesses are *strictly local*. We refer to these computations as *map-like computations*. More formally, we say that all array accesses within a tensor comprehension of the form

```
{ iv -> expr_1(iv) | constraints_1;
   ...
   iv -> expr_n(iv) | constraints_n }
```

are *strictly local*, if and only if all sub-expressions within the result-defining expressions `expr_1(iv)`, ..., `expr_n(iv)` that contain a reference to a relatively free array-variable a are of the form `a[iv]`.

Clearly, our addition example from the previous sections constitutes such a map-like computation.

4.2 Stencil-computations

While map-like operations are frequently used, a much larger group of operations exposes a more relaxed form of locality which we call *stencil locality*. The idea here is to allow for constant offsets. We call all array accesses within a tensor comprehension like the one above *stencil local*, if and only if all references to relatively free variables a are of the form `a[iv+off]` and there exist two constants `cl` and `cu` such that for all `off` we have `cl <= off < cu` on all components of `off`.

With this relaxed notion of locality, our work becomes applicable to a very large set of applications ranging from convolutions for image processing, over numerical methods for approximating PDEs to implementations of deep learning techniques. The price for allowing constant offsets is a need for more sophisticated streaming as for any chunk of results a slightly bigger chunk of input data is needed.

To reflect typical application scenarios, we look for a slightly more complex application pattern. We want to capture situations where such a computation is embedded into a sequential loop, i.e., we call a code pattern *stencil-computation* if it is of the form:

```
for (i=start; i<stop; i++) {
    a = { iv -> expr_1(a, iv) | constraints_1;
          ...
          iv -> expr_n(a, iv) | constraints_n };
}
```

However, we also include occurrences of standalone tensor comprehensions with stencil-local accesses. We simply consider them special cases with only one iteration.

5 IMPLEMENTATION

5.1 General strategy

For both algorithmic patterns, we leverage CUDA's streams and CUDA's asynchronous communications. As we describe in Section 3, these framework features can be combined to run multiple streams enabling transfers between host and device to happen concurrently to kernel executions. Key to this approach is that we can identify independently executable sets of inputs, outputs and kernel invocations. Once they are identified, we can task several independent streams to handle these sets: Each individual stream is tasked to transfer a portion of the host buffer to the device buffer (1), launch a kernel on that subset of the data (2) and then transfer the computed results back to the host (3).

In contrast to Section 3, where it is all about latency hiding, we now want to look at an extension of this idea that enables dealing with out-of-core computations, *i.e.*, computations whose arguments or results do not fit onto the device in their entirety. The key observation is that a streaming setup ideally suits this demand. All that is needed is to envision the device memory to act as a sliding window that can be moved along the larger host buffer. After partitioning the host data, we assign a stream to each chunk and initiate the aforementioned workflow. While the device starts executing, the sliding window is moved to the right until all data is processed. Since each stream is essentially a queue of operations, we can for each stream already queue up the workflow for another equally or smaller-sized region on the host. This process is then repeated until the entire host data is partitioned and each partition assigned to a stream.

5.2 Map-like computations

Figure 4 shows our general strategy applied to map-like computations with a single argument. The host buffer is partitioned into four chunks, each indicated with a different colour, and two streams are used to process two chunks on the device at a time. The unmarked colours indicate initial values and the colours marked with a cross are finished values produced by a kernel invocation. Since by definition map-like computations make only strictly local accesses into their arguments (see Section 4), there can be no dependencies between streams, allowing for completed chunks to be evicted from the device.

Figure 5 shows the main code pattern we propose for map-like computations. First, it is necessary to calculate the number of *rounds* that are required to compute the entire host buffer. Each round

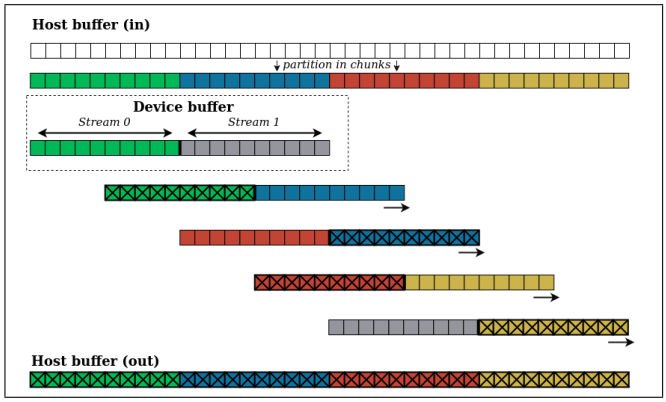

Figure 4: An abstract visualisation of the streamed map-like implementation.

consists of a host-to-device (H2D) memory transfer phase, a kernel invocation phase and a device-to-host (D2H) memory transfer phase. Looking back at Figure 4, the first round for stream 0 would then be the loading, computing and unloading of the green chunk, and for stream 1 the loading, computing and unloading of the blue chunk.

Line 1 shows how, given buffer sizes *host_n* and *dev_n*, we determine the total number of rounds *r*. We then compute the chunk size *chunk_n* by dividing the device-buffer size by the number of streams *ns* that we want to run concurrently. For readability of the paper, we assume that the host buffer is divisible by the device buffer, and the device buffer is divisible by the number of streams. However, our full templates do not build on these assumptions but handle the edge cases appropriately.

Also note here, that we need a minimum of two streams to enable overlapping of communication and computation; however, in practice, higher numbers are needed to maximise the latency hiding effect.

```
1 r = (host_n + dev_n - 1) / dev_n;  \\ number of rounds
2 chunk_n = dev_n / ns; \\ size of individual chunks
3 for (r_j = 0; r_j < r; r_j++){
4     \\ H2D memory transfers
5     \\ kernel invocations
6     \\ D2H memory transfers
7 }
8 cudaDeviceSynchronize();
```

Figure 5: Implementing the sliding window for ns streams

In the sequel, we pick up on our addition example from Section 2 and Section 3 again, to show what the code looks like for the three different phases in case of a two argument map. We start with the host-to-device transfers; they are shown in Figure 6. In Figure 6a, we see the original, non-streamed version, transferring both arguments in their entirety synchronously to the device. Figure 6b shows the streamed code for host-to-device transfers in current round *r_j*, which replaces line 4 in Figure 5. The depicted for-loop starting on line 1 iterates over all streams and initiates two transfers per stream.

