UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Bachelor's thesis

(aemylis, Æmilie Cholewa-Madsen, rjk148)
(sortraev, Anders Lietzen Holst, wlc376)

Teaching the Futhark compiler block and register tiled matrix multiplication

Supervisor: Cosmin E. Oancea
Secondary supervisor: Troels Henriksen

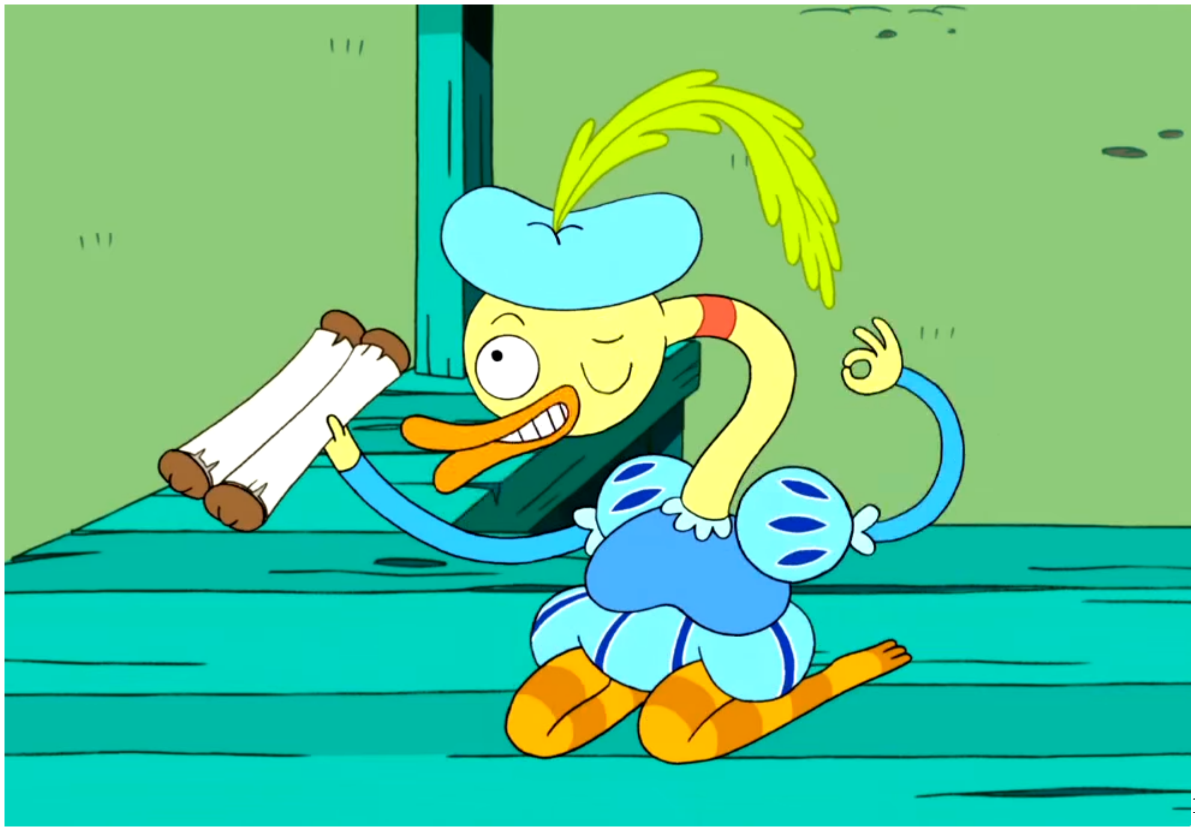
2020-06-08

Abstract

Matrix multiplication is a widely used and inherently parallel operator. Futhark, a high-level and purely functional programming language, generates efficient GPU code for matrix multiplication using block tiling. In this thesis, we explore how performance of matrix multiplication programs can improve from the block and register tiling code transformation, with the goal of implementing the optimization into the loop tiling pass of the Futhark compiler. With an offset in theoretical code transformation, we show how to apply block and register tiling to a pseudocode representation of an ordinary, parallel matrix multiplication program in order to improve its temporal locality of reference. We then show just how the program benefits in memory performance, and how the same transformation applies to a range of similarly structured programs. In a high-level overview, we present the Kernels stage of the Futhark compiler and design `RegTileReturns`; an addition to the compiler necessary in order to fully support the block and register tiling transformation. We discuss non-trivial parts of the implementation, including optimizations made to the handling of *residual input*, as well as our strategies for validation testing and benchmarking. Test results show promising performance speedups over the existing block-tiling across all tests on an Nvidia RTX 2080ti, whilst our implementation performs roughly as good as block-tiling on an older Nvidia GTX 780ti. We successfully implement block and register tiling of ordinary matrix multiplication in the Futhark compiler, but do not manage to generalize the transformation to other types of programs.

Resumé

Matrixmultiplikation er en meget brugt og i sagens natur parallel operator. Futhark, et højniveau, rent funktionelt programmeringssprog, genererer effektiv GPU-kode til matrixmultiplikation ved hjælp af blokflisebelægning. I dette projekt undersøger vi, hvordan ydeevnen af matrixmultiplikationsprogrammer kan forbedres vha. blok- og registerflisebelægningskodetransformation, med det formål, at implementere optimeringen i løkkeflisebelægningsgennemløbet i Futhark-oversætteren. Med afsæt i teoretisk kodetransformation viser vi, hvordan man udfører blok- og registerflisebelægning på en pseudokoderepræsentation af et ordinært, parallelt matrixmultiplikationsprogram for at forbedre dets temporale referencelokalitet. Vi viser derefter, hvordan programmets hukommelseseffektivitet drager fordel af transformationen, og hvordan den samme transformation gælder for en række lignende strukturerede programmer. På højt niveau præsenterer vi Kernels-stadiet af Futhark-oversætteren og designer `RegTileReturns`; en tilføjelse til oversætteren, der er nødvendig for fuldt ud at understøtte blok- og registerflisebelægningstransformationen. Vi diskuterer ikke-trivielle dele af implementeringen, inklusiv optimeringer af håndteringen af overskydende inddata, samt vores strategier for valideringsprøver og ydelsesmålinger. Testresultater viser lovende forbedringer i ydeevne over den eksisterende implementering af blokflisebelægning på tværs af alle afviklede prøver på et Nvidia RTX 2080ti-kort, mens der ydes omtrent lige så godt som blokflisebelægningen på et ældre Nvidia GTX 780ti-kort. Vi implementerer succesfuldt blok- og registerflisebelægning af ordinær matrixmultiplikation i Futhark-oversætteren, men lykkes ikke med, at generalisere transformationen til andre typer programmer.

[1]

[1] Screenshot from © Cartoon Networks *Adventure Time*, episode S02E03: "*Blood Under The Skin*"

Contents

0 Introduction

0.1 Motivation

Central in many data processing tasks from across the natural sciences - such as linear transformations used in signal/image processing; adjacency matrices used in graph analysis, curve fitting and classification algorithms seen in statistics, and so on - matrix multiplication (MM) is arguably the most important, or, at least, the most frequently used matrix operation. If within a given program a lot of processor time is going to be spent in matrix multiplication routines, then naturally, there is a benefit to optimizing this part of the code. Better performance can be achieved by writing programs for parallel hardware such as a GPU, and we know that matrix multiplication can be implemented as a dot product in a **map** nest, the latter of which is inherently parallel [1].

However, despite the large degree of inherent parallelism in matrix multiplication, hardware limitations, such as the slow speed of main memory, impose a bottleneck on performance gained from simply parallelizing. In general, the performance of GPU programs is in very large part dictated by memory efficient programming [2] [1]. For most types of problems, spatial locality is easily obtained through coalesced access; conversely, temporal locality is typically not as straightforwardly exploited.

Tiling is a code transformation which comprises stripmining, interchanging and distributing parallel loops, or **map** nests in the functional setting. The transformations changes the order of computation, without changing the semantics, in such a way that locality of reference can be improved. Whether tiling actually improves performance, and by how much, depends on both the program and the hardware it is executed on.

In matrix multiplication, single elements from input matrices are reused multiple times in computing the product; this would indicate an opportunity for temporal locality which is not exploited in the naive implementation, but which could be in a tiled version.

Futhark is a high-level language designed for easy GPU code generation. As of now, the Futhark compiler recognizes patterns of matrix multiplication in the high-level Futhark code and generates efficient GPU code using block tiling. A handwritten OpenCL kernel for matrix multiplication using both block and register tiling has shown significant improvement over a block tiled kernel generated by the Futhark compiler.

0.2 Project Overview

The goals of this project are to analyze the two tiling techniques *block tiling* and *register tiling* and their optimizations to parallel matrix multiplication programs; to examine and expand the Futhark Kernels IR with support for register tiling in addition to block tiling. This will all enable prototyping a block and register tiling pass in the Futhark optimization stage.

In section 1, we first give some needed background on an abstract GPU hardware model resembling that of the types of devices we target.

Then, in section 2, we review basic theory on tiling transformations in a step-by-step block and register tiling transformation of an ordinary matrix multiplication program, before analyzing how the tiled program might translate to a GPU kernel as well as how it might improve temporal locality in the program. At the end of this section, we discuss how to generalize the optimization to a wide range of programs similar to MM.

Section 3 goes in-depth with key features of the Kernels stage of the Futhark compiler, in which we implement the new tiling pass. Here, we also discuss the design of an addition to the Kernels IR in the new `KernelResult`, `RegTileReturns`. This is used in an abstract sketch of what a block and register tiled matrix multiplication program looks like in the Kernels IR.

Implementation of the aforementioned prototype is inspired by the block tiling pass already present in the compiler. In section 4, we describe problems encountered in implementing block and register tiling and their solutions, as well as the limitations of our implementation.

Finally, in sections 5 and 6, we validate and benchmark our implementation before evaluating the project.

---

1 On programming for GPU hardware

In this first section, we find it fitting to present the GPU hardware and execution model, and to talk about the GPU memory hierarchy. Even though Futhark is designed to be hardware agnostic, it primarily targets CUDA/OpenCL [2], and then so do we; whenever we discuss GPU hardware and programming for such devices throughout the report, we shall refer to a hardware model that is abstract but compatible with CUDA/OpenCL hardware models.

With respect to the GPU memory hierarchy, we give only a *very abstract* overview of the memory hierarchy, and focus only on the *global, local*, and *private* memory layers.

The field of GPU programming has borrowed a lot of names from other fields and overloaded these with new meanings; in this section, we will disambiguate some of the necessary terminology used in the report.

---

[2] `https://futhark-lang.org/`

1.1 GPU hardware and execution levels

Table 1 is a reference sheet for the names we use to refer to the different hardware levels of the GPU [2].

Level	**Description**
Thread	The smallest unit of execution on the GPU; like a CPU core, each thread has its own number of registers, but for example differs in that it shares an I-cache with a warp of nearby threads. Each thread has a unique ID in its group.
Warp	A warp consists of exactly 32 threads, which share an I-cache and which thus execute in lockstep. Threads in a warp can communicate very fast with each other, and make global memory accesses in unison.
Group	Threads are divided into 1, 2, or 3-dimensional groups of up to 1024 threads; since threads exist in warps of 32, group dimensions will typically be multiples of 32. Threads in a group share the same space of *local memory*.
Grid	The singular 1, 2, or 3-dimensional *grid* holds all groups of threads, but only a certain number of groups can be scheduled at any given time. Threads across all groups in the grid share the same *global memory*.
Kernel	A function specifically for execution on the GPU. A kernel is simultaneously executed by all threads in the grid, and thus up to one kernel may run at any given time.

Table 1: GPU-lingo reference sheet - hardware levels. [2]

Hardware levels

GPU computations are performed inside so-called *kernels*. As stated in table 1, a kernel is executed simultaneously by all scheduled threads on the GPU, and so the notion of hardware levels might seem ambiguous or redundant, but each hardware level does have distinct structural characteristics; however, for the purposes of this project, we will primarily need to think of an execution level as an indication of the hardware-level at which the result of an operation is manifested - more on this in section 1.2.

Execution levels

We might find it convenient to think of a computation as taking place on a certain *execution level*; we distinguish between *grid-*, *group-*, and *thread*-levels of execution, and computation on these three execution levels correspond to inter-group, intra-group, and thread-private operations, respectively.

When programming a parallel application for execution on a GPU, one can think of execution levels as a conceptualization of levels of parallelism in the application - in other words, levels of parallelism can be mapped directly to execution levels of execution.

As a basic example, two nested levels of parallelism surrounding a procedure could be mapped to a 1-dimensional grid of 1-dimensional groups as such:

```
foreach group in grid
  foreach thread in group
    do procedure
```

Similarly, four nested levels of parallelism could be just as easily mapped to a two-dimensional grid of two-dimensional groups as such:

```
foreach row_of_groups in grid
  foreach group in row_of_groups

    foreach row_of_threads in group
      foreach thread in row_of_threads
        do procedure
```

We shall see the latter example realized later in the report.

1.2 The GPU memory hierarchy

Table 2 is a reference sheet for the names we use to refer to the different levels in the GPU memory hierarchy.

Level	**Description**
Register memory	The fastest, smallest type of memory, private to the individual thread. The number of registers per thread varies according to the number of scheduled threads and groups. As a general rule, we assume it to be no more than 64. [3]
Private memory	A somewhat ambiguous term which we use to specify thread-private memory which can - and which we would ideally like to see - be stored in register memory. We use this term to make the distinction that not all thread-private memory is eligible for register storage (eg. when register spilling occurs [4]).
Local memory	User-managed, on-chip memory shared among threads in a group. Much larger and slower than register memory, but much faster and smaller than global memory. The size of local memory is 48 KiB per group. [5]
Global memory	The main, off-chip DRAM memory of the GPU, shared among all threads in the grid, and accessible by the host device, eg. a CPU. The size of global memory varies greatly, but is eg. 3 and 10 GB, respectively, for the GPUs we use.

Table 2: GPU-lingo reference sheet 2 - memory hierarchy

[3]Each SM has 64K registers (128K for CC (compute capability) 3.7). a maximum of 64K/255 registers can be used per group/thread for CC 3.2+ devices; else 32K/63 per group/thread [3].

[4]Registers are spilled to off-chip (ie. near global memory) thread-private memory - in OpenCL terminology, ”private memory” (equivalent to CUDA’s ”local memory”) is used to refer specifically to this off-chip memory.

[5]CC 7.0+ devices have more, but must dynamically allocate requests of > 48 KiB [3].

1.3 Global memory coalescing on the GPU

In some over-simplification, *global memory coalescing* is the parallel hardware analogue to spatial locality of reference in the CPU setting.

