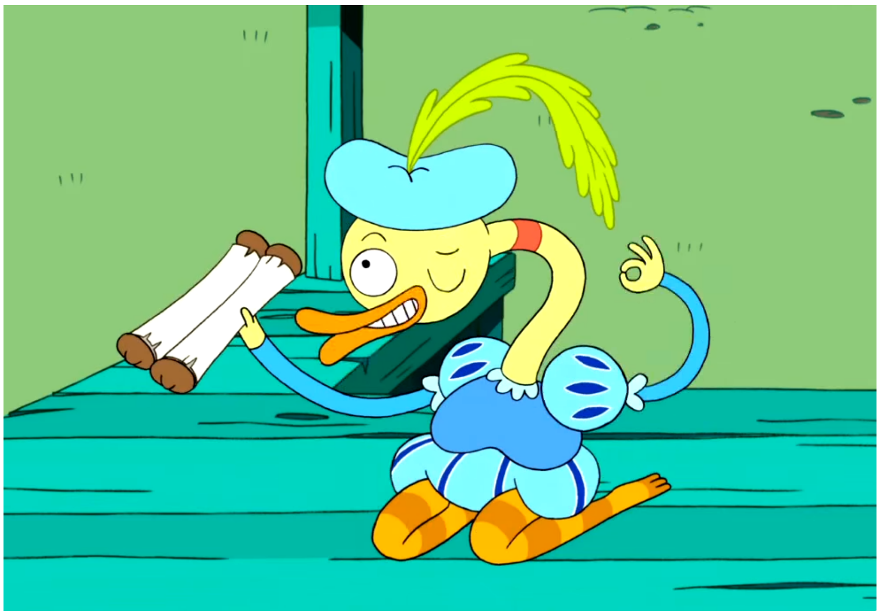
UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE



Master's thesis

sortraev (Anders L. Holst)

Optimizing Tensor Contractions for GPU Execution in Futhark

Advisor: Cosmin E. Oancea

2024-05-31

Abstract

The tensor contraction, a higher-dimensional analogue to the matrix multiplication, is a widely used basic building block that is not only suitable for efficient GPU execution due to its highly parallel nature, but also ripe for locality of reference optimizations due to a high degree of data reuse. Futhark, a highly optimizing compiler targeting GPU hardware, generates efficient 2D block/register tiled code for GEMM-like programs, but does not apply the transformation to arbitrary contractions. With an offset in tensor contraction and GPU code transformation theory, we detail how we successfully implemented block/register tiling of arbitrary tensor contractions into the Futhark compiler, using generic LMAD copies to stage input data and a number of other minor optimizations, and describe some of the problems overcome in doing so as well as the roadblocks and limitations which unfortunately remain. Using a small benchmarking plan we examine the practical benefits of the transformation, using a hand-written prototype kernel and a GPU code generator for high-performance tensor contractions as points of reference – the implementation performs well, reaching between 68% and 98% of the reference programs, but the opportunities for optimization are many. Finally, we present some ideas for future work in both improving and generalizing the implementation.

Acknowledgements

I wish to acknowledge:

My advisor, Cosmin E. Oancea, for his confidence in me when my projects seemed most hopeless, for his matter-of-fact approach to advising, for his excellent work as teacher and course coordinator in my time as student and TA at DIKU, and finally, for his honest impressions of life and work in research.

Troels Henriksen, of whom I have been a big fan since the very first CompSys live coding lecture during my second year, which must have been my first proper to low-level programming, and, in particular, C, which grew to become my favorite language. Thank you for much inspiration throughout my time at DIKU, but especially for the on-point hotline support during all three of my Futhark-related projects.

My friends Malthe, Jeppe, Louise, and Jens, for their emotional support throughout my almost seven years at DIKU. Without you … well, I probably would have still finished, but thanks anyway!

Finally, and with a special thanks: `aemylis` (Æmilie), for the many fun, interesting, difficult, and, not least of all, inspiring days and nights at DIKU during our bachelor's, on which I often reflect. *tjk tjk.*

Contents

1 Introduction and motivation

The tensor contraction is a basic building block of many software applications in various domains of the computational sciences, including, but not limited to, quantum chemistry modeling, fluid dynamic simulation, stress simulation in bridge engineering, probability, and, in particular, tensor networks in e.g. machine learning. The large computational complexity of most tensor contractions is a motivation in itself to develop efficient practical applications.

Futhark[1][2] is an optimizing compiler primarily targeting GPU hardware. As it would happen, Futhark is also the name of a much related, high-level array programming language based on a programming model emphasizing data parallelism – but for the entirety of this thesis report we shall concern ourselves primarily with the former. While still in active and ongoing development, the Futhark compiler already employs a variety of sophisticated (GPU) code transformations, and together with the programming language, it is already showing promising and interesting results, albeit mostly in academic applications. But there are still many windows of opportunity for optimization in the compiler. One such window is in the loop tiling pass of the compiler, and has to do with block/register tiling of exactly tensor contraction expressions. *Cue thesis.*

At present the compiler will identify matrix multiplication-like expressions in the source language and produce effficient GPU code using a memory optimization called *block/register tiling*[3]. Matrix multiplication can be viewed as a specialization of the tensor contraction, and they present similar opportunities for data reuse optimization. In this thesis project, we explore how the block/register tiling transformation can be generalized to arbitrary tensor contractions in the Futhark compiler and to what gain. We additionally explore a small number of optimizations and benchmark profile our implementation.

The intellectual contribution is by no means novel, but rather a collection of important observations about the prospects of successfully implementing and generalizing the optimization in the context of Futhark specifically, as well as documentation of some of the technical hurdles discovered (and, in some cases, overcome) in development. The practical contribution (i.e. the code product) should be considered a basis for further development, or, at the least, a proof of concept.

2 Related work

Many different techniques for computing tensor contactions exist, and we shall not discuss all of them here, nor go into very fine detail. In some broad terms, the techniques can be grouped in two categories: Those which use explicit transposition of input/output to ensure efficient memory access, and those which perform direct contraction without transposition and obtain efficient access through other means [4][5].

An example of the former is TTGT (**T**ranspose-**T**ranspose-**G**EMM-**T**ranspose), which transposes each of the operand tensors (hence the initial two T's) to suitable permutations in which GEMM can be applied with efficient 1-stride to the innermost modes (i.e. the innermost dimensions of the tensors in memory); then GEMM is applied, and the result is transposed again to the desired output permutation. The technique can be efficient if an optimized BLAS implementation of GEMM is used, but all three transpositions are pure overhead and require extra memory[6][4].

An example of the latter is GETT (**G**EMM-like **T**ensor-**T**ensor multiplication) [4], which uses the cache hierarchy and a number of additional index sets based on the operand modalities to implicitly transpose inputs as they are loaded, whereafter a loop-wrapped macro-kernel performs in-cache GEMM of submatrices. In other words, this technique resembles high-performance GEMMs, but where the extra dimensionality is handled via careful packing. GETT is mainly targeted at CPU.

COGENT (**CO**de **GEN**erator for **T**ensors) [5][6], the initial inspiration for this project, belongs to the latter category of direct contraction, and resembles to some degree GETT. COGENT is characterized by its approach to parameterized code generation: the mapping of software parallelism to hardware parallelism is based on static analysis of a representative problem instance, and hence postpones code generation until the problem instance is known. This differs from most other implementation (including ours, as we shall see), where typically all code versions are generated before-hand based on features of the source program alone, and later picked between only once the actual problem instance is known. Hence the benefit of COGENT is that it can generate kernels more tightly specialized to individual contractions, but generalization is not as simple as tweaking a set of parameters. In addition, the GPU is now the target, hence different techniques are used: Analysis of reuse directions in the iteration space of the three tensors (two operands, one result) are used to map parallellism in the given contraction to GPU parallellism in a fashion that best utilizes the GPU memory hierarchy.

3 Background

In this section, we introduce the theoretical background for the implementation, before diving into technical implementation details in section 4.

First, some basic theory on tensor contractions, GPU hardware and GPU programming, and the block/register tiling optimizatoin. Next, we introduce some of the concepts we will be working with in implementation, as well as those parts and intricacies of the Futhark compiler most pertinent to the implementation.

3.1 Terminology

We use CUDA terminology[7] in all discussions of GPU hardware and GPU code (with some addenda; see below), which we will assume the reader is familiar with. We then introduce the following additional terminology and abbreviations:

Tblock (CUDA) thread block (to disambiguate it from e.g. the "block" in block tiling).

Shmem (CUDA) shared memory.

Private memory (CUDA) register memory and local memory. See section 3.3.3.

FVI fastest varying index (wrt. n-dimensional arrays and loop nests).

FVTI fastest varying **thread** index (wrt. n-dimensional tblocks and thread indices).

Redomap map/reduce composition.

(GE)MM (generalized) matrix multiplication.

TC tensor contraction.

$\boldsymbol{\mathcal{X}}_{ijk}$ an upper-case and bold calligraphic symbol denotes a tensor; the subscript is its indices.

$\langle \dots \rangle$ angular brackets denote an ordered sequence (analogous to a list in code).

This terminology will be used throughout *the entire report*, not just this section.

3.2 Tensor theory: notation and contraction

We adopt and adapt notation and definitions used in [5] and [6].

The n'th order tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{d_1 \times \cdots \times d_n}$ can be seen as a higher-dimensional generalization of the matrix, and is characterized by its order and its n *modes* d_i for $i = 1 \dots n$, where the size of each d_i is called the *extent* of that mode, and where each is associated with a uniquely identifying name; hence the tensor order is its number of modes/indices. As an example, a 4th order tensor is akin to a 4-dimensional matrix of matrices, and may be denoted by:

$$\boldsymbol{\mathcal{X}}_{ijkl}, \tag{1}$$

where the indices $ijkl$ represent the 4 modes of the tensor – in a piece of code, we might represent $\boldsymbol{\mathcal{X}}_{ijkl}$ with a 4-dimensional array.

Please note that there is no *immediate* relation between the *name* and the *extent* of a mode; the names are primarily for identification (in terms of permutation), and two modes of equal extent will still have different identifying names.

3.2.1 Tensor contraction

One very common operation on tensors is the *tensor contraction* (TC), which we may intuitively understand as a higher-dimensional analog to the matrix product. product. Let $\boldsymbol{\mathcal{X}}_I$ and $\boldsymbol{\mathcal{Y}}_J$ be tensors of some sequences of modes I and J, and define

$$\begin{aligned} K &= I \cap J \supseteq \emptyset \\ L &= I \cup J \setminus K, \end{aligned} \tag{2}$$

s.t. K are the *contracted modes*, i.e. the set of modes common to $\boldsymbol{\mathcal{X}}_I$ and $\boldsymbol{\mathcal{Y}}_J$, and L the *free modes*, i.e. the union of modes unique to each operand tensor[1].

Then the contraction is a tensor of modes L and it is given by:

$$\boldsymbol{\mathcal{Z}}_L = \sum_K \boldsymbol{\mathcal{X}}_I \boldsymbol{\mathcal{Y}}_J. \tag{3}$$

In other words, $\boldsymbol{\mathcal{Z}}_L$ is the summation over repeated modes of the contraction operands, and its order is equal to the number of free modes in the contraction. As mentioned earlier, we may freely arrange the order of elements of L, and for two tensors to be eligible for contraction along a common mode, the extent of that mode must also be common among the operand tensors.

Since TC is summation over repeated modes, we typically use Einsum(-like) notation to express TCs. As an example, the matrix product $\mathbf{C} = \mathbf{AB}$ can be expressed as a special case of TC using two second order tensors $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ as such:

$$\boldsymbol{\mathcal{C}}_{ij} = \boldsymbol{\mathcal{A}}_{ik} \boldsymbol{\mathcal{B}}_{kj}. \tag{4}$$

3.2.1.1 Asymptotic complexity of TC

The TC is an asymptotically expensive operation. Whereas MM of square matrices grows "simply" with $\mathcal{O}(n^3)$, the contraction of hypercubic tensors[2] of K contracted modes and L free modes has complexity $\mathcal{O}(n^{|K|+|L|})$. As an example from [5], for the six-dimensional contraction of two order 4 tensors in CCSD(T), we have $|K| = 1$ and $|L| = 6$ and hence a complexity

[1] Note that whereas I and J are sequences to emphasize order, L is viewed as a set to emphasize the fact that we may freely arrange the layout of the contraction and until we do it is not ordered, while K is viewed as a set simply because all modes in K are logically contracted "simultaneously" (the notion of any temporal ordering of contractions does not really make sense in a purely mathematical context).

[2] Tensors with $d_i = d_j$ for all i, j.

of $\mathcal{O}(n^7)$ – granted, relatively small $n \leq 28$ were used in all experimental results presented in [5]. Granted – as also noted in [5] – asymptotic cost models can be ambiguous in regards to TCs due to high modality and low extent in most practical applications.

3.2.2 Tensor Terminology for ~~Dummies~~ Computer Scientists

As we quickly approach the implementation section, we will soon come to think of tensors as **multi-dimensional arrays in memory**, and hence we shall for the remainder of this report adopt a lingo more familiar to computer scientists for discussing tensors. Instead of *mode*/*modes* and *extent*, we use simply *index*/*indices* and *size*, and instead of tensor *order*, we shall refer to the *dimensionality* or *rank* of a tensor, even if the latter is already an established term in both tensor theory and linear algebra. Similarly, as we will eventually generalize code generation to arbitrary operators, we sometimes use *reduction* in place of *summation.*

3.3 GPU memory hierarchy

The GPU memory hierarchy is conceptually similar to that of the CPU, however with some key differences that the GPU programmer should be aware of in order to obtain good performance. In this brief and abstracted introduction to the GPU memory hierarchy, we use the **CUDA memory model** as a basis. Hence this introduction to the GPU memory hierarchy is based on the official CUDA programming guide[7].

The three important layers of the GPU memory hierarchy are *global memory*, *shared memory*, and *(thread-)private memory.*

3.3.1 Global memory and coalescing

Global memory is by far the largest and slowest layer, and holds memory shared among executing tblocks, as well as input data to kernel functions. Global memory has its own intermediate cache hierarchy which we shall not concern ourselves with. When accessing global memory, we are, however, interested in *global memory coalescing*, which, for our purposes, is analogous to cache locality on the CPU.

When threads in a warp access global memory, the GPU will attempt to combine those accesses into as few *coalesced* accesses as possible. A single global memory transactions issued by a warp involves 32, 64, or 128 contiguous (and, importantly, aligned) bytes, and if all threads in the warp access addresses within that range then the transaction is perfectly coalesced.

For our purposes, we will strive to have consecutive threads in a warp access consecutive accesses in global memory. Hence, to access global memory most efficiently in e.g. a loop, the per-iteration intra-thread stride should be unitary, while the per-thread intra-iteration stride may e.g. equal the size of the tblock.

As an example, the below copy of a 1D slice of size `TBLOCK_SIZE * T` size slice from and to arrays `glb_mem` and `shmem` is *uncoalesced*:

```
forall (tid = 0; tid < TBLOCK_SIZE; i++) // for each thread in the tblock
  for (i = 0; i < T; i++)
    shmem[tid * T + i] = glb_mem[offset + tid * T + i];
```

because neighbouring threads access memory T addresses apart in each loop iteration, while the below change obtains coalesced access:

```
forall (tid = 0; tid < TBLOCK_SIZE; i++) // for each thread in the tblock
  for (i = 0; i < T; i++)
    shmem[i * TBLOCK_SIZE + tid] = glb_mem[offset + i * TBLOCK_SIZE + tid];
```

since the per-iteration inter-thread stride is 1. Note that in the above two examples, the same slice of memory is copied and with the same layout in `shmem`, only each cooperating thread has been responsible for different elements.

3.3.2 Shared memory and bank conflicts

Shared memory is a layer of tblock-wide memory which can be used to share data among threads within a single tblock in e.g. cooperative copies and computations. The default size of shared memory is 48 KiB[3], and hence it should be considered a limited and precious resource. On the other hand, shmem accesses are orders of magnitude faster than global memory accesses.

Physically the memory is divided into 32 banks called *shared memory banks*, each holding multiple 32-bit words. Successive addresses map to successive banks (with wrap-around), and in a single transaction, a single 32-bit word of each of the 32 banks may be simultaneously read or written by the 32 threads in a warp. Since only a single word from each bank may be accessed in one transaction, multiple accesses to different addresses mapped to the same bank must be serialized, and this is called a bank conflict. Hence to use shared memory most efficiently, we must deliberately assert that threads in each warp do not cause bank conflicts. When we use (logically) multi-dimensional shared memory arrays, this can often be achieved by padding the size of the inner dimension, essentially skewing the inter-thread access pattern. This, of course, comes at the expense of overhead in shared memory usage, but the trade-off is typically worth the extra space, so long as it does not bar the kernel from launching for a given shared memory configuration which would otherwise be valid without padding.

Note that we may see bank conflicts even when the inner dimension (more precisely, the access stride between adjacent threads) is *larger* than 32. As an example, if adjacent threads access shared memory with a stride of, say, $m = 36$, then we get bank conflicts, as can be seen by examining the banks accessed by threads in a warp:

$$\Big\langle (\mathbf{tid} * 36) \bmod 32 \ : \ \mathbf{tid} \in \langle 0, 1, \ldots \rangle \Big\rangle = \big\langle \underbrace{0, 4, 8, 12, 16, 20, 24, 28}_{8-\text{cycle}}, 0, 4, 8, \ldots \big\rangle, \tag{5}$$

[3]Newer devices have *more*, but accessing >48 KiB requires dynamic allocation and a manual per-kernel opt-in[7]; Futhark-compiled binaries does this when necessary.

where $\langle \dots \rangle$ denotes an ordered sequence. Hence the banks accessed are on an 8-cycle for increasing **tid** when $m = 36$, and we get $32/8 = 4$-way bank conflicts for threads in the warp.

In any case, we can always avoid bank conflicts by choosing m to be odd. To see why, note that the only distinct prime factor of 32 is 2. This tells us that 32 must be coprime with any number not divisible by 2, which is the definition of an odd number. Since $m + 1$ is odd whenever m is even, we need only ever to pad with a single element to avoid conflicts.

3.3.2.1 Space overhead in multi-dimensional padding

We have seen that to avoid bank conflicts, we may pad the size of inner dimensions to an odd number. Note then that a product is odd **iff** all of its factors are odd. This implies that when we have multiple dimensions which need padding, we may in the worst case have to pad all of them (if they are all even to begin with, that is). This can quickly lead to space blowup if the number of padding dimensions is large.

To quantify this overhead, consider a sequence of shmem dimension sizes $\langle S_1, \dots, S_n \rangle$, where we assume $S_i \geq 1$ for all dimensions i, since otherwise the array is empty. Let j be the dimension indexed by the FVTI, and assume that the inner dimensions $\langle j+1, \dots, n \rangle$ *do* need padding, i.e. that the product $\prod_{i=j+1}^{n} S_i$ is even. We wish to derive a *lower* bound on the overhead incurred from padding (in the case where at least some padding is necessary, that is), so we assume that exactly one dimension $k > j$ needs padding.

The size of the array *after* padding can then be expressed as a product where the one term S_k is padded by substituting $(S_k + 1)$:

$$\begin{aligned}
\left[\prod_{i=1}^{k-1} S_i\right] \cdot \underbrace{(S_k+1)}_{\text{pad dim } k} \cdot \left[\prod_{i=k+1}^{n} S_i\right] &= \left[\prod_{i=1}^{n} S_i\right] \cdot \frac{1}{S_k} \cdot (S_k + 1) \\
&= \left[\prod_{i=1}^{n} S_i\right] + \left[\prod_{i=1}^{n} S_i\right] \cdot \frac{1}{S_k} \\
&= \left[\prod_{i=1}^{n} S_i\right] + \underbrace{\left[\prod_{i=1}^{k-1} S_i\right] \cdot \left[\prod_{i=k+1}^{n} S_i\right]}_{\text{overhead}}. \qquad (6)
\end{aligned}$$

Evidently the overhead incurred from padding can be large for large-rank arrays. In section 4.4.1, we discuss (and present our implementation of) a simple optimization to shmem overhead which can be made when the shmem tile is represented as a flat array.

3.3.3 Register memory, register spilling, and private memory

The smallest and fastest layer in the memory hierarchy are the thread registers, however, this is also the least predictable type of memory, since when the register allocator runs out of registers, it is free to (and will without warning) spill variables to so-called *local memory*, which is slow and off-chip memory comparable in speed to global memory. It can often be difficult to predict placement and to prevent register spilling, since the number of registers per thread is not fixed *per se*, but rather is just one variable in an equation including e.g. tblock size and the number of executing warps per multiprocessor, and the register allocator may lower the cap on registers if beneficial for occupancy. As programmers, the best we can do is to make conscious and deliberate choices in the code, especially when declaring thread-private arrays (e.g. register tiles), but it is also possible to aid the compiler by setting a per-kernel register cap *or* launch bounds (but not both). The Futhark compiler uses the latter, which better allows for user-controlled tiling parameters.