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

(a) Non streamed H2D

⇓

```
1 for (i = 0; i < ns; i++){
2     cudaMemcpyAsync(dev_a[i * chunk_n],
3                     a[r_j * dev_n + i * chunk_n],
4                     chunk_n, h2d, stream[i]);
5     cudaMemcpyAsync(dev_b[i * chunk_n],
6                     b[r_j * dev_n + i * chunk_n],
7                     chunk_n, h2d, stream[i]);
8 }
```

(b) Streamed H2D

Figure 6: H2D transformations for map-like computations

```
1 dev_a = kernel<<< host_n >>>(dev_a, dev_b, host_n);
```

(a) Non streamed kernel invocation

⇓

```
1 for (i = 0; i < ns; i++){
2     kernel<<< chunk_n, stream[i] >>> (dev_a, dev_b, chunk_n,
3                                       i*chunk_n);
4 }
```

(b) Streamed kernel invocation

Figure 7: Kernel invocation transformations for map-like computations

Figure 7 shows how the non-streamed kernel invocation is transformed into its streaming counterpart. Again, we use a for-loop that iterates over the number of streams to insert kernel invocations on the chunks within each stream (Figure 7b). Note that we pass the chunk size and the offset with respect to the first index as parameters instead of the entire host-buffer size. In the corresponding kernel code, shown in Figure 8, these parameters are being used to identify the data that needs to be computed with.

```
1 kernel(dev_a, dev_b, n){
2     tid = blockIdx.x * blockDim.x + threadIdx.x;
3
4     if(tid < n)
5         dev_a[tid] = fn(dev_a[tid], dev_b[tid]);
6 }
```

(a) Non streamed kernel

⇓

```
1 kernel(dev_a, dev_b, n, offset){
2     tid = blockIdx.x * blockDim.x + threadIdx.x + offset;
3
4     if(tid < n + offset)
5         dev_a[tid] = fn(dev_a[tid], dev_b[tid]);
6 }
```

(b) Streamed kernel

Figure 8: Kernel transformations for map-like computations

Finally, we have to transform the device-to-host transfers. Figure 9 shows this transformation. The streamed code for *D2H* memory transfers is similar to the code for the *H2D* transformations from Figure 6. Given the number of streams *ns*, it is possible to produce the desired streamed transfers using a for-loop.

```
cudaMemcpy(a, dev_a, host_n, d2h);
```

(a) Non streamed D2H

⇓

```
for (i = 0; i < ns; i++){
    cudaMemcpyAsync(a[r_j * dev_n + i * chunk_n],
                    dev_a[i * chunk_n],
                    chunk_n, d2h, stream[i]);
}
```

(b) Streamed D2H

Figure 9: D2H transformations for map-like computations

5.3 Stencil computations

When moving from map-like computations to stencil computations, we lose the property that all array accesses are strictly local. This means that when computing a chunk of the result we typically need more input data than just the corresponding chunk of all inputs.

From our characterisation of stencil operations in Section 4, we know that for each computation of a result position *iv*, we need a neighbourhood of a fixed size around that position. To cater for that situation, we create buffers on the device that are larger than the chunks of the results and we load overlapping portions from the input into them.

To illustrate our approach for stencil computations, we will consider a very simple form of stencil computation, a three-point stencil operation, where we compute a vector from an existing one by the average of a value and both its neighbours. A corresponding SaC code would be:

```
{ i -> (a[i-1] + a[i] + a[i+1]) / 3.0 | [1] <= i < [n-1];
  i -> a[i]                                            };
```

For this example, we can see that for computing all but the first and last chunk we need exactly one element from the previous chunk and one element from the subsequent chunk in order to be able to compute all elements for the given chunk. This observation gives rise to a variant of our map-like scheme as shown in Figure 10. Using the same colouring and marking of chunks as in the map-like approach, we now can see that our device-buffer chunks not only take the values from the corresponding input chunk but also one element from a chunk to the left (if present) and one element from the chunk to the right (if present). Only the elements corresponding to the chunk are being computed within each individual stream, writing back exactly that portion to the host.

While this approach is rather simple to implement it has a very high level of communication in relation to the amount of computation needed. To ameliorate the situation, it is well-known from code generators for stencil codes [7, 24] that it is advantageous to improve temporal locality through tiling in case the stencil operations are applied repeatedly.

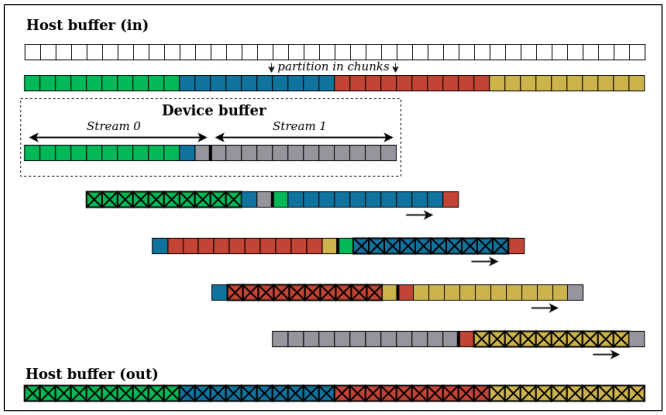

Figure 10: An abstract visualisation of how a single three-point stencil operation can be streamed.

5.4 Iterative stencil computations

Again, we look at a three-point stencil computation but now embedded into a sequential loop:

```
for (t=0; t<t_max; t++)
    a = { i -> (a[i-1] + a[i] + a[i+1]) / 3.0
              | [1] <= i < [n-1];
          i -> a[i]                              };
```

To improve temporal locality in the streaming version, the basic idea is to perform several iterations on each chunk before evicting the chunk from the device buffer. Given that for each stencil operation we need more input than we can produce output, this implies that we can compute fewer and fewer elements as we progress in the iterations. If we consider the iterations an additional axis of the computation, then such a computation in the three-point stencil case forms a trapezoidal shape of values, shrinking by one element per iteration on each side of the chunk.

Depending on the number of iterations we plan to perform on each chunk before moving on to the next one, this increases the extra space required in the buffer from 2 elements for a single step to *2*n* for *n* steps. Besides the increase in size, it also implies the loss of partial computations on the extra elements as the neighbouring chunk will require exactly the same computations. While this might be acceptable in the case of a three-point stencil on a 1-dimensional array, the space and compute overhead becomes quickly untenable in case of stencils on higher-dimensional arrays. To avoid such overhead for high numbers of *n*, we use a more elaborate scheme.