As stated in table 1, threads in a warp make global memory requests in unison. The GPU will attempt to *coalesce* reads and writes made by a warp from/to global memory into as few transactions as possible to minimize bandwidth. However, coalescing is only possible when a warp requests memory locations within a single cache line in global memory; if they do not, then multiple transactions are necessary [4].

Below illustration [6] shows how three threads `t0`, `t1`, and `t2` access nine addresses in memory across three loop iterations marked in red, yellow, and blue, respectively, in uncoalesced and coalesced fashion:

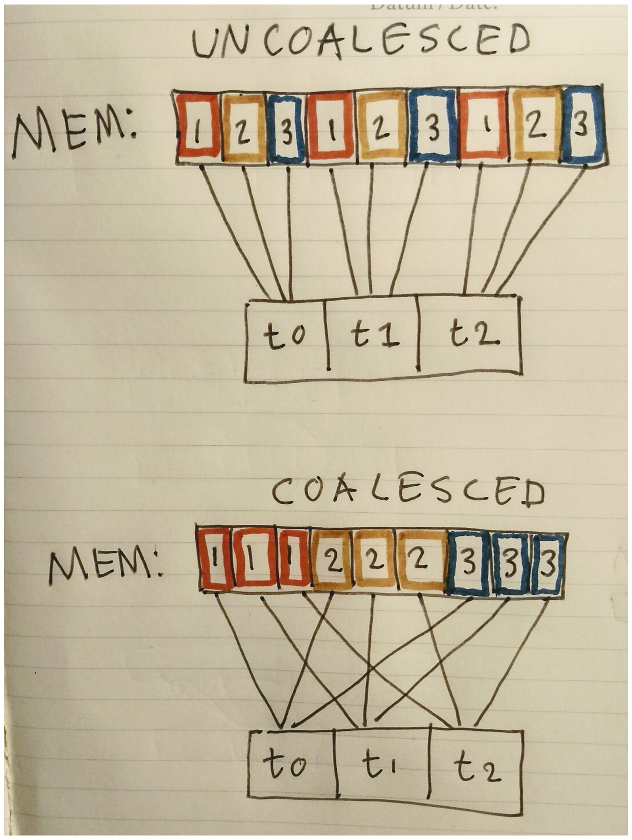


Figure 1: Coalesced and uncoalesced access

.

[6] *We have in our midst a bachelor of the arts.*

If we assume a one-dimensional group of size `T = 3`, then the functions describing the above access patterns would be:

```
// uncoalesced access pattern.
for (i = 0; i < T; i++)
  foo = mem[thread_index + i*T];
```

```
// coalesced access pattern.
for (i = 0; i < T; i++)
  foo = mem[thread_index*T + i];
```

In the uncoalesced access case, each thread goes in a sequential stride across iterations, but since other threads simultaneously access memory `T` addresses away, this effectively becomes a stride `T` access pattern - not ideal. In the coalesced access case, individual threads access memory in a `T`-stride across iterations, which results in a stride-1 access pattern within a given iteration.

- **Collective copy**

As stated, threads in warps should ideally access adjacent locations in global memory - but what if within a given procedure each thread requires data that is in a sequentially layed out (ie. in a stride-1 pattern) in global memory?

Recall that threads in a group share the same space of fast *local memory*. Local memory does not suffer nearly the same latencies from uncoalesced accesses as global memory does, so it may be beneficial to perform a coalesced read from global memory into local memory before doing an uncoalesced read (eg. a permutation of the first read) from local memory into thread-private memory [2].

This is called *collective copying*, since threads read elements of global memory which are actually needed by other threads. More on this in sections 3.2.1 and 4.3.

2 The block and register tiling transformation and its benefits

Having given background on GPU hardware, we want to introduce theory on tiling. We wish to show and explain how to transform a program to make it eligible for tiling, aswell as the tiling itself, and we shall also explain why the transformation is legal. Then, we want to analyze how a tiled program can be mapped to hardware and how tiling can improve performance.

Below, in the left-most snippet, is a basic MM (matrix multiplication) program [7] in the Futhark source language.

```
map (\A_row ->
  map (\B_col ->
      map2 (*) A_row B_col
        |> reduce (+) 0
      ) (transpose B)
    ) A
```

⇒

```
map (\A_row ->
  map (\B_col ->
      loop acc = 0 for k < U do
        acc + A_row[k] * B_col[k]
      ) (transpose B)
    ) A
```

⇒

```
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    acc = 0
    for (k = 0; k < U; k++)
      acc += A[i, k] * B[k, j];
    C[i, j] = acc;
```

The program consists of a map nest over some U-dimensional row and column vectors of A and B; inside this map nest is a pair-wise multiplication of each row and column, followed by a summation of products - in other words, here is a dot product. This is expressed as a **map-reduce**-composition, or **redomap**. In the middle snippet, this **redomap** is translated to a regular for loop over the U dimension, and in the right-most snippet, the Futhark program is translated to C-like pseudocode.

In this section, we consider the transformation of the ordinary MM program as it appears in the right-most snippet. In the end, we discuss how block and register tiling[8] can be beneficial in a broader sense, as the transformation applies to not only MM, but a range of programs which follow a similar structure.

We use C-like pseudocode for the remainder of this section. We expect a lot of nested loops, so for brevity, we use indentation to indicate scope, and whitespace for readability. Changes and important features are highlighted in gray as above, and annotated with in-code comments.

[7] The function implicitly takes as input two arrays A and B of dimensions M * U and U * N, respectively, and stores the result is some array C of dimensions M * N.

[8] Register tiling is also known as *unroll and jam.*

2.1 Step-by-step block/register tiling transformation

In this section, we present a step-by-step walkthrough of the block and register tiling transformation of a ”naive” MM program.

- **Step 0: Loop dependency analysis**

Consider listing 1; pseudocode for ”naive” matrix multiplication, as presented previously. The two outer `i`- and `j`-loops are parallel. This is because `acc` is private for these two loops, and because in the `(i,j)`'th iteration, only `C[i, j]` is written, so each iteration is independent of the other, and they can be executed in parallel. On the other hand, the inner loop of index `k` is not; here, `acc` is read from and then written to on every iteration, so there is a write-after-read dependency aswell as a read-after-write (ie. a true) dependency.

This is also apparent in inspection of the dependency matrix [9] for the program. Dependency vectors for each statement are shown in comments to the pseudocode. The two outer loops are parallel since the first two columns in the dependency matrix are all ’=’. The inner loop is not parallel, since its column in the dependency matrix contains a ’<’ not preceded by another ’<’, meaning this dependency is not carried by any outer loop[1].

```
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    acc = 0;                    ! [=, =]

    for (k = 0; k < U; k++)
      acc += A[i, k] * B[k, j];  ! [=, =, <]
    C[i, j] = acc;              ! [=, =]
```

Listing 1: Dependency analysis of naive matrix multiplication

As per the loop dependency analysis, the two outer loops can safely be parallelized. The inner loop of index `k` is kept sequential [10].

[9] A dependency matrix is one whose columns correspond to loops in a program, and whose row vectors correspond to statements nested in these loops; each element is either a ’=’, corresponding to an intra-iteration dependency, ’>’, or ’<’, corresponding to dependencies on future and past iterations, respectively[1].

[10] The inner `k` loop could, in fact, also be parallelized to some degree, by realizing that it is a **redomap** (it fits the form `acc = acc + exp`, where `exp` does not contain `acc`)[1], but as we shall see, this transformation and its implementation into Futhark is the matter of future work and so we do not pursue it in this analysis.

```
forall (i = 0; i < M; i++)       ! parallelized
  forall (j = 0; j < N; j++)     ! parallelized
    acc = 0;

    for (k = 0; k < U; k++)      ! left sequential
      acc += A[i, k] * B[k, j];
    C[i, j] = acc;
```

Listing 2: Outer loops parallelized

- **Step 1: Stripmining**

  The first step of loop tiling is to stripmine. Stripmining is a loop transformation which splits the iteration space of a single, normalized loop into chunks, or *tiles*, of some fixed tile size `T`. The loop is transformed into two perfectly nested loops, where the outer loop goes in stride `T`, and the inner loop "fills the gaps" with a stride of 1 [1].

  This code tranformation is always safe since it does not change the order (or semantics) of operations.

  A loop can be stripmined twice by first stripmining with some tile size `T2`, and then stripmining the resulting stride-1 loop with some other tile size `T`. This process can be repeated to tile a loop an arbitrary number of times. [11]

  We wish to perform the following stripminings:

  - the outermost loop of index `i` is doubly stripmined with strides of `Ty*Ry` and `Ry` in two new loops of indices `iii` and `ii`, respectively. This tiles the `M` rows of the first operand matrix `A`.
  - the second-outermost loop of index `j` is doubly stripmined with strides of `Tx*Rx` and `Rx` in two new loops of indices `jjj` and `jj`, respectively. This tiles the `N` columns of the second operand matrix `B`.
  - the inner loop of index `k` is stripmined with a stride of `Tk` in a new loop of index `kk`. This tiles the common dimension `U`.

[11] We can, of course, make no assumptions about the iteration space of the loop to be stripmined, so we will need to check bounds in case the outer tile size does not divide the iteration space it tiles. However, to avoid checking bounds on inner tilings, we simply let the outer tile sizes be multiples of the inner tile sizes.

- **Tile size parameters**

The reader should acquaint themselves with parameters `Ty, Tx, Tk, Ry,` and `Rx`.

We shall for the remainder of the report refer to `Ty*Ry` and `Ry` as the *outer block tile* and *register tile sizes*, respectively; similarly, we call `Tx*Rx` and `Rx` the *inner block tile* and *register tile sizes*, respectively, and `Tk` the "**redomap** tiling" size, since this parameter tiles the innermost sequential dimension pertaining to the **redomap**.

With respect to block/register tiling, `Ty` and `Tx` in themselves are not tiling parameters, but rather specify group dimensions. However, in block tiling, as we shall also touch upon, these specify the inner and outer block tile sizes, respectively.

The loops resulting from stripmining the outer loops remain parallel while the inner loop remains sequential.

```
forall (iii = 0; iii < M; iii += Ty*Ry)
  forall (ii = iii; ii < min(M, iii + Ty*Ry); ii += Ry)        ! double stripmine of outer i loop
    forall (i = ii; i < min(M, ii + Ry); i++)

      forall (jjj = 0; jjj < N; jjj += Tx*Rx)
        forall (jj = jjj; jj < min(N, jjj + Tx*Rx); jj += Rx)  ! double stripmine of middle j loop
          forall (j = jj; j < min(N, jj + Rx); j++)

            acc = 0;
            for (kk = 0; kk < U; kk += Tk)
              for (k = kk; k < min(U, kk + Tk); k++)           ! single stripmine of inner redomap loop
                acc += A[i, k] * B[k, j];

            C[i, j] = acc;
```

Listing 3: Stripmining of all loops

- **Step 2: Loop interchange**

  Next step of block tiling the two outer loops is loop interchange. Loops of stride 1 (ie. those of index `i` and `j`) are interchanged innermost. Loops of indices `ii` and `jj` are interchanged inside those of indices `iii` and `jjj`. This code transformation is legal because it is always safe to interchange a parallel loop inwards in a perfect loop nest. [12]

```
forall (iii = 0; iii < M; iii += Ty*Ry)                      ! outer tilings are moved outermost
  forall (jjj = 0; jjj < N; jjj += Tx*Rx)

    forall (ii = iii; ii < min(M, iii + Ty*Ry); ii += Ry)    ! inner tilings are interchanged
      forall (jj = jjj; jj < min(N, jjj + Tx*Rx); jj += Rx)  ! to second-outermost position

        forall (i = ii; i < min(M, ii + Ry); i++)            ! stride 1 loops interchanged innermost
          forall (j = jj; j < min(N, jj + Rx); j++)

            acc = 0;
            for (kk = 0; kk < U; kk += Tk)
              for (k = kk; k < min(U, kk + Tk); k++)
                acc += A[i, k] * B[k, j];

            C[i, j] = acc;
```

Listing 4: Interchanging parallel loops

[12]In general, loop interchange is legal if the transformation does not result in an invalid direction matrix, meaning no row has '>' as their first non-'='-symbol. Parallel loops will always remain parallel when interchanged inwards. We can see this by looking at their column in the direction matrix. The '='-symbols does not change and any non-'='-symbol will still be preceded by a dependency carrying '<'-symbol, since inwards interchanging corresponds to interchaging the column to the right [1].

- **Step 3a: Array expansion of acc before distribution**

  We want to distibute the perfect loop nest consisting of the loops with indices ii, jj, i, and j over the statements inside it - the zero initialization of acc; the dot product, and write to C.

  Because acc is private for each iteration we need to move declaration of acc outside the loop nest and expand it into an array. In this way, instructions that used to be executed in the same iteration can access the same private index into acc across the distributed loops.

  The dimensions of the expanded array correspond to the dimensions of the four loops in the loop nest that is to be dstributed, such that each iteration has a private element in the array.

```
forall (iii = 0; iii < M; iii += Ty*Ry)
  forall (jjj = 0; jjj < N; jjj += Tx*Rx)

    acc[Ty, Tx, Ry, Rx];            ! 4D array expansion of acc

    forall (ii = iii; ii < min(M, iii + Ty*Ry); ii += Ry)
      forall (jj = jjj; jj < min(N, jjj + Tx*Rx); jj += Rx)

        forall (i = ii; i < min(M, ii + Ry); i++)
          forall (j = jj; j < min(N, jj + Rx); j++)

            acc[ii-iii, jj-jjj, i-ii, j-jj] = 0;
            for (kk = 0; kk < U; kk += Tk)
              for (k = kk; k < min(U, kk + Tk); k++)
                acc[ii-iii, jj-jjj, i-ii, j-jj] += A[i, k] * B[k, j];

            C[i, j] = [ii-iii, jj-jjj, i-ii, j-jj];
```

Listing 5: Array expansion of acc

- **Step 3b: Loop distribution**

  We can now distribute the loop nest over the three statements inside: zero-initialization of `acc`, the inner **redomap** loop nest, and the write to `C`. This distribution is legal as long as the order of statements is preserved, since there are no circular dependencies between statements; only dependencies on preceding statements [1].

```
forall (iii = 0; iii < M; iii += Ty*Ry)
  forall (jjj = 0; jjj < N; jjj += Tx*Rx)

    acc[Ty, Tx, Ry, Rx];

    forall (ii = iii; ii < min(M, iii + Ty*Ry); ii += Ry)       ! ii, jj, i, and j distributed
      forall (jj = jjj; jj < min(N, jjj + Tx*Rx); jj += Rx)     ! over zero init of acc
        forall (i = ii; i < min(M, ii + Ry); i++)
          forall (j = jj; j < min(N, jj + Rx); j++)
            acc[ii-iii, jj-jjj, i-ii, j-jj] = 0;

    forall (ii = iii; ii < min(M, iii + Ty*Ry); ii += Ry)       ! ii, jj, i, and j distributed
      forall (jj = jjj; jj < min(N, jjj + Tx*Rx); jj += Rx)     ! over the redomap loop nest
        forall (i = ii; i < min(M, ii + Ry); i++)
          forall (j = jj; j < min(N, jj + Rx); j++)

            for (kk = 0; kk < U; kk += Tk)
              for (k = kk; k < min(U, kk + Tk); k++)
                acc[ii-iii, jj-jjj, i-ii, j-jj] += A[i, k] * B[k, j];

    forall (ii = iii; ii < min(M, iii + Ty*Ry); ii += Ry)       ! ii, jj, i, and j distributed
      forall (jj = jjj; jj < min(N, jjj + Tx*Rx); jj += Rx)     ! over write to result array
        forall (i = ii; i < min(M, ii + Ry); i++)
          forall (j = jj; j < min(N, jj + Rx); j++)
            C[i, j] = acc[ii-iii, jj-jjj, i-ii, j-jj];
```

Listing 6: Distributing parallel loops of indices `ii`, `jj`, `i`, `j`

- **Step 4: More loop interchange**

  We now want to interchange the outer parallel `ii` and `jj` loops inside the outer sequential `kk` loop, and the inner parallel `i` and `j`-loops inside the inner sequential `k` loop.