Hence we use the term *private memory* to mean memory which can be – and which we would therefore ideally see – allocated to register memory, but for which we are technically not in control of placement.

3.3.4 Collective copying

As explained, in order to obtain efficient access to global memory, adjacent threads in a warp should ideally access adjacent memory locations. However, some GPU algorithms might require individual threads to process consecutive elements from global memory. Rather than accessing global memory inefficiently, threads in a tblock can perform a *collective copy* using shared memory as a staging buffer, after which threads may retrieve from shared memory the required elements using a strided access pattern, which shared memory does not penalize [7].

3.4 Block/register tiling transformation

One of the GPU code transformation techniques we will be employing is called *block/register tiling*. In this section we give a step-by-step walkthrough of the transformation of a regular MM program, since the transformation is difficult to exemplify in pseudocode for arbitrary TCs, but the technique generalizes to higher dimensions. The theory behind is based primarily on [8], with some GPU hardware specifics derived from [9], and some parts of the step-by-step walkthrough lifted from the author's own work in [3].

The walkthrough is rather lengthy and may be read cursorily.

3.4.1 Loop stripmining and normalization

Loop stripmining is a loop transformation which splits a size N iteration space of some loop (parallel or otherwise) into Q chunks of size N/Q, using an inner loop of size Q to "fill in the gaps", as in example 1. Q is called a *stripmining factor*, but we shall for the remainder of the report call it a *tile parameter*, and Q is typically used to denote a tile parameter for a sequential dimension, whereas a parallel dimension tile parameter is denoted by T.

If Q divides N, then the transformation is always safe to perform without additional boundary guards. We then prefer to normalize all resulting loops. A normalized loop is one which starts at index 0 and goes with a 1-stride up to some immutable upper bound invariant to the loop variable. Normalized loops can promote loop unrolling, but in terms of the block/register

```
for (i = 0; i < N; i++)            ⇒    for (ii = 0; ii < N; ii += Q)
  f(i);                                 for (i = ii; i < min(N, ii + Q); i++)
                                          f(i);
```

Example 1: Stripmining of a simple for loop by some factor Q.

```
for (ii = 0; ii < ceil(N / Q); ii++)
for (i = 0; i < Q; i++)
  if (ii * Q + i < N)
    f(ii * Q + i);
```

Example 2: Normalization of the loops in the loop nest in the RHS of example 1.

tiling transformation, normalization is necessary to obtain a clear mapping of software parallelism to hardware parallelism. Example 2 shows the normalization of stripmined loops.

3.4.2 Block tiling and block/register tiling

Block tiling is the method of stripmining multiple consecutive innermost loops in a perfect loop nest and interchanging *inwards* the resulting stride-1 loops. The transformation is safe whenever it would be safe to interchange the loops pre-stripmining. Example 3 shows a simple MM program, while example 4 shows what the program looks like after the block tiling transformation.

In some cases, block tiling can be used to optimize spatial locality, i.e. if under the original traversal of the iteration space the program exhibits bad access patterns (e.g. uncoalesced access). This can be identified with access pattern analysis. In loop nests with significant data reuse, block tiling can improve spatial locality: If multiple iterations access the same memory, then we can rearrange the iteration space and improve temporal locality by moving (the execution of) those iterations closer together in time. Data reuse can be identified by inspection of variance in the loop nest (if an array is invariant to a dimension, then there is reuse).

In terms of TC, block tiling is primarily an optimization to temporal locality due to the high degree of data reuse in most TCs, but it will also improve spatial locality in many cases, since it enables efficient (and coalesced) reads of tiles from global memory in all cases (given proper tiling parameters).

To further improve temporal locality in code with heavy data reuse, we can employ the register layer of the memory hierarchy and perform register tiling on top of block tiling. The idea is to further stripmine some (perhaps already stripmined) outer parallel dimension, and then to interchange inwards and *sequentialize* the resulting loop, such that its iteration space can be mapped to a single thread. We call the tiling parameter a *register tile parameter*, and denote it R to distinguish it from T tiles, whose resulting loops remain parallel, and from Q tiles, which tile inherently sequential dimensions, e.g. reduction dimension(s) in a TC or MM. Example 5 shows the program after this final stripmining and interchange, but *before* sequentializing the R-tiled loops and hence *before* the actual register tiling.

A number of steps remain before the code can be mapped to the GPU. At this point, we need

```
forall (a = 0; a < Na; a++)
  forall (b = 0; b < Nb; b++)
    // redomap
    acc = 0;
    for (q = 0; q < Nq; q++)
      acc += X[a, q] * Y[q, b];
    Z[a, b] = acc;
```

Example 3: Simple program implementing $\mathcal{Z}_{ab} = \mathcal{X}_{aq} * \mathcal{Y}_{qb}$, i.e. MM. The outer two `forall`-loops are parallel, while the innermost is *not* due to a RAW hazard on `Z[a, b]`.

```
forall (aa = 0; aa < Na; aa += Ta)
  forall (bb = 0; bb < Nb; bb += Tb)

    forall (a = aa; a < min(Na, aa + Ta); a++)
      forall (b = bb; b < min(Nb, bb + Tb); b++)

        // redomap
        acc = 0;
        for (qq = 0; qq < Nq; qq += Qq)
          for (q = qq; q < min(Nq, qq + Qq); q++)
            acc += X[a, q] * Y[q, b];
        Z[a, b] = acc;
```

Example 4: Block tiling of the program in example 3. Parallel loops each tiled with a T tile, and sequential loop tiled with a Q tile; parallel stride-1 loops interchanged inwards. No loop normalization.

to distribute all but the outermost two loops over the redomap, i.e. the initialization of acc, the accumulation step, and the update of Z. Example 6 shows this transformation, and concludes block/register tiling. The transformation promotes both spatial locality and sequentialization, since each thread now performs an amount of extra sequential work equal to the register tile size, which can also be beneficial in case of oversaturation.

3.4.3 Mapping block/register tiling to hardware

The pseudocode shown in example 6 is still missing some transformation before it can be implemented in GPU code. First, we would normalize and unroll the innermost stride-1 loops s.t. each thread can store a size $R_a \times R_b$ register tile in private memory, and to remove some of the overhead in the loops.

Then, in order to enforce efficient reuse, instead of reading directly from X and Y in the redomap accumuation step, we instead insert copies of tiles from arrays in global memory to shared memory at the start of the qq loop, whence we can read in the accumulation step and save a large factor of global memory reads. These copies can be implemented in general as described in section 3.5.1.

```
forall (aaa = 0; aaa < Na; aaa += Ta * Ra)
  forall (bbb = 0; bbb < Nb; bbb += Tb * Tb)

    forall (aa = aaa; aa < min(Na, aaa + Ta * Ra); aa += Ra)
      forall (bb = bbb; bb < min(Nb, bbb + Tb * Rb); bb += Rb)

        forall (a = aa; a < min(Na, aa + Ra); a++);
          forall (b = bb; b < min(Nb, bb + Rb); b++);
            // redomap
            acc = 0;
            for (qq = 0; qq < Nq; qq += Qq)
              for (q = qq; q < min(Nq, qq + Qq); q++)
                acc += X[a, q] * Y[q, b];
            Z[a, b] = acc;
```

Example 5: Further stripmining of the parallel loops in in example 4. Here, the innermost aa and bb loops are additionally tiled with an R tile. The resulting stride-1 loops are interchanged inwards (note: not innermost) but remain parallel for the moment.

Finally, we map the parallel `forall` loops to GPU hardware as per the comments in example 6 – in general, the outermost parallel loops tiled with both a T and an R tile will be mapped to the GPU grid, while the innermost parallel loops tiled singly with a T tile are mapped to the tblock, and for a contraction of k free indices, we have a k-dimensional grid and tblock.

The flat grid and tblock sizes can be described by:

$$\begin{aligned} \text{gridSize} &= \prod_{i \in D} T_i, \\ \text{numTblocks} &= \prod_{i \in D} \left\lceil \frac{N_i}{T_i R_i} \right\rceil, \end{aligned} \tag{7}$$

where D is the sequence of outer (i.e. non-reduction) dimensions in the TC, and this tells us that for high-rank TCs, some (or most) tile parameters will necessarily be set to 1 (or close) due to hardware constraints on tblock size, shared memory per tblock, and other resources available. For some segspace dimension i, setting $R_i = 1 < T_i$ then corresponds to fully parallelizing dimension i on the tblock, while setting $T_i = 1 < R_i$ corresponds to mapping i on the grid with a factor R_i of sequentialization, while setting $T_i = R_i = 1$ corresponds to fully mapping dimension i onto the grid.

3.4.4 Choice of tiling parameters

When choosing tiling parameters, it is often beneficial to choose as large values as will fit in the working set(s) (e.g. shared memory or the amount of thread registers), in order to promote reuse as much as possible. However, for the GPU in particular, this is not always the case. Because tile sizes are usually dependent on the size of the tblock, shared memory, and

```
forall (aaa = 0; aaa < Na; aaa += Ta * Ra)   // outer grid
  forall (bbb = 0; bbb < Nb; bbb += Tb * Tb) // inner grid

    acc[Ta][Tb][Ra][Rb]; // array expansion of acc
    forall (aa = aaa; aa < min(Na, aaa + Ta * Ra); aa += Ra)   // outer tblock
      forall (bb = bbb; bb < min(Nb, bbb + Tb * Rb); bb += Rb) // inner tblock
        for (a = aa; a < min(Na, aa + Ra); a++)   // unroll
          for (b = bb; b < min(Nb, bb + Rb); b++) // unroll
            // initialize accumulator
            acc[aa-aaa, bb-bbb, a-aa, b-bb] = 0;

    forall (aa = aaa; aa < min(Na, aaa + Ta * Ra); aa += Ra)   // outer tblock
      forall (bb = bbb; bb < min(Nb, bbb + Tb * Rb); bb += Rb) // inner tblock
        // redomap accumulation
        for (qq = 0; qq < Nq; qq += Qq)
          for (q = qq; q < min(Nq, qq + Qq); q++)     // unroll
            for (a = aa; a < min(Na, aa + Ra); a++)   // unroll
              for (b = bb; b < min(Nb, bb + Rb); b++) // unroll
                acc[aa-aaa, bb-bbb, a-aa, b-bb] += X[a, q] * Y[q, b];

    forall (aa = aaa; aa < min(Na, aaa + Ta * Ra); aa += Ra)   // outer tblock
      forall (bb = bbb; bb < min(Nb, bbb + Tb * Rb); bb += Rb) // inner tblock
        for (a = aa; a < min(Na, aa + Ra); a++)   // unroll
          for (b = bb; b < min(Nb, bb + Rb); b++) // unroll
            // redomap result write-back
            Z[a, b] = acc[aa-aaa, bb-bbb, a-aa, b-bb];
```

Example 6: Final block/register tiled MM program, obtained from example 5 via distribution of parallel loops of indices aa, bb, a, b over the inner redomap, including array expansion of acc, followed by sequentialization of the a and b loops. Comments show a possible mapping of software parallelism to GPU hardware parallelism, and which sequential loops may be unrolled.

the number of registers used per thread, the choice of tile parameters must conform with the hardware bounds on all three of these *simultaneously*. Further, even if all three bounds are respected, there can often be a significant benefit to occupancy in decreasing (or, for that matter, *increasing*) some tile parameters. This is one of many factors which makes static selection of tile parameters difficult.

3.5 LMADs for describing tiles and copies

We want some generic and simple way to express and work with slices of memory for when we will eventually be copying arbitrary-rank tiles from global to shared memory. The LMAD (**l**inear **m**emory **a**ccess **d**escriptor) is an interesting concept with many uses, one of which we can use to generically describe *and copy* tiles. This condensed and somewhat simplified summary of LMADs is based primarily on [10], but the original paper [11] gives a good introduction to the uses of LMADs in the Futhark compiler.

The LMAD for an n-dimensional slice of some n-dimensional array (i.e. each dimension is indexed fully) is defined by a flat offset (in number of elements) into that array at which the slice starts, and, for each of the n dimensions, the size of the slice along that dimension and the stride between elements in that dimension (i.e. the distance, measured in number of elements in the flat representation, between two elements adjacent in that dimension):

$$\mathcal{L} = \Big(\tau,\ \big\langle\sigma_1, \ldots, \sigma_n\big\rangle,\ \langle\delta_1, \ldots, \delta_n\rangle\Big), \tag{8}$$

where τ is the flat offset, and σ_i and δ_i are the size and stride for the i'th dimension. Note that τ, σ_i, and δ_i are all positive integers. Each LMAD $\mathcal{L}$ describes a set of points in 1D space, which we may express mathematically by the formula:

$$\text{points}\big(\mathcal{L}\big) = \Big\{\tau + i_1\delta_1 + \cdots + i_n\delta_n\ \Big|\ 0 \le i_1 < \sigma_1,\ \ldots,\ 0 \le i_n < \sigma_n\Big\}, \tag{9}$$

However, we shall eventually think of LMADs as index functions (see section 3.5.1). Returning to eq. (8), we may describe the read of a tile of size $T_a \times T_b \times T_c$ from some larger array $\boldsymbol{\mathcal{X}}$ of size $N_a \times N_b \times N_c$ (with $T_i \le N_i$ for each i) starting at offset π, with the LMAD:

$$L_{\boldsymbol{\mathcal{X}}} = \Big(\pi,\ \big\langle T_a,\ T_b,\ T_c\big\rangle,\ \langle N_b \cdot N_c,\ N_c,\ 1\rangle\Big). \tag{10}$$

Conversely, if, say, the *destination* of that copy is an array $\boldsymbol{\mathcal{X}}_{\text{shr}}$ (in e.g. shared memory) of size $T_a \times T_b \times T_c$ – i.e. exactly equal to the tile size – then we may describe the write destination slice of memory with:

$$\mathcal{L}_{\boldsymbol{\mathcal{X}}_{\text{shr}}} = \Big(0,\ \big\langle T_a,\ T_b,\ T_c\big\rangle,\ \langle T_b \cdot T_c,\ T_c,\ 1\rangle\Big). \tag{11}$$

Since the memory slices we will be describing with LMADs and subsequently copying are all hypercubic tiles, traditional triplet-notation would be sufficient, however since Futhark IR uses LMADs for all index functions, so shall we in reasoning about copies in our program.

3.5.1 Generic LMAD copying

As stated, we wish to use LMADs to easily and generically generate code for copying slices from global to shared memory. The interesting thing with LMADs is that each LMAD also defines a corresponding index function for accessing the associated array. For example, the index function corresponding to the LMAD in eq. (10) is a 3-ary function given by:

$$\begin{aligned} L_{\boldsymbol{\mathcal{X}}}(a, b, c) &= \pi + a \cdot N_b \cdot N_c + b \cdot N_c + c \cdot 1 \\ &= \pi + (a \cdot N_b + b) \cdot N_c + c. \end{aligned} \tag{12}$$

We may find it useful to define a unary index function $L'_{\boldsymbol{\mathcal{X}}}$ which can be used to easily and uniquely map values from a 1D index space – say, a flat thread index, **tid** – to flat indices into

```
1  function lmad_copy_tile(src_arr, dst_arr, src_lmad, dst_lmad) {
2    dims = dst_lmad.dims;
3    tile_size = product(dims);
4
5    for (i = tid; i < tile_size; i += TBLOCK_SIZE) {
6      inds = unflatten(i, dims);
7
8      src_ind_flat = src_lmad.offset;
9      dst_ind_flat = dst_lmad.offset;
10     for (k = 0; k < len(dims); k++) {
11       src_ind_flat += inds[k] * src_lmad.strides[k];
12       dst_ind_flat += inds[k] * dst_lmad.strides[k];
13     }
14     dst_arr[dst_ind_flat] = src_arr[src_ind_flat];
15   }
16 }
```

Listing 1: Pseudocode for parallel LMAD copy of arbitrary-rank tiles. We assume that dimensions of the two LMADs are of equal rank and size. Global memory boundary guards omitted for brevity.

$\boldsymbol{\mathcal{X}}$. To do so, we first *unflatten* **tid** wrt. the LMAD dimensions by:

$$\begin{aligned}\mathbf{tid}_a &= \mathbf{tid}/(T_b \cdot T_c)\\ \mathbf{tid}_b &= (\mathbf{tid} \bmod (T_b \cdot T_c))/T_c\\ \mathbf{tid}_c &= (\mathbf{tid} \bmod (T_b \cdot T_c)) \bmod T_c,\end{aligned} \tag{13}$$

and then, define L' by:

$$L'_{\boldsymbol{\mathcal{X}}}(\mathbf{tid}) = \pi + (\mathbf{tid}_a \cdot N_b + \mathbf{tid}_b) \cdot N_c + \mathbf{tid}_c, \tag{14}$$

with $\mathrm{dom}(L'_{\boldsymbol{\mathcal{X}}}) = \{0, 1, \dots, T_a T_b T_c - 1\}$. This is useful, because (spoiler alert) it means that we can use a k-dimensional thread block to easily and generically read d-dimensional tiles of global memory, for arbitrary k and d, by flattening the thread block index and applying L'. Neat!

3.5.1.1 Efficient parallel implementation of LMAD copying

Listing 1 shows pseudocode for an implementation of a parallel and generic LMAD copy of tiles for intra-tblock execution. The outer loop is called a *virtualization loop*, and handles both the case where the tblock is larger than the tile, as well as the converse. Note that if all LMAD dimensions are compile-time constant, then we obtain an efficient implementation by unrolling the product, the unflattening of indices, and the inner loop (lines 3, 6, and 10-13, respectively). If one or both arrays are statically sized, e.g. a shmem array, then strides can also be constant folded, and further, if the destination LMAD dims and tblock size are statically known, then the outer loop can be normalized and unrolled.

```
 1  def aqc_qb_cba [a][b][c][q] (X: [a][q][c]f32) (Y: [b][q]f32) : [c][b][a]f32 =
 2    let xsss: [c][a][q]f32 = transpose (map transpose X)
 3    let yss = Y
 4    in
 5      map (\xss ->
 6        map (\ys ->
 7          map (\xs ->
 8            map2 (*) xs ys |> f32.sum -- sequential redomap.
 9          ) xss
10        ) yss
11      ) xsss
```

Listing 2: Futhark code for the TC $\boldsymbol{\mathcal{Z}}_{cba} = \boldsymbol{\mathcal{X}}_{aqc}\boldsymbol{\mathcal{Y}}_{qb}$. The outer map nest (lines 5-7) constitutes the kernel segspace (assuming the inner `map2` is fused with the reduction and thus not part of the segspace). The redomap in line 8 is sequential. Note the explicit rearrangement of first operand tensor (line 2) in order to place the reduction dimension innermost, and to allow conforming with the desired segspace dimensions.

3.6 Futhark compiler background

In this section, we give *brief* and *very high-level* descriptions of those modules and features of the Futhark compiler which directly pertain to our implementation, as well as some interesting subtleties, the knowledge of which may come in handy in implementation.

The implementation will take place entirely in the loop tiling pass of the optimization stage of the compiler.

3.6.1 Segspaces and array layout rearrangement

Listing 2 shows a Futhark implementation of the TC $\boldsymbol{\mathcal{Z}}_{cba} = \boldsymbol{\mathcal{X}}_{adc}\boldsymbol{\mathcal{Y}}_{db}$. The three outer maps in lines 4-6 constitute what we call the kernel *segspace*, and their ordering is important, because they determine the layout of the result. In this case, the desired output layout is `[c][b][a]`, and so the maps are nested in precisely that order. If e.g. we were to interchange the outer two maps in the source code, then the segspace dimensions, and hence the result (size) type, would become instead `[b][c][a]`. Additionally, since the inner redomap requires 1D slices, the reduction dimension must be innermost on either operand array.

To facilitate this ordering in the map nest, it can sometimes be necessary to rearrange the layout of one or more input arrays. For example, in listing 2, we have to rearrange `xsss'`, since the desired result layout requires that its `[c]` dimension comes before its `[a]` dimension, and because the reduction dimension `[q]` must be interchanged inwards[4]. Alternatively, we can get the slices we need using explicit indexing in maps over `iota` arrays, but this is generally

[4]Such a rearrangement of k dimensions is always possible using $\mathcal{O}(k^2)$ transpositions. A transposition at level i corresponds to a swap of indices i and $(i + 1)$, so we can obtain the sequence of swaps which produce a rearrangement by e.g. bubble sorting its inverse permutation. For an example, see `tc_gen.py` at `github.com/sortraev/msc_thesis_public`.

considered an anti-pattern in Futhark since it inhibits certain optimizations, and is in most cases avoidable.