The key idea is to keep all partial computations and to alternate between phases that compute trapezoidal shapes and inverted trapezoidal shapes that compute the missing values between them. We refer to the trapezoids in the first phase as *even trapezoids* and to the inverted trapezoids of the elements between them as *odd trapezoids* as they correspond to the even and odd chunks of overall computations, respectively. Figures 11 and 12 demonstrate these two phases.

Figure 11 shows the first phase of computing the even trapezoids. In the host data, we mark the even chunks in green, blue, red, and yellow. In between, we have the starting values of the odd chunks, coloured in purple. The discrepancy in size between the odd and

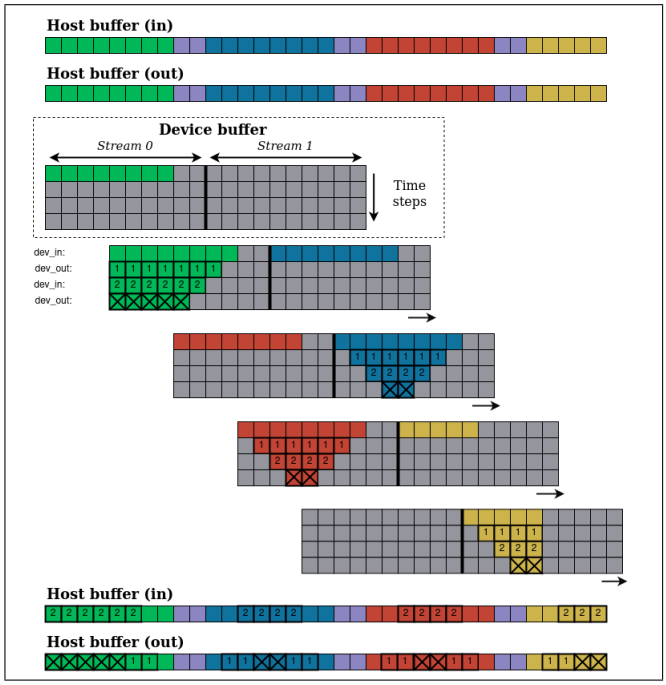

Figure 11: An abstract visualisation of how even trapezoids are computed in the streamed stencil implementation.

even chunks reflects that the former will shrink in size over the cause of iterations, whereas the latter will grow.

Again, we utilise the idea of a sliding window, but this time every other chunk of data is skipped depending on the current phase. After loading the first chunk (green fields in Figure 11) into the device buffer *dev_in*, we start executing three iterations of the three-point-stencil computation. While doing so, we alternate between *dev_in* and *dev_out*. The next block in Figure 11 shows the computed intermediate values in fields with bold edges and the computed final values additionally marked with a cross. We put numbers 1 and 2 into the fields corresponding to intermediate values to indicate the iteration they belong to. Once the three iterations are performed, **both** device buffers are being transferred back to the corresponding host buffers. As can be seen in the host buffers in the bottom of the figure, both buffers contain values from different iterations.

While the three iterations are being computed on chunk 0, chunk 2 (blue fields in Figure 11) is being loaded into the next stream's device buffers. Likewise, these are being computed while the results from the first stream are being transferred back to the host, and so forth. The bottom of the figure shows the final state after the first phase is completed. At this point, a global synchronisation ensures completion of the entire phase before computation of the second phase commences.

As shown on top of Figure 12, the results from the first phase serve as starting point for the second phase. When loading chunk number 1 of both the in- and output buffer, we see that we do not only load the purple elements of the chunk itself, but also some of the partially computed fields of the neighbouring chunks; some green values from chunk 0 and some blue values from chunk 2. A closer look at the loaded values and their inscribed iteration

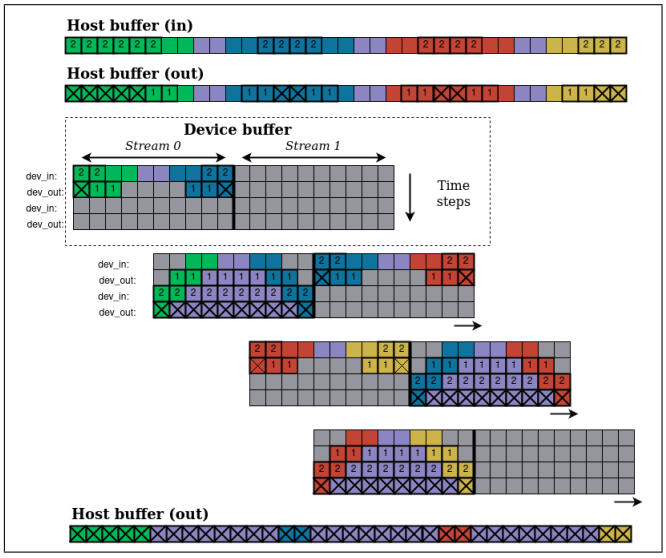

Figure 12: An abstract visualisation of how odd trapezoids are computed in the streamed stencil implementation.

numbers reveals that different values are being utilised at different iterations, allowing the next purple values of the chunk to be computed without re-computing any values of the first phase. While the computation takes place, the next chunk (chunk number 3) is being loaded. During the computation of that chunk, we evict both device buffers from the first stream back to the host, now leading to a vector whose values consist exclusively of values with three stencil operations performed on them.

Once the second phase is finished, we once again perform a global synchronisation before potentially restarting all over again.

Despite being conceptually rather different from the streaming of map-like operations described in Section 5.2 the proposed code for streaming iterative stencil-computations looks rather similar.

Figure 13 shows the basic setup. Again we start by computing the number of rounds needed. This is followed by a for-loop that iterates over the rounds performing the typical triplet of actions within each stream: host-to-device transfers, kernel executions, and device-to-host transfers. The key difference here being that we have two of these sliding window implementations, one for the even chunks and one for the odd chunks, separated by a full device synchronisation. To improve readability, we omit handling the edge cases that involve the very first and very last value of the host buffer, and in terms of divisibility we make the same assumption as for the map-like computation.

The necessary adaptations of the kernel are shown in Figure 14. Again similar to the map-like computations, we use an offset to identify the elements that need to be computed. However, since we need to compute ranges that vary with the iteration that we perform, we now use two offsets: *offset-l* for identifying the starting point and *offset_r* for identifying the end point of our kernel execution.