  This is legal with the same argumentation as for the previous loop interchange; that it is always legal to interchange parallel loops inwards in a perfect loop nest, and here, loops of indices `ii`, `jj`, `i`, and `j` are parallel and reside in a perfect loop nest with the loops of indices `kk` and `k`.

```
forall (iii = 0; iii < M; iii += Ty*Ry)
  forall (jjj = 0; jjj < N; jjj += Tx*Rx)

    acc[Ty, Tx, Ry, Rx];
    forall (ii = iii; ii < min(M, iii + Ty*Ry); ii += Ry)
      forall (jj = jjj; jj < min(N, jjj + Tx*Rx); jj += Rx)
        forall (i = ii; i < min(M, ii + Ry); i++)
          forall (j = jj; j < min(N, jj + Rx); j++)
            acc[ii-iii, jj-jjj, i-ii, j-jj] = 0;

    for (kk = 0; kk < U; kk += Tk)
      forall (ii = iii; ii < min(M, iii + Ty*Ry); ii += Ry)
        forall (jj = jjj; jj < min(N, jjj + Tx*Rx); jj += Rx) ! redomap loops and par-
          for (k = kk; k < min(U, kk + Tk); k++)              ! allelism interchanged
            forall (i = ii; i < min(M, ii + Ry); i++)
              forall (j = jj; j < min(N, jj + Rx); j++)
                acc[ii-iii, jj-jjj, i-ii, j-jj] += A[i, k] * B[k, j];

    forall (ii = iii; ii < min(M, iii + Ty*Ry); ii += Ry)
      forall (jj = jjj; jj < min(N, jjj + Tx*Rx); jj += Rx)
        forall (i = ii; i < min(M, ii + Ry); i++)
          forall (j = jj; j < min(N, jj + Rx); j++)
            C[i, j] = acc[ii-iii, jj-jjj, i-ii, j-jj];
```

Listing 7: Interchanging inner parallelism inside sequential `kk` and `k` loops

- **Step 5a: Sequentialize innermost stride-1 `forall` loops**

  Eventually, we are going to want to load slices of A and B into private (register) memory. To do this, we need to sequentialize one or more of the (now parallel) innermost stride-1 loops.

  We choose the innermost loops of indices `i` and `j`, which have `Ry` and `Rx` iterations, respectively, corresponding to the innermost two dimensions of `acc`, such that these loops can be unrolled per thread and the innermost two dimensions of `acc` be scalarized and stored in register memory. In listing 8 below, we sequentialize the innermost loop dimensions of indices `i` and `j`.

```
forall (iii = 0; iii < M; iii += Ty*Ry)
  forall (jjj = 0; jjj < N; jjj += Tx*Rx)

    acc[Ty, Tx, Ry, Rx];
    forall (ii = iii; ii < min(M, iii + Ty*Ry); ii += Ry)
      forall (jj = jjj; jj < min(N, jjj + Tx*Rx); jj += Rx)
        for (i = ii; i < min(M, ii + Ry); i++)                  ! sequentialized
          for (j = jj; j < min(N, jj + Rx); j++)                ! sequentialized
            acc[ii-iii, jj-jjj, i-ii, j-jj] = 0;

    for (kk = 0; kk < U; kk += Tk)
      forall (ii = iii; ii < min(M, iii + Ty*Ry); ii += Ry)
        forall (jj = jjj; jj < min(N, jjj + Tx*Rx); jj += Rx)
          for (k = kk; k < min(U, kk + Tk); k++)
            for (i = ii; i < min(M, ii + Ry); i++)              ! sequentialized
              for (j = jj; j < min(N, jj + Rx); j++)            ! sequentialized
                acc[ii-iii, jj-jjj, i-ii, j-jj] += A[i, k] * B[k, j];

    forall (ii = iii; ii < min(M, iii + Ty*Ry); ii += Ry)
      forall (jj = jjj; jj < min(N, jjj + Tx*Rx); jj += Rx)
        for (i = ii; i < min(M, ii + Ry); i++)                  ! sequentialized
          for (j = jj; j < min(N, jj + Rx); j++)                ! sequentialized
            C[i, j] = acc[ii-iii, jj-jjj, i-ii, j-jj];
```

Listing 8: Sequentializing innermost loops of indices `i` and `j`

- **Step 5b: Unroll sequential stride-1 loops**

In order to scalarize `acc` (so the two inner dimesions can be stored in registers), we need to unroll the loops of indices `i` and `j` [13]. Loop unrolling can be performed on any loop which has compile-time constant bounds, and the first step in unrolling is normalization [1].

- **Loop normalization and unrolling**

A normalized loop is one whose loop variable starts at 0 and goes to some upper bound with a stride of 1; note that compile-time constant bounds are not required. Loop normalization is the code transformation of a loop such that it adheres to these criteria, and is performed as such:

```
for (i = lo; i < hi; i += k)  =>  for (i0 = 0; i0 < ceil((hi-lo)/k); i0++)
  loop_body(i);               =>    i = lo + i0*k;
                              =>    loop_body(i);
```

Immediately inside the loop body and before it it used, the loop variable is ”de-normalized”, such that its value in a given iteration is maintained.

As an example, we show normalization of the loop of index `k` - even though it is not related to scalarization of `acc`, we we eventually want to unroll this loop as well for performance reasons. For brevity, we start by moving the assertion of `k < U` inside the loop body:

```
for (k = kk; k < min(U, kk + Tk); k++)  =>  for (k = kk; k < kk + Tk; k++)
  loop_body(k);                         =>    if (k < U) loop_body(k);
```

And now, normalization:

```
for (k = kk; k < kk + Tk; k++) => for (k = 0; k < ceil((kk + Tk - kk) / 1); k++)
  if (k < U) loop_body(k);     =>   if (kk + k*1 < U) loop_body(kk + k*1);

                               => for (k = 0; k < Tk; k++)
                               =>   if (kk + k < U) loop_body(kk + k);
```

With the loop normalized, it is ready to be unrolled. This is where the compile-time constantness requirement comes into play: the compiler must know how many iterations to unroll. Below listing shows how `Tk` iterations of the `k` loop are unrolled:

```
for (k = 0; k < Tk; k++)               =>  if (kk < U)      loop_body(kk);
  if (kk + k < U) loop_body(kk + k);   =>  if (kk + 1 < U) loop_body(kk + 1);
                                       =>  if (kk + 2 < U) loop_body(kk + 2);
                                       =>  ...
                                       =>  if (kk + Tk - 1 < U) loop_body(kk + Tk - 1);
```

[13] Loop unrolling also has the performance benefit of removing a number of loop variable increments and loop condition evaluations equal to the number of iterations of the original loop, which of course comes at the cost of replicated code.

However, since we make no assumptions as to the tile sizes at the point of code transformation, we must leave unrolling for later stages (ie. of whatever compiler that eventually performs the actual transformation) and for now simply normalize.

Performing now loop normalization on all sequential loops of stride 1:

```
forall (iii = 0; iii < M; iii += Ty*Ry)
  forall (jjj = 0; jjj < N; jjj += Tx*Rx)

    acc[Ty, Tx, Ry, Rx];
    forall (ii = iii; ii < min(M, iii + Ty*Ry); ii += Ry)
      forall (jj = jjj; jj < min(N, jjj + Tx*Rx); jj += Rx)
        for (i = 0; i < Ry; i++)                        ! normalized
          for (j = 0; j < Rx; j++)                      ! normalized
            if (ii + i < M && jj + j < N)               ! inserted bounds check
              acc[ii-iii, jj-jjj, i, j] = 0;

    for (kk = 0; kk < U; kk += Tk)
      forall (ii = iii; ii < min(M, iii + Ty*Ry); ii += Ry)
        forall (jj = jjj; jj < min(N, jjj + Tx*Rx); jj += Rx)
          for (k = 0; k < Tk; k++)                      ! normalized
            if (kk + k < U)                             ! inserted bounds check
              for (i = 0; i < Ry; i++)                  ! normalized
                for (j = 0; j < Rx; j++)                ! normalized
                  if (ii + i < M && jj + j < N)         ! inserted bounds check
                    acc[ii-iii, jj-jjj, i, j] += A[ii+i, kk+k] * B[kk+k, jj+j]; ! updated indexing

    forall (ii = iii; ii < min(M, iii + Ty*Ry); ii += Ry)
      forall (jj = jjj; jj < min(N, jjj + Tx*Rx); jj += Rx)
        for (i = 0; i < Ry; i++)                        ! normalized
          for (j = 0; j < Rx; j++)                      ! normalized
            if (ii + i < M && jj + j < N)               ! inserted bounds check
              C[i+ii, j+jj] = acc[ii-iii, jj-jjj, i, j];
```

Listing 9: Normalizing sequential loops of indices `i`, `j`, and `k`

This step concludes the block and register tiling code transformation.

2.2 Mapping transformed program to abstract hardware

Having finished code transformation of naive matrix multiplication into a block and register tiled equivalent in pseudocode, we now want to examine how the transformed program most efficiently can be mapped to *abstract* parallel hardware, as described in section 1 - after all, the end goal is to implement our program for high-performance execution on a GPU. This will help us to analyze how tiling improve temporal locality of reference and give insigt in how to translate the psuedocode into a GPU kernel.

Below listing 10 shows the code after the block and register tiling tranformation - only now, it is annotated with comments indicating which loops are parallel, and for each parallel loop; which hardware level this particular loop corresponds to. Highlighted lines show where tiles are copied to and from global, local, and private memory. This is all elaborated on in the following subsections.

```
forall (iii = 0; iii < M; iii += Ty*Ry)                          ! parallel - group offset y
  forall (jjj = 0; jjj < N; jjj += Tx*Rx)                        ! parallel - group offset x

    acc[Ty, Tx, Ry, Rx];                                         ! inner two dimensions scalarized
    forall (ii = iii; ii < min(M, iii + Ty*Ry); ii += Ry)        ! parallel - thread offset y
      forall (jj = jjj; jj < min(N, jjj + Tx*Rx); jj += Rx)      ! parallel - thread offset x
        for (i = 0; i < Ry; i++)                                 ! sequential - unrolled
          for (j = 0; j < Rx; j++)                               ! sequential - unrolled
            if (ii + i < M && jj + j < N)
              acc[ii-iii, jj-jjj, i, j] = 0;

    for (kk = 0; kk < U; kk += Tk)                               ! sequential

      A_loc = A[iii : iii + Ty*Ry, kk : kk + Tk]                 ! threads in a group collectively copy 2D slices
      B_loc = B[kk : kk + Tk, jjj : jjj + Tx*Rx]                 ! of A and B from global to local memory

      forall (ii = iii; ii < min(M, iii + Ty*Ry); ii += Ry)      ! parallel - thread offset y
        forall (jj = jjj; jj < min(N, jjj + Tx*Rx); jj += Rx)    ! parallel - thread offset x

          for (k = 0; k < Tk; k++)                               ! sequential
            if (kk + k < U)

              A_reg = A_loc[ii : ii + Ry, k]                     ! each thread copies 1D slices of A_loc and
              B_reg = B_loc[k, jj : jj + Rx]                     ! B_loc from local to private (register) memory.

              for (i = 0; i < Ry; i++)                           ! sequential - unrolled
                for (j = 0; j < Rx; j++)                         ! sequential - unrolled
                  if (ii + i < M && jj + j < N)
                    acc[ii-iii, jj-jjj, i, j] += A_reg[i] * B_reg[j] ! read from private memory

    forall (ii = iii; ii < min(M, iii + Ty*Ry); ii += Ry)        ! parallel - thread offset y
      forall (jj = jjj; jj < min(N, jjj + Tx*Rx); jj += Rx)      ! parallel - thread offset x
        for (i = 0; i < Ry; i++)                                 ! sequential - unrolled
          for (j = 0; j < Rx; j++)                               ! sequential - unrolled
            if (ii + i < M && jj + j < N)
              C[ii+i, jj+j] = acc[ii-iii, jj-jjj, i, j];
```

Listing 10: Mapping transformed code to abstract hardware

2.2.1 Mapping parallelism to hardware levels

The two outermost forall-loops of indices iii and jjj can be mapped to a two dimensional grid. These loops have strides of Ty*Ry and Tx*Rx (ie. 2D block tile stride), respectively, which can be seen as the *block tile offset* into A and B in global memory, respectively, of a particular group.

The iii and jjj loops, of course, have ⌈M / Ty*Ry⌉ and ⌈N / Tx*Rx⌉ number of iterations, respectively; if we let these be the dimensions of the grid (ie. the number of groups in each direction), such that each group perform one of the iterations, then each group can de-normalize their group indices into block tile offsets by:

```
blockTileOffset_y = groupId_y * Ty * Ry <=> iii,
blockTileOffset_x = groupId_x * Tx * Rx <=> jjj
```

The forall-loops of indices ii and jj can be mapped to two dimensional groups. These loops have strides of Ry and Rx (ie. 2D register tile stride), respectively, which can be seen as the *register tile offset* into A_loc and B_loc in local memory, respectively, of a particular thread.