A composition of (possibly nested) transpositions can be represented as a single `rearrange` in the Futhark IR[5]. When an array is rearranged, its LMAD is transformed accordingly, and the new layout may or may not be manifested in memory, depending on the results of other compiler passes/stages (but, as we shall discuss in implementation, section 4, the strategy we implement assumes that it is *not*).

3.6.2 The loop tiling pass

As previously mentioned the compiler already performs block/register tiling of (GE)MM-like expressions in the source code – the implementation module is called `BlkRegTiling`, and we shall henceforth use this name in reference to the current implementation. For more in-depth descriptions of the original design of and strategies used in `BlkRegTiling`, see [3].

The module is part of the loop tiling pass, and this is also where the new implementation will take place. The loop tiling pass is one of the first passes in the GPU-specific optimization pipeline, and follows a number of standard pipeline passes (simplification, CSE, dead code removal, SOAC fusion, etc.), kernel extraction, and the `optimiseGenRed` pass, which transforms a generalized reduction into (amongst other things) a map nest with a tileable redomap inside.

At this point, the available information includes a `KernelBody`, whose kernel statements includes the redomap[6], which in turn holds load statements for the input array slices, as well as map and reduction lambdas for the redomap. The `KernelBody` is itself carried in a `SegMap`, which also holds information about the execution segspace and the kernel result. From each load statement in the redomap, we may – directly or indirectly – access information about the base array whence the redomap operand slice comes, including its dimensions, LMAD information (if any; this information exists only if the array layout has been rearranged), and the segspace dimensions on which the array is variant.

3.6.3 Futhark multi-versioning

Note: Since we will only be generating a single kernel, the ins and outs of Futhark multi-versioning are not particularly important for the implementation, but since the original inspiration for this project was to explore the possibilities of implementing the COGENT strategy (as presented in [6]), which can be viewed as a sort of multi-versioning strategy (see section 3.7), it is *important to at least have a sense of how Futhark does multi-versioning. This section may be read cursorily.*

For a given Futhark source program, the Futhark compiler will often generate multiple different (GPU) code versions. The compiled binary will then *at runtime* choose the code version

[5]A source language counterpart would be very convenient in this caes, but such a function would require rank polymorphism, which is not currently supported.

[6]More specifically, it holds a so-called *screma* (**sc**an/**re**duce/**ma**p combination) which, if the scan part is null, corresponds to a redomap.

best suited for the given problem instance. Between 2-6 different code versions are typically generated for most programs.

Many such code versions are generated by *incremental flattening* [12], a rather sophisticated, interesting, and novel compiler transformation developed specifically for Futhark. In broad and oversimplifying terms it means that for a given source program, the compiler will generate multiple code versions based on the (regular) nested parallelism in the source program and its possible mappings to hardware parallelism. As an example, the kernel produced by `BlkRegTiling` is called an *intra-tblock* (segmap) kernel because it operates at the intra-tblock level (as will the new kernel we eventually generate; see section 4), and this particular kernel is chosen whenever there is sufficient *outer parallelism* in the input data (in this case, the "outer parallelism" is the two parallel dimensions on top of the redomap, corresponding to the dimensions of the result product).

The important takeaway from this is that even though different code versions are made to better suit different (representative) problem instances, their generation is based on levels of parallelism in the source program, and *not* on any particular (representative) problem instance. This is, of course, because the kernel is generated at *Futhark source* compile time, while the problem instance is not known until host code runtime (more specifically, between the time of kernel *generation* and kernel *compilation*).

3.6.4 The `BlkRegTiling` module

One of the IR code versions for a regular GEMM or GEMM-like source expression is one in which the outer two parallel dimensions are mapped to hardware parallelism and the inner redomap is kept sequential and mapped to threads. This IR code version can then be 2D block/register tiled according as described in section 3.4, and this is precisely what the current `BlkRegTiling` module performs.

More specifically, `BlkRegTiling` will perform 2D block/register tiling of GEMM-like IR expressions loosely fitting the pattern of this pseudocode:

```
map (\xs ->
  map (\ys ->
    let redomap_res = redomap xs ys
    -- code2 is a sequence of expressions variant on
    -- redomap_res, but on which redomap_res is invariant.
    let code2 = ...
    ...
    in code2
  ) yss
) xss
```

where `xss` and `yss` are both 2D arrays; `redomap` is some map/reduce composition; and `code2` is some sequence of expressions variant on the redomap result, but on which the redomap is not variant. Note that `code2` may contain expressions which precede the redomap *syntactically* (i.e. in source code), but not logically (i.e. in terms of variance).

We concretize the pattern in a set of *firing conditions* for the expression:

1. the expression has a redomap (quasi-)perfectly nested inside two outer parallel dimensions on which the result is variant, possibly surrounded by `code2`, i.e. expressions variant on the redomap result(s) but on which the redomap is invariant;
2. the redomap takes exactly two 1D array slices as input, each of which is variant to exactly one of the two outer parallel dimensions on which the result is variant; and
3. for each of the two parallel dimensions on which the result is variant, there is exactly one redomap input array variant to this dimension.

Condition 2 implies that each redomap array is a slice of some larger base array (the dimension on which it is variant is the dimension which indexes this array), while conditions 2 and 3 in conjunction ensure an opportunity for data reuse optimization. A small handful of additional conditions apply, such as restrictions on data types, but these are not particularly relevant to our implementation.

The `BlkRegTiling` module then produces GPU code with structure similar to that of the pseudocode in example 6, with the addenda in section 3.4.3.

3.6.4.1 Relevant features of and optimizations in `BlkRegTiling`

The existing `BlkRegTiling` produces efficient code for MM expressions, so we would do good to examine the techniques used here. The module makes a number of smaller optimizations, including splitting the main reduction loop into a prologue and an epilogue in order to remove a number of boundary checks in the prologue, and padding shmem to prevent bank conflicts. This padding is implemented largely as described in section 3.3.2, and the implementation code is simple since it deals only with 2D arrays. Padding is *always* used, meaning bank conflicts may accidentally be introduced if tile parameters were already chosen s.t. the inner dimension size was odd (which is very unlikely but valid nonetheless).

But perhaps the most interesting feature about `BlkRegTiling` – at least as far as our implementation goes – is its method for copying tiles from global to shared memory. Whereas in our implementation the kernel will be reading arbitrary-rank tiles using generic LMAD copies as discussed in section 3.5.1, `BlkRegTiling` can use a more efficient implementation: Because the tblock and both of the tiles are always 2D, it can map the read of each 2D tile directly onto the 2D tblock without logically reshaping the tblock dimensions, which, as we shall see in benchmarking (section 6.3), saves significant overhead.

3.7 COGENT strategy wrt. Futhark

This section based entirely on [6] and [5].

Given a TC expression and a representative problem instance, COGENT generates efficient GPU kernel code for the TC tailored specifically to the given and similar problem instances. It does so by a static estimation of the optimal mapping of software parallelism to hardware parallelism, based on a model-driven pruning of the configuration space of different such mappings, and a cost model for the data movement needed for a given mapping[6]. Furthermore, it performs fusion of TCs where applicable, but this is not relevant to our project since we will not be looking into fusion.

While COGENT *does* generate multiple kernels, these kernels follow the same general structure and differ only in their handling of partial tiles. These variants handle: No partial tiles, partial tiles in the reduction dimension, partial tiles in one or more outer dimensions, and partial tiles in the reduction and one or more outer dimensions.

However, whereas Futhark generates all code versions based on the source program and offers additional specialization via tuning parameters (e.g. tile parameters) at runtime, COGENT requires knowledge of the problem instance in order to run the pruning model. This is well-suited for e.g. long running programs with a fixed problem instance, such as in a tensor network, but for programs with varying inputs, COGENT would effectively function as an interpreter generating kernels on the fly, and this may or may not impose a significant overhead (although we cannot say to what extent a singly generated COGENT kernel could generalize to other representative problem instances).

In any case, we quite early in the project decided *not* to go ahead with exploring how to implement the COGENT method in Futhark, since generating new kernels at (host code) runtime would essentially require embedding (a port of) COGENT inside Futhark binaries, which is not in line with the Futhark model, where all code is generated at compile time and parameterized at (host code) runtime.

On the other hand, there might very well still be benefit to be had in exploring how similar methods could be implemented inside the Futhark compiler, which might then be used to generate a smaller handful of COGENT-like TC kernels at *compile* time, or perhaps there might be inspiration to be had in regards to implementing (model-driven) autotuning of tile parameters in Futhark – but this will remain future work.

4 Implementation

In this section, I present my plan for implementation, including a detailed description of the chosen strategy for code transformation/generation and some of the lower-level choices made, and discuss some of the challenges and hurdles I met (and, in most cases, overcame) during implementation.

Listing 3 shows a Futhark implementation of (a generalization of) the TC:

$$\mathcal{Z}_{bickja} = \mathcal{X}_{jqai} * \mathcal{Y}_{bcqk}, \tag{15}$$

which will be used as a running example throughout the implementation sections.

For the purposes of design and experimentation, as well as a point of reference to eventually compare the generated code with, we begin the implementation process with a prototype kernel in CUDA/C++. The prototype also implements the TC in eq. (15) and resembles largely the pseudocode in listing 5 (presented shortly), but implements also some of the optimizations and tweaks we eventually implement in the compiler (except for the optimization discussed in section 4.5). Conversely, the prototype uses no optimizations that the Futhark compiler could not employ. However, the actual prototype code presents no insight on its own, and besides this it is much too long to include in this report, and so we instead refer to our appendices at `github.com/sortraev/msc_thesis_public`.

4.1 Limitations

Before delving into implementation details, we first account for the following limitations in our implementation:

Limitation 1 like `BlkRegTiling`, we assume exactly two redomap input arrays in the TC expression.

Limitation 2 we assume TCs with only a single redomap dimension (in other words, only one contracted index), for the simple fact that support for this is outside the scope of this project.

Limitation 3 similar to `BlkRegTiling`, we do not support 1D redomap arrays, hence e.g. matrix/vector products and tensor/vector contractions are not supported.

Limitation 4 our implementation assumes size type parameters on operand tensors in a TC expression are unique, even if two dimensions are equal

Limitation 5 due to time constraints, we do not implement support for `code2`, i.e. any additional scalar statements in the segspace on which the redomap is independent, nor support for additional outer parallel dimensions on which all or none of the redomap arrays are variant.

Limitation 6 due to a bug out of scope of our project, we are forced to assume that redomap arrays are rearranged using only compositions of transpositions, s.t. the layout rearrangment can be expressed with a single `rearrange` in the IR (see section 4.7.2.2).

```
1  def jqai_bcqk_bickja 't_x 't_y 't_z [b][i][c][k][j][a][q]
2    (redomap: [q]t_x -> [q]t_y -> t_z)
3    (X: [j][q][a][i]t_x)
4    (Y: [b][c][q][k]t_y)
5    : [b][i][c][k][j][a]t_z =
6    let xssss: [i][j][a][q]t_x =
7      X
8      |> map transpose
9      |> map (map transpose)
10     |> map transpose
11     |> transpose
12   let yssss: [b][c][k][q]t_y = map (map transpose) Y
13   in
14     map (\ysss ->
15       map (\xsss ->
16         map (\yss ->
17           map (\ys ->
18             map (\xss ->
19               map (\xs ->
20                 redomap xs ys
21               ) xss
22             ) xsss
23           ) yss
24         ) ysss
25       ) xssss
26     ) yssss
```

Listing 3: Futhark implementation of the TC in eq. (15), except here the contraction is generalized to arbitrary redomaps. We can specialize this function to perform regular 32-bit float TC by setting `redomap = \xs ys -> map2 (*) xs ys |> f32.sum`. Note in particular the explicit rearrangements of input arrays to conform with the desired result dimensions (lines 6-12) – these rearrangements exist primarily as logical rearrangements in the IR, and are only manifested in the generated GPU code should the compiler deem them beneficial. For more on this, see section 4.7.2.1.

Note that we cannot guarantee that this list of limitations is exhaustive. All of these limitations could in principle be lifted – see section 6.3.

4.2 General code generation strategy

As explained in section 3.7, the strategy employed by COGENT is not suitable for implementation in Futhark, since we wish to generate a single kernel per TC expression, which can then be parameterized to suit different problem instances. Instead, in order to generate a generic kernel, the strategy we settle on is to generalize the method used in `BlkRegTiling` to arbitrary dimensions.

Listing 5 shows pseudocode for the kernel we wish to implement, albeit with some abstractions which we shall disambiguate as we describe their implementation in section 4.7, but we first give a broad overview in this section.

TCs will be fully block/register tiled, meaning *all* dimensions in the segspace will be tiled with a T and an R tile, while the (singular, as per the restrictions; see section 4.1) redomap dimension is tiled once with a Q tile. As explained in section 3.4.3, a contraction of n free indices implies an n-D thread block and an n-D grid of tblocks (similarly, it implies an n-D segspace in the IR), but most of these dimensions will likely be unit (or very small) when n is large due to hardware constraints on tiling parameters.

Whereas e.g. COGENT uses a model to determine an efficient mapping of software parallelism to hardware parallelism, our model is significantly simpler: The n-dimensional segspace in the TC expression is mapped directly to the n-dimensional tblock, s.t. the i'th innermost dimension of the segspace is indexed by the i'th innermost thread index. This for example ensures coalesced writes to the result tensor, but the main motivator is simplicity.

4.2.1 Shmem tiles and copying from global to shmem

The prototype will use LMAD copies to move tiles from global to shared memory, as outlined in section 3.5.1, since this enables easy and generic code generation for efficiently copying arbitrary dimension tiles. This method also ensures coalesced reads from global memory, so long as tile sizes are chosen appropriately. Hence implementation of the `lmad_copy_tile` function used in lines 20-21 of listing 5 can be based on listing 1.

For the LMADs describing a tile from a given input array, the size of each LMAD dimension will match the tile parameters tiling that dimension, i.e. T_iR_i for some parallel dimension i, or Q_q for the sequential dimension q, and the layout of LMAD dimensions will reflect those of the base array. The strides for each LMAD are determined similarly, but depend, of course, on the underlying array.

As an example, for the first operand array in the TC in listing 3, $\boldsymbol{\mathcal{X}}$, the two LMADs describing reads/writes of tiles are:

$$\begin{aligned} L_{\boldsymbol{\mathcal{X}}} &= \Big(\pi,\ \big\langle T_jR_j,\ Q_q,\ T_aR_a,\ T_iR_i\big\rangle,\ \langle N_qN_aN_i,\ N_aN_i,\ N_i,\ 1\rangle\Big), \\ L_{\boldsymbol{\mathcal{X}}_{\mathsf{shr}}} &= \Big(0,\ \big\langle T_jR_j,\ Q_q,\ T_aR_a,\ T_iR_i\big\rangle,\ \langle Q_qT_aR_aT_iR_i,\ T_aR_aT_iR_i,\ T_iR_i,\ 1\rangle\Big). \end{aligned} \tag{16}$$

where π is the flat offset into $\boldsymbol{\mathcal{X}}$ of the start of the particular tile.

4.2.2 Register tile, accumulation, and write-back

Initialization of register tiles, the redomap accumulation step, and the final write-back of register tiles will be implemented quite straight-forwardly from the pseudocode given in example 6, except, of course, now generalized to arbitrary-rank register tiles.

Listing 4 shows pseudocode for the abstract `redomap_accumulate` function, specifically for the TC in eq. (15) – code generation for `init_reg_tile` and `write_reg_tile`, the initialization and final write-back of register tiles, is similar.

```
function redomap_accumulate(thd_reg_tile, s_A, s_B, Q, R_tiles) {
  (Rb, Ri, Rc, Rk, Rj, Ra) = R_tiles
  // outer loop based on the Q tile.
  for (q = 0; q < Q; q++)
    // inner loop nest based on R tiles, which come from the segspace.
    for (b = 0; b < Rb; b++)
    for (i = 0; i < Ri; i++)
    for (c = 0; c < Rc; c++)
    for (k = 0; k < Rk; k++)
    for (j = 0; j < Rj; j++)
    for (a = 0; a < Ra; a++)
      thd_reg_tile[b][i][c][k][j][a] +=
        s_A[idx_s_A(j, q, a, i)] * s_B[idx_s_B(b, c, q, k)];
}
```

Listing 4: Pseudocode for the redomap accumulation step, as would be generated specifically for the example TC in listing 3 ($\mathcal{Z}_{bickja} = \mathcal{X}_{jqai} * \mathcal{Y}_{bcqk}$). Performs a contraction of two tiles in shmem and accumulates to the thread-private register tile accumulator. For a Q-tile and a sequence of R-tiles (one for each dimension in the segspace), we generate a loop nest with an outer loop iterating the Q tile, and an inner loop nest iterating the R tiles. `idx_s_A` and `idx_s_B` are abstract index functions.

4.3 Copying tiles from global to shared memory

The copying of tiles from global to shared memory is, in broad terms, implemented straightforwardly from the description given in section 3.5.1, and we implement almost one-to-one the pseudocode in listing 1 – but with global memory boundary guards this time, obviously, whereas shmem boundaries are handled by the virtualization loop. The implementation code is quite tedious, but in the large scheme of things it is a trivial implementation of aforementioned strategies, hence we do not go into too much detail with it.

This leaves two performance problems to solve: Effective and efficient handling of partial tiles, and avoiding bank conflicts on shmem accesses.

4.3.1 Handling partial tiles

In section 3.4 we assumed that all tile sizes evenly divided their respective input dimensions. When we lift this assumption, we sometimes have so-called *partial tiles.* We can have partial tiles in both the parallel and the redomap dimension, and each of these must be handled explicitly. In particular, we must attend to two issues: Avoiding out-of-bounds global memory accesses, and how to go about shared memory and the register tile accumulation when outside these bounds, since the register tile accumulation can be corrupted if we attempt to reduce garbage values from share memory.

Handling out-of-bounds accesses to global memory is trivial, since it requires only to insert a boundary guard on global memory accesses (both in the redomap step and the final write-

```
1  kernel tensor_contraction(g_A, g_B, g_C, T_tiles, R_tiles, Q) {
2    // declare shmem (which depends on tile dims) and LMAD's describing tiles.
3    s_A, s_B = declare_shmem(T_tiles, R_tiles, Q);
4    lmad_s_A = LMAD(0, s_A.dims, strides(s_A.dims));
5    lmad_s_B = LMAD(0, s_B.dims, strides(s_B.dims));
6
7    // NE-initialize register tile of dimensions R_tiles.
8    thd_reg_tile = init_reg_tile(R_tiles);
9
10   num_full_sequential_tiles = ceil(common_dim / Q);
11   for (k = 0; k < num_full_sequential_tiles; k++) {
12     // compute this tblock's offset into g_A and g_B.
13     g_A_offs, g_B_offs = tblock_offsets(g_A, g_B);
14
15     // compute LMADs for the global memory arrays.
16     lmad_g_A = LMAD(g_A_offset, s_A.dims, strides(g_A.dims));
17     lmad_g_B = LMAD(g_B_offset, s_B.dims, strides(g_B.dims));
18
19     // copy tiles from global to shared memory.
20     lmad_copy_tile(g_A, s_A, lmad_g_A, lmad_s_A);
21     lmad_copy_tile(g_B, s_B, lmad_g_B, lmad_s_B);
22     // sync tblock.
23
24     // accumulate partial contraction to thread's register tile.
25     redomap_accumulate(thd_reg_tile, s_A, s_B, Q, R_tiles);
26     // sync tblock.
27   }
28   // write this thread's register tile to output tensor.
29   write_reg_tile(g_C, thd_reg_tile, R_tiles);
30 }
```

Listing 5: High-level pseudocode for the GPU kernel we wish to generate, with abstract funtcions substituted for those parts of the code which must be parameterized over the TC segspace. The `lmad_copy_tile` function (lines 20-21) may be implemented as in listing 1. The kernel is executed by $\prod\lceil N_i/T_iR_i\rceil$ tblocks, each of $\prod T_i$ threads.

back).

Handling shared memory turned out to be more subtle. For regular TCs, with multiplication and addition for map function and reduction operator, respectively, we need no explicit handling of partial tiles, because we can simply fill shared memory with zeros in indices outside global memory bounds – later, in the redomap step, these padding zeros are effectively ignored, since $x * 0 = 0 * x = 0$ for all x, and since 0 is the neutral element for addition, the result affects not the reduction. In general, we can safely ignore partial tiles whenever there exist values a and b s.t. $f(a, y) = f(x, b) = \mathbf{ne}$ for any x and y, where $\mathbf{ne}$ is the reduction neutral element.

But such values can be difficult and expensive to determine statically, assuming they exist for the given operators. For example, the fact that map function bodies can contain arbitrary code does not make this analysis easier, so we quickly decided that it was not feasible for the scope of the project.