As a consequence of the offset approach, the key steering of the computation happens in the code that launches the kernels. It is depicted in Figure 15. Here, we can see how the offsets are being computed from the current stream index *i*, the current iteration *k*,

```
r_n = dev_n - ns * (t + 2);
r = (host_n + r_n - 1 ) / r_n;
```

(a) Shared constants

```
for(i = 0; i < r; i++){
    for(j = 0; j < ns; j++)
        if(i % 2 == j % 2)
            \\ h2d
    \\ kernels
    \\ d2h
}
cudaDeviceSynchronize();
```

(b) Even trapezoids

```
for(i = 0; i < r; i++){
    for(j = 0; j < ns; j++)
        if(i % 2 != j % 2)
            \\ h2d
    \\ kernels
    \\ d2h
}
cudaDeviceSynchronize();
```

(c) Odd trapezoids

Figure 13: Code structure for streamed stencil computations

```
kernel(dev_in, dev_out, n){
    tid = blockIdx.x * blockDim.x + threadIdx.x;

    if(tid > 0 && tid < n - 1)
        dev_out[tid] = fn(dev_in[tid-1], dev_in[tid],
                          dev_in[tid+1]);
}
```

(a) Non streamed kernel

⇓

```
kernel(dev_in, dev_out, offset_l, offset_r){
    tid = blockIdx.x * blockDim.x + threadIdx.x + offset_l;

    if(tid < offset_r)
        dev_out[tid] = fn(dev_in[tid-1], dev_in[tid],
                          dev_in[tid+1]);
}
```

(b) Streamed kernel

Figure 14: Kernel transformations for stencil computations

the total number of iterations per pair of phases *t*, and the chunk size *chunk_n*. We also see the explicit swapping of device input and output buffers *dev_in* and *dev_out*.

Finally, we have the code for copying the device buffers back to the host in Figure 16. Note that in contrast to the map-like operations, here, we have to copy the content of both device buffers back to the corresponding host buffers.

6 EXPERIMENTAL EVALUATION

6.1 Experimental setup

The key question we try to answer is: can we hope to achieve a reasonable performance when moving from an in-core problem size

```
for (k = 1; k <= t; k++){
  for (i = 0; i < ns; i++){
    if(r_j % 2 == i % 2)
      kernel<<< chunk_n, stream[i] >>> (dev_in, dev_out,
                                        i*(chunk_n + t + 2) + k,
                                  (i+1)*(chunk_n + t + 2) - 2 - k);
  }

  if(k < t)
    swap(dev_in, dev_out);
}
```

(a) Even trapezoids

Figure 15: Kernel invocations for streamed stencils

```
for (i = 0; i < ns; i++){
    if(r_j % 2 == i % 2){
        \\ send final and intermediate values to host out
        cudaMemcpyAsync(out[r_j * r_n + i * chunk_n + 1],
                        dev_out[i * (chunk_n + t + 2) + 1],
                        chunk_n + t - 2, d2h, stream[i]);

        \\ send intermediate values to host in
        cudaMemcpyAsync(in[r_j * r_n + i * chunk_n + 1],
                        dev_in[i * (chunk_n + t + 2) + 1],
                        t, d2h, stream[i]);

        cudaMemcpyAsync(in[r_j * r_n + (i+1) * chunk_n + 1],
                  dev_in[(i+1) * (chunk_n + t + 2) - t - 3],
                  t, d2h, stream[i]);
    }
}
```

(a) Even trapezoids

Figure 16: D2H transfers for streamed stencils

to an out-of core problem size? If so, does this work for all kinds of map-like and iterative stencil operations, or does it only work for certain classes of applications? Further questions are: Does it make sense to use streaming for some in-core cases as well? If so, for which kind of applications? How does the choice of the number of streams affect the overall performance?

To tackle these questions, we first have to identify how we can possibly quantify 'reasonable' performance. An out-of-core implementation requires the orchestration and handling of many transfers and kernel invocations rather than just a single triplet of host-to-device transfer, kernel invocation, and device-to-host transfer as it is needed for the in-core counterpart. This overhead can possibly be offset by gains in latency hiding whose effect depends on the computation to communication ratio of the problem investigated.

Looking at both, map-like computations and iterative stencil computations, we conduct the following experiments: We take C++ implementations for both, the streamed and the non-streamed version of our two examples from Section 5. Both of these have a very low ratio between computation and communication between host and device as can be seen from Table 1.

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

Table 1: An overview of the number of floating-point operations, memory reads and memory writes in each kernel execution of the different algorithms.

We run different problem sizes ranging from vector sizes of 20MB up to 25GB and we compare the overall performance between the non-streamed version (for all sizes possible) and the streamed version. To judge the overall performance in absolute terms, we do not only look at the GFLOPS/sec, but we also look at the effective memory bandwidth between GPU and device memory that we achieve. In particular the latter measure is crucial here, given that both, vector additions and three-point-stencils are memory bound. In order to increase the ratio of computation to communication between host and device, we also look at increasing numbers of iterations over each of the two computations ranging from 1 to 3883 iterations. Note here, that for the map-like operation we intentionally do not move the iteration into the kernel as that would shift our kernel from being memory bound to being compute bound and, thus, would take away the effective memory bandwidth between GPU and device memory as upper physical bound that we can compare against irrespective of the computation to communication ratio between host and device.

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. Essentially, the measured run-time covers the precise moment the first host-to-device transfer is initiated until the final device-to-host transfer has completed.

Most of the time, the standard deviation of these runs is between 0.01%-2%, so we use the mean value 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, which 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 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 2: An overview of the system used for the experiments.