The ii and jj loops, have Ty and Tx iterations, respectively; if we let these be the group dimensions (ie. the number of threads per groups in each direction), such that each thread perform one of the iterations, then each thread can de-normalize their thread indices into register tile offsets into the block tile by:

```
regTileOffset_y = threadId_y * Ry <=> ii,
regTileOffset_x = threadId_x * Rx <=> jj
```

2.2.2 Copying slices to local and private memory

The for-loop of index kk (on line 15) tiles the common dimension of the input matricies. Tiling the loops have changed the order in which the result is computed. This means that, for each iteration of this loop, 2D slices:

```
A[iii : iii+Ty*Ry, kk : kk+Tk],
B[kk : kk+Tk, jjj : jjj+Tx*Rx]
```

are used in all iterations of the inner loop of index k. These slices, which have dimensions Ty*Ry*Tk and Tx*Rx*Tk, can be collectively copied (see section 4.3) by each group from A and B in global memory to A_loc and B_loc in local memory (lines 17-18).

For each iteration of the for-loop of index `k` (line 20), 1D slices `A_loc[ii : ii+Ry, k]` and `B_loc[k, jj : jj+Rx]` are used for all iterations of the inner loops of indices `i` and `j`. These slices, of sizes `Ry` and `Rx`, can be copied from `A_loc` and `B_loc` in local memory to `A_reg` and `B_reg` in private memory (lines 23-24).

2.2.3 Scalarizing `acc`

`acc` has been array expanded to an array of size `Ty*Tx*Ry*Rx` (declared on line 4) in which each group accumulates its block tile result. The size of the two outer dimensions of `acc` matches the number of threads per group and each thread produces an `Ry*Rx` 2D register tile of this result. Since none of the elements are accessed by more than a single thread, `acc` is thread-private such that each thread has a `Ry*Rx` private array. The loops of indices `i` and `j` (lines 7-8, 26-27, 33-34) iterate this register tile. These loops can be unrolled, which enables scalarizing the two innermost dimensions of `acc` and storing them in private memory.

2.2.4 Tile sizes and hardware limitations

Certain hardware constants put natural restriction on tile sizes and also on certain *combinations* of tile sizes. We should consider these resitrictions in code generation.

This means that tiling parameters need to be tuned to the specific hardware where the program are executed.

- **Tiling restriction: group dimensions**

  In two-dimensional block/register tiling, tile parameters `Ty` and `Tx` specify the y- and x-group-dimensions.

  Recall from table 1 that the maximum number of threads in a group is 1024, meaning the multiple of dimensions in a 1, 2, or 3-dimensional group can be no greater than 1024; this would imply that `Ty` and `Tx` must always be chosen such that `Ty * Tx <= 1024`.

  For square work groups, this would mean a bound of `Ty = Tx <= 32`.

- **Tiling restriction: size of local memory**

  The size of local memory also puts a natural restriction on the combinations of tile size parameters. As stated in table 2, the bound on local memory is 48 KiB per group, but if `lmem_bound` is some arbitrary upper bound on local memory, then the set of combinations of tiling parameters are limited to those which satisfy the equation:

```
Tk * (t_size_A*Ty*Ry + t_size_B*Tx*Rx) <= lmem_bound,
```

where `t_size_A` and `t_size_B` are the sizes in bytes of the data types `A` and `B` consist of.

In lines 14-15 of listing 10, we copy two block tiles of sizes `Tx∗Rx∗Tk` and `Ty∗Ry∗Tk` from `A` and `B` in global memory to local memory; assuming 32-bit data types in `A` and `B`, this means a total of `4∗Tk ∗ (TxRx + TyRy)` bytes of local memory is necessary to hold these block tiles.

In this example, choosing `Ty = Tx = 32` and `Ry = Rx = 2` (ie. the largest group size but the smallest non-trivial register tile size) would imply `Tk <= 96`.

- **Tiling restriction: number of registers per thread**

Lastly, the number of available registers per thread sets a restriction on the register tile sizes, `Ry` and `Rx`. The block/register tiled kernel is going to copy slices of `A` and `B` into arrays in private memory of sizes `Ry` and `Rx`, in addition to maintaining a `Ry∗Rx` array of accumulated result values to be written to `C`.

This means that *at least* `Ry + Rx + Ry∗Rx` registers are needed to accomodate full scalarization of private memory arrays - not accounting for any constant number of additional registers needed for temporary values.

As stated, off-chip thread-private memory is used in case of register spilling, so we must be careful not to choose `Ry` and `Rx` too large. For example, if we assume a maximum of 64 registers per thread (as per table 2) and no additional temporary registers are needed, then `4∗4`, `4∗12`, `6∗6`, or `4∗8` register tiles are valid, while `8∗8` is not.

2.3 How tiling improves temporal locality for MM

We have now seen how the parallel loops in the tiled program can be mapped to the hardware levels and where slices of the input matrices can be copied from global to local memory and again in smaller slices from local to private memory. In this section, we will analyze how this improves the temporal locality of reference by counting the number of memory accesses to see how much reusing values reduce the number of expensive readings. We want to compare the block and register tiled code to the naive program (as in listing 1) and a version which is block tiled but *not* register tiled.

◦ **Block tiled program**

To obtain a program which is block-, but not register tiled we can follow the same transformation steps as in section 2.1 with a few modifications.

In step 1 of the transformation, we instead *singly* stripmine the parallel loops with strides `Ty` and `Tx`, respectively (equivalent to choosing register tile sizes `Ry = Rx = 1` and omitting the resulting single-iteration loops).

This makes step 5 (loop unrolling) redundant since now there is no inner sequentialism. Instead, each thread produces only a single value in the result rather than a two-dimensional tile.

When the tile sizes do not divide the input dimensions some threads will not perform meaningful work and might access memory redundantly depending on how partial tiles are handled. We will assume that the tile sizes do divide the input dimensions in this section. The following table 3 shows the total number of accesses to global and local memory in MM with and without tiling. They are explained and compared below.

Memory access	**Naive**	**Block tiled**	**Block and register tiled**
Global memory reads	$\mathtt{2MNU}$	$\frac{\mathtt{MNU}}{\mathtt{Ty}} + \frac{\mathtt{MNU}}{\mathtt{Tx}}$	$\frac{\mathtt{MNU}}{\mathtt{TyRy}} + \frac{\mathtt{MNU}}{\mathtt{TxRx}}$
Global memory writes	$\mathtt{MN}$	$\mathtt{MN}$	$\mathtt{MN}$
Local memory reads	0	$\mathtt{2MNU}$	$\frac{\mathtt{MNU}}{\mathtt{Ry}} + \frac{\mathtt{MNU}}{\mathtt{Rx}}$
Local memory writes	0	$\frac{\mathtt{MNU}}{\mathtt{Ty}} + \frac{\mathtt{MNU}}{\mathtt{Tx}}$	$\frac{\mathtt{MNU}}{\mathtt{TyRy}} + \frac{\mathtt{MNU}}{\mathtt{TxRx}}$

Table 3

- **Copying from global memory to local memory:** When copying block tiles from global to local memory $\frac{\mathrm{M}}{\mathrm{TyRy}} \cdot \frac{\mathrm{N}}{\mathrm{TxRx}}$ groups copy two tiles of sizes TyRyTk and TxRxTk for each of the $\frac{\mathrm{U}}{\mathrm{Tk}}$ iterations of the loop of index kk. This copies a total of $\frac{\mathrm{MNU}}{\mathrm{TyRy}} + \frac{\mathrm{MNU}}{\mathrm{TxRx}}$ elements read from global and written to local. Each copied element is temporarily stored in a register.

- **Copying from local memory to private memory:** When copying register tiles from local to private memory each of the $\mathrm{Ty} \cdot \mathrm{Tx}$ threads in each of the $\frac{\mathrm{M}}{\mathrm{TyRy}} \cdot \frac{\mathrm{N}}{\mathrm{TxRx}}$ groups copy slices of sizes Ry and Rx for each of the $\frac{\mathrm{U}}{\mathrm{Tk}}$ iterations of the kk-loop and Tk iterations of the nested k-loop. This copies a total of $\frac{\mathrm{MNU}}{\mathrm{Ry}} + \frac{\mathrm{MNU}}{\mathrm{Rx}}$ elements read from local memory.

- **Accumulating the dot product:** The operation computing the accumulation of the dot products is executed a total number of MNU times acc is read and written each time, and since acc is scalarized these are accesses to private memory. Two operands originating from A and B are also read once in each accumulation; with block tiling these can be read from local rather than from global memory, and with register tiling they can be read from private memory. In this case, it requires a total of 4MNU private memory accesses.

2.3.1 Block tiling vs block and register tiling

Block tiling reduces number of global reads by a factor of the block tile size. On the other hand it also introduces $(2+\mathrm{Ty}^{-1}+\mathrm{Tx}^{-1}) \cdot \mathrm{MNU}$ local accesses. This trade-off pays off since local memory accesses are much cheaper than global memory accesses.

Adding register tiling expands the block tile sizes by factors of Ry and Rx, further reducing the number of global memory reads. It also reduces the number of local memory accesses by this factor. The number of threads per group is still $\text{Tx} \cdot \text{Tx}$ and each thread now produces a tile of the result instead of just a single value, which means fewer groups are needed to compute the MM result and more of the work is sequential.

However, even if the program is not register tiled, some elements in the result might not be computed in parallel. There is a hardware limitation on the number of threads which can be physically manifested at a time. If the input is large enough that all available parallelism is exploited, some computations will need to be sequential anyway. In this case, we expect that register tiling the program will speed up these sequential computations and improve performance.

Register tiling might not be beneficial if the input is small. Spawning a thread imposes some overhead in itself, so having threads perform some sequential work can make sense, but in this case, this should be covered by the sequential computation of the dot product.

Since global accesses are reduced with a factor of the block tile sizes, we want these to be as large as possible. However, the larger the groups, the larger a number of threads potentially will perform redundant work in partial tiles when tile sizes do not divide input dimensions. For instance, in a case where input dimensions M and N are half the size of the block tiles, 75% of the threads of the single executing group would be wasted.

A drawback to both block and register tiling is that the transformations increase the size and complexity of the code. There might be a small negative impact on performance due to instruction caching but the greatest problem with this is that it worsens the readability and maintainability of the code. This is an argument for having a compiler automatically generate tiled code instead of writing it by hand.

2.4 Generalizing block and register tiling

So far we have considered tiling for ordinary MM. If we examine the transformation steps we notice they are indifferent to certain parts of the program, which means that the transformation could be applied to other similar programs as well. Ideally, when implementing the transformation into the Futhark compiler, we want the optimization to apply to as many different input Futhark programs as possible.

In this section, we search for a more general pattern of programs which could benefit from block- and register tiling.

2.4.1 Example: GEMM

Generalized matrix multiplication, or GEMM, computes the product $C = \alpha AB + \beta C'$, where C and C' both are `M * N` matrices.

```
let C = map2 (\A_row C'_row ->
             map2 (\B_col c' ->
                  let ab = map2 (*) A_row B_col
                           |> reduce (+) 0
                  in  alpha * ab + beta * c'
                  ) (transpose B) C'_row
             ) A C'
```

Ordinary MM can be seen as a specific case of GEMM with $\alpha = 1$ and $\beta = 0$, but with no `C'` as input. Like ordinary MM, the temporal locality of GEMM could be improved by block and register tiling. We want to be able to handle the additional operations before and after the **redomap** computation (ie. the load from `C` and the scaling of `ab` and `c'`) in a way that does not negatively impact performance.

2.4.2 Generalizing the redomap

In the above GEMM example, the **redomap** takes two input 1D-arrays (a row of `A` and a column of `B`), pair-wisely multiplies these and sums products, resulting in the dot product of the two vectors.

- **redomap input arrays**

In general, for 2D tiling to be beneficial, this **redomap** expression should have at least two input arrays `arr1` and `arr2`, where `arr1` is invariant to the innermost loop and `arr2` is invariant to the second innermost loop. Invariant here means that indexing of this particular array does not depend on the given loop variable, such that the same element is used for all iterations of that loop. Exactly because the same elements are used for multiple iterations of the two innermost loops, there exists an opportunity to improve temporal locality so we can expect tiling these loops to be beneficial.

If there exists a loop to which no input array is invariant, there would be no benefit to tiling this loop. Attempting to tile it would not break the program but might damage performance by for example redundant copies to local memory.

- **redomap lambda and result types**

  The program should still be tileable even if the **map** and **reduce** operators in the **redomap** are something other than `(*)` and `(+)`, respectively.

  Depending on input array dimensions and these operators, the result type of the **redomap** might not be scalar. The result of the **redomap** is accumulated in private memory, and a non-scalar **redomap** result takes up more space.

  If the `Ry*Rx` per-thread accumulator is not able to fit in register memory, then values are spilled to off-chip thread-private memory global memory instead. However, if the size of the result is non-scalar but still small, eg. a tuple of 32-bit types, then register tiling might still be beneficial, but register tiles would have to be smaller than would otherwise be possible with scalar values.

2.4.3 Generalizing the parallel loop nest

Rather than recognizing only programs with exactly two outer loops, the optimization should work with a deeper loop nest, for instance in case of batch matrix multiplication where multiple matrix multiplications are executed in a loop nest. The loops we want to tile should be the innermost loops in the nest.

2.4.4 Generalizing statements preceding and following the redomap

To generalize the pattern even further, we consider the statements preceding and following the redomap.

For ordinary MM, preceding statements consists only of loads from `A` and `B`, and no statements follow. For GEMM, additionally contains a read from `C'` preceding the redomap, while the **redomap** is followed by statements scaling its result by some value α and adding `c`.

In general we would like to tile loops where the body of the loopnest follows this pattern:

```
code1
redomap
code2
```

However, if, `code1` and `code2` can be arbitrary, it can be hard to predict how they affect the benefit of tiling transformation, so we might need some restrictions or modifications before we choose to apply the tiling transformation.

- **Rearranging statements**

  There might be some statements in `code1` which the **redomap** does not depend on. These can safely be moved to after the **redomap**. For GEMM, if we do not move the reading from `C`, we would have to keep them in registers while computing the **redomap** and they would take up space which could have been used elsewhere or cause other values in private memory to be spilled to global memory. When these statements are concatenated with `code2` they might exhibit some pattern which can be further optimized in a later pass.

- **Non-scalar `code2` result**

  A non-scalar `code2` result faces the same problems as a non-scalar **redomap** result discussed above.

  `code2` could potentially produce its non-scalar result directly in global memory rather than storing it temporarily in private memory; however, the Futhark IR dictates that a kernel produces all of its results as the same hardware level.