4.3.1.1 Prologue/epilogue treatment of partial tiles

The simplest solution to handling shmem in case of out-of-bounds is to wrap the redomap phase in boundary guards reflecting those on global memory, s.t. garbage values are explicitly ignored. Since partial tiles in the parallel dimensions are ignored during the write-back due to global memory boundary guards, we need only handle boundary guards on the reduction dimension.

However, these checks naturally affect performance. As a mitigation, we can unroll the last iteration of the outer reduction loop (which iterates the reduction dimension) and remove the boundary check for all but this unrolled iteration. The code almost doubles in size, but code size is rarely a concern.

For TCs with large reduction dimensions, the cost of the epilogue may be amortized by the higher number of prologue iterations, whereas this may not necessarily be the case for smaller TCs. As an example, consider a TQ with $Q_q = 16$, where q is the reduction dimension. If $N_q = 1000$, then we would have $\lfloor 1000/16 \rfloor = 62$ prologue iterations and a single epilogue iteration, whereas if $N_q = 31$, then we we would have exactly one of each. Hence the prologue/epilogue treatment might be detrimental to performance for certain TCs and problem instances if the epilogue happens to be redundant.

In any case, we decide on this solution, since it is preferable to computing the entire contraction using epilogue iterations, and leave it to future work to implement the analysis discussed in section 4.3.1. In the meantime, we offer a source-level attribute `#[no_epilogue]` which can be used to manually and explicitly disable the epilogue for when it is known to be redundant – however, this is prone to user-error so it is primarily for benchmark comparison.

4.4 Avoiding shared memory bank conflicts

We wish to avoid shmem bank conflicts by padding shmem tiles as best as possible. Similar to `BlkRegTiling`, we wish to pad inner dimensions of the shmem tile for the input array indexed by the FVTI, but we now want to pad arbitrary-rank shmem arrays, and ideally only when it is necessary.

4.4.1 Improved padding of multi-dimensional shmem arrays

In section 3.3.2.1, we discussed the possible overhead in shmem padding and showed that it can be large for large-rank tiles. Fortunately it is not difficult to avoid a large part of this overhead: The idea is to flatten the multi-dimensional shmem array and pad the inner dimensions *together.* As an example, instead of padding the array $S_a \times S_b \times S_c$ to $S_a \times (S_b + 1) \times (S_c + 1)$, we flatten the dimensions and pad to $S_a \cdot (S_b \cdot S_c + 1)$.

To see why this improves shmem usage over the method discussed in section 3.3.2.1, consider again a sequence of shmem dimension sizes $\langle S_1, \ldots, S_n \rangle$, with $S_i \geq 1$ for all i. Let j be the dimension indexed by the FVTI, and assume that the inner dimensions $\langle j+1, \ldots, n \rangle$ need padding. We pad by splitting the product in two and padding the product of inner dimensions as such:

$$\left[\prod_{i=1}^{j} S_i\right] \cdot \underbrace{\left(\left[\prod_{i=j+1}^{n} S_i\right] + 1\right)}_{\text{pad dims } j+1 \ldots n} \quad = \quad \left[\prod_{i=1}^{n} S_i\right] + \underbrace{\left[\prod_{i=1}^{j} S_i\right]}_{\text{overhead}}. \tag{17}$$

Recall eq. (6), the lower bound on shmem overhead incurred by the naive padding method:

$$\text{Eq. (6):} \qquad \left[\prod_{i=1}^{n} S_i\right] + \left[\prod_{i=1}^{k-1} S_i\right] \cdot \left[\prod_{i=k+1}^{n} S_i\right],$$

where $k > j$ is the padded dimension. From $j \leq k-1$ we have $\prod_{i=1}^{j} S_i \leq \prod_{i=1}^{k-1} S_i$, and by the assumption of positive sizes S_i, we have also $\prod_{i=k+1}^{n} S_i \geq 1$, which gives:

$$\underbrace{\left[\prod_{i=1}^{n} S_i\right] + \left[\prod_{i=1}^{j} S_i\right]}_{\text{Eq. (17)}} \quad \leq \quad \underbrace{\left[\prod_{i=1}^{n} S_i\right] + \left[\prod_{i=1}^{k-1} S_i\right] \cdot \left[\prod_{i=k+1}^{n} S_i\right]}_{\text{Eq. (6)}}. \tag{18}$$

Hence we can save on shmem usage, but at the obvious consequence that for arrays with more than one padded dimension, the padded array can no longer be represented in its original rank, since arrays must be regular. While this adds no overhead in index computation in the generated code (since the array representation would be flattened by later compiler stages anyway), it *does* complicate our implementation code since shmem arrays must now be treated as flat arrays throughout the module, but the priority is the quality of generated code so we of course implement this method.

4.4.2 Padding implementation

First, padding should only be applied when necessary and beneficial, since if it is not then we may *introduce* conflicts by padding. Hence the padding term used for some flat size s is:

$$\text{padTerm}(s) = \begin{cases} 1 & \text{if } s \bmod 2 = 0 \\ 0 & \text{otherwise} \end{cases} \quad = \quad 1 - (s \bmod 2). \tag{19}$$

For shmem arrays whose dimensions depend on tiling parameters, s is not known until host code runtime, and hence the decision of whether to pad is made in host code.

As we pad a shmem array, we must of course update its associated LMAD strides to reflect the new physical layout. Let $\mathbf{S} = \langle S_1, \ldots, S_n \rangle$ be the LMAD dimensions for the array for which we wish to compute padded strides, and denote by j the index at which we wish to pad the inner dimensions. Let also $(+\!\!+)$ denote concatenation of sequences – then we compute padded strides according to the function:

$$\text{stridesPad}\left(\mathbf{S},\ j\right) = \text{outerStridesPad}\left(\mathbf{S},\ j\right) +\!\!+ \ \text{innerStrides}\left(\mathbf{S},\ j\right), \tag{20}$$

where:

$$\text{innerSizePad}(\mathbf{S},\ j) = \left[\prod_{i=j+1}^{n} S_i\right] + \text{padTerm}\left(\prod_{i=j+1}^{n} S_i\right) \tag{21}$$

$$\text{outerStridesPad}(\mathbf{S},\ j) = \left\langle \left[\prod_{i=k}^{j} S_i\right] \cdot \text{innerSizePad} \ \middle|\ k \in \langle 2,\ \ldots,\ j+1 \rangle \right\rangle \tag{22}$$

$$\text{innerStrides}(\mathbf{S},\ j) = \left\langle \left[\prod_{i=l}^{n} S_i\right] \ \middle|\ l \in \langle j+1,\ \ldots,\ n+1 \rangle \right\rangle \tag{23}$$

The basic idea is this: first, we split shmem dimensions at j; then, using eq. (21), compute the flat size of the inner dimensions with padding; using eq. (22), compute strides for the outer dimensions, taking into account the flat size of the padded inner dimensions; using eq. (23), compute *inner* strides, i.e. those unaffected by padding; finally, concatenate outer and inner strides (eq. (20)).

Listing 6 shows the compiler implementation of shmem padding, including both the decision of which arrays should be padded, and the computing of padded strides. Note that even if no padding is needed, this process does not incur any overhead since all of these computations are constant folded by the kernel compiler when the tile dimensions are constant at kernel compilation.

4.5 Special case optimization for regular MM

Early testing indicated some possible overhead in using LMAD copies in regular MM programs as compared to the method used in `BlkRegTiling` (as described in section 3.6.4.1). We suspect

```
1  let variant_dim_inds = map (`L.elemIndex` segspace_dims) arr_dims
2  let innerProducts = scanr (*) 1 . tail
3  ~(shmem_size_flat, shmem_strides) <-
4    -- Determine candidacy for padding.
5    case Just inner_dim_ind `L.elemIndex` init variant_dim_inds of
6      Just i -> do
7        -- Split tiles on the index at which inner tiles need padding.
8        let (outer_dims, inner_dims) = splitAt (i + 1) tile_dims
9        -- Pad if the inner dims is of even size.
10       let pad_term = 1 - (product inner_dims `rem` 2)
11       let inner_size_flat = product inner_dims + pad_term
12
13       -- The outer strides take into account the inner size, but drop the
14       -- innermost 1-stride, since this belongs to the inner strides.
15       let outer_strides = init $ innerProducts $ outer_dims ++ [inner_size_flat]
16       let inner_strides = innerProducts inner_dims
17       let size_flat = product outer_dims * inner_size_flat
18       pure (size_flat, outer_strides ++ inner_strides)
19     _ -> pure (product tile_dims, innerProducts tile_dims)
```

Listing 6: Compiler implementation of shmem padding (simplified). Line 5 determines whether an array is candidate for padding by checking whether any of its non-innermost dimensions is variant on the innermost segspace dimension (in other words, indexed by the FVTI). Lines 8-17 splits tile dimensions into those which need padding, and outer dimensions, and computes flat size and LMAD strides for the array accordingly. Note that candidacy for padding is determined at compile time, while the actual padding, if necessary, is applied at (host code) runtime. Also, note that the actual implementation code is a little more tedious, since we prefer to bind values to expressions s.t. they are more easily identified in the generated code.

it might be interesting, and perhaps beneficial, to make a special case optimization for the case where the TC is a regular MM, although we use the word "optimization" tentatively since we have yet to benchmark the solution. In this case we replace the flat LMAD copy loop with a 2-nested loop mapping each 2D tile onto the 2D tblock, hence mimicing `BlkRegTiling`. This might save the overhead of logically reshaping the tblock going into the LMAD copy.

The optimization fires whenever both operand tensors are 2D (by the assumption of a single reduction dimension, this would also imply the result is 2D). We are unsure of whether this method can generalize to higher dimensions, so we do not pursue this.

4.6 Firing conditions for the transformation

We wish to formulate a set of conditions on which to pattern match IR expressions to test eligibility for the transformation. Since TC is a generalization of MM, so will these conditions generalize the firing conditions in the `BlkRegTiling` module as described in section 3.6.4. Due to the arbitrary dimensionality of the segmap in a TC expression, it is difficult to illustrate the firing conditions in a pseudocode example, although such an example would be analogous to the example in section 3.6.4.

In any case, firing conditions for an IR expression to be eligible for the transformation are:

1. the expression has a redomap (quasi-)perfectly nested inside $n \geq 2$ parallel dimensions on which the result is variant, possibly surrounded in the map nest by `code2`, i.e. expressions variant on the redomap result(s) but on which the redomap is invariant;
2. the redomap takes exactly two 1D array slices as input, each of which is variant to at most $(n-1)$ of the n parallel dimensions; and
3. for each of the n parallel dimensions on which the result is variant, there is exactly one redomap input array variant to this dimension.

Condition 2 asserts that no array is variant to all parallel dimensions, since then there is no data reuse on this array, while condition 3 asserts that the given parallel dimension is part of the contraction, since remaining outer parallel dimensions should not be tiled but rather interchanged outwards and mapped entirely onto the (CUDA) grid. Condition 3 also implies that each redomap input array is variant to at least 1 parallel dimension.

Note that condition 1 rejects vector/matrix products[7] since here we would have $n = 1$, while condition 2 more generally rejects vector/tensor contractions (including vector/matrix products), since here the vector and tensor would be variant to 0 and n outer dimensions, respectively. These assumptions can be lifted, but this is outside the scope of this project.

There may or may not be other faults or excessive restrictions in this set of firing conditions, and so we leave it to future work (section 6.3) to revisit and concretize them.

4.7 Hurdles in implementation

When taken in *isolation*, most steps taken in implementation up until this point have been quite straight-forward. What has made implementation difficult (and, at times, tedious) have been to fit the pieces together, and, more importantly, to fit our transformations and analyses into the given intermediate representation, where, as it turns out, some necessary information were hard to come by without some rather nasty hacks.

In this section, we go into more depth with some of the intricacies in implementing the transformation into the Futhark compiler specifically.

[7] It also rejects the vector/vector product, but here we have no data reuse so this case is not relevant.

4.7.1 Obtaining input array information

In order to facilitate code transformation/generation, we first need a number of different pieces of information on each input array to the redomap. ?? shows the Haskell data type used to gather and manage all such information for a given redomap array, and these include:

1. `baseArr`, `baseArrDims`, and `arrLoadStm`: a reference to the base array in global memory whence to read input data; the dimensions of said array, used to generate boundary guards on global memory accesses; and a load statement to execute reads from the array;
2. `lmadPerm`: if the original array layout has been rearranged, whether it be in the source code of a previous compiler stage, then we need somehow to reverse-engineer the layout, which we can do using the LMAD permutation. This is necessary in order to correctly map the layout of this array onto the multi-dimensional thread block during the copy stage (more on this in section 4.7.2);
3. `varDimInds`: information on the array's variance to the different segspace dimensions, which is necessary in order to extract information pertinent to the given array from among the information associated with the segspace (see ??), such as when generating loop nests where this array is dependent only on a subset of loop variables, and to identify shmem arrays candidate for padding – see section 4.7.3;
4. `tileDims`, `shmemSizeFlat`, and `shmemStrides`: the logical dimensions of the tile for this array, as well as the flat physical size including padding of the shmem array holding it, and the LMAD strides used for indexing it (see section 4.4.1);
5. `shmemElemType`: the element type of the array, used for the initial declaration of its shmem tile, and for filling shmem with blanks when out-of-bounds on global memory;

Most of this information can be derived trivially from other information, while some of it must be extracted from the environment and from input IR expressions.

4.7.2 Extracting array layout information

When we eventually begin to read tiles from global to shared memory using LMAD copies, we will require that the innermost dimension of the global memory tile be mapped to the FVTI of the executing tblock in order to obtain coalesced access – see section 4.3. In order to achieve this, we need to know the layout of the array in global memory, which must be accessed only indirectly via the load statement for each redomap array carried into the module in the redomap construct since it is not carried explicitly. From this we can then query the type environment to obtain its layout.

However, as it turns out this information includes any rearrangements made to the array layout (either in the source code or earlier compiler stages) and does not necessarily reflect the actual layout in memory. We then had to find a way to reverse-engineer the array layout.

4.7.2.1 Reverse-engineering layout permutations

Before entering the `TCTiling` module, the loop tiling pass estimates index function transformations for each array name in the to-be-optimized statements by scanning the statements for any reshaping and rearranging operations and gathers the estimates in an environment that is then passed on to the `TCTiling` module. We call this information an *estimate*, since true LMAD information is not attached until the `GPUMem` representation, and because it unfortunately is not always reliable, as we shall see in section 4.7.2.2.

But let us assume for a moment that it is. Given the rearranged array dimensions and the strides in the LMAD information, we can reverse-engineer the original layout as such: For each permutation of the array layout dimensions, compute its corresponding strides; exactly one set of computed strides is guaranteed to equal the known strides, and the permutation which produces this set of strides is then the original layout. The reverse-engineered array layout can then be used to correctly map global memory reads to the tblock indices.

The actual compiler implementation of this matching of permutations to known strides is a little tedious. By associativity and commutativity of multiplication, we of course know that e.g. $a \cdot (b \cdot c) = c \cdot (b \cdot a)$ for all numbers a, b, c, however, in the IR a stride would be represented by a `PrimExp` expression[8], and these properties do not hold for multiplication of `PrimExps`, hence we have e.g. $\text{Mul}(P_a,\ \text{Mul}(P_b, P_c)) \neq \text{Mul}(\text{Mul}(P_c, P_b), P_a)$ for `PrimExps` $P_a,\ P_b,\ P_c$.

Our solution to this problem is to "flatten" `PrimExps` when they happen to represent product expressions, i.e. when all factors are either a binary multiplication expression *or* a non-recursive `PrimExp` constructor, and then to sort the factors in the product. This e.g. means

$$\begin{aligned} \text{flatten}\left(\text{Mul}\big(P_a,\ \text{Mul}(P_b, P_c)\big)\right) &= \text{flatten}\left(\text{Mul}\big(\text{Mul}(P_c, P_b),\ P_a\big)\right) \\ &= \text{product}\left(\{P_a,\ P_b,\ P_c\}\right). \end{aligned} \tag{24}$$

Note that leaf and constant `PrimExps`, as well as any recursive `PrimExps` that is not binary multiplication, are stored as opaque expressions in the product and not flattened. This unfortunately has the obvious pitfall that we cannot guarantee to match strides which are not simple products. It is unclear whether this case is even possible; nevertheless, as is it is a potential weakness.

Ideally we would have either: A safe and reliable way to extract layout information for redomap input arrays, e.g. by having it carried in either the redomap construct or the redomap arrays; *or* a generic way to generate (e.g. tblock-wide) segmaps without specifying exactly which dimensions of the segmap is mapped to which dimensions on the executing tblock – this would allow us to express an arbitrary n-dimensional tile copy without worrying about how the copy is mapped to the tblock.

[8]A `PrimExp` can be a leaf expression, a constant value, a binary operator recursively applied to two `PrimExps`, and a number of other things.

```
entry_jqai_bcqk_bickja (X : [J][Q][A][I]f32, ...) = {
  let {X_out : [I][J][A][Q]f32} = rearrange((3, 2, 0, 1), X)
  ...
}

LMAD estimation:
  [LMADDim {ldShape = I, ldStride = 1        },
   LMADDim {ldShape = J, ldStride = Q * A * I},
   LMADDim {ldShape = A, ldStride = I        },
   LMADDim {ldShape = Q, ldStride = A * I    }]
```

(a) Using $\left[\text{map } T \circ \text{map (map } T) \circ T \circ \text{map } T \circ \text{map (map } T)\right] \boldsymbol{\mathcal{X}}$, as in listing 3.

```
entry_jqai_bcqk_bickja (X : [J][Q][A][I]f32, ...) = {
  let {X_reshape0  : [J * Q][A][I]f32} = reshape([J * Q][A][I], X)
  let {X_rearrange : [I][A][J * Q]f32} = rearrange((2, 1, 0), X_reshape0)
  let {X_out   : [I][A][J][Q]f32}  = reshape([I][A][J][Q], X_rearrange)
  ...
}

LMAD estimation:
  [LMADDim {ldShape = I, ldStride = J * A * Q},
   LMADDim {ldShape = J, ldStride = A * Q    },
   LMADDim {ldShape = A, ldStride = Q        },
   LMADDim {ldShape = Q, ldStride = 1        }]
```

(b) Using $\left[\text{map (map unflatten)} \circ T \circ \text{map } T \circ T \circ \text{flatten}\right] \boldsymbol{\mathcal{X}}$.

Listing 9: Futhark IR code at the point immediately preceding loop tiling, for the two different methods of rearranging $\boldsymbol{\mathcal{X}}$ in listing 3, and the LMAD estimations generated for each by the loop tiling pass before commencing TC tiling, where T = transpose. Note that the strides computed in (b) correspond to the layout of the array as if the source-level rearrangement was actually manifested, whereas the LMAD in (a) is as we expect.
IR output generated using `futhark dev`, and LMADs using debug prints.

```haskell
1  findLMADPerm :: Env -> VName -> Maybe [Int]
2  findLMADPerm (_, ixfn_env) arr = do
3    -- Lookup LMAD for array.
4    lmad <- LMAD.dims <$> M.lookup arr ixfn_env
5    let shape    = map (untyped . LMAD.ldShape) lmad
6        strides0 = map (toFlatPrimExp . untyped . LMAD.ldStride) lmad
7    -- Test each permutation against known strides; pick first succeeding.
8    msum $ map (isPermutationOf strides0 . getStrides) $ permutations shape
9    where
10     getStrides = map toFlatPrimExp . scanr binopMul . tail
11     ...
```

Listing 7: Compiler implementation of the reverse-engineering of LMAD permutations. Function `getStrides` (line 10) generates stride expressions as the cumulative products of inner dimensions of a given shape, appending an innermost stride of 1 before flattening the resulting strides using `toFlatPrimExp` as shown in listing 7.

4.7.2.2 Assumption on rearrangements in the IR

As mentioned the LMAD information computed at the start of the loop tiling pass is not always reliable for rearranged arrays. As far as we can tell this estimated LMAD information can happen to be incorrect when the layout transformation cannot be expressed as a single rearrangement in the IR, but as a sequence of rearrangements and reshaping. We decide that a solution to this problem is **outside the scope of our project**, and hence decide instead to only test programs where redomap arrays are rearranged using only compositions of transpositions and make a note of it for future work.

Listing 9 shows examples of the estimated LMADs resulting from two different methods of rearranging the first operand array in listing 3.

4.7.3 Use of variance indices

Throughout code generation, we will at various points need to generate expressions parameterized over the segspace dimensions. Because the two redomap arrays are variant to disjoint subsequences of segspace dimensions and neither of them is variant to all of them, we at various points need to be able to partition the information derived from the segspace between the two arrays.