6.2 Performance results

In Figure 17, the effect of input size on the performance of the map-like computation in terms of GFLOP/s and bandwidth is shown. The left y-axis shows the GFLOP/s and the right y-axis shows the

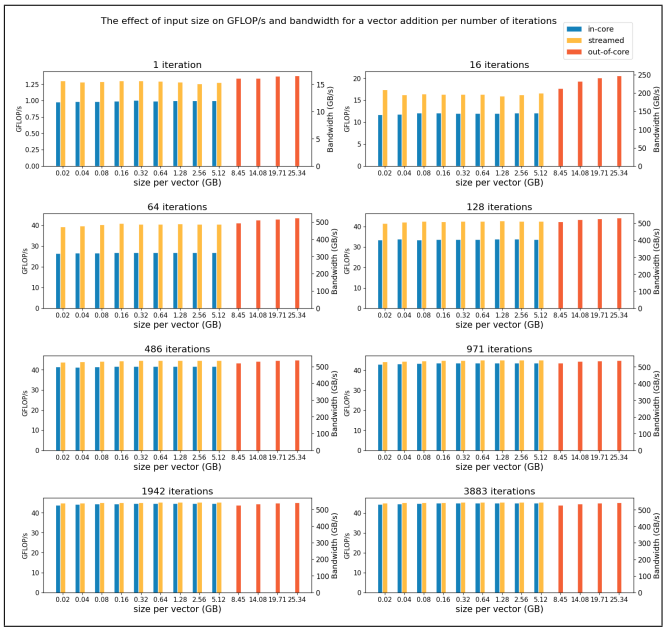

Figure 17: The effect of input size on the performance of an in-core, streamed and out-of-core map-like computation at different iterations

bandwidth. Looking at the top-left chart, we can see that for one iteration, i.e., in the extreme case regarding the computation to communication ratio, the streamed version outperforms the non-streamed version by almost 30%. This is clearly an effect of the latency hiding that stems from the streaming. Interestingly, the overall performance actually increases as we transition from in-core to out-of-core. We can also observe that the entire computation is very much dominated by the transfers between host and device as our kernel performance only achieves an effective device memory bandwidth of roughly 16 GB/s.

As we start shifting the computation to communication ratio by increasing the number of iterations, we can see how the dominance of the host-device communication slowly diminishes. From roughly 64 iterations onward, we see that we reach about 80% of the effective device memory bandwidth. Interestingly, we still see that the latency hiding of the streamed version outweighs any overheads caused by the use of streaming as the streamed version yields about 30% better performance.

We look at further shifts towards computations to see whether we at some point reach a situation where the non-streamed version outperforms the streamed version due to streaming overheads. At 3883 iterations, the non-streamed version almost catches up, but is still slightly less performing.

From these findings, we can see that the overhead of streaming is negligible in comparison to the gains in latency hiding. For map-like computations, streaming does not only provide excellent out-of-core performance, it also outperforms the in-core version due to latency hiding of the communication between the host and the device. Furthermore, it turns out that these effects seem to be independent of the problem size.

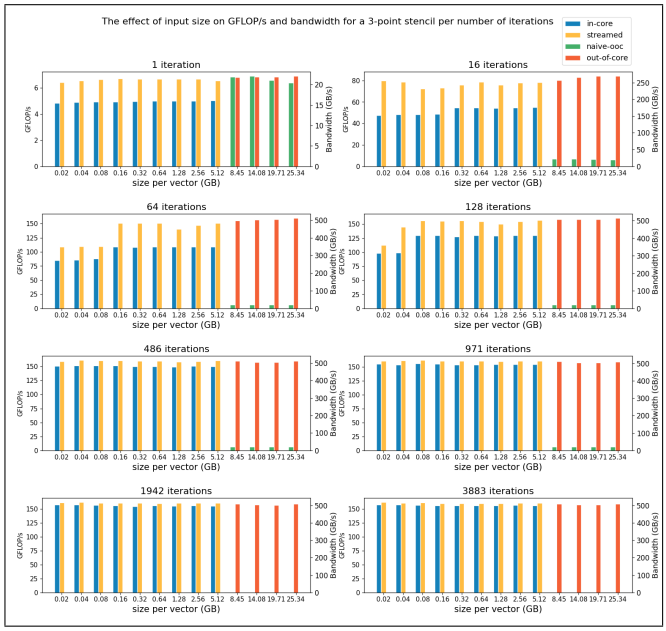

Figure 18: The effect of input size on the performance of an in-core, streamed and out-of-core 3-point stencil computation at different iterations.

Figure 18 shows the same experiments for the three-point stencil codes. Overall, we see the same effects: the streaming version outperforms the non-streaming version due to latency hiding, an effect that slowly diminishes as we increase the number of iterations. Even with 3883 the non-streaming version has not fully caught up with the streaming version. From roughly 64 iterations onward, we see that the streaming version reaches its maximum of 80% of the memory bandwidth

Notice here though, that this only works out since in **all** examples we only communicate the entire vector once from host to device and once back. While that is trivial in the map case, in the stencil case, this requires the elaborate trapezoidal scheme from Section 5.3. To demonstrate this aspect, we have added the green bars referred to as naive out-of-core. This implementation is based on a single step at a time as sketched in Figure 10. For this implementation we see competitive performance for one iteration but no improvements for multiple iterations as there are complete transfers between host and device and back for each single iteration.

6.3 Impact of streaming granularity

We conduct several experiments to identify the impact of the number of streams used on the overall performance achieved. We use a subset of the experiments from the previous subsection varying the numbers of streams from 2 to 128. Figure 19 shows our findings in the context of our vector addition experiments. The figure contains four rows of plots. Each row pertains to a vector size, starting from small to large. While the top two rows show measurements for in-core streaming, the bottom two rows shows the measurements of our out-of-core experiments. Furthermore, the first column of plots shows our results at one iteration and the second column of plots show our results for 3883 iterations.