- **Multiple subsequent redomaps**

  When `code1` or `code2` contain additional **redomap**s, they might not benefit from the same tiling depending on whether their input arrays are invariant to the same dimensions of the map next. If these **redomap**s are independent, the loop nest could be distributed over them and the tiling applied to whichever loop nest which would benefit from it.

---

3 Inside the Futhark compiler

To aid us in implementation, we would like to create a pseudocode sketch of what the program should look like in the IR after it leaves our code transformation.

In preparations, we also need to examine the Futhark compiler as is to determine what it needs (ie. what it is missing) in order to accomodate the block/register tiling code transformation we arrived at in section 2.2.

In this section, we eventually work towards visualizing the transformed code in the IR proper, but first, we give an abstract overview of those aspects of the Futhark most important to the implementation, and present an addition to the Kernels IR in `RegTileReturns`.

3.1 A tiny bit of the Futhark Kernels IR

In this subsection, we explain those of the non-trivial features of the intermediate representation which are relevant to our implementation - ie. what is expressible at this stage and how, and what type of information is conveyed to later compiler stages. Where relevant, we will discuss what the IR is missing before we can hope to implement block and register tiling.

The entire block/register tiling transformation will be performed in the **loop tiling** pass of the **Kernels** stage of the compiler, in which loop tiling is performed - as such, this is the only optimization pass we will have to worry about, but we nevertheless with to give the reader a little context. The below diagram shows an abstract view of the compiler, with the Kernels stage focused:

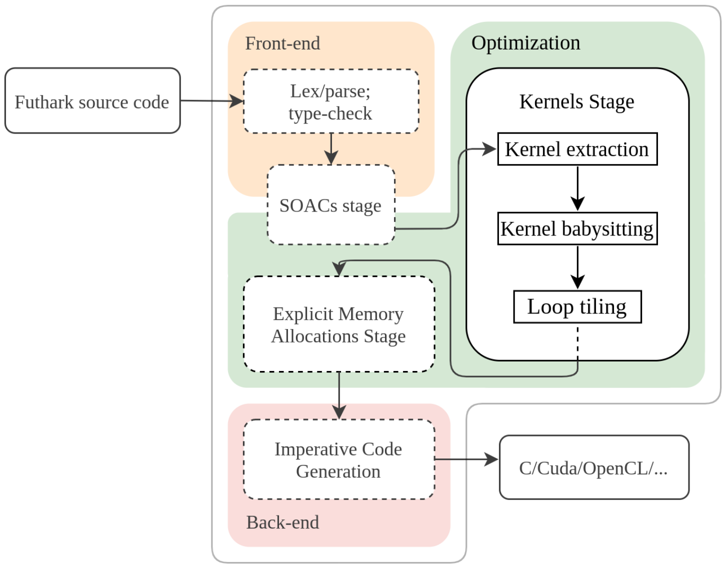


Figure 2: Abstract diagram of the Futhark compiler; Kernels stage highlighted.

Based on a more detailed diagram from [5].

Within the Kernels stage, the loop tiling pass is preceded by **kernel babysitting**, in which sequential memory access patterns inside kernels are inspected and sub-arrays are transposed if necessary to achieve coalesced memory access, in turn preceded by **kernel extraction**, in which any nested parallelism left-over from the prior stage (the SOACs stage, which we will not discuss) is flattened [6].

The Kernels IR is both typed and sized [6].

3.1.1 Kernels IR: grammar

We have chosen to devise a simplified version of the IR used in the Kernels stage of the Futhark copiler, containing only those features and constructs relevant to the produced sketch. A grammar for this version is shown in below figure 3; the grammar descibes the syntax of expressions in the IR. In contrast to the actual Kernels IR, this simplified version is not typed.

```
exp        :=  0
            |  id | id [exp]
            |  let' in exp
            |  exp binOp exp
            |  segmap segLevel context { exp }
            |  loop (id = exp) for bound do { exp }
            |  id with [id, ..., id] <- exp
            |  collectiveCopy id [exp]
            |  privateCopy id [exp]

let'       :=  let id = exp let'
            |  let id = exp

binOp      :=  + | * | /
bound      :=  id < exp
context    :=  (bound, ..., bound)
segLevel   :=  group | threadPrivate
```

Figure 3: Grammar of the simplified version of the Kernels IR which we use.

3.1.2 Kernels IR: segmap

In flattening, a perfect map nest of depth $k \geq 1$ is translated into a so-called **segmap** (segmented **map**) construct; a parallel construct operating over a k-level *mapnest context*, with each level corresponding to a dimension of the map nest [7]. Each level specifies a formal parameter to the function, aswell as the array whence it comes.

As such, the **segmap** is quite simply the flattened representation of the **map** nest. Below is an example of the translation of a source language **map** nest into an IR **segmap**:

```
map (\bs ->                 =>
  map (\b -> f b) bs        =>   segmap {bs in bss} {b in bs}
    ) bss                   =>     (f b)
```

The mapnest context `{bs in bss} {b in bs}` should be read as a nested ”foreach bs in bss; foreach b in bs”. But the above is actually an abstraction. In the IR, which at this point is sized, each level is an index variable/index space boundary pair corresponding to a dimension of the mapped array. These then explicitly index the formal parameters to the **segmap** inside the body:

```
segmap {bs in bss} {b in bs}  <=>  segmap {i < bss.dimY} {j < bss.dimX}
  (f b)                       <=>    (f bss[i, j])
```

where `bss.dimY` and `bss.dimX` are the inner and outer dimensions of `bss`. The correspondence to a nest of parallel **forall** loops holds:

```
segmap {i < bss.dimY} {j < bss.dimX}  <=>  forall (i = 0; i < bss.dimY; i++)
  (f bss[i, j])                       <=>    forall (j = 0; j < bss.dimX; j++)
                                      <=>      f(bss[i, j]);
```

In addition to this, each **segmap** is annotated with an *execution level* (*segLevel* in the grammar), indicating at what hardware-level this particular **segmap** executes; see section 1. We distinguish between execution levels `group`, `thread`, and `threadPrivate`, which correspond to inter-group, intra-group, and thread private levels of execution.

3.1.3 Kernels IR: for loops and in-place updates

In section 2.1 we learned about loop normalization. The Futhark IR, as well as the source language, supports only normalized for loops - this is no hindrance, however, as most of the loops in the transformed code are going to benefit from unrolling and scalarization, which requires normalization.

The general Futhark loop construct maintains a merge variable, which is initialized to some value before the loop and updated at the end of each iteration; think tail-recursion. The IR supports in-place updates of a merge variable (eg. if this is an array) if the merge initializor is not used again later.

The below example shows how an array `arr_init` (declared earlier) is zeroed-out in-place inside a Futhark (source language or IR) loop:

```
let arr_updated = loop (arr_merge = arr_init)
                  for i < arr_bound do
                    arr_merge with [i] <- 0
```

3.1.4 Kernels IR: `KernelResult`

The Kernels IR introduces a special construct for expressing the return value of one or more groups of threads executing a kernel: `KernelResult`. The collaborative result of a number of threads is typically an array in memory, so a `KernelResult` also conveys all information necessary for determining how to construct just that array in memory - ie. what and where each thread writes.

To accomodate block and register tiling, we need a `KernelResult` which can support the case where:

1. each thread writes multiple values (ie. a register tile)
2. write values originate from private (eg. register) memory
3. per-thread write indices are non-contiguous (because multi-dimensional tiles means strided write)

But as of the start of this project, the Kernels IR was missing a `KernelResult` suitable for this case. The closest candidates were:

- `TileReturns`: each group of threads collaborate in writing a 2D slice of an array. Each physical thread is expected to write at most one element, so this is not suitable for register tiling, but this is used in the existing block tiling.
- `ConcatReturns`: each thread produces a chunk of the result, and per-thread chunks are concatenated to form the result. Chunks do not have to be concatenated in thread-order, but individual chunks can not be split up and interleaved with others so multi-dimensional register tiles are impossible.
- `WriteReturns`: each thread is given an arbitrary number of index/value pairs and writes these values to the corresponding indices of some result array. This immediately seems promising, however `WriteReturns` requires a compile-time statically known number of writes per thread, which is incompatible with user-configurable tile sizes.

In the following section, we discuss our work on designing the new `KernelResult`.

3.1.5 Kernels IR: `RegTileReturns`

Since none of the existing `KernelResults` are suitable for the transformation, we need to design a new constructor which can convey all information necessary to produce an index function for the per-thread write indices into the result array, as well as a boundary guard surrounding the write-back as seen in the pseudocode for the code transformation in section 2.1.

We call the proposed new constructor `RegTileReturns` and look to the existing `TileReturns` for inspiration.

`TileReturns` takes two parameters: the first being a list of tuples whose first element is a dimension in the result array and whose second element is the tile size with which this dimension is tiled, and the second, the name of the tile this group writes (that is, the accumulated result over all tiles this group has processed). The latter list of tuples is alone enough to generate an index function which, at run-time, computes the individual thread's scalar write index given group and thread indices.

Let's look at an example with two-dimensional tiling: if the result has dimensions `M * N` and each group has dimensions `Ty * Tx` and is responsible for a tile of the same size (remember `TileReturns` assumes threads write at most one element), then the list `[(M, Ty), (N, Tx)]` carries sufficient information to generate the index function:

```
write_index_y = group_offset_y + thread_offset_y,
```

where `group_offset_y` and `thread_offset_y` are the y-direction offsets of this particular group within the result array and this particular thread within the `Ty * Tx` tile (and similarly for the x-direction), and where

```
 group_offset_y = groupId_y * Ty,
thread_offset_y = threadId_y,
```

and similarly for the x-direction. These indices are then flattened to:

```
write_index_flat = write_index_y * N + write_index_x
                 = (groupId_y * Ty + threadId_y) * N
                  + groupId_x * Tx + threadId_x.
```

Along with the unflattened write indices, the result dimensions are then all that is necessary to generate code for the mentioned boundary guard.

As discussed multiple times throughout sections 2, with the block/register tiling transformation, each thread is now individually reponisble for a Ry × Rx tile. The group dimensions remain Ty * Tx, and as such each group is collectively responsible for a tile of size (Ty*Ry) * (Tx*Rx). The group and thread offsets now become:

```
  group_offset_y = groupId_y * Ty * Ry,
 thread_offset_y = threadId_y * Ry,
```

and similarly for the x-direction. These should then be flattened to:

```
write_index_flat = write_index_y * N + write_index_x
                 = (groupId_y * Ty * Ry + threadId_y * Ry) * N
                  + groupId_x * Tx * Rx + threadId_x * Rx.
```

It would appear that the input dimensions, along with the group and register tiles corresponding to that dimension, is all that is needed to generate per-thread write-back index functions.

As for the boundary guard, we of course still only need to assert that each unflattened index falls within its corresponding result dimensions, which would already be included by the above.

We then design RegTileReturns to take two parameters: a list of triples whose three elements are a dimension in the result array, the group tile, and the register tile size corresponding to that dimension, and the name of the per-group result tile.

This design should generalize to n-dimensional tiling, but using again the two-dimensional tiling example from earlier, then here the list [(M, Ty, Ry), (N, Tx, Rx)] would generate the desired index functions.

3.2 Sketching block and register tiled matrix multiplication in the IR

We are ready to present our IR sketch of the transformed code as it should look after leaving the loop tiling pass.

However, before we dive into it, we make a couple of assumptions to ease understanding and to focus attention on the less complex implementation details for the time being, and thus the sketch presented here is **simplified**.

We shall, of course, later remove these assumptions when we discuss implementation in more detail, but for now refer to appendix A for the full, unsimplified sketch.

3.2.1 IR sketch: added abstractions

First of all, we add two abstractions to the IR: `collectiveCopy`, which can be used by threads in a group to collectively copy a slice from global memory to local memory in coalesced fashion, and `privateCopy`, which can be used to let individual threads simultaneously copy a slice from local to private memory. Implementations of the former will be discussed in section 4.3.

Second, we remove ourselves from the notion of residual input (from stripmining, as discussed in 2.1) and assume that input sizes are divided evenly by tile sizes (`Ty*Ry` divides `M`; `Tx*Rx` divides `N`; `Tk` divides `U`), such that we do not have to worry about boundary checks yet in the sketch. Boundary checks will become an important part of implementation once we move away from this assumption, as we will discuss in section 4.2.

- **On `collectiveCopy` in the IR**

  In ordinary matrix multiplication, we know that the second operand matrix `B` is semantically transposed before kernel execution, and as such the index function for coalesced access is easily given. However, in general - that is, when the program is something different from matrix multiplication - we cannot know anything about the second operand *array* or its layout in memory (at least not at this stage in the compiler) and can only make qualified guesses which result in coalesced access in the most cases.

  Ideally, the IR would feature a built-in abstraction for specifying a collective copy from/to global memory, which could then be resolved in a later compiler stage. However, at this point, the IR *does not* feature such an abstraction, so for the sake of this abstract presentation, we add our own in `collectiveCopy`. In section 4.3, we show how we actually generate code similar to that which would ideally be auto-generated, but which only guarantees coalesced access in the case where `B` is transposed, and thus definitively leave `collectiveCopy` for future work.

3.2.2 IR sketch: walkthrough

In this subsubsection, we walk through our IR sketch (seen below for reference), highlighting non-trivial parts afterwards.

```
-- host code goes here. assume B is transposed at
-- this point, such that A :: [M][U], B :: [N][U].
let res = segmap group (gid_y < gridDim_y, gid_x < gridDim_x) {
  -- each group computes block tile offset in both dimensions
  let iii = gid_y * Ty * Ry
  let jjj = gid_x * Tx * Rx

  -- each thread produces an Ry*Rx private memory array,
  -- initialized with neutral elements to the reduction.
  let grp_acc_init =
    segmap threadPrivate (ltid_y < Ty, ltid_x < Tx) {
      let thd_acc_init = scratch([Ry][Rx], t)
      in loop (thd_acc_merge = thd_acc_init) for i < Ry do {
        loop (thd_acc_merge' = thd_acc_merge) for j < Rx do {
          thd_acc_merge' with [i, j] <- 0
        }
      }
    }


  let group_res =
  loop (grp_acc_merge = grp_acc_init) for kk0 < U/Tk do {
    let kk = kk0 * Tk

    -- threads in a group collectively copy 2D slices A[iii : iii + Ty*Ry, kk : kk+Tk]
    -- and B[jjj : jjj + Tx*Rx, kk : kk+Tk] from global to local memory.
    let A_loc = collectiveCopy A[iii : iii + Ty*Ry, kk : kk + Tk]
    let B_loc = collectiveCopy B[jjj : jjj + Tx*Rx, kk : kk + Tk]

    let thread_res = loop (grp_acc_merge' = grp_acc_merge) for k < Tk do {

      -- each thread copies 1D slices A_loc[ltid_y*Ry : ltid_y*Ry + Ry, k]
      -- and B_loc[ltid_x*Rx : ltid_x*Rx + Rx, k] from local to private mem.
      let asss = privateCopy A_loc[ltid_y*Ry : ltid_y*Ry + Ry, k]
      let bsss = privateCopy B_loc[ltid_x*Rx : ltid_x*Rx + Rx, k]

      -- each thread computes redomap over its register tile.
      in segmap threadPrivate (ltid_y < Ty, ltid_x < Tx) {
        let as = asss[ltid_y, ltid_x]
        let bs = bsss[ltid_y, ltid_x]
        let thd_acc_init = grp_acc_merge'[ltid_y, ltid_x]

        in loop (thd_acc_merge = thd_acc_init) for i < Ry do {
          loop (thd_acc_merge' = thd_acc_merge) for j < Rx do {
            let map_res = as[i] * bs[j]
            let red_res = thd_acc_merge'[i, j] + map_res
            in thd_acc_merge' with [i, j] <- red_res
          }
        }
      }
    }
    in thread_res
  }
  in RegTileReturns [(M, Ty, Ry), (N, Tx, Rx)] group_res
}
```

- **Lines 1-2:** The code transformation should first produce some host code, fetching tile sizes and computing kernel invariants such as grid dimensions, local memory size, and such - this is omitted here but indicated by comments. Also, note that with ordinary matrix multiplication, `B` will be transposed at this point.