Denote again by $D_\mathbf{S}$ the dimensions of some segspace $\mathbf{S}$. We return to the running example of listing 3, where in this case we have $D_\mathbf{S} = \langle b, i, c, k, j, a \rangle$. Consider then listing 4, the redomap accumulation step for this particular TC, where here the inner six loops are derived from $\mathbf{S}$. Each redomap array is variant to 3 dimensions in $\mathbf{S}$, and hence to only 3 of the R loop variables in scope in the loop nest (in addition to q from the outermost loop over the Q tile, but this is independent from $\mathbf{S}$). We now need somehow to partition the set of loop variables.

```
1 data FlatPrimExp = Product [FlatPrimExp] | OpaquePrimExp (PrimExp VName)
2   deriving (Eq, Ord)
3 toFlatPrimExp :: PrimExp VName -> FlatPrimExp
4 toFlatPrimExp = Product . sort . extractFactors . flattenMulOps
5   where
6     flattenMulOps (BinOpExp Mul {} e1 e2) = Product $ map toFlatPrimExp [e1, e2]
7     flattenMulOps e = OpaquePrimExp e
8     extractFactors (Product es) = concatMap extractFactors es
9     extractFactors e = [e]
```

Listing 8: Compiler implementation of “flattening” of `PrimExp` product expressions. `flattenMulOps` (lines 6-7) largely implements flattening as described in section 4.7.2.1 (exemplified in eq. (24)), while `extractFactors` (lines 8-9) join factors across products s.t. they can be sorted (using `Eq PrimExp`) going into an equality check.

In the context of the compiler, we can extract variance information from the environment, and this would tell us that the redomap arrays are variant onto dimensions $\langle i, j, a\rangle$ and $\langle b, c, k\rangle$, respectively. For each array, we can cross-reference this information with the set of loop variables *if* we also have a mapping of segspace dimensions to loop varibles. However, explicitly creating such a mapping can be tedious if it has to be done every time.

Instead, we found it convenient to encode the variance information for a given redomap array in terms of the (zero-based) index of each segspace dimension on which this array is variant. We will for the remainder of this report refer to this information as *variance indices*, and for the two redomap arrays in the running example, the variance indices are $\langle 4, 5, 1\rangle$ and $\langle 0, 2, 3\rangle$, respectively, since the mapping of $D_\mathbf{S}$ to indices is $\langle b : 0, i : 1, c : 2, k : 3, j : 4, a : 5\rangle$.

In general, to compute variance indices for an array $\boldsymbol{\mathcal{X}}$ with dimensions $D_{\boldsymbol{\mathcal{X}}}$ into a segspace $\mathbf{S}$ with dimensions $D_\mathbf{S}$, we may use the formula:

$$\text{varInds}_\mathbf{S}\left(\boldsymbol{\mathcal{X}}\right) = \underset{d \in D_{\boldsymbol{\mathcal{X}}}}{\left(+\!\!+\right)} \text{elemIndices}_{D_\mathbf{S}}(d), \tag{25}$$

where $(+\!\!+)$ is again sequence concatenation, and $\text{elemIndices}_{D_\mathbf{S}}(d)$ is the sequence of indices of all occurences d in $D_\mathbf{S}$. Recall from section 3.2 that tensor indices are uniquely labelled – hence d must occur in $D_\mathbf{S}$ at most once, and note that this formula produces a sequence of variance indices ordered wrt. the layout of $\boldsymbol{\mathcal{X}}$.

Then, given an indexed sequence of information $\mathbf{I} = \langle I_0, I_1, \ldots\rangle$ of any type, where $\mathbf{I}$ is derived from, and hence ordered wrt., some segspace $\mathbf{S}$, we may extract, or *gather*, those elements of $\mathbf{I}$ associated with an array $\boldsymbol{\mathcal{X}}$ by the formula:

$$\text{gather}_\mathbf{S}\left(\mathbf{I},\ \boldsymbol{\mathcal{X}}\right) = \left\langle I_i \;\middle|\; i \in \text{varInds}_\mathbf{S}\left(\boldsymbol{\mathcal{X}}\right) \right\rangle \tag{26}$$

In our compiler implementation, variance indices are computed once in the initial derivation of array information, and information derived from it is implicitly indexed.

5 Testing

In this section, we first present, discuss, and justify our benchmarking plan for the new `TCTiling` module, and later, present, analyze, and conclude upon the results of benchmarking. To reproduce benchmarking results, please see `github.com/sortraev/msc_thesis_public`.

5.1 Validation testing

Since the code is still in development, and because much work yet remains in addressing limitations in the implementation (as documented in section 4.1), we do not go very much into validation testing at this point, except to write a small suite of programs with isolated TCs that can be used to verify changes and to convince ourselves of the validity of benchmarking results. The validation suite consists of the 3 TCs later used in benchmarking: (including the three TCs used in benchmarking; see section 5.3.2), each of which is run against a handful of representative test inputs (including partial tiles in all dimensions, unit dimensions, and other important cases) for different tile parameterizations (both "nice" and uncommon tile sizes).

All validation tests pass for the compiler versions and GPU used in benchmarking (see section 5.3.1). To view and reproduce tests, please see `github.com/sortraev/msc_thesis_public`.

5.2 Benchmarking goals

The primary goal with benchmarking is to explore whether the chosen strategy has merit, and, of course, whether it is beneficial to move further with it in Futhark or whether other strategies should be explored instead. To do so, we benchmark against a point of reference with COGENT. Even if the main contribution of COGENT is fusion of TCs[5], their code generator also produces efficient code for isolated TCs[6], and besides this, COGENT was the initial inspiration for this project, so it is an interesting reference point. We additionally test our Futhark generated code against the prototype kernel in order to explore whether there is room for improvement. Finally, since our implementation attempts to generalize the existing `BlkRegTiling` module for 2D tiling, we of course wish to compare our implementation with the reference implementation for simple MM programs.

We also wish to benchmark performance effects of some (but not all) of the optimizations made: Specifically,the effects of manually handling partial tiles in an epilogue, as described in section 4.3.1, since if it turns out that the epilogue dominates for TCs with small reduction dimensions, then this is good motivation for putting work into eliminating the prologue/epilogue treatment where doing so is valid; and the special case MM optimization as described in section 4.5, since if results show that the LMAD copy adds excessive overhead in this case then we may want to look into more special case optimizations.

The tile configuration space is very large for most TCs, and finding optimal parameters for a specific kernel is infeasible. Hence, wrt. tile parameterization, the goal of our benchmarking is not to quantify the *optimal* performance of our kernel, but rather to formulate a set of heuristics and make a best effort approximation on which to gauge *potential* performance of the kernel.

5.3 Benchmarking plan

We wish to limit the number of test cases as much as possible while still providing realiable and meaningful measurements for analysis. In this section we present some of the different factors in the benchmarking suite.

5.3.1 Code and compiler versions used

We test three different version of the Futhark-generated kernel: `TCTiling_EPILOGUE`, a baseline kernel which handles partial tiles in an epilogue; `TCTiling_NO_EPILOGUE`, which implicitly ignores partial tiles and foregoes the epilogue; and `TCTiling_MM-opt`, which uses an epilogue but which implements the special case MM optimization as mentioned in section 4.5. The first two are compiled by commit `6cc23cd2d` [9] of the compiler, while the latter is compiled using commit `9894ecee1` [10].

We test two different versions of the prototype, here called `prototype_EPILOGUE` and `prototype_NO_EPILOGUE`, similar to the first two `TCTiling` kernel versions. These can be found at `github.com/sortraev/msc_thesis_public`. We do not test the MM specific optimization in the prototype.

For COGENT tests, we generate one new kernel per contraction using the COGENT code generator, as presented in [6], available at `github.com/kimjsung/CGO2019-AE`.

All benchmarking tests are run on an nVIDIA A100 GPU with 40GB of RAM and CC 8.0.

5.3.2 Test contractions

We test 6 different TC programs, where 4 are MM programs and 2 are larger-rank TCs. To benchmark larger contractions, we look to [5] for inspiration in regards to test contractions. Here, COGENT is tested against two sets of TCs, **sd1** and **sd2**. For all TCs in **sd1**, q is innermost on the first operand and outermost on the second operand, while for all TCs in **sd2**, q is innermost on both operands. For this reason we choose to test only one TC from each set, even if the individual TCs in each set may have other distinguishing features.

We pick one TC from each set at random[11]:

$$\textbf{sd1\_7}: \quad \boldsymbol{\mathcal{Z}}_{abcijk} = \boldsymbol{\mathcal{X}}_{icaq} * \boldsymbol{\mathcal{Y}}_{qbjk}, \tag{27}$$

$$\textbf{sd2\_3}: \quad \boldsymbol{\mathcal{Z}}_{abcijk} = \boldsymbol{\mathcal{X}}_{kiaq} * \boldsymbol{\mathcal{Y}}_{bcjq}. \tag{28}$$

[9] `github.com/diku-dk/futhark/tree/6cc23cd2d`

[10] `github.com/diku-dk/futhark/tree/9894ecee1`

[11] Note that the authors of COGENT [5] label tensor indices *innermost first* and uses different names for contraction indices between different TCs, whereas we label indices *outermost first* and consistently use q for the contraction index – hence the TC expressions may appear different where they are not.

For the MM tests, we use all four permutations of operand tensors in the regular MM:

$$\mathbf{MM0}: \quad \mathcal{Z}_{ab} = \mathcal{X}_{aq} * \mathcal{Y}_{bq}, \tag{29}$$
$$\mathbf{MM1}: \quad \mathcal{Z}_{ab} = \mathcal{X}_{aq} * \mathcal{Y}_{qb}, \tag{30}$$
$$\mathbf{MM2}: \quad \mathcal{Z}_{ab} = \mathcal{X}_{qa} * \mathcal{Y}_{bq}, \tag{31}$$
$$\mathbf{MM3}: \quad \mathcal{Z}_{ab} = \mathcal{X}_{qa} * \mathcal{Y}_{qb}. \tag{32}$$

5.3.3 Test input datasets

We test the **sd1_7** and **sd2_3** programs against each of the below datasets:

Dataset 1 $(N_a, N_b, N_c, N_i, N_j, N_k, N_q) = (32, 32, 32, 32, 32, 32, 32)$: This represents a balanced workload, with two hypercubic operand tensors and "nice" dimension sizes, i.e. all dimensions are divided evenly by tile sizes (except for $R = 5$). Used to test performance when there are no partial tiles

Dataset 2 $(N_a, N_b, N_c, N_i, N_j, N_k, N_q) = (32, 32, 32, 32, 32, 32, 31)$: Similar workload to dataset 1, but with partial tiles in the reduction dimension. Used to test effects on performance of the epilogue treatment

Dataset 3 $(N_a, N_b, N_c, N_i, N_j, N_k, N_q) = (31, 31, 31, 31, 31, 31, 31)$: Similar workload to datasets 1, but with partial tiles in all dimensions. Used to compare `TCTiling` with a COGENT kernel that has all boundary checks enabled

Dataset 4 $(N_a, N_b, N_c, N_i, N_j, N_k, N_q) = (16, 16, 16, 16, 16, 16, 2048)$: Same workload as dataset 1, but with a disproportionately large reduction dimension

Note that the number 31 in datasets 2 and 3 was chosen for the simple fact that it is the closest prime to 32, which guarantees partial tiles in all dimensions i for which $T_i R_i > 1$ at a similar workload.

All **MM** tests are run with the same dataset of square matrices of dimensions 4096×4096.

5.3.4 Performance metric

Performance is measured in arithmetic throughput in TFLOPs/s (tera-FLOPs per second). This is a fitting metric in this case since the number of FLOPs is constant for any problem instance, whereas e.g. memory throughput is an ambiguous metric, since the amount of data reuse varies for different configurations.

5.3.5 Tile parameter search

As stated a number of times at this point, the tile parameter space is large for large-rank TCs, and since Futhark as of yet does *not* support autotuning of tile parameters, we must manually choose parameters used in benchmarking. Hence we simply brute-force search for good tile parameters for use in testing.

For the MM program, we set a single, simple heuristic: T and Q tiles come from the set $\{8, 16, 32\}$, and R tiles from the set $\{1, 2, 4, 5, 8\}$, for a total of $3^3 \cdot 5^2 = 675$ configurations. We do not justify it besides to say that these are values which have worked well with the existing 2D tiling implementation. All of these combinations are valid, although some may be inefficient due to e.g. register spilling (for example, for $(T_a, T_b, R_a, R_b) = (32, 32, 8, 8)$, we necessarily go over the soft limit of 65536 registers per tblock[7]).

For the remaining two test programs, **sd1_7** and **sd2_3**, we have significantly more parameters, and we in fact must also extend the T and Q parameter space to $T, Q \in \{1, 2, 4, 8, 16, 32\}$ and additionally $Q \in \{64, 128\}$ for dataset 4, since the higher number of parameters requires higher modularity. The R set remains the same. We then apply the following set of "*common sense and best bets*" heuristics (some of which are derived from hardware restrictions):

1. since we test only input datasets with balanced input tensors, we want to balance shmem s.t. no shmem array is alloted more than 75% of the total shmem usage (before padding).
2. total shmem usage including padding does not exceed 163 KiB per tblock (the maximum amount of dynamic shmem for the A100 we use in testing[7]).
3. both arrays must have at least one non-unitary R tile;.
4. the tblock size must be a power of 2 between 32 and 1024.
5. the configuration achieves an occupancy of at least 80%, as per the CUDA occupancy calculator[12], factoring in a conservative overhead of 8 registers besides the register tile.

Some of the heuristics may be implied by the occupancy heuristic, but we keep them in nonetheless. Using these heuristics we prune to some number of configurations, whence we randomly sample 2000 configurations for each of our Futhark generated and prototype kernels, and in turn select the best performing configuration(s). Finally, based on the initial findings, we exclude certain parameter/value pairs and re-run search with a more fine-tuned sample of configurations for each kernel – as an example, we found that $R_k > 1$ virtually never gave good results for any of the kernels, so this was excluded.

The search is run for each kernel/problem instance combination, and the best performing is used in the final results. The 25 best performing configurations for each kernel/problem instance can be found at `github.com/sortraev/msc_thesis_public`.

5.3.6 COGENT kernel generation

We generate one fresh COGENT kernel using the COGENT code generator[13] for each of the 8 combinations of test program and dataset. However, some of the generated kernels are identical, and hence the number of distinct COGENT kernels used is only 4: One kernel for each test program to be used for datasets 1-3, and one kernel for each test program for dataset 4.

[12] `docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator`

[13] `github.com/kimjsung/CGO2019-AE`

5.4 Benchmarking limitations

Our benchmarking plan is very limited in scope. First and foremost, benchmarking is naturally limited by the limitations in our implementation, as presented in section 4.1. However, the main limiting factor in benchmarking is time (in particular the parameter search takes a long time), and hence our benchmarking plan is additionally limited by:

1. we test only input datasets with equal-size operand tensors
2. all datasets tested are of similar workload
3. we test only contractions with equal-rank operand tensors
4. we do not profile the effects of shmem padding
5. we do not test optimal tiling parameters, since finding such parameters is infeasible for virtually any kernel (rather we test the best parameters under a set of heuristics, as described in section 5.3.5)
6. we do not profile "average case" performance of our kernels, i.e. the performance we might expect from generalizing a single set of parameters to different problem instances, or the performance an end-user might expect from choosing tiling parameters at random

We leave this list of limitations as future work, but do not guarantee that it is exhaustive.

5.5 Benchmarking results

5.5.1 Parameter search results

We perform the parameter search as described in section 5.3.5. The results of the search is a long list of configuration/measurement pairs for each kernel/problem instance pair, from which we choose configurations used in benchmarking. The top 10 best performing configurations for each kernel/problem instance pair can be found at `github.com/sortraev/msc_thesis_public`.

5.5.2 sd benchmark results

For the `TCTiling` kernels, speedups are reported as "$\substack{\text{(speedup vs. prototype)} \\ \text{(speedup vs COGENT)}}$", while for the prototype kernels speedups are simply wrt. COGENT.

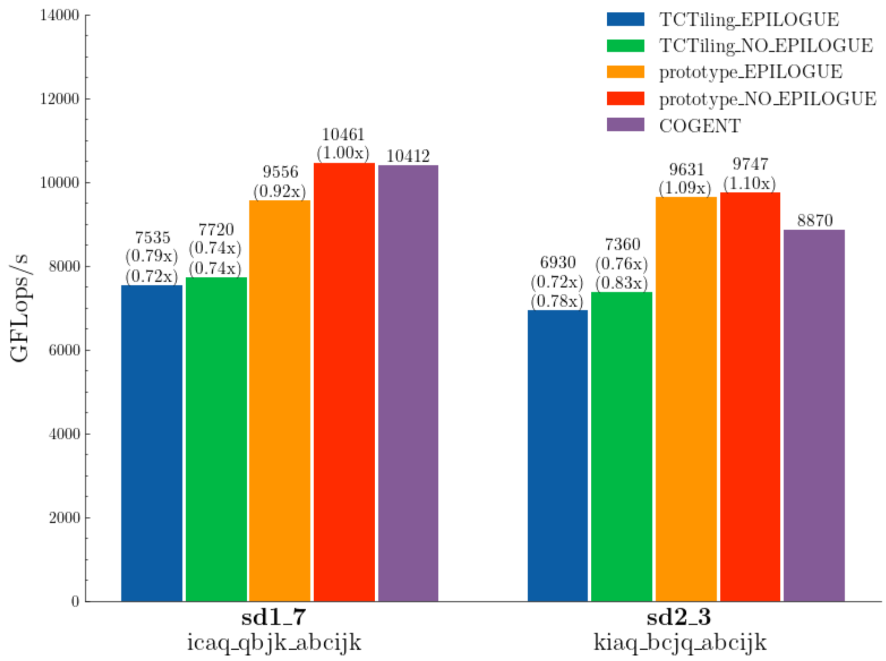


Figure 2: sd tests, dataset 1 (no partial tiles).

5.5.2.1 Dataset 1: No partial tiles

Figure 2 shows the results of **sd** tests on dataset 1. All kernels generally handle **sd1_7** better than **sd2_3**.

The prototype kernels each perform quite well as compared to the COGENT kernel for both datasets, with speedups between 0.92x-1.1x. `TCTiling` kernels fall a behind COGENT with speedups of roughly 0.73x and 0.8x for **sd1_7** and **sd2_3**, respectively, and about as far behind prototype kernels, indicating that there may be some performance to be gained in `TCTiling`.

`TCTIling_NO_EPILOGUE` performs better than `TCTiling_EPILOGUE` for both contractions, even though there are no partial tiles in these datasets (meaning the epilogue is never run).

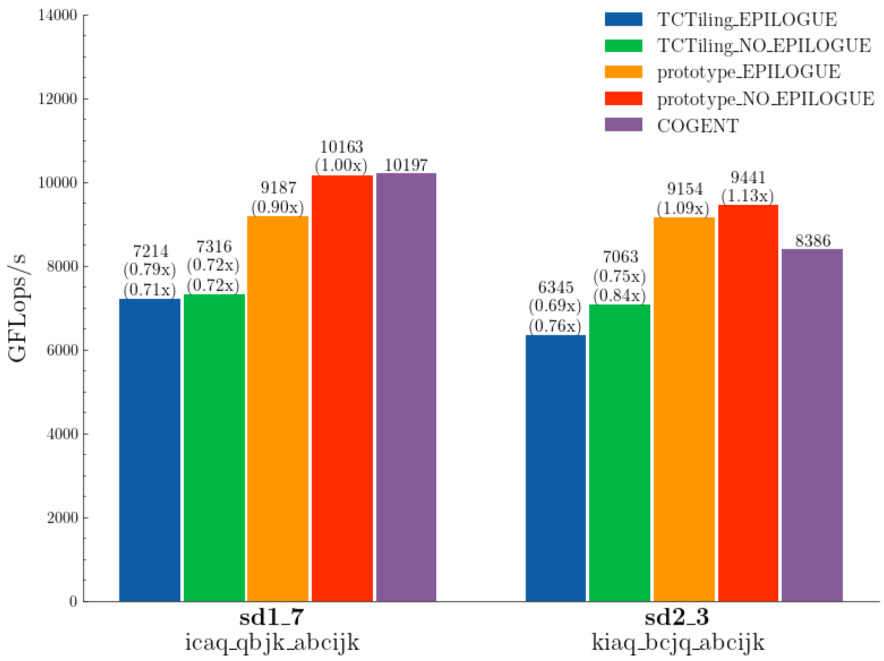


Figure 3: sd tests, dataset 2 (partial tiles in the reduction dimension).

5.5.2.2 Dataset 2: Partial tiles in the reduction dimension

Figure 3 shows the results of **sd** tests on dataset 2. All kernels perform a little worse than for dataset 1, and the relative performance between kernels is largely the same.

The gap between `TCTiling_EPILOGUE` and `TCTiling_NO_EPILOGUE` for **sd1_7** is surprisingly small, considering we now have partial tiles and hence the epilogue is run. However, looking at the tile parameters used in these tests, we see that $Q_q = 4$ for this test, and hence the prologue is run $\lfloor 31/4 \rfloor = 7$ times against the 1 epilogue, so this could be indication that the cost of the epilogue can be amortized when the Q tile is chosen appropriately. On the other hand, the gap between the two kernels for the **sd2_3** dataset is now *larger* than for dataset 1, which might indicate the contrary.