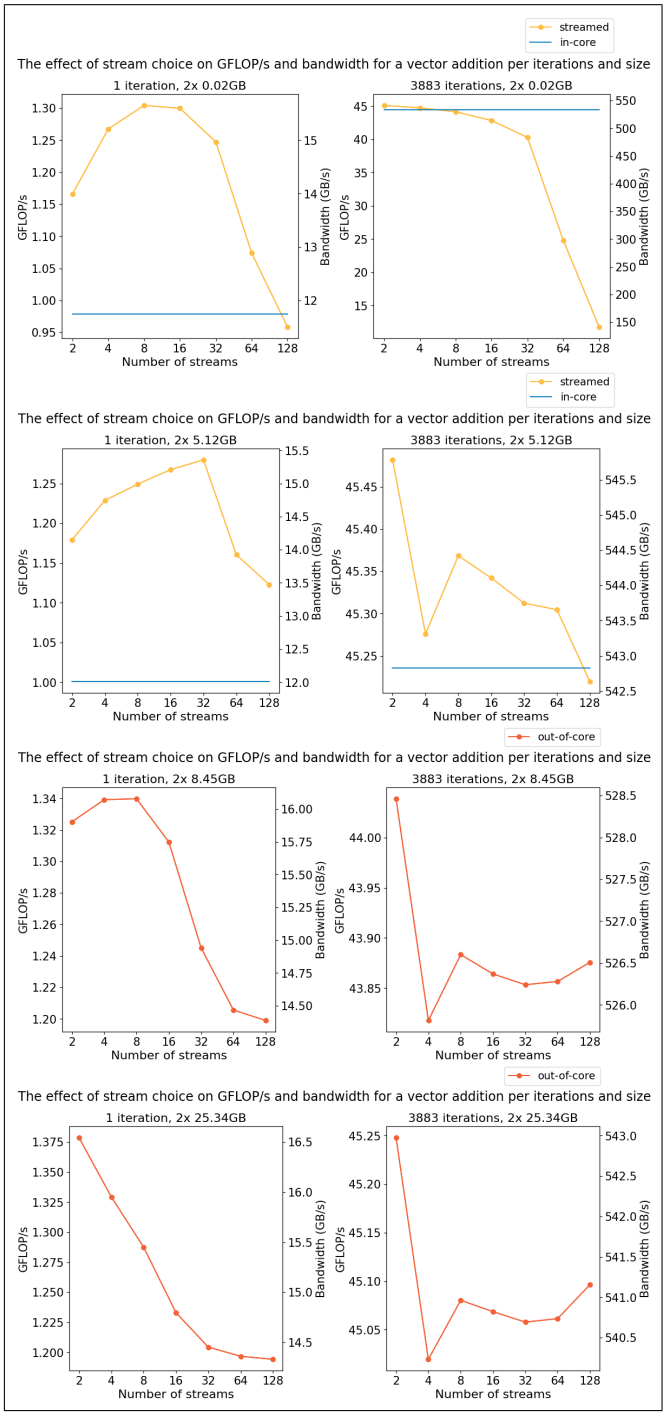

Figure 19: The effect of number of streams choice on the performance of the map-like computation.

We can see that several aspects impact the performance of the algorithm. Smaller arrays and fewer iterations favour a larger number of streams, unless the resulting chunks get *too* small (see top-left plot of Figure 19). This observation suggests that, when the transfer-to-compute ratio is low, there are multiple opportunities for overlapping communication and computation rather than exclusively at the very first and very last chunk. On the other hand, larger arrays and iterations perform better with a small number of streams, which is particularly visible in the bottom two rows and the second column of Figure 19.

Notice here that the importance of the number of streams diminishes at a high number of iterations, since the discrepancy between the extremes is below 1% (except when the chunk size gets *too* small). Overall, we observe that the choice for the number of streams is not significant.

Similar to the map-like computation, the performance impact of the choice of streams is measured for the stencil computation, which is shown in Figure 20.

The layout of the figure is identical to that of Figure 19, containing four rows and two columns of plots, where the rows vary in input size and the columns in number of iterations.

Interestingly, our measurements for the stencil computation paint a nearly identical picture to the measurements of the map-like computation. Clearly, the choice regarding the number of streams again matters more for a lower number of iterations. On top of that, we see once more that a low number of iterations in combination with smaller arrays benefit from eight or more streams, whereas larger arrays and iterations prefer a smaller number.

7 RELATED WORK

The differences in performance between CUDA's communication models in the context of code generation has been studied before. In [25], the authors look at different communication models and their performance impact for a range of different GPU architectures. The authors find that the choice of communication model is hardware specific and that making the right choices becomes increasingly important as the compute-to-communication ratio gets lower. Streaming or even out-of-core computations are not considered in that work. In this paper, we look exclusively at applications with a low compute-to-communication ratio. We observe that streaming generally improves the overall performance but has the most pronounced effect on smaller problem sizes and when being performed only a few times.

A lot of research has been done on optimizing specifically stencil computations on the GPU. In particular, [20] and [21] share similarities with our work since the authors look at stencil computations utilising multiple GPUs. In [20], multiple CUDA streams are used to overlap the communication of border regions between neighbouring GPUs with the computation of the regular non-border regions on each GPU. Additionally, the authors use several CPU threads to reduce the overhead from kernel launches. [21] builds on this approach 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 [20] and [21] do not stream chunks of data on the individual GPUs and are not directly targeting general out-of-core execution.

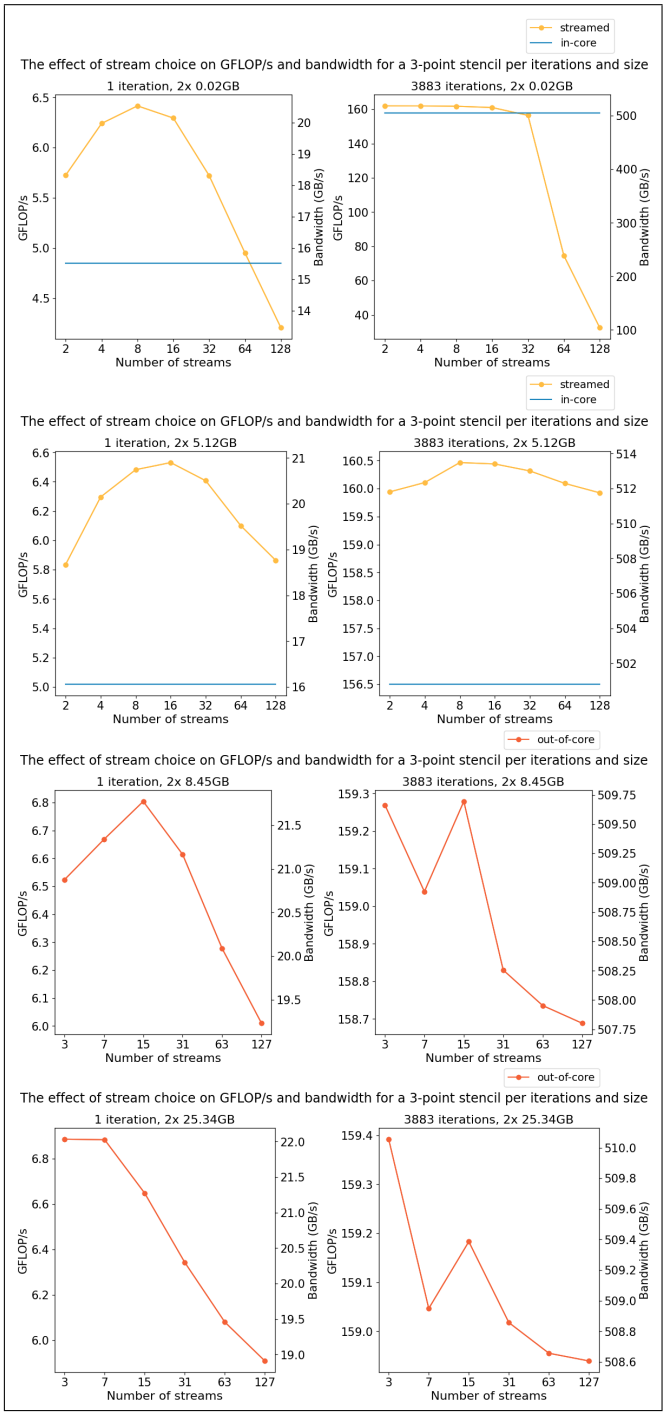

Figure 20: The effect of number of streams choice on the performance of the three point stencil computation.