- **Line 3:** The actual kernel comprises a group-level **segmap** with context corresponding to the grid dimensions (as derived in section 2.2).

- **Lines 11-15:** Each thread should initialize its register tile of private memory with the neutral element to the reduction operator of the later **redomap**. Since this is per-thread and pertains to private memory only, it should be done in a thread-private **segmap** over work-group dimensions `Ty*Tx`, with a sequential loop nest inside iterating the 2D register tile.

- **Lines 22-53:** The main sequential loop of index `kk`, which iterates the block tiles and accumulates the result for each block in `grp_acc_merge`.

- **Line 23:** Here is an example of de-normalization of a loop; the loop of index `kk0` is normalized, whereas the original loop of index `kk` has a stride `Tk`, so we multiply `kk0` with `Tk`.

- **Lines 27-28:** Here, we insert the abstract `collectiveCopy` to perform a group-wide, coalesced copy of one entire block tile from global to local memory. As stated, more on this later.

- **Lines 30-51:** The inner sequential loop of index `k`, which iterates the recently copied block tile, accumulating the result each time threads have finished processing their register tiles.

- **Lines 34-35:** Threads individually copy slices of local memory into private memory using `privateCopy` - again, more on this later.

- **Line 38:** Now that each thread has copied over their private slices of A and B from local memory, they are ready to compute the **redomap**. Since this is per-thread and all computations are made using values stored in memory, this is performed using a thread-private **segmap** over the group dimensions.

- **Lines 43-49:** Each threads computes its own redomap in a sequential loop nest of dimensions `Ry*Rx`, since this is exactly the size of the register tile. Here, the loop body is the **redomap** of an ordinary matrix multiplication, but recall from section 2.4 that this can take many forms.

Lines 54: Here, we insert an instance of `RegTileReturns` exactly as presented and discussed in the previous section. Later compiler stages will handle code generation from this point on.

4 Implementation

This section documents solutions to select non-trivial problems encountered during implementation of the code transformation into the Futhark compiler.

- **Source and installation**

  We build upon our own branch of the Futhark repository, which can be cloned from:
  `https://github.com/diku-dk/futhark/tree/aemilie_anders-block_reg_tile.git`

  To install our branch of the compiler, simply follow the steps given in section 1.2 of the official installation guide found at `https://futhark.readthedocs.io/en/latest/installation.html`, substituting our git link for the one listed in the guide. Installation requires an existing installation of `stack`.

  We also include the full Haskell source code of our implementation in appendix D.

4.1 Implementation: starting point

In this first implementation subsection, we describe features related to our implementation, but which should not be accredited to us; rather, our supervisors. These are parts of the implementation which we are not responsible for but build on top of, and so we will not dwell on the details of these, but simply give credit where it is due.

4.1.1 Code pattern matching

As explained in section 2.4, the code transformation applies to a wide variety of programs. The very first step in the program should be to actually recognize these patterns.

At the start of the project, Cosmin, our main supervisor, provided us with a couple hundred lines of code in the module `Futhark.Optimise.BlkRegTiling` to use as a starting point. The module consisted of a number of helper functions and a single exported function `mm_BlkRegTiling`. This function took (and takes) as input an IR statement containing a let binding of an IR `segmap`. The function would then pattern match the body of the `segmap` to determine whether it applies for the tiling transformation. The body of this function, which would be executed on a successful pattern match of the `segmap`, was of course left empty for us to finish.

This code, as well as our additions, can be found in appendix D.

4.1.2 Back-end support for `RegTileReturns`

In section 3.1.5 we discussed the design of `RegTileReturns`, a new `KernelResult` constructor which could carry all information required by later compiler stages in order to generate index functions for the final write to a result array.

Support for the new `RegTileReturns` would have to be implemented in the imperative code generation back-end of the Futhark compiler [14], rendering the task both out of scope of the project - as we have only worked in the `Kernels` stage of the compiler (see figure 2) - not to mention out of range of our abilities.

When finished designing the new constructor, we conveyed the design to our secondary advisor Troels, who made most of the necessary changes to the back-end; as of the writing of this report, code generation for the *boundary check* on the final write to the result array is still missing. More on this in section 4.5.

4.1.3 Register tile sizes in the compiler

Later, in looking over generated code with our supervisors, we noticed how in the generated code, arrays which were supposed to reside in private memory did not always; rather, these were almost always being placed in local memory by the **Explicit Memory Allocations** compiler stage. The problem here was that the Futhark compiler had not been told to *treat* register tile sizes as being compile-time constant despite them being unknown at this time [15]. This, too, has been implemented by our secondary advisor Troels.

---

[14] Specifically the `Futhark.CodeGen.ImpGen.Kernels.Base` module.

[15] Actually, since Futhark programs support run-time configurable tuning parameters, register tile sizes are not known until the user gives them as command-line input, after which a GPU kernel is compiled.

4.2 Implementation: handling residual input

In performing the stripmining transformation in section 2.1, we inserted boundary checks to handle the cases where a tile size does not evenly divide the iteration space it tiles. Later, in presenting our IR sketch in section 3.2, we brushed over the notion of *residual input*: When computing the product of two matrices `A` and `B` of sizes `M*U` and `U*N`, respectively, using tiling, tile sizes do not necessarily divide input dimensions. When they do not, we have *residual input* at the boundaries of the matrices; we call these tiles *partial tiles*.

In this subsection, we first discuss each of the types of residual input and how we handle them. Then, we present an optimization to the number of boundary checks involved in this which we have implemented.

We might sometimes implicitly refer to certain loops or constructs of code; when we do, we refer to the final version of the code transformation as it appeared in listing 10. We suggest the reader to reacquaint themselves with the tiling parameters presented in section 2.1, as we shall also make reference to these throughout the this section.

4.2.1 Identifying and handling partial tiles

Regardless of the particular type of residual input we are dealing with, we have two general concerns to address.

First of all, we must assert not to over-step boundaries of arrays in global memory during reads and writes, and so some boundary checks are necessary. These boundary checks can be *expensive*, so we want to make sure that we perform as little redundant checks as possible.

Second, since we can only allocate a static amount of local memory for all groups in the grid, so those groups which are responsible for processing partial tiles will naturally not always fill up local memory with meaningful values (in fact, some groups in some cases will never). Since Futhark is a functional language, we always need to write *something* to local memory, so here, padding values are inserted.

We then identify three types of residual input which we need to handle:

- #0 residual input in the common dimension `U`, which occurs when the **redomap** tile `Tk` does not divide `U`.
- #1 residual input in the `M`-dimension, which occurs when the outer block tile `Ty*Ry` does not divide `M`.
- #2 residual input in the `N`-dimension, which occurs when the inner block tile `Tx*Rx` does not divide `N`.

- **#0: Residuals in the common dimension `U`**

  In iterating the common dimension `U`, the outer loop variable `kk` specifies the offset in the `U` dimension of a particular tile, while the inner loop of index `k` iterates that tile. If `Tk` does not divide `U`, then the inner `k`-loop might overstep `U`. We need to assert this does not happen, which amounts to an assertion of `(kk + k) < U` guarding computations inside the **redomap**.

  Because the innermost sequential `i`- and `j`-loops are invariant to both `kk` and `k`, this guard can be placed before these loops to avoid writing padding values from local to register memory, as well as to avoid accumulating garbage results to `acc` (as we shall see later, the latter is not a problem in ordinary matrix multiplication, whereas in infinitely many other cases it is fatal).

Since the common dimension is fully iterated by all groups, we know that each thread will experience the same amount of residual input, and that each group processes the same number of tiles in the `U` dimension; precisely ⌈`U/Tk`⌉.

- **#1: Residuals in the `M` dimension**

  This type of residual not only needs to be considered inside the `redomap`, but also in the read of `A` from global to local memory, as well as in the final write to `C`.

  However, whereas type #0 residuals need to be handled by each group inside the **redomap**, this type of residual is only experienced by groups residing on the edge of the grid in the `y` direction - ie. groups for which `group_index_y == grid_dim_y - 1`.

- **#2: Residuals in the `N` dimension**

  This type is symmetric to type #1 residuals and is thus handled similarly.

4.2.2 Optimization: handling `U` dimension residuals in an epilogue

Each type of residual input imposes a lot of boundary checks which can potentially be a negative for the performance of the program, even if memory is always going to be the dominating bottle-neck.

As mentioned earlier, since each group iterates the common dimension `U` fully, each group processes exactly ⌈`U/Tk`⌉ tiles in the `U` dimension, `U/Tk` of which are full tiles; if there is residual input, then each group will finish iteration in a partial tile.

Threads are, of course, guaranteed to stay within bounds when iterating full tiles, so here, boundary checks related to the `U` dimension can safely be omitted. The idea is then to unroll the final iteration of the outer sequential `kk`-loop and to remove boundary checks from all but this unrolled iteration - we call this the *epilogue* [16]. Below is an example - notice how inside the epilogue, on line 5, `kk` is set to the largest multiple of `Tk` less than or equal to `U`:

[16] *Because it picks up where the main storyline left off and is sometimes redundant!*

```
1  for (kk = 0; kk < ceil(U/Tk); kk += Tk)  =>  for (kk = 0; kk < U/Tk; kk += Tk)
2    /* kk loop body */                      =>    // kk loop body, with U
3                                            =>    // boundary checks omitted.
4                                            =>  if (U % Tk)
5                                            =>    // start of epilogue
6                                            =>    kk = (U/Tk) * Tk;
7                                            =>    // kk loop body
```

[17]

This optimization comes at a payoff in code size: the kk loop body constitutes the majority of the program as is, so unrolling the last iteration in an epilogue means nearly doubling the code, however, we choose to implement this optimization in our code generation.

[17] The branch with condition `U % Tk` on line 4 of the snippet is actually redundant, as `U` boundary checks inside the epilogue provide this guard when there is no residual. We omit it in code generation since the presence of residual input much more often than not is the case.

4.3 Implementation: collective copy

In section 3.2, we presented our simplified sketch of what the transformed program should look like in the IR as it leaves Kernels stage. In the sketch, we omitted details on the implementation of the `collectiveCopy` construction, which was simply an added abstraction to ease understanding and not an actual feature of the Kernels IR.

In this section, we discuss implementation of this collective copy to local memory, which guarantees coalesced access when input matrix `B` is transposed.

4.3.1 Achieving coalesced access to global memory

We learned in section 1.3 how to ensure coalesced access to global memory when multiple threads access at the same time.

For the purposes of this section, we return our attention to the transformed MM program as it appeared in listing 10; the final step in which we first inserted abstract copies into the C-like pseudocode of the block/register tiled program. We shall use this as the setting for demonstrating how to achieve coalesced access, focusing on the copy of `A` from global to local memory (the copy from `B` is symmetric).

Recall from section 2.2 that `A_loc` has dimensions `Ty*Ry*Tk`. Now, we want to replace the abstract slice copy of `A` to `A_loc` in line 14 of listing 10 with pseudocode representing an actual implementation. Below snippet shows that pseudocode:

```
1   A_loc[Ty*Ry, Tk];
2   for (i0 = 0; i0 < Ry; i0++)
3     for (k0 = 0; k0 < ceil(Tk/Tx); k0++)
4       i = i0 * Ty + threadId_y;  // threads go in stride ty * tx,
5       k = k0 * Tx + threadId_x;  // offset by their indices.
6
7       A_row_idx = iii + i;       // iii and kk denote block
8       A_col_idx = kk  + k;       // tile offsets into A.
9
10      if (A_row_idx < M && A_col_idx < U)       // global memory boundary guard.
11          A_loc[i, k] = A[A_row_idx, A_col_idx];
12      else
13          A_loc[i, k] = 0;
```

The copy is performed in two nested and normalized for loops of dimensions `Ry*ceil(Tk/Tx)` (lines 2-3); the loops have these dimensions such that when they are de-normalized in lines 4-5, they have dimensions `Ty*Ry*Tk` (assuming `Tx` divides `Tk`), which is precisely the size of `A_loc`.

To achieve coalesced access to `A`, we want each thread to go in a group-dimensional stride across loop iterations; this is achieved in lines 4-5, where thread indices are added to count as local thread offset (recall section 1.3).

4.3.2 Implementing collective copy in the IR

Recall also from section 3 that at the Kernels stage of the compiler, we do not have knowledge of the manifestation in memory of certain arrays. Here, we also dicussed the possibility of an abstract `collectiveCopy` construction; we leave this for future work.

In implementing the collective copy in the compiler, we first notice a problem imposed by the requirement of configurable tile sizes:

- **Handling out of bounds local memory writes**

In explaining the pseudocode for the collective copy arrived at above, we mentioned that the nested for loops of dimensions `Ry*ceil(Tk/Tx)` share common dimensions with `A_loc`. However, this is only the case when `Tx` divides `Tk`.

When `Tx` does not divide `Tk`, we run the risk of overstepping local memory boundaries during writes. An easy fix is to enforce `Tk` to be a multiple of `Tx`, but this would be at the sacrifice of full flexibility in tiling parameter tuning [18].