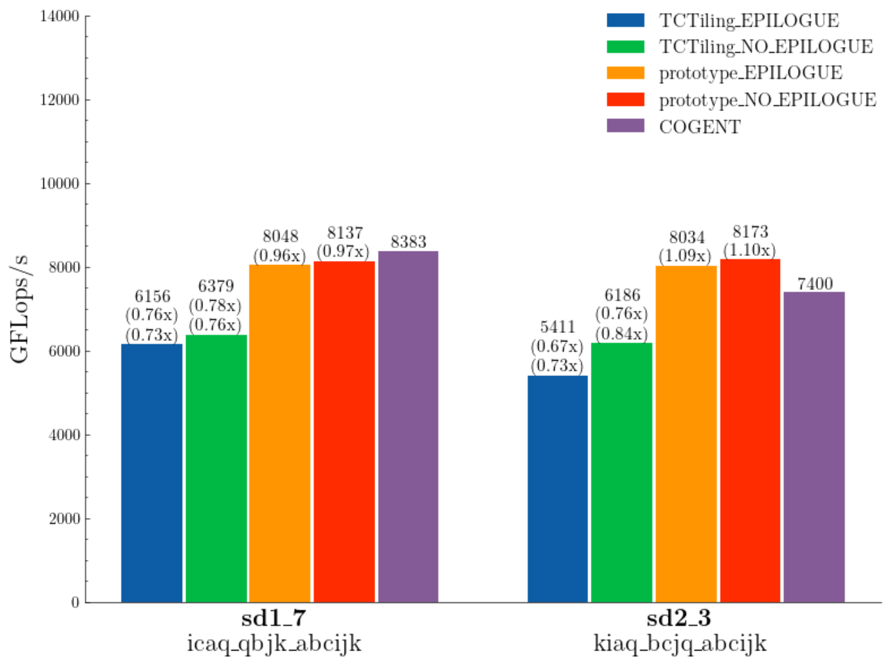


Figure 4: sd tests, dataset 3 (partial tiles in all dimensions).

5.5.2.3 Dataset 3: Partial tiles in all dimensions

Figure 4 shows the results of **sd** dataset 3 tests. Interestingly, but not surprising, all kernels perform significantly worse than for dataset 2, with a wider margin than between datasets 1 and 2.

Again, we see largely the same relative performance between our compiler implementation and the reference kernels, with the two `TCTiling` kernels again reaching in the range of 0.73x and 0.8x vs. COGENT. This is noteworthy, because for this dataset we would have expected our kernel to lessen the gap to COGENT, since here COGENT launches a kernel more comparable to ours (due to extra inserted boundary checks).

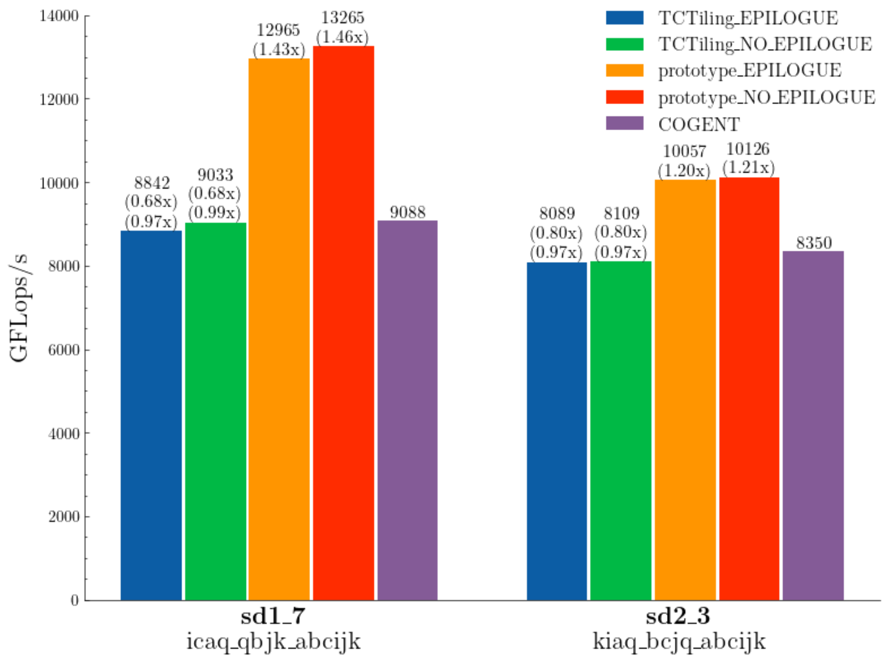


Figure 5: sd tests, dataset 4 (no partial tiles; large reduction dimension).

5.5.2.4 Dataset 4: Large reduction dimension

Finally, fig. 5 shows the results of **sd** dataset 4 tests, and this plot paints a different picture from the previous three tests.

The `TCTiling` kernels now perform within a very small margin to COGENT, which may indicate a number of things. For one, considering `TCTiling` performs significantly better on this dataset than any of the previous, it may indicate that `TCTiling` favors higher thread-sequentialism.

On the other hand, COGENT performs a little worse here than for dataset 1, which has the same workload and also no partial tiles. This may be a consequence of the fact that COGENT chooses Q tiles (in their paper called TB_k) from the set $\{4, 8, 16\}$[6], which is not particularly modular. However, considering that $Q_q = 16$ was used by `TCTiling` for **sd1_7**, this may not be the issue in this case (for reference, $Q_q = 128$ was used for **sd2_3**).

The particularly high performance of the prototype kernels – as compared to COGENT and its own performance for the previous three datasets – is further indication that our strategy favors large reduction dimensions (i.e. high degrees of thread-sequential work), which makes sense considering saturation.

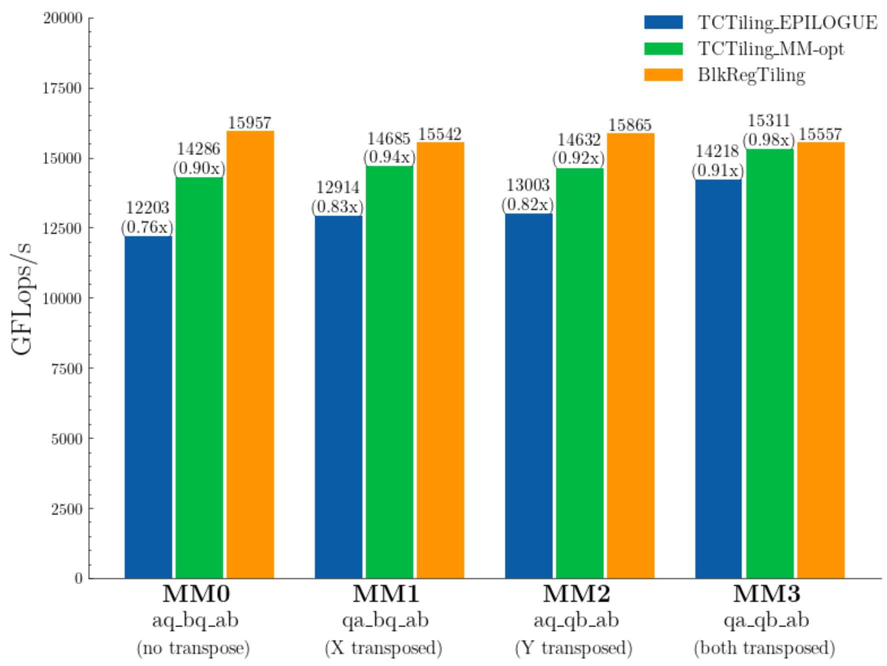


Figure 6: MM tests. Speedups relative to `BlkRegTiling`.

5.5.3 MM benchmark results

Figure 6 shows the results of of the benchmark comparison of our implementation against the existing `BlkRegTiling` module for the single dataset of 4096×4096 matrices. *Note that* `TCTiling_NO_EPILOGUE` *is not tested since* `BlkRegTiling` *uses an epilogue.*

Evidently `TCTiling_EPILOGUE`, which uses a flat LMAD copy, suffers significant speed-downs as compared to `BlkRegTiling` for all four MM variants, but especially for **MM0** which sees a speedup of 0.76x.

Meanwhile, `TCTiling_MM-opt`, which forgoes the LMAD copy, improves significantly on the previous result, albeit not quite meeting `BlkRegTiling` despite the fact that the generated code is very similar between the two kernels. In any case, this is quite clear of an indication that the LMAD copy has some unnecessary overhead in at least one very common case, but it also warrants looking into other differences between the two implementations.

6 Evaluation

6.1 Implementation evaluation

We successfully implemented block/register tiling of arbitrary tensor contraction expressions and generalized upon the existing `BlkRegTiling` module, albeit with some limitations on the source-level expression, as detailed in section 4.1.

The generated code matches exactly the expectation, as per the handwritten CUDA prototype, and we successfully implemented the optimizations we set out to, including shmem padding and the prologue/epilogue treatment.

Aside from aforementioned limitations, we believe the overall quality of the code product is fairly high and facilitates further development. The code is also well-documented with comments. However, as presented in the implementation sections, there were numerous problems which we had to overcome during development, sometimes with tedious work-arounds. These unfortunately have made for a less than perfectly reliable module – for example, the transformation can fail (or, at least, fail to produce efficient code) if either redomap array has had its layout information transformed with more than a simple IR `rearrange` (i.e. permutation).

6.2 Benchmarking results evaluation

As presented in section 6.3 our benchmarking suite has a lot of glaring limitations. This is of course unfortunate, because meaningful conclusions require broad and generalizable results. In particular, it is unfortunate that we did not test more different input datasets with irregular dimensions.

In any case, the benchmarking we do have does leave us with a couple of interesting findings: It seems there is some benefit to omitting the epilogue where doing so is valid, even if the benefit can be quite small for some cases; the margin between our implementation and the prototype indicates that there is significant room for micro-optimization, since the prototype uses no strategies or optimizations which could not be implemented in the Futhark compiler; the **MM** tests showed us that it might eventually be possible to replace the `BlkRegTiling` module with no loss of performance even in this special case, however some work remains in (micro-)optimizing the implementation; and finally, the **sd** dataset 4 and **MM** test results indicated that there may be performance gains to be had in additional sequentialization when the reduction dimension is small, e.g. by having threads process multiple smaller tiles and iterating the reduction dimension multiple times.

In general, we believe our benchmarking has shown a definite potential for the implemented strategy, even if some tweaks and changes are necessary in order to bring the Futhark-generated code up to speed with the prototype, and even if additional testing is necessary before we can rule out other strategies.

6.3 Future work

The body of future work can be divided into three categories: general bug fixing and further generalization of the implementation; performance optimizations; and entirely new functionality, and while ideally they should be adressed in this order, we present them here in only roughly that order. Note that some of our proposals for future work are more justified than others.

Generalize the implementation

As mentioned in section 4.1, we decided to down-prioritize support for non-empty `code2` (i.e. any code statements/expression in the segmap variant on the redomap, but on which the redomap is not variant) and additional parallel dimensions on top of the segmap on which none *or* all of the redomap arrays are variant due to a lack of time and because we decided these features were not necessary in order to examine feasibility of the strategy. However, we believe these features can be quickly implemented and may take inspiration from the existing implementation in `BlkRegTiling`.

Concretize firing conditions for the transformation

The current pattern matching rules for determining whether an expression fits the requirements for TC tiling are derived largely from the existing pattern matching rules for 2D tiling, but modified to fit the definition a TC (wrt. free and contracted indices), as described in section 4.6. For one, these conditions for one reject vector/tensor contractions (including vector/matrix products), and we cannot guarantee that there are not other excessive restrictions – or, conversely, that the conditions are too permissive – hence we suggest to look into a more rigorous derivation of the firing conditions.

Better access to crucial information in the IR

In section 4.7 we presented and discussed a number of tedious problems unearthed during implementation into the compiler, some of which required quite brittle hacks to get around, perhaps most notably the problem of reverse-engineering LMAD information in the GPU IR, where it is not usually available. To better and more reliably facilitate the transformation, some work should be put into reconsidering how certain pieces of information are conveyed in the IR, such as information on redomap arrays.

In might even be beneficial to look into migrating the module to a later compiler stage, e.g. to the GPUMem IR where true LMAD information is available, although we cannot say for sure whether this is even feasible since we really are only familiar with the GPU IR.

More exhaustive benchmark testing of the implementation

Our benchmarking was very limited in scope, even if we did find enough evidence that we felt safe to claim that the strategy has potential (see). In the future, however, it might be both helpful and meaningful to develop a more deliberate and exhaustive benchmarking suite. In we gave a list of limitations on benchmarking which should ideally be addressed, but it would also be beneficial to look into benchmark comparison with other TC libraries besides COGENT.

Explore different methods for copying from global memory

The copy of tiles from global to shared memory was implemented as a flat LMAD copy, in order to make it generic in rank of the read array and the executing tblock (sections 3.5.1 and 4.3), whereas the existing `BlkRegTiling` module uses a method specifically tailored to the 2D case (section 3.6.4.1). While early exploration showed that the LMAD copy was better suited for expressing arbitrary-rank copies, we eventually decided to implement the `BlkRegTiling` method as a special case optimization, as described in section 4.5, and benchmarking showed this to be very beneficial.

Hence it would be interesting to further and more rigorously examine whether this strategy, or one similar to it, can be generalized or applied to more special cases.

As an addendum to this, it might be wise to see if any other micro-optimizations are made in `BlkRegTiling` that we might have missed, since, again, the specialization makes no assumptions that should not also be valid by the generalization.

COGENT-like handling of partial tiles

Recall from section 3.7 that COGENT generates kernels only once the problem instance is known (hence it is not directly applicable to Futhark). This enables the code generator to always choose tile parameters s.t. they divide the input dimensions, provided that the dimensions have divisors which make for suitable tile parameters. In any case, this makes for an excellent common case optimization, since it allows them to generate kernels specifically tailored to the four general cases of partial tiles: No partial tiles; partial tiles in the reduction dimension; partial tiles in one or more outer dimension; and partial tiles in both the reduction and one or more outer dimension. A suitable kernel is then chosen at runtime.

Futhark could also be made to generate four such kernels for each TC, and then choose a suitable kernel at runtime. It might not be beneficial in cases where the problem is not known, since then tile parameters cannot be specifically tailored to the problem instance, and hence the boundary guard-free kernels will rarely be chosen, but it still interesting to look into.

Automatic or default tile parameterization

By default, Futhark executables set T (parallel) and R (register) tiles to 16 and 4, respectively, if the user does not manually provide parameters or a tuning file. While 16 and 4 happen to be good (and sometimes optimal) values for 2D block/register tiling[14], they are problematic for 3 dimensions and up for a number of reasons.

First, recall from section 3.4.3 that the flat tblock size is always the product of T tiles. If the tiling dimensionality is 3 or greater, then the default $T = 16$ will prevent the kernel from

[14] Ie. block/register tiled kernels with a 2D result, such as regular MM programs; here, $(T, R) = (16, 4)$ would imply a thread block of $T^2 = 256$ threads and a register tile of flat size 16.

launching due to requesting too large of a tblock, given a maximum tblock size of 1024. As is, the Futhark generated host code does not detect this, so the program fails with a CUDA runtime error, where ideally it should detect this and attempt one of the other kernels. However, when the dimensionality is high enough, default tile sizes will result in requesting too much shared memory, and in this case the kernel will not even be picked in the first place.

Second, say T tiles are chosen s.t. the tblock size is valid. If R tiles are chosen poorly then we instead run the risk of requesting too much shared memory or too many registers per thread. Requesting too much shared memory will not result in error, since host code will pick up on this and choose another kernel, but it can be a problem for performance if the next picked kernel is unsuited to the problem instance, and a sudden degradation in performance from a small change in R tile parameterization might not be obvious to the user. Requesting too many registers can result in a CUDA runtime launch error or, perhaps worse, register spilling to CUDA local memory, which can tremendously degrade performance.

To avoid these problems, we propose two mitigations: Based on a given problem instance and set of tile parameters, it should always be possible to automatically determine appropriate (albeit not optimal) default values for the tiling parameters using heuristics. It can be hard to choose good values before knowing the problem instance (as an example, recall from section 3.7 that COGENT[6] obtains good tile parameterization by choosing values for preset representative problem instances), but it should always be possible to choose a set of parameters that allow the kernel run, avoids excessive register spilling, balances shared memory usage between the two shared memory arrays, and obtains coalesced writes to the output tensor.

In summary, bad parameterization can lead to both CUDA runtime errors, host code foregoing the kernel in favor of other kernels possibly less suited for the problem instance (thereby nullifying the optimization), and the kernel launching with degenerate performance. This may be motivation enough in itself, but finally, there is also a benefit to hiding low level semantics from the programmer. Users should not be expected to be familiar with the low-level GPU algorithm(s) onto which their source code is mapped, and manually tweaking tuning parameters requires a significant understanding of the low-level GPU code.

Remove redundant LMAD copies following kernels

The following was not mentioned in the report, since it is not directly relevant to our project, but it is future work nonetheless: At present the compiler will sometimes insert redundant GPU-to-GPU LMAD copies following our kernel for certain TCs. According to Troels Henriksen (head maintainer of the Futhark code base), these copies are likely inserted as a conservative measure to assert row-major layout of the result tensor. We have not been able to discern a meaningful pattern as to what triggers these redundant copies, however it seems to occur mostly for larger-rank TCs, such as the **sd1_7** and **sd2_3** programs used in testing.

As is these copies are unfortunately quite visible in the overall performance of our implementation (for those programs for which they are inserted), so some time would be spent well looking into their source and why they are inserted.

7 Conclusion

This thesis has shown how an optimizing GPU compiler might identify and exploit opportunities for data reuse in tensor contraction expressions and generate efficient block/register tiled code for GPU execution, albeit with a number of restrictions to the contraction expression and the subsequent code statements.

More specifically, the main contribution of this thesis is `TCTiling`, a new, work-in-progress tensor contraction tiling module to the optimization stage of the Futhark compiler, which successfully compiles tensor contraction and contraction-like expressions to efficient GPU code, with optimizations made to the handling of partial tiles via a prologue/epilogue treatment, and to reducing shared memory bank conflicts via padding.

Perhaps most importantly, the thesis has uncovered, presented, and discussed a number of challenges in doing so specifically in the context of the Futhark compiler, as well as solutions to some of these hindrances. However, as discussed as part of future work, some unfortunately remain, such as the problem of instability across tiling parameters, and the hassles of accessing certain crucial information in the IR.

Despite validation testing succeeding, benchmark testing showing promising results, and the generated code matching that of the prototype CUDA kernel, there is still room for improvement and optimization, with our generated kernel reaching roughly 72-75% and 75-80% the performance of our handwritten prototype kernels and COGENT-generated kernels[5][6], respectively, for near-hypercubic problem instances, and 68% and 98% the performance of our prototype kernels and COGENT-generated kernels, respectively, for a problem instance with more sequential work.

The implementation sought to replace and generalize upon `BlkRegTiling` – the existing 2D block/register tiling module specifically targeting GEMM-like programs – and came respectably close, reaching between 90% to 98% the performance of `BlkRegTiling` for four different matrix multiplication programs.

Still, a lot of work remains in both profiling and generalizing the implementation, exploring entirely new avenues of optimization, and in refactoring the code to fit more comfortably inside the given IR and compiler stage, but this thesis has shown that there is good and definite potential in doing so.

8 References

[1] Troels Henriksen. *Design and Implementation of the Futhark Programming Language.* PhD thesis, DIKU, November 2017. URL `https://futhark-lang.org/publications/troels-henriksen-phd-thesis.pdf`. Accessed May 7, 2024.

[2] *Futhark programming language official website.* URL `https://www.futhark-lang.org/`. Accessed: May 7, 2024.

[3] Æmilie Cholewa-Madsen and Anders Lietzen Holst. *Teaching the Futhark compiler block and register tiled matrix multiplication.* BSc thesis, DIKU, June 2020. URL `https://futhark-lang.org/student-projects/aemilie-anders-bsc-thesis.pdf`. Accessed: May 7, 2024.

[4] Paul Springer and Paolo Bientinesi. Design of a High-Performance GEMM-like Tensor–Tensor Multiplication. *ACM Trans. Math. Softw.*, 44(3), January 2018. ISSN 0098-3500. doi: 10.1145/3157733. URL `https://doi.org/10.1145/3157733`.

[5] J. Kim, A. Sukumaran-Rajam, C. Hong, A. Panyala, R. K. Srivastava, S. Krishnamoorthy, and P. Sadayappan. Optimizing Tensor Contractions in CCSD(T) for Efficient Execution on GPUs. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, page 96–106, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357838. doi: 10.1145/3205289.3205296. URL `https://doi.org/10.1145/3205289.3205296`.