Research on the automation of streaming arrays through the GPU exists. Both [2] and [14] look at extensions for the embedded array language Accelerate [1], which allow the programmer to explicitly choose to stream a limited set of operations through the GPU. [14] is most closely related to our paper, as it specifically

looks at arrays similar to arrays in SaC; namely so-called regular arrays in the context of Accelerate, which are arrays that do not contain other arrays as elements. The authors use a similar chunk-based approach to scheduling the streams, but communication and computation are not overlapped.

Furthermore, the benchmarks presented in [14] exhibit a correlation between the chunk size and the achieved performance. For example, a dot product operation reaches optimal performance at approximately sixteen million elements per chunk. On the other hand, we 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 we try to overlap as much communication and computation as possible. This matches the authors' suggestion in [14] that, in general, there could be a significant improvement in performance by overlapping communication and computation as done in this paper.

Besides our specific context of automating the streaming of arrays through GPUs, there is a much broader related body of research out there that looks at stream programming in general. One example of that is [11], here the authors present the fundamentals of Arc, which supports the expression and compilation of long-running stream operators in the context of hardware accelerated continuous batch and stream analytics.

More recently, a few studies related to out-of-core stencil computations on the GPU have been published. For instance, [9] 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 temporal locality improving techniques combined with memory-saving optimizations and communication overlap to achieve higher performance. The sub-domains are computed sequentially and the border region dependencies are resolved with redundant computations.

In contrast, we calculate several sub-domains concurrently by streaming them through the GPU and this is done without redundant computations. [9] 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 [9] experiment with 3D stencils and also utilise additional disk memory from the host in scenarios where the RAM capacity is not sufficient, so a direct comparison cannot be made.

Moreover, [10] follows up on [9] by extending the out-of-core implementation to multiple GPUs. In [10], MPI is used to split the problem domain in sub-domains to be handled by separate GPUs, which each then utilise the approach from [9]. 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 introduce their algorithm into a domain specific language, but do not yet provide rewriting schematics or code generation examples.

Finally, [3] introduces a library (HHRT) that functions as a wrapper around MPI and CUDA to assists the programmer with the non-trivial memory swapping process in [9, 10]. 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.

8 CONCLUSION

This paper tackles the problem of mapping data-parallel array computations that operate on data larger than the device memory space of a GPU into out-of-core computations on these accelerators. As GPUs do not support virtual memory spaces such transformations are exclusively possible through a transformation of the array codes into streaming codes.

We identify two algorithmic patterns that can be transformed into streaming codes, provide CUDA code templates for streaming such algorithms and demonstrate the effectiveness of the streaming codes through a systemic performance evaluation. Given that such a streaming limits the opportunities for reusing data on the device, it implicitly also lowers the overall compute to communication ratio. We tackle this problem by leveraging CUDA streams to overlap kernel executions with concurrent memory transfers.

We systematically vary the computation to communication ratio and the problem sizes in order to investigate how the streamed code compares to non-streamed counterparts. We can not only show that the out-of-core computations in all cases achieve the same relative performance, but they even outperform the non-streamed versions in case all data fits into the device memory of the GPU. We can identify clearly that latency hiding of the transfer between host and device is the key factor here.

The optimal number of streams depends on several factors, the size of the data as well as the number of iterations over all data have an impact on this parameter. In general, we can see that smaller arrays and fewer iterations benefit larger numbers of threads (typically 8 or more) whereas large numbers of iterations favour fewer streams. However, the number of streams chosen turns out to only have a rather small effect on the overall run-time.

The overall drive of this research is to identify code patterns suitable for code generation that can process arrays that are larger than the memory that is available on the GPU without an overwhelming loss of performance. Surprisingly, we find that it is not only possible to create competitive streaming code, but that this code also considerably outperforms the non-streaming code in case the device memory is big enough to hold all data.

These insights suggest that the code generation for languages such as SaC should make use of the proposed templates irrespective of the size of arrays that are involved. While our hand-coded examples only implemented rather specific cases, the templates can directly be used for code generation for the full generality of the patterns identified in Section 4. Even the fact that our template for the stencil operation only operates on one-dimensional arrays is not a limiting factor since the code generated from SaC is based on a flattened internal representation. Multi-component indices into multi-dimensional arrays are anyways being translated into offsets in the unrolling of the array elements.

An interesting direction for further research clearly is an extension of the set of code pattern that can be handled. A straightforward candidate are reduction operations, in particular when being fused with map-like operations.

REFERENCES

[1] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming* (Austin, Texas, USA) *(DAMP '11)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/1926354.1926358

[2] Robert Clifton-Everest, Trevor L. McDonell, Manuel M. T. Chakravarty, and Gabriele Keller. 2017. Streaming Irregular Arrays. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell* (Oxford, UK) *(Haskell 2017)*. Association for Computing Machinery, New York, NY, USA, 174–185. https://doi.org/10.1145/3122955.3122971

[3] Toshio Endo and Guanghao Jin. 2014. Software technologies coping with memory hierarchy of GPGPU clusters for stencil computations. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, New-York, 132–139. https://doi.org/10.1109/CLUSTER.2014.6968747

[4] Clemens Grelck. 2005. Shared memory multiprocessor support for functional array processing in SAC. *J. Funct. Program.* 15 (05 2005), 353–401. https://doi.org/10.1017/S0956796805005538

[5] Jing Guo. 2012. *Fully automated transformation of hardware-agnostic, data-parallel programs for host-driven executions on GPUs.* Ph. D. Dissertation. University of Hertfordshire, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.630034

[6] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. 2011. Breaking the GPU Programming Barrier with the Auto-Parallelising SAC Compiler. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming* (Austin, Texas, USA) *(DAMP '11)*. Association for Computing Machinery, New York, NY, USA, 15–24. https://doi.org/10.1145/1926354.1926359

[7] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High Performance Stencil Code Generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 100–112. https://doi.org/10.1145/3168824

[8] Troels Henriksen. 2017. *Design and Implementation of the Futhark Programming Language.* Ph. D. Dissertation. Department of Computer Science, Faculty of Science, University of Copenhagen.