As such, we must guard local memory in a bounds check:

```
A_loc[Ty*Ry, Tk];
for (i0 = 0; i0 < Ry; i0++)
    for (k0 = 0; k0 < ceil(Tk/Tx); k0++)
        i = i0 * Ty + threadId_y;  // threads go in stride ty * tx,
        k = k0 * Tx + threadId_x;  // offset by their indices.

        A_row_idx = iii + i;
        A_col_idx = kk  + k;

        if (k < Tk) // local memory boundary check.
            if (A_row_idx < M && A_col_idx < U) // global memory boundary check.
                A_loc[i, k] = A[A_row_idx, A_col_idx];
            else
                A_loc[i, k] = 0;
        else
            // out of bounds on local mem? don't write! do nothing.
```

As seen in the snippet, when we are out of bounds of local memory, we simply do nothing. But here we run into another problem:

- **Handling non-writes in a functional IR** The IR is still functional at the Kernels stage. This implies that we can not have if-statements with empty branches as in the above snippet. The solution then to this is to use a `KernelResult` construct of the IR called `WriteReturns` (which we briefly described in section `section:ir-theory` in explaining why it is unsuitable [19] for the cases now supported by `RegTileReturns`).

[18] Modulo hardware restrictions; see section 2.2.

[19] `WriteReturns` required a compile-time statically known number of writes per thread, unsuitable with run-time configurable tile sizes.

With `WriteReturns`, each thread specifies a number of index/value pairs and write these values to those indices of a result array; Based on some per-thread computed condition, individual threads obtain either non-negative indices or `-1`; the latter corresponding to a non-write of the associated value [20].

In this case, there is one index/value pair per thread, and the result array is `A_loc`. The condition is chosen to be whether the computed index is within bounds, which is redundant seeing as `WriteReturns` already inserts a bounds guard. Nevertheless, this proved to be the way to go as long as we did not have access to an abstract `collectiveCopy` or something similar.

Using `WriteReturns` also has the added benefit of greatly easing index computation into local memory arrays, especially when `Tx` does not divide `Tk`, since the constructor takes only flat indexed arrays and produces only flat write indices [21].

4.4 Implementation summary

Upon finishing implementation, we began inspecting the generated OpenCL code. After the changes mentioned in 4.1.3, the generated code seemed satisfying: Arrays correctly manifested where we expected them to be; un-flattened group and thread indices computed only once at the beginning of the kernel; no redundant index computations made; etc.

Aside from the extra bounds checks imposed by `WriteReturns`, we were satisfied to start validation and benchmark testing.

4.5 Implementation limitations

In this section, we discuss those of the proposed implementation goals which we did not manage to satisfy in the alloted time.

Disclaimer: *we shall make no claim that the list that follows is exhaustive.* Rather, it should be considered an inspiration for future work.

4.5.1 Lack of generality

In section 2.4 we discussed how tiling applies in a more general case. While the goal of the project from the start was to implement code generation for block and register tiled matrix multiplication, we hoped to build a more generic implementation which could handle for example GEMM and batch matrix multiplication.

At the time of hand-in and writing this report, our implementation *does not* handle non-trivial `code1`'s (ie. when `code1` contains more than input two array load statements), nor does

[20]The avid reader shall recognize this as the parallel **scatter** operator.

[21]At later compiler stages, all indices are flattened, but not necessarily at this stage.

it handle a non-empty `code2` following the **redomap**. This prevents us from correctly tiling programs such as GEMM or batch MM.

Our implementation also does not correctly handle cases in which the **segmap** to be tiled is itself nested inside additional, outer **map** dimensions. This also prevents us from correctly tiling programs such as batch MM.

However, we have added pattern match guards to `mm_BlkRegTiling` which should detect these cases and let the compiler fall-back to the existing block tiling pass.

4.5.2 Missing bounds check in `RegTileReturns`

As mentioned in section 4.1.2, the code generation back-end support for `RegTileReturns` is still missing a boundary check on the final write to a result array.

In the following sections, in validating and benchmark our implementation, we are thus forced to artificially insert this bounds check into the generated OpenCL kernel code.

We argue that this is still valid in terms of producing credible validation and benchmark results for the reason that the boundary checks inserted are identical to what will eventually be auto-generated in the back-end, and also that it is permissible with regards to the project goals since the scope of this project was never to acquaint ourselves or to make changes to the compiler outside the Kernels stage.

---

5 Validation testing

Having reviewed the limitations of our implementation, we now want to validate those parts of our program which we do expect to be functional; in particular, programs with the pattern of ordinary matrix multiplication, but also programs with slight variations from this pattern.

In this section, we discuss our validation strategy and test plan, and report and evaluate the results of validation testing. For a guide on reproduction of tests, see appendix B.

5.1 Validation: strategy

In order to pass validation, our implementation needs to handle different types and sizes of input; different tile size combinations; and a small number of different programs. As such, there are a lot of factors which must come into play in an exhaustive validation.

In this subsection, we present the devised test plan and discuss to what degree it is satisfying.

5.1.1 Validation strategy: different types of input and tiling parameters

- **Tiling parametrs and residual input**

We want to test our implementation with different types and sizes of input, as well as with different combinations of tiling parameters. Our implementation should correctly handle cases both with and without partial tiles in the input.

Input sizes and tiling parameters are interdependent, since a certain combination of tiling parameters may result in the presence of partial tiles for some inputs while it may not for others, and vice versa. With respect to a ”nice” set of tiling parameters in which `Ty` and `Tx` both divide `Tk`, we devise our validation input set such that we *separately* test inputs covering all possible combinations of the three types of residuals; namely `M`, `U`, and `N` dimension residuals. However, all input sizes are run with every set of tiling parameters.

- **Tiling parameters and local memory boundaries**

Another important assertion to make pertains to local memory boundaries: as discussed in section 4.3, a consequence of configurable tile sizes happens when either `Ty` or `Tx` (or both) does not divide `Tk` - in these cases, there is a risk of out of bounds writes to local memory when reading from global memory. For each test we run across all input sizes, we test the aforementioned ”nice” case, which is the case in which both `Ty` and `Tx` divide `Tk`; cases where one of them does, but the other does not; and cases where neither divide `Tk`.

In summary, below is a table of each test input set along with a brief description:

`(M,     U,     N)`	Assuming `(Ty, Tx, Tk, Ry, Rx) == (16, 16, 32, 8, 4)`
`(2,      3,     4)`	A single, partial register tile.
`(15,    29,    27)`	A single, partial block tile.
`(128,   32,    64)`	One group, one block tile per group, no residual.
`(128, 103,    64)`	One group, multiple block tiles/group, no residual.
`(512,   32, 1024)`	Multiple groups, one block tile per group, no residual.
`(512, 128, 1024)`	Multiple groups, multiple block tiles pergroup, no residual.
`(513, 128, 1024)`	Multiple groups, residual in M dim.
`(512, 129, 1024)`	Multiple groups, residual in U dim.
`(512, 128, 1025)`	Multiple groups, residual in N dim.
`(513, 129, 1024)`	Multiple groups, residual in M and U dims.
`(513, 128, 1025)`	Multiple groups, residual in M and N dims.
`(512, 129, 1025)`	Multiple groups, residual in N and U dims.
`(513, 129, 1025)`	Multiple groups, residual in all dims.

Table 4: Validation test input set

and similarly, for the tiling parameters we use:

`(Ty, Tx, Tk, Ry, Ry)`	Description
`(16, 16, 32,  8,  4)`	”Nice” tiling parameters.
`(13, 16, 16,  8,  4)`	`Ty` does not divide `Tk`.
`(16, 13, 16,  8,  4)`	`Tx` does not divide `Tk`.
`(13, 16, 16,  8,  4)`	Neither `Ty` nor `Tx` divides `Tk`.
`(19, 16, 16,  8,  4)`	`Ty` greater than `Tk`.
`(19, 16, 16,  8,  4)`	`Tx` greater than `Tk`.
`(19, 19, 16,  8,  4)`	Both `Ty` and `Tx` greater than `Tk`.

Table 5: Validation test tiling parameter combinations

We argue that if our program correctly handles one case of each combination of input size and tiling parameters, then it handles all such cases.

5.1.2 Validation strategy: different programs

We want to test our implementation with a couple of different programs:

- **Ordinary MM**

  We first and foremost want to validate an ordinary MM program, as this is the program we will be benchmarking in the next section. Here, we simply assert that our implementation computes the correct matrix product.

- **MM-like program with integer division**

  We wish to assert that we never make computations on inserted padding values in local memory due to missing boundary guards. To do this, we run a test of an MM-like program, but with integer division substituted for the **map** lambda.

  This program is tested with input matrices of only non-zero elements, so if a zero division exception is thrown, we can be certain that it is due to our implementation not correctly handling boundaries.

- **MM-like program with mixed types**

  To assert that we handle inputs other than 32-bit floats, we test a program whose input matrices have element types `i16` and `f64`, and whose return array has element type `bool`.

  In addition, we let the reduction operator be logical AND to test whether we correctly initialize `acc` in register memory when this should not simply be zero-initialized.

5.1.3 Validation strategy: execution

We run our test suite using `futhark-bench`, Futhark's benchmark utility, which: auto-generates random input arrays in given dimensions; compiles a sequential version of the given test program, which is used computed expected results over each input set; runs a benchmark of the program; validates the result produced by the tested program against the generated, expected result.

Each test in our validation suite is run once.

5.2 Validation: results

All validation tests pass successfully.

---

6 Benchmark testing

An important aspect of our project - the sole motivation, in fact - is improving the performance of MM (and similarly structured) programs generated by the Futhark compiler.

Even though our implementation is lacking in numerous areas (as discussed in section 4.5), we wish to benchmark those features which do work; as seen in the previous section, our implementation *does* validate for ordinary MM (and other, simple programs), so this is what we benchmark.

In this section, we discuss our benchmark strategy and test plan, and report and evaluate the results of benchmarking. For a guide to reproduction of our benchmark results, we once again refer to B (our test repository mentioned there also contains all programs mentioned in this section).

6.1 Benchmarking: strategy

A lot of factors do and should come into play in benchmarking our implementation. First and foremost, we wish to benchmark using different sizes of input, but that is not all - this section details our benchmarking strategy.

6.1.1 Strategy: comparing different implementations

We are going to compare our generated block/register tiled kernel with the following programs:

- **Naive parallel MM kernel**

  We wish to benchmark a naive parallel MM kernel to examine the effects of block and register tiling compared to a kernel with no memory optimizations. The naive MM kernel is a one-to-one correspondence between the source language map nest with a **redomap** inside, in that it spawns a grid of `M * N` threads, each of which sequentially computes a `U`-element dot product in a single for loop. This kernel has extremely bad memory access patterns and should perform very slow compared to both the block tiled and block/register tiled kernels.

- **Existing block tiled kernel**

  We benchmark the block tiled kernel currently produced by the Futhark compiler. Ideally, we should see our block/register tiled kernel outperform it, because if it does not, then the implementatoin does not particularly justify itself.

  However, we *do* expect the block tiled kernel to be faster for small inputs, due to the lower overhead in the block tiled kernel - in comparing our implementation to the block tiled kernel, we will then look for a *sweet-spot* where our implementation starts outperforming.

- **Handwritten kernel with MM-specific optimizations**

  Lastly, we wish to benchmark a handwritten and human-optimized matrix multiplication kernel to see. Futhark is designed to easily generate efficient GPU code, and so to answer the question of whether the ease of use of Futhark outweighs the payoff in performance, we test against an optimal kernel written by a human programmer.

  Like our implementation, the handwritten kernel employs an epilogue for the handling of residuals, but differs greatly from our auto-generated kernel in a number of other ways. Most notably, the handwritten, human-optimized kernel:

  - omits a number of auxiliary private memory arrays and redundant copying from/to these, which are otherwise required by the Futhark compiler to generate since Futhark is a functional language.
  - omits a number of redundant thread-synchronization barriers on local memory (such as a barrier between loads of `A` and `B` from global to local memory) which the Futhark compiler currently does not attempt to identify and remove.
  - assumes that `Tx` and `Ty` both divide `Tk`, such that local memory boundaries can safely be ignored when writing to local memory.
  - utilizes minor, non-functional control flow, in that a break-statement exits the `k` loop of the epilogue once all subsequent iterations can be (human-)inferred to be redundant.

  In addition, the handwritten kernel utilizes a number of optimizations which are only valid in the case of *ordinary matrix multiplication* and an infinite number of similar programs, but which are unsafe in infinitely many other cases. We shall go in detail with these optimizations in the benchmarking discussion, section 7.3.

6.1.2 Strategy: different types of input

Our implementation should be fast for all types of input - while we expect it to be for some, there are still optimizations to be implemented for certain types of input (especially for the handling of inputs with very large inner dimensions; see section 7.3).

However, due to limited time and scope, we see ourselves forced to benchmark only a single type of input: near-square matrices of dimensions evenly distributed in the range `(M, N, U) ∈ (214, 272, 263)...(4294, 4220, 4229)`. Input dimensions are randomly generated to ensure (with great probability) that we always have partial tiles in all dimensions.

In section 6.3 we discuss this limitation.

6.1.3 Strategy: using optimal tiling parameters

We ideally want to benchmark our implementation using optimal tiling parameters. Since our program features five different tiling parameters (`Ty, Tx, Tk, Ry, Rx`), determining the optimal combination can be tedious, considering that each parameter can be set independently and that each type of input demands a different type of tiling parameters.

While the size of local memory (as well as other hardware constants) does technically set natural restrictions on the number of possible combinations [22], we choose to limit the search space significantly and search only those tile size combinations where:

$$\text{Ty, Tx, Tk} \in \{12,\ 16,\ 24,\ 32\}$$

$$\text{Ry, Rx} \in \{4,\ 6,\ 8,\ 12\}$$

for a total of 1024 combinations.

We write an automated script to search for optimal tile combinations. To ease the search and subsequent benchmarking, we want to find a *single set of optimal parameters* for each GPU we benchmark on, and in searching for optimal parameters, we test using only the same input set of near-square matrices as discussed above - again, this is far from ideal, but a consequence of the scope of the project.

The block tiled kernel has a single tiling parameter, ”`TILE`” - we also search for the optimal value of this, but do so separately.

[22] Ignoring all but local memory restraints, the number of combinations is given by the number of solutions to the equation `t_size * Tk * (Ty*Ry + Tx*Rx) <= lmem_bound`, where `t_size` is the size of the data type used and `lmem_bound` is the bound on local memory size.