[6] J. Kim, A. Sukumaran-Rajam, V. Thumma, S. Krishnamoorthy, A. Panyala, L.-N. Pouchet, A. Rountev, and P. Sadayappan. A Code Generator for High-Performance Tensor Contractions on GPUs. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 85–95, 2019. doi: 10.1109/CGO.2019.8661182.

[7] *CUDA C++ Programming Guide.* NVIDIA Corporation, 12.4 edition. URL `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`. Accessed: May 8, 2024.

[8] Cosmin E. Oancea. Lecture Notes for the Software Track of the PMPH Course. Course curriculum, University of Copenhagen, 2018.

[9] Michel Dubois, Murali Annavaram, and Per Stenstrom. *Parallel Computer Organization and Design.* Cambridge University Press, 2012. ISBN 978-521-88675-8.

[10] Philip Munksgaard. *Static and Dynamic Analyses for Efficient GPU Execution.* PhD thesis, DIKU, 2023. URL `https://di.ku.dk/english/research/phd/phd-theses/2023/Philip_Munksgaard_Thesis.pdf`. Accessed May 10, 2024.

[11] P. Munksgaard, T. Henriksen, P. Sadayappan, and C. Oancea. Memory optimizations in an array language. In *2022 SC22: International Conference for High Performance Computing,*

Networking, Storage and Analysis (SC) (SC), pages 424–438, Los Alamitos, CA, USA, nov 2022. IEEE Computer Society. URL `https://doi.ieeecomputersociety.org/`.

[12] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. Incremental flattening for nested data parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 53–67, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6225-2. doi: 10.1145/3293883.3295707. URL `http://doi.acm.org/10.1145/3293883.3295707`.

thanks for reading. :)

Appendix

All appendices – including prototype kernels, COGENT kernels, the results of parameter searches, and validation tests – can be found at `github.com/sortraev/msc_thesis_public`. For reasons of formalia, we include a full code dump of our implementation here.

A Implementation code

Below code dump contains the entirety of our implementation code for the new `TCTiling module`, as it appears in the commit `9894ecee1`.

```haskell
{-# LANGUAGE TypeFamilies #-}

module Futhark.Optimise.TCTiling (doTCTiling) where

import Control.Monad
import Data.Char
import Data.List qualified as L
import Data.Map.Strict qualified as M
import Futhark.Analysis.PrimExp
import Futhark.IR.GPU
import Futhark.IR.Mem.LMAD qualified as LMAD
import Futhark.Optimise.BlkRegTiling (matchCodeStreamCode, processIndirections)
import Futhark.Optimise.TileLoops.Shared
import Futhark.Tools
import Futhark.Transform.Rename

forM2 :: Monad m => [a] -> [b] -> (a -> b -> m c) -> m [c]
forM2 xs ys f = zipWithM f xs ys

forM3 :: Monad m => [a] -> [b] -> [c] -> (a -> b -> c -> m d) -> m [d]
forM3 xs ys zs f = forM (zip3 xs ys zs) (\(a, b, c) -> f a b c)

se0, se1, se2 :: SubExp
se0 = intConst Int64 0
se1 = intConst Int64 1
se2 = intConst Int64 2

seglvl_thd :: SegLevel
seglvl_thd = SegThreadInBlock $ SegNoVirtFull $ SegSeqDims []

reductionLoopBody ::
  TCEnv ->
  VName ->
  VName ->
  [VName] ->
  Bool ->
  Builder GPU [VName]
reductionLoopBody tc_env qq0 reg_tiles_in shr_arrs_in is_prologue = do
  qq <- letExp "qq" =<< toExp (le64 qq0 * pe64 tile_Q)

  redomap_inputs_shr <- forM2 shr_arrs_in arr_infos $ copyGlb2Shr qq
  reg_tiles_out <- accumulateRegTile qq redomap_inputs_shr
  pure $ reg_tiles_out : redomap_inputs_shr
  where
    arr_infos = arrsInfo tc_env
    kernel_params = kernelParams tc_env
    tile_Q = tileQ kernel_params
    tiles_T = tilesT kernel_params
    tiles_R = tilesR kernel_params
    tblock_dims = tblockDims kernel_params
    common_dim = commonDim kernel_params
    tblock_size_flat = tblockSizeFlat kernel_params

    is_MM = all ((== 2) . length . tileDims) arr_infos && length tblock_dims == 2

    copyGlb2Shr :: VName -> VName -> ArrInfo -> Builder GPU VName
    copyGlb2Shr qq shr_arr arr_info = do
      -- Setup parameters for the WithAcc.
      cert_p <- newParam "cert_p" $ Prim Unit
      t <- stripArray (shapeRank smem_shape) <$> lookupType shr_arr
      acc_p <-
        newParam (baseString shr_arr) $
          Acc (paramName cert_p) smem_shape [t] NoUniqueness

      lam <- mkLambda [cert_p, acc_p] $
        case is_MM of
          True -> do
            -- In the special MM case, we generate a loop nest similar to the
            -- one in BlkRegTiling, i.e. a 2-loop nest of dimensions:
            --
            --   ceil(s0 / Ta), ceil(s1 / Tb)
            --
            -- where [s0][s1] is the size of the shmem slice and (Ta, Tb) are
            -- the tblock dimensions. As an example, for the regular MM, i.e.
            -- Z_ab = X_aq * Y_qb, the first operand shmem array has dimensions
            -- [Ta * Ra][Tq], and hence the loop nest has dimensions:
            --
            --   Ra, ceil(Tq / Tb).
            loop_bounds <-
              zipWithM
                ( \tile_dim tblock_dim ->
                    letSubExp "loop_bound" =<< ceilDiv tile_dim tblock_dim
                )
                tile_dims
                (tblockDims kernel_params)

            fmap varsRes $
              segMapND "foo" seglvl_thd ResultNoSimplify tblock_dims $ \ltids ->
                fmap (varsRes . (: [])) $
                  forLoopNest_ loop_bounds (paramName acc_p) $ \loop_inds acc_merge -> do
                    inds' <-
                      forM3
                        loop_inds
                        tblock_dims
                        ltids
                        ( \loop_ind dim ltid ->
                            letExp "ind" =<< toExp (le64 loop_ind * pe64 dim + le64 ltid)
                        )
                    copyLoopBody acc_merge undefined inds'
          _ -> do
            -- In the general case, we use a flat LMAD copy.
            --
            -- The strategy is to flatten the tblock and then unflatten it to fit the
            -- dimensions of the array in shared memory, using a virtualization loop
            -- in case the tile is larger than the tblock, and a boundary guard for
            -- the converse. This is easily achieved using SegVirt, but whereas
            -- SegVirt wraps the entire loop body in an `if (i < tile_size_flat)`
          -- guard, we want that guard only on the write to shared memory. Hence
          -- we must manually build the virtualization loop, which unfortunately
          -- bloats the code a bit here.
          tile_size_flat <- letSubExp "tile_size_flat" =<< toExp $ product tile_dims'
          iters <- letSubExp "virt_iters" =<< ceilDiv tile_size_flat tblock_size_flat
          fmap varsRes $
            segMap1D "foo" seglvl_thd ResultNoSimplify tile_size_flat $ \ltid ->
              fmap (varsRes . (: [])) $
                forLoop_ iters (paramName acc_p) $ \i0 acc_merge -> do
                  i <- letExp "flat_virttid" =<< toExp (le64 i0 * pe64 tblock_size_flat + le64 ltid)
                  unflat_inds <-
                    forM (unflattenIndex tile_dims' $ le64 i) $
                      letExp "unflat_ind" <=< toExp
                  copyLoopBody acc_merge i unflat_inds

    letExp (baseString shr_arr) $
      WithAcc [(smem_shape, [shr_arr], Nothing)] lam

    where
      smem_strides = smemStrides arr_info
      smem_shape = Shape [smemSizeFlat arr_info]
      tile_dims = tileDims arr_info
      tile_dims' = map pe64 tile_dims
      tblock_offsets = arrGather_ arr_info (tblockOffsets tc_env) (Var qq)
      base_arr_dims = baseArrDims arr_info
      base_arr = baseArr arr_info

      copyLoopBody :: VName -> VName -> [VName] -> Builder GPU VName
      copyLoopBody acc i inds = do

        -- The shared mem indices are simply the unflattened indices, while
        -- the global mem indices need to have tblock offsets added onto them.
        glb_inds <-
          forM2 tblock_offsets inds $ \tb_offset ind ->
            letExp "glb_ind" =<< toExp (pe64 tb_offset + le64 ind)

        -- Perform a boundary check and read from the global mem array!
        in_bounds <-
          letExp "in_bounds"
            =<< toExp
              ( foldr (.&&.) true $
                  zipWith
                    (\ind dim -> le64 ind .<. pe64 dim)
                    glb_inds
                    base_arr_dims
              )

        -- We initially permuted base array dimensions to match the actual
        -- layout in memory, such that we were able to map it to the thread
        -- block. However, we must make to sure re-permute it before executing
        -- the read, since the `index` function assumes the indices are given
        -- in order of the *rearranged* array. Insane, I know.
        let glb_inds_perm = arrPerm arr_info glb_inds
        glb_elem <-
          letExp (baseString base_arr)
            =<< eIf
              (toExp in_bounds)
              ( index "glb_elem" base_arr glb_inds_perm
                  >>= resultBodyM . (: []) . Var
              )
              -- Here, we simply insert a zero (or zero-like value) into
              -- smem whenever we are outside bounds. However, this only
```

```
                -- succeeds in certain cases, unless we explicitly handle
                -- residual tiles in an epilogue (which we do).
                -- See note [SmemZeroPaddingOnGlobalMemOOB].
                (eBody [eBlank $ Prim $ smemElemType arr_info])


          -- Flat smem index (including padding, if any).
          shr_ind_flat <-
            letTupExp' "shr_ind_flat" <=< toExp . sum $
              zipWith (\ind s -> le64 ind * pe64 s) inds smem_strides

          -- Finally, update shared mem array accumulator.
          letExp "acc_out"
            =<< eIf
              (toExp =<< smem_bounds_check)
              ( resultBodyM . map Var <=< letTupExp "acc_updated" . BasicOp $
                  UpdateAcc
                    Unsafe
                    acc
                    shr_ind_flat
                    [Var glb_elem]
              )
              (resultBodyM [Var acc])

          where
            smem_bounds_check
              | is_MM = smemBoundsCheck inds tile_dims
              | otherwise = do
                  tile_size_flat <- letSubExp "tile_size_flat" <=< toExp $ product tile_dims'
                  smemBoundsCheck [i] [tile_size_flat]
            smemBoundsCheck inds' dims =
              fmap (foldr (.&&.) true) $
                forM3
                  tblock_dims
                  dims
                  inds'
                  ( \tblock_dim tile_dim ind -> do
                      tile_fits_tblock <-
                        fmap ((.==. 0) . le64) . letExp "tile_fits_tblock" . BasicOp $
                          BinOp (SRem Int64 Unsafe) tile_dim tblock_dim
                      pure $ tile_fits_tblock .||. le64 ind .<. pe64 tile_dim
                  )


    accumulateRegTile :: VName -> [VName] -> Builder GPU VName
    accumulateRegTile qq redomap_inputs_shr =
      segMapND_ "reg_tiles_out" seglvl_thd ResultPrivate tiles_T $ \ltids -> do
        reg_tile_in <- index "reg_tile_in" reg_tiles_in ltids
        fmap ((: []) . varRes) $
          forLoop_ tile_Q reg_tile_in $ \q reg_tile_in' ->
            letExp "reg_tile_acc"
              =<< eIf
                ( toExp $
                    -- if we are in the prologue, accumulate unconditionally!
                    fromBool is_prologue
                      .||. le64 qq + le64 q .<. pe64 common_dim
                )
                ( resultBody . (: []) . Var
                    <$> accumulateRegTileInnerLoopNest ltids q reg_tile_in'
                )
                (resultBodyM [Var reg_tile_in'])
      where
```

```haskell
        accumulateRegTileInnerLoopNest :: [VName] -> VName -> VName -> Builder GPU VName
        accumulateRegTileInnerLoopNest ltids q reg_tile_in =
          forLoopNest_ tiles_R reg_tile_in $ \loop_inds reg_tile_merge -> do
            -- Compute lists of indices for each redomap operand. For each
            -- dimension, we need an index of the form `ltid * reg_tile +
            -- loop_ind`, so for the reduction dimension, use a dummy ltid and
            -- reg_tile.
            dummy_ltid <- letExp "dummy_ltid_q" =<< toExp se0
            let dummy_regtile = se1
            shr_inds_flat <- forM arr_infos $ \arr -> do
              let ltids' = arrGather_ arr ltids dummy_ltid
              let tiles_R' = arrGather_ arr tiles_R dummy_regtile
              let loop_inds' = arrGather_ arr loop_inds q
              inds <-
                forM3 ltids' tiles_R' loop_inds' $ \ltid tile loop_ind ->
                  letSubExp "shr_ind" =<< toExp (le64 ltid * pe64 tile + le64 loop_ind)
              letTupExp "shr_ind_flat" <=< toExp . sum $
                zipWith
                  (\ind s -> le64 ind * le64 s)
                  inds
                  (smemStrides arr)

            -- Compute map and reduction results and update the register tile.
            map_f <- renameLambda $ mapLam tc_env
            red_op <- renameLambda $ redLam tc_env

            map_operands <- forM2 redomap_inputs_shr shr_inds_flat $ \arr inds ->
              eSubExp . Var <$> index (baseString arr ++ "_elem") arr inds
            map_res <- eLambda map_f map_operands

            acc <- eSubExp . Var <$> index "acc" reg_tile_merge loop_inds
            red_res <- eLambda red_op $ acc : map (eSubExp . resSubExp) map_res

            update "res" reg_tile_merge loop_inds $ resSubExp $ head red_res

doTCTiling :: Env -> Stm GPU -> TileM (Maybe (Stms GPU, Stm GPU))
doTCTiling env (Let pat aux (Op (SegOp (SegMap SegThread {} seg_space ts old_kbody))))
  | KernelBody () kstms [Returns ResultMaySimplify certs (Var _res_name)] <- old_kbody,
    -- we don't want to tile the kernel if it is going to have expensive
    -- boundary checks.
    -- TODO: why, though? what else do we do in this case?
    certs == mempty,
    -- the kernel should have exactly one primtyped result.
    [res_t] <- ts,
    primType res_t,
    all_gtids_dims <- unSegSpace seg_space,
    -- TODO: for now, I test only source programs with no outer parallel
    --       dimensions, ie. all dims in the segspace pertain to the
    --       contraction.
    --       find out how to reliably extract the inner dims of the segspace.
    --       perhaps inner dims are all those onto which the kernel result is
    --       variant and at least (or exactly) one redomap array is variant?
    (rem_outer_gtids_dims, inner_gtids_dims) <- ([], all_gtids_dims), -- TODO: placeholder.
    (gtids, inner_dims) <- unzip inner_gtids_dims,
    -- check that the kernel fits the pattern:
    -- some code1; one Screma SOAC; some code2,
    -- where code2 may contain additional Scremas but code1 may not.
    -- TODO: do we assume only one Screma in kstms? does it even matter?
    Just (code1, screma_stmt@(Let pat_redomap _ (Op _)), code2') <-
      matchCodeStreamCode kstms,
    -- checks that the Screma SOAC is actually a redomap and normalizes it
    Just (common_dim, redomap_arrs, (_is_comm, red_lam, red_nes, map_lam)) <-
```

```haskell
      isTileableRedomap screma_stmt,
    -- TODO: Cosmin's implementation mentioned rearranging the below couple of
    --       conditions. better look into this.
    -- check that exactly two 1D arrays are streamed through redomap,
    -- and the result of redomap is one scalar
    length redomap_arrs == 2,
    [red_ne] <- red_nes,
    [red_t, _] <- map paramDec $ lambdaParams red_lam,
    primType red_t,
    map_ts@[_, _] <- map paramDec $ lambdaParams map_lam,
    all primType map_ts,
    initial_variance <- M.map mempty $ scopeOfSegSpace seg_space,
    variance <- varianceInStms initial_variance kstms,
    -- assert that all redomap arrays are variant to some, but not all innermost
    -- dimensions of the kernel.
    -- TODO: find out whether/where/how to use the returned information.
    Just _var_inds_per_arr <- variantDimsPerArr variance redomap_arrs gtids,
    -- TODO: all of the below guards are lifted from Cosmin's code.
    --       find out which of them are relevant here, and whether they need to
    --       be changed/generalized.
    --       as far as I can tell, it all pertains to the handling of `code2`,
    --       so I'll let it sit for now.
    -- get the variables on which the first result of redomap depends on
    (redomap_orig_res : _) <- patNames pat_redomap,
    Just red_res_variance <- M.lookup redomap_orig_res variance, -- variance of the reduce result
    -- we furthermore check that code1 is only formed by
    -- 1. statements that slice some globally-declared arrays
    --    to produce the input for the redomap, and
    -- 2. potentially some statements on which the redomap
    --    is independent; these are recorded in `code2''`
    Just (code2'', table_load_stms) <-
      processIndirections code1 redomap_arrs red_res_variance,
    -- extract the stms loading slices from redomap arrays and check that there
    -- is one such stm for each redomap array.
    Just load_stms <- mapM (`M.lookup` table_load_stms) redomap_arrs = do
      let _code2 = code2' <> code2''
      let map_prim_ts = map elemType map_ts

      -- TODO: for now, we manually disable the prologue/epilogue treatment when
      -- suitable. However, ideally this would be done automatically, or not at
      -- all, if there turns out to be a better method, or if the epilogue is
      -- not sufficiently detrimental to performance that it is necessary.
      let use_epilogue = not $ AttrName "no_epilogue" `inAttrs` stmAuxAttrs aux

      (new_kernel, host_stms) <- runBuilder $ do
        kernel_params@( TCKernelParams
                          _gtids
                          _inner_dims
                          _common_dim
                          _inner_dim_names
                          tiles_T
                          tiles_R
                          _tiles_TR
                          tile_Q
                          grid_dims
                          grid_size_flat
                          _tblock_dims
                          tblock_size_flat
                          tbids
                          tbid_flat
                        ) <-
          makeTCKernelParams gtids inner_dims common_dim

      (ret_seggroup, stms_seggroup) <- runBuilder $ do
        tc_env <- makeTCEnv env kernel_params load_stms map_lam red_lam map_prim_ts

        -- Zero-initialize register tile.
        reg_tiles_init <- segMapND_ "reg_tiles" seglvl_thd ResultPrivate tiles_T $ \_ -> do
          reg_tile_init <- scratch "reg_tile_init" (elemType res_t) tiles_R
          css <- forLoopNest_ tiles_R reg_tile_init $ \loop_inds merge ->
            update "reg_tile" merge loop_inds red_ne
          pure [varRes css]

        -- Declare shared memory arrays.
        shr_arrs_init <-
          forM (arrsInfo tc_env) $ \arr ->
            scratch
              ("shr_" ++ baseString (baseArr arr))
              (smemElemType arr)
              [smemSizeFlat arr]

        ~(reg_tiles_res : _) <-
          case use_epilogue of
            True -> do
              myDebugM "Compiling TC expression WITH epilogue"
              num_full_Q_tiles <-
                letExp "num_full_Q_tiles" . BasicOp $
                  BinOp (SQuot Int64 Unsafe) common_dim tile_Q
              residual_input <-
                letExp "residual_input" . BasicOp $
                  BinOp (SRem Int64 Unsafe) common_dim tile_Q

              ~prologue_res@(reg_tiles' : shr_arrs') <-
                forLoop (Var num_full_Q_tiles) (reg_tiles_init : shr_arrs_init) $
                  \qq0 (reg_tiles_merge : shr_arrs_merge) ->
                    reductionLoopBody tc_env qq0 reg_tiles_merge shr_arrs_merge True

              letTupExp "reduction_res"
                =<< eIf
                  (toExp $ le64 residual_input .==. 0)
                  (resultBodyM $ map Var prologue_res)
                  ( resultBody . map Var
                      <$> reductionLoopBody tc_env num_full_Q_tiles reg_tiles' shr_arrs' False
                  )
            _ -> do
              myDebugM "Compiling TC expression WITHOUT epilogue"
              num_q_tiles <- letSubExp "num_Q_tiles" =<< ceilDiv common_dim tile_Q
              forLoop num_q_tiles (reg_tiles_init : shr_arrs_init) $
                \qq0 (reg_tiles_merge : shr_arrs_merge) ->
                  reductionLoopBody tc_env qq0 reg_tiles_merge shr_arrs_merge True

        let regtile_ret_dims =
              map ((,se1,se1) . snd) rem_outer_gtids_dims
                ++ zip3 inner_dims tiles_T tiles_R
        pure [RegTileReturns mempty regtile_ret_dims reg_tiles_res]
      -- END KERNEL BUILDER

      let grid = KernelGrid (Count grid_size_flat) (Count tblock_size_flat)
          level' = SegBlock SegNoVirt (Just grid)
          space' = SegSpace tbid_flat (rem_outer_gtids_dims ++ zip tbids grid_dims)
          kbody' = KernelBody () stms_seggroup ret_seggroup
      pure $ Let pat aux $ Op $ SegOp $ SegMap level' space' ts kbody'
    -- END HOST BUILDER