[9] Guanghao Jin, Toshio Endo, and Satoshi Matsuoka. 2013. 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*. IEEE, New-York, 1080–1087. https://doi.org/10.1109/IPDPSW.2013.58

[10] Guanghao Jin, Toshio Endo, and Satoshi Matsuoka. 2013. 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)*. IEEE, New-York, 1–8. https://doi.org/10.1109/CLUSTER.2013.6702633

[11] Lars Kroll, Klas Segeljakt, Paris Carbone, Christian Schulte, and Seif Haridi. 2019. Arc: An IR for Batch and Stream Programming. In *Proceedings of the 17th ACM SIGPLAN International Symposium on Database Programming Languages* (Phoenix, AZ, USA) *(DBPL 2019)*. Association for Computing Machinery, New York, NY, USA, 53–58. https://doi.org/10.1145/3315507.3330199

[12] Hao Li, Di Yu, Anand Kumar, and Yi-Cheng Tu. 2014. Performance modeling in CUDA streams — A means for high-throughput data processing. In *2014 IEEE International Conference on Big Data (Big Data)*. Insitute of Electrical and Electronics Engineers, New York, NY, USA, 301–310. https://doi.org/10.1109/BigData.2014.7004245

[13] Thomas Macht and Clemens Grelck. 2019. SAC Goes Cluster: Fully Implicit Distributed Computing. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, New York, 996–1006. https://doi.org/10.1109/IPDPS.2019.00107

[14] Frederik M. Madsen, Robert Clifton-Everest, Manuel M. T. Chakravarty, and Gabriele Keller. 2015. Functional Array Streams. In *Proceedings of the 4th ACM SIGPLAN Workshop on Functional High-Performance Computing* (Vancouver, BC, Canada) *(FHPC 2015)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/2808091.2808094

[15] Wojciech Michal Pawlak. 2021. *Derivative Pricing and Risk Management Applications.* Ph. D. Dissertation. University of Copenhagen, Universitetsparken 5, 2100 Kobenhavn.

[16] Daniel Rolls, Carl Joslin, Alexei Kudryavtsev, Sven-Bodo Scholz, and Alex Shafarenko. 2009. Numerical Simulations of Unsteady Shock Wave Interactions Using SaC and Fortran-90. In *Parallel Computing Technologies*, Victor Malyshkin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 445–456.

[17] Sven-Bodo Scholz. 2003. Single Assignment C: Efficient Support for High-Level Array Operations in a Functional Setting. *J. Funct. Program.* 13, 6 (Nov. 2003), 1005–1059. https://doi.org/10.1017/S0956796802004458

[18] Sven-Bodo Scholz, Stephan Herhut, Frank Penczek, Clemens Grelck, Artem Shinkarov, and Hans-Nikolai Viessmann. 2021. Single assignment C tutorial. https://sac-home.org/_media/docs:tutorial.pdf.

[19] Sven-Bodo Scholz and Artjoms Šinkarovs. 2019. Tensor Comprehensions in SaC. In *Proceedings of the 31st Symposium on the Implementation and Application of Functional Programming Languages* (Singapore) *(IFL '19)*. ACM, New York, NY, USA, Article 15, 13 pages. https://doi.org/10.1145/3412932.3412947

[20] Mohammed Sourouri, Tor Gillberg, Scott B. Baden, and Xing Cai. 2014. Effective multi-GPU communication using multiple CUDA streams and threads. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. Insitute of Electrical and Electronics Engineers, New York, NY, USA, 981–986. https://doi.org/10.1109/PADSW.2014.7097919

[21] Mohammed Sourouri, Johannes Langguth, Filippo Spiga, Scott B. Baden, and Xing Cai. 2015. CPU+GPU Programming of Stencil Computations for Resource-Efficient Use of GPU Clusters. In *2015 IEEE 18th International Conference on Computational Science and Engineering*. IEEE, New York, 17–26. https://doi.org/10.1109/CSE.2015.33

[22] Michel Steuwer. 2015. *Improving programmability and performance portability on many-core processors.* Ph. D. Dissertation. University of Münster. https://www.lift-project.org/publications/2015/steuwer15phdthesis.pdf

[23] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. LIFT: A functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, New-York, 74–85. https://doi.org/10.1109/CGO.2017.7863730

[24] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The Pochoir Stencil Compiler. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/1989493.1989508

[25] Hans-Nikolai Vießmann and Sven-Bodo Scholz. 2020. Effective Host-GPU Memory Management Through Code Generation. In *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages* (Canterbury, United Kingdom) *(IFL 2020)*. Association for Computing Machinery, New York, NY, USA, 138–149. https://doi.org/10.1145/3462172.3462199

[26] Hans-Nikolai Viessmann, Sven-Bodo Scholz, Artjoms Sinkarovs, Brian Bainbridge, Brian Hamilton, and Simon Flower. 2015. Making Fortran Legacy Code More Functional: Using the BGS Geomagnetic Field Modelling System As an Example. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages* (Koblenz, Germany) *(IFL '15)*. ACM, New York, NY, USA, Article 11, 13 pages. https://doi.org/10.1145/2897336.2897348

[27] V. Wieser, C. Grelck, P. Haslinger, J. Guo, F. Korzeniowski, R. Bernecky, B. Moser, and S.B. Scholz. 2012. Combining High Productivity and High Performance in Image Processing Using Single Assignment C on Multi-core CPUs and Many-core GPUs. *Journal of Electronic Imaging* 21, 2 (2012). https://doi.org/10.1117/1.JEI.21.2.021116

[28] Artjoms Šinkarovs, Hans Viessmann, and Sven-Bodo Scholz. 2021. Array Languages Make Neural Networks Fast. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (Virtual,Canada) *(ARRAY 2021)*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3315454.3464312