```

```haskell
      pure $ Just (host_stms, new_kernel)
doTCTiling _seg_space _kstms = pure Nothing

-- | Given a variance table, a list of array names, and a list of inner dims
-- (actually, the list of gtids for said inner dims); asserts that each array is
-- variant to at least 1 and not all inner dims, and that at least one array is
-- variant to each inner dim. If these assertions hold; returns list of indices
-- of variant dims for each array.
-- TODO: Dimensions on which all redomap arrays are variant should be
-- interchanged outwards.
variantDimsPerArr ::
  VarianceTable ->
  [VName] ->
  [VName] ->
  Maybe [[Int]]
variantDimsPerArr variance arrs gtids = do
  let var_inds_per_arr = map variantInnerDimsForArr arrs
  let var_gtids_per_arr = map (gather gtids) var_inds_per_arr

  -- Interchange those dimensions of the segspace on which all redomap arrays
  -- are variant outwards.
  let (outer_dims, tc_dims) =
        L.partition
          -- Check that given dim is in var_dims of all arrays.
          (\dim -> all (elem dim) var_gtids_per_arr)
          gtids
  let segspace_dims' = outer_dims ++ tc_dims
  let segspace_perm = gtids `isPermutationOf` segspace_dims'

  -- assert that all arrays are variant to some, but not all dims.
  -- TODO: is below check sufficient to check this assertion?
  --       perhaps this assertion should be (or already is) made elsewhere.
  guard $ all ((`elem` [1 .. n_dims - 1]) . length) var_inds_per_arr

  -- for each dim, assert that at least one array is variant to this dim.
  -- TODO: is there a better, more correct, or safer way to assert this?
  -- Actually, I think this can safely be assumed to already hold, due to these
  -- parallel dimensions already having been interchanged outwards in an earlier
  -- compiler stage, but I might be wrong on this.
  guard $ all (`elem` concat var_gtids_per_arr) gtids

  -- assert no overlap in variance between arrays.
  -- TODO: is this check necessary or even desired? for exactly 2 redomap
  -- arrays, overlap in variance means all redomap arrays are variant to the
  -- given parallel dimension, and thus it would have been interchanged outwards
  -- (given the above TODO is implemented).
  -- guard $ allUnique $ concat var_inds_per_arr

  pure var_inds_per_arr
  where
    n_dims = length gtids
    variantInnerDimsForArr arr =
      let arr_variance = M.findWithDefault mempty arr variance
       in L.findIndices (`nameIn` arr_variance) gtids
    -- allUnique (x : xs) = x `notElem` xs && allUnique xs
    -- allUnique _ = True

-- | All the various kernel parameters and related information we need to
-- declare and/or compute in host code.
data TCKernelParams = TCKernelParams
  { -- Gtids and sizes of those dimensions of the inner segspace which we are tiling.
    innerGtids :: [VName],
    innerDims :: [SubExp],
    commonDim :: SubExp,
    -- Not strictly necessary, but nice to have for consistent names throughout
    -- the generated code.
    innerDimNames :: [String],
    -- T, R, and TR tiles for each inner dimension.
    tilesT :: [SubExp],
    tilesR :: [SubExp],
    tilesTR :: [SubExp],
    -- Tile size for the sequential (reduction) dimension.
    tileQ :: SubExp,
    -- Grid and tblock parameters.
    gridDims :: [SubExp],
    gridSizeFlat :: SubExp,
    tblockDims :: [SubExp],
    tblockSizeFlat :: SubExp,
    -- VNames for the tblock id's.
    tbidVns :: [VName],
    tbidFlatVn :: VName
  }
  deriving (Show)

-- | All of the information needed for code generation in kernel scope. Also
-- carries the kernel parameters declared in host scope.
data TCEnv = TCEnv
  { kernelParams :: TCKernelParams,
    -- Block offset for each dimension in the result.
    tblockOffsets :: [SubExp],
    -- Lambdas for the map function and reduction operators for the contraction.
    mapLam :: Lambda GPU,
    redLam :: Lambda GPU,
    -- For each reduction array, the information needed to handle this
    -- particular array during code generation.
    arrsInfo :: [ArrInfo]
  }
  deriving (Show)

-- | All the information needed to handle a given operand array.
-- TODO: give a proper name to this one.
data ArrInfo = ArrInfo
  { baseArr :: VName,
    baseArrDims :: [SubExp],
    arrLoadStm :: Stm GPU,
    lmadPerm :: [Int],
    varDimInds :: [Maybe Int],
    tileDims :: [SubExp],
    smemSizeFlat :: SubExp,
    smemStrides :: [SubExp],
    smemElemType :: PrimType
  }
  deriving (Show)

gather :: [a] -> [Int] -> [a]
gather xs = map (xs !!) . filter (`elem` indices xs)

gather_ :: [a] -> a -> [Maybe Int] -> [a]
gather_ xs x = map (maybe x (xs !!) . checkIdx)
  where
    checkIdx i
      | Just j <- i, j `elem` indices xs = i
      | otherwise = Nothing

```

```haskell
arrGather_ :: ArrInfo -> [a] -> a -> [a]
arrGather_ info src x = gather_ src x $ varDimInds info

arrPerm :: ArrInfo -> [a] -> [a]
arrPerm info xs = gather xs $ lmadPerm info

makeTCKernelParams ::
  [VName] ->
  [SubExp] ->
  SubExp ->
  Builder GPU TCKernelParams
makeTCKernelParams gtids inner_dims_se common_dim_se = do
  -- various names.
  tile_common_dim_vn <- newVName $ "T_" ++ common_dim_name
  tile_T_vns <- mapM (newVName . ("T_" ++)) inner_dim_names
  tile_R_vns <- mapM (newVName . ("R_" ++)) inner_dim_names
  tbids <- mapM (newVName . ("tbid_" ++)) inner_dim_names
  tbid_flat <- newVName "tbid_flat"

  -- tile sizes.
  tile_Q <- letTileSE SizeTile tile_common_dim_vn
  tiles_T <- mapM (letTileSE SizeTile) tile_T_vns
  tiles_R <- mapM (letTileSE SizeRegTile) tile_R_vns
  tiles_TR <-
    zipWithM (\t r -> toExp $ pe64 t * pe64 r) tiles_T tiles_R
      >>= zipWithM letSubExp (map ("TR_" ++) inner_dim_names)

  -- grid and tblock stuff.
  grid_dims <-
    zipWithM ceilDiv inner_dims_se tiles_TR
      >>= zipWithM letSubExp (map ("grid_dim_" ++) inner_dim_names)
  grid_size_flat <-
    letSubExp "grid_size_flat"
      =<< toExp (product $ map pe64 grid_dims)

  let tblock_dims = tiles_T
  tblock_size_flat <-
    letSubExp "tblock_size_flat"
      =<< toExp (product $ map pe64 tiles_T)

  pure $
    TCKernelParams
      gtids
      inner_dims_se
      common_dim_se
      inner_dim_names
      tiles_T
      tiles_R
      tiles_TR
      tile_Q
      grid_dims
      grid_size_flat
      tblock_dims
      tblock_size_flat
      tbids
      tbid_flat
  where
    inner_dim_names
      | Just name_strs <- mapM getNameStrFor inner_dims_se = name_strs
      | otherwise = map show $ indices inner_dims_se
    common_dim_name = maybe "Q" id $ getNameStrFor common_dim_se

    getNameStrFor (Var v) = Just $ filter isAscii $ baseString v
    getNameStrFor _ = Nothing

    letTileSE tile_type v =
      letSubExp (baseString v) $ Op $ SizeOp $ GetSize (baseName v) tile_type


data FlatPrimExp = Product [FlatPrimExp] | OpaquePrimExp (PrimExp VName)
  deriving (Eq, Ord)

-- TODO: should rewrite this to not use L.permutations, since it is O(n!) for
-- arrays of `n` dims. For n <= 6 dims this is fine-ish, but for ~7 and up it
-- quickly becomes a problem. Can also find the correct permutation using
-- iterative search in quadratic-ish time.
findLMADPerm :: Env -> VName -> Builder GPU [Int]
findLMADPerm (_, ixfn_env) arr = do
  case maybe_lmad_perm of
    Just res -> pure res
    _ -> indices . arrayDims <$> lookupType arr
  where
    maybe_lmad_perm = do
      lmad <- LMAD.dims <$> M.lookup arr ixfn_env
      let shape = map (untyped . LMAD.ldShape) lmad
          strides0 = map (toFlatPrimExp . untyped . LMAD.ldStride) lmad
      -- Test each permutation against the known strides; pick first succeeding.
      msum $ map (isPermutationOf strides0 . strides) $ L.permutations shape

    strides = map toFlatPrimExp . (++ [val1]) . scanr1 binopMul . tail
    binopMul = BinOpExp $ Mul Int64 OverflowUndef
    val1 = ValueExp $ IntValue $ Int64Value 1

    -- Flattens a nested PrimExp (if that PrimExp happens to represent a simple
    -- product) to a [FlatPrimExp], which can then be sorted to check for
    -- equality. Used to more reliably check equality between LMAD strides.
    -- See note [FlattenPrimExps].
    toFlatPrimExp :: PrimExp VName -> FlatPrimExp
    toFlatPrimExp = Product . L.sort . flattenProducts . flattenMulOps
      where
        flattenMulOps (BinOpExp Mul {} e1 e2) = Product $ map toFlatPrimExp [e1, e2]
        flattenMulOps e = OpaquePrimExp e

        flattenProducts (Product es) = concatMap flattenProducts es
        flattenProducts e = [e]

makeTCEnv ::
  Env ->
  TCKernelParams ->
  [Stm GPU] ->
  Lambda GPU ->
  Lambda GPU ->
  [PrimType] ->
  Builder GPU TCEnv
makeTCEnv env kernel_params load_stms map_lam red_lam _map_ts = do

  tblock_offsets <-
    forM3 inner_dim_names tbids tiles_TR $
      \dim_name tbid tile_TR ->
        letSubExp ("tb_offset_" ++ dim_name)
          =<< toExp (le64 tbid * pe64 tile_TR)

  fmap (TCEnv kernel_params tblock_offsets map_lam red_lam)
    $ forM
```

```
      load_stms
    $ \load_stm -> do
      -- TODO: should probably gather all of the comments made here in a note.

      -- We need to extract the dimensions of each input array, and
      -- unfortunately the Redomap passed into this module only indirectly
      -- carries this information, as part of the kernel stms loading each
      -- redomap input slice. It'd be more convenient if the Redomap carried not
      -- only the VNames of its operands slices, but also the base arrays (if
      -- any) whence each slice comes, or at least the layout thereof.
      --
      -- In any case, knowledge of the actual layout of a given input array is
      -- necessary in order to correctly map the global-to-smem tile copy to
      -- the tblock dimensions (to obtain coalesced access on both smem
      -- inputs), as well as to generate the boundary guard on the read, and to
      -- match tile sizes to each input array, since these are not simply
      -- (M: (Ty, Ry)), (N: (Tx, Rx)), and (U: Tk) as in the 2D case.
      --
      -- Additionally, as it turned out, it was simply more convenient to load
      -- directly from these base arrays, rather than binding computed indices
      -- to gtids and inserting load statements.
      let base_arr = getArrayFromLoadStm load_stm
      arr_t <- lookupType base_arr
      let dims' = arrayDims arr_t
      let smem_elem_type = elemType arr_t

      -- In fact, we need not only the layout for each array, but also the index
      -- in the segspace of each dimension, s.t. later we may extract tblock
      -- offsets, loop variables, and other information associated with this
      -- given smem array. Below mess accomplishes this:
      --
      -- First, for each dimension in the array, determine the index into
      -- inner_dims of this dimension. Note that the indices computed here are
      -- the same as those returned by `variantDimsPerArr` for the given array,
      -- but in different order -- those computed by `variantDimsPerArr` are
      -- ordered by their occurence in the map nest (outermost first), while
      -- these are ordered by the array layout (outermost first).
      --
      -- Then, later in code generation, when we compute e.g. a set of tblock
      -- offsets or a set of loop indices based on the segspace, we can, for
      -- each input array, extract the tblock offsets and loop indices
      -- corresponding to this particular array.
      --
      -- Unfortunately, it is not quite as simple as that. If the array layout
      -- has been rearranged at some point before reaching this module, then we
      -- must reverse-engineer the original array layout from associated LMAD
      -- information. However, since LMADs do not carry permutation information,
      -- we must reverse-engineer it by trying all possible permutations of the
      -- known dimensions for the array (see function `findLMADPerm`). Again,
      -- none of this would be necessary if information on the base array was
      -- available somehow.
      -- If an array has not been rearranged, the identity permutation is
      -- recorded.
      lmad_perm <- findLMADPerm env base_arr
      let inv_lmad_perm = map snd $ L.sort $ zip lmad_perm [0 ..]

      let base_arr_dims = gather dims' inv_lmad_perm
      -- TODO: handle the case where multiple dimensions have the same name.
      let var_inds = map (`L.elemIndex` inner_dims) base_arr_dims

      -- Then, for each dimension of each array, extract the TR tile and
      -- tblock offset corresponding to this dimension. For the tile
    -- dims, we insert tile_Q in the index of the array dim not
    -- represented in inner_dims.
    let tile_dims = gather_ tiles_TR tile_Q var_inds
    let tile_dims_pe = map pe64 tile_dims

    -- Determine whether this array is a candidate for padding. If so, we need
    -- to take this into account in its flat size and the computed strides.
    let innerProducts = scanr (*) 1 . tail
    ~(smem_size_flat', smem_strides') <-

      -- An array is candidate for padding if one of its dimensions is indexed
      -- by the inner thread index UNLESS that dimension happens to also be
      -- innermost on the shared array.
      case Just inner_dim_ind `L.elemIndex` init var_inds of
        Just i -> do
          -- Split on the index at which the inner tiles need padding.
          let (outer_smem_dims, inner_smem_dims) = splitAt (i + 1) tile_dims_pe

          -- We only need padding when the inner size is a multiple of 2, so
          -- the padding term is `1 - (size_pre_pad % 2)`.
          -- TODO: I bind these two because I can't seem to get `rem` to work
          -- with TPrimExps. is there a better way?
          size_pre_pad <- letSubExp "size_pre_pad" =<< toExp (product inner_smem_dims)
          tmp <- letSubExp "tmp" $ BasicOp $ BinOp (SRem Int64 Unsafe) size_pre_pad se2
          pad_term <- letSubExp "pad_term" =<< toExp (1 - pe64 tmp)

          let inner_size_flat = pe64 size_pre_pad + pe64 pad_term

              outer_strides = init $ innerProducts $ outer_smem_dims ++ [inner_size_flat]
              inner_strides = innerProducts inner_smem_dims

              size_flat = product outer_smem_dims * inner_size_flat
          pure (size_flat, outer_strides ++ inner_strides)

        _ -> pure (product tile_dims_pe, innerProducts tile_dims_pe)

    smem_size_flat <- letSubExp "smem_size_flat" =<< toExp smem_size_flat'
    smem_strides <- mapM (letSubExp "smem_stride" <=< toExp) smem_strides'

    pure $
      ArrInfo
        base_arr
        base_arr_dims
        load_stm
        lmad_perm
        var_inds
        tile_dims
        smem_size_flat
        smem_strides
        smem_elem_type

  where
    getArrayFromLoadStm :: Stm GPU -> VName
    getArrayFromLoadStm (Let _ _ (BasicOp (Index arr _))) = arr
    getArrayFromLoadStm stm =
      error $
        "getArrayFromLoadStm error: expected a BasicOp Index stm, got: "
          ++ prettyString stm

    tbids = tbidVns kernel_params
    tiles_TR = tilesTR kernel_params
    tile_Q = tileQ kernel_params
    inner_dim_names = innerDimNames kernel_params
    inner_dims = innerDims kernel_params
    inner_dim_ind = length inner_dims - 1


-- Note [SmemZeroPaddingOnGlobalMemOOB]
-- When copying from global to shared memory, we need to handle out-of-bounds
-- reads from global memory. For the time being, we write a padding value to
-- shared memory. This padding value is a zero (or zero-like) value from the
-- corresponding element type.
--
-- However, this "solution" succeeds only when the following condition holds:
--
-- `f zero_0 _ = f _ zero_1 = red_ne`
--
-- where `f` is the map function; `zero_0` and `zero_1` are the zero-like values
-- for the two given smem array element types; and `red_ne` is the reduce
-- neutral element.
--
-- This is seldom the case in general, however it happens to hold for regular
-- tensor contraction and MM, hence it is used for testing for the time being.
--
-- The simple solution (and the one implemented) is the prologue/epilogue
-- treatment, in which the last iteration of the main reduction loop is unrolled
-- and a boundary guard corresponding to the one we had on global memory is
-- inserted into the register tile accumulation step s.t. we never process
-- garbage values in the reduction (or, at least, they do not affect those
-- entries of the register tile which are eventually written to global mem).
-- However, this will inevitably affect performance, and the difference is more
-- noticeable the less full tiles we have in the common dimension.
--
-- As an example, for regular MM of 2000x2000 matrices with a reduction dim tile
-- of Tk = 32, we will have floor(2000 / 32) = 62 full tiles and 1 partial tile,
-- so here the epilogue is largely amortized by the size of the prologue. But
-- for tensor contractions of higher-rank tensors, each dimension typically is
-- not very large. If we have, say, 30x30x30x30 tensors and a reduction dim tile
-- of Tk = 16, then we will have 1 full tile and 1 partial tile, and now the
-- epilogue dominates.
--
--
-- Another solution is to statically examine whether `zero_0` and `zero_1` exist
-- s.t. the above condition holds, but this analysis can be difficult or
-- impossible, and the values may not even exist.
--
-- Alternatively (on Cosmin's suggestion), the user can manually pass a padding
-- value as an attribute in the Futhark source code. Personally, I think this is
-- very hacky, obscure to most users, error-prone, and an anti-pattern. Also,
-- attributes only support passing integral values, not float values.

-- There is a big TODO in figuring out the best solution to this problem which
-- will also generalize best to arbitrary contractions.

-- Note [FlattenPrimExps]
-- In reverse-engineering LMAD permutations, we need to check equality between
-- LMAD strides. To do so, we in turn need to check equality between product
-- expressions. From commutativity and distributivity of multiplication, we of
-- course expect the two strides lists:
--
-- `[(a * b) * c, b * c, c, 1]`
--
-- and
--
```

```
-- `[(c * b) * a, c * b, c, 1]`
--
-- to be equal, since we have (a * b) * c = (c * b) * a, and so on.
--
-- However, the `Eq` instance for `PrimExp`s is not quite so sophisticated, so
-- we need a way to "normalize" product `PrimExp`s. To accomplish this, we
-- "flatten" nested `Mul` expressions and sort them (using the `Ord` instance
-- for `PrimExp`).
--
-- Example: Before flattening, the three `PrimExp` expressions:
--
-- exp1 = `BinOpExp Mul (BinOpExp Mul a b) c`
-- exp2 = `BinOpExp Mul (BinOpExp Mul c b) a`
-- exp3 = `BinOpExp Mul a (BinOpExp Mul b c)`
--
-- where a, b, c are `PrimExp`, would not test equal. However, all three
-- expressions flatten to:
--
-- `Product [OpaquePrimExp a, OpaquePrimExp b, OpaquePrimExp c]`
--
-- and hence we have `(exp1 == exp2) && (exp2 == exp3)`. Yay!
--
--
-- Note that if any of the expressions `a, b, c` are nested non-`Mul` `PrimExp`s
-- where ordering matters, then the flattening and sorting is not reliable.
-- As an example, the two expressions:
--
-- exp4 = `BinOpExp Mul (BinOpExp Add a (BinOpExp Mul b c)) d`
-- exp5 = `BinOpExp Mul (BinOpExp Add (BinOpExp Mul b c) a) d`
--
-- would "flatten" to
--
-- `Product [OpaquePrimExp (BinOpExp Add a (BinOpExp Mul b c)), OpaquePrimExp d]`
-- and
-- `Product [OpaquePrimExp (BinOpExp Add (BinOpExp Mul b c) a), OpaquePrimExp d]`
--
-- respectively, which would not test equal, meaning that in terms of testing
-- equality, this flattening is only reliable for simple product `PrimExp`s.
--
-- Hence it should be considered a proof of concept, and there is a big TODO in
-- making this reliable.
```