For each combination of implementation and the two GPUs we are going to benchmark, we find the optimal tiling parameters to be:

- Existing block tiled kernel, both GPUs: TILE = 32
- Our block/register tiled kernel
  - 2080ti: (Ty, Tx, Tk, Ry, Rx) = (16, 16, 16, 8, 4)
  - 780ti: (Ty, Tx, Tk, Ry, Rx) = (16, 16, 16, 2, 2)
- Handwritten kernel, both GPUs: (Ty, Tx, Tk, Ry, Rx) = (16, 32, 32, 8, 4)

6.1.4 Benchmarking strategy: different GPUs

We have been granted access to two types of GPUs on a GPU cluster at DIKU: a GeForce GTX 780ti and a GeForce RTX 2080ti. We shall not go into detail with technical specifications, but rather simply emphasize that the 780ti is already a 7 year old device, and that the 2080ti is considerably faster with for example more sophisticated latency hiding [23].

We wish to run all of our benchmarks on both machines to examine whether we get to examine the effects of different hardware on speed- ups/downs.

---

[23]We refer to technical specs in [3]; the 780ti and 2080ti have compute capabilities 3.5 and 7.5, respectively.

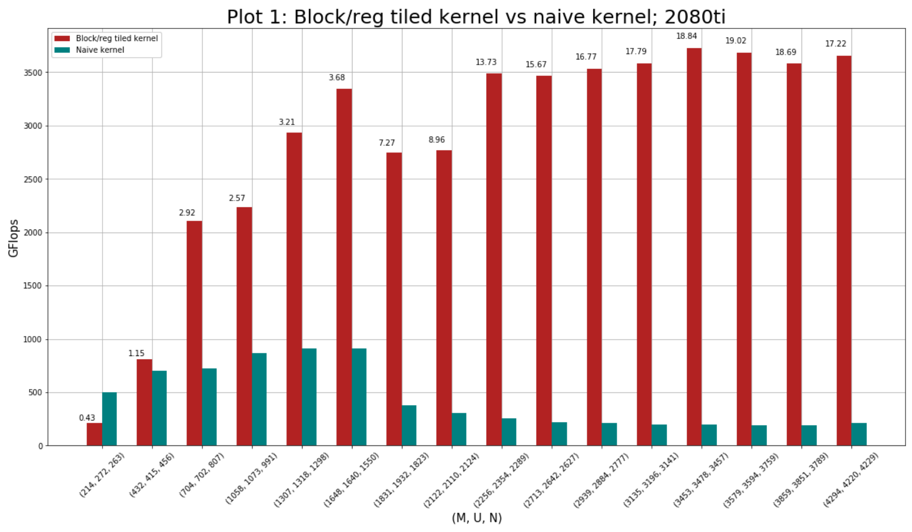
6.2 Benchmarking: results

In this section, we report the results of our benchmark tests in pretty graphs. For hard numbers, including deviations in measurement, see appendix C.

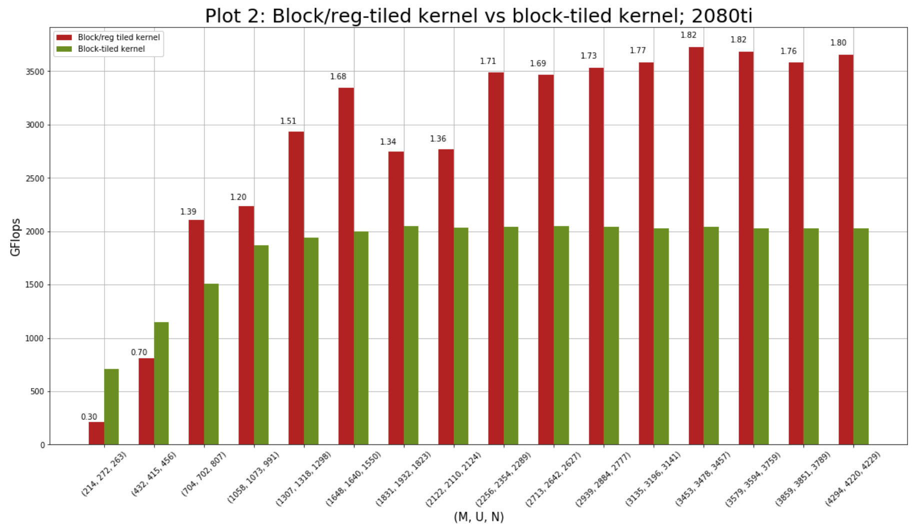
For each GPU we run our benchmarks on, we plot our block/register tiled kernel against the three programs discussed in the previous section. In addition, we compare the block-tiled kernel with the hand-written block/register tiled kernel to get *a sense of* the upper bound on the benefits in block/register tiling.

Above each set of bars in the plots we indicate the factor of speed-up of our implementation versus whichever kernel it is being compared to; below each plot, we give a short discussion of the measured results.

6.2.1 Benchmark plots: 2080ti results

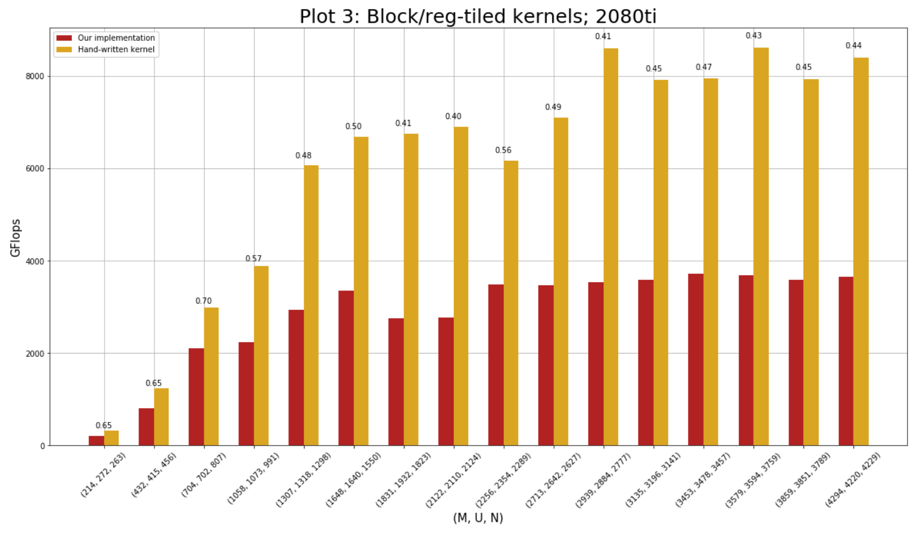


- **Plot 1:** As expected, our kernel *greatly* outperforms the naive, parallel kernel over all inputs - save for the very smallest input, where the naive kernel is over twice as fast. This is explained by the very low overhead of the naive kernel as compared to that of our implementation.



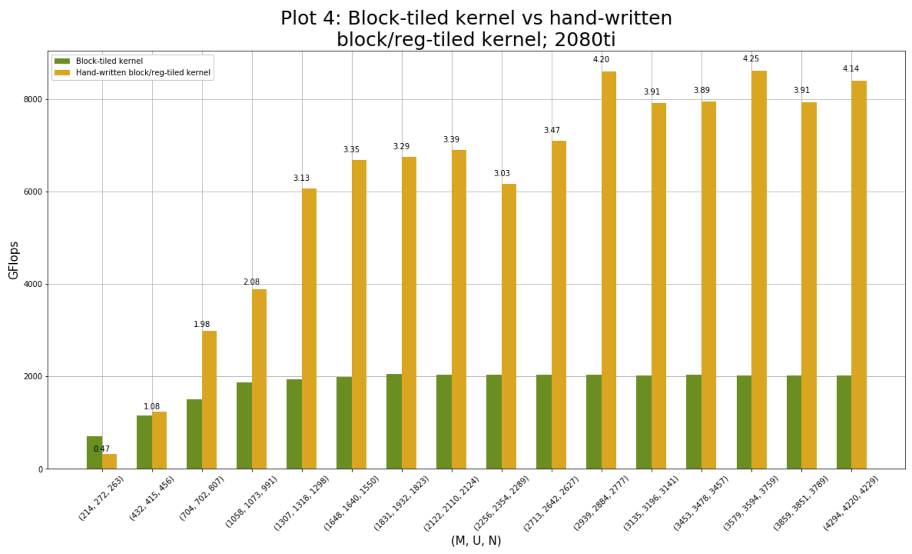
• **Plot 2:** Our implementation outperforms the existing block tiling from the third input set and onwards, indicating a sweet-spot for square inputs at around M, U, N ~600.

From this point on, our implementation obtains speedup factors of x1.2 to x1.82 over the block-tiled kernel. Speedups are more consistent for the larger inputs, indicating that the extra locality benefits from register tiling begin to converge for larger inputs.



• **Plot 3:** With speeddowns as low as x0.41, the hand-written kernel greatly beats our implementation over all input sizes - this is of course expected given the optimizations employed by the hand-written kernel.

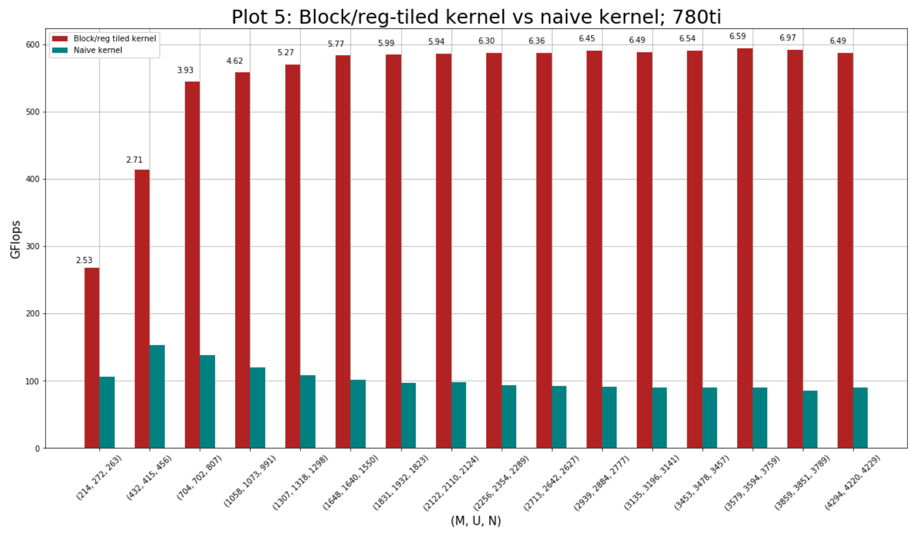
However, it is still interesting to see the hand-written kernel outperform our implementation by roughly as much as our implementation outperformed the block-tiled kernel.



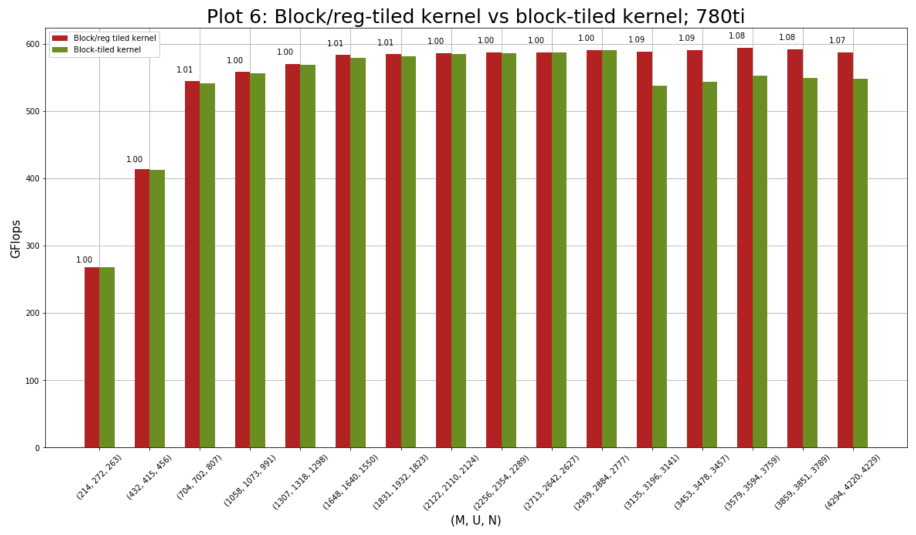
- **Plot 4:** The hand-written kernel beats the existing block-tiled kernel by up to factors of x4.25, and often by factors greater than 4.

This would tell us that there might be room for much improvement in our implementation.

6.2.2 Benchmark plots: 780ti results



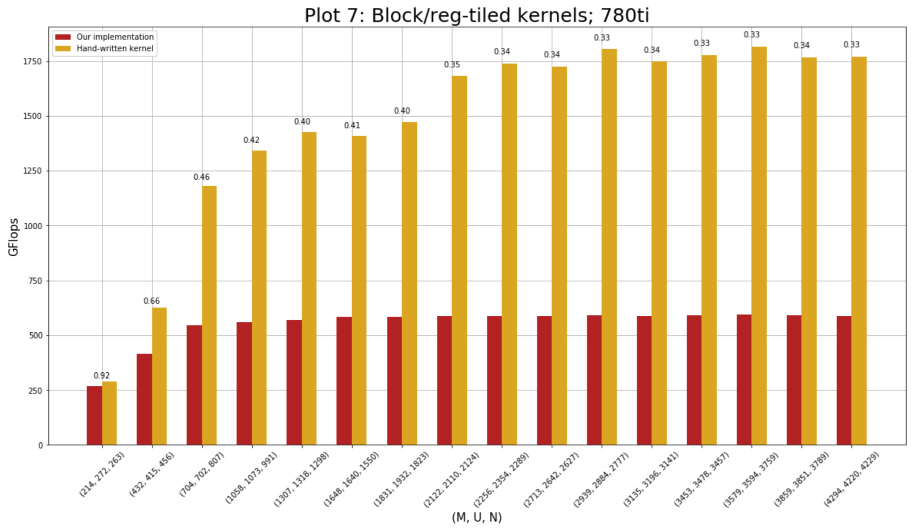
• **Plot 5:** Again, equally as expected as plot 1, our kernel outperforms the naive kernel on the 780ti aswell - however, here the speedups max out at x6.97 as opposed to x19.02 on the 2080ti. This might indicate that the locality benefits in block/register tiling are not as greatly felt on lesser hardware.



• **Plot 6:** Out of all benchmark results, these are perhaps the most interesting, yet the most ambivalent: On the 780ti, our implementation *barely* outperforms the block-tiled kernel over all inputs, with speedup factors between 1 and 1.09.

While the measurements for the largest five input sets show speedups which in isolation might appear considerable and *could indicate a sweet-spot for very large inputs*, these results are largely rendered insignificant by a stalemate across all smaller inputs.

When seen in the light of the results made on the 2080ti, this could also indicate an overhead in block/register tiling that is not particularly outweighed by locality benefits on the lesser hardware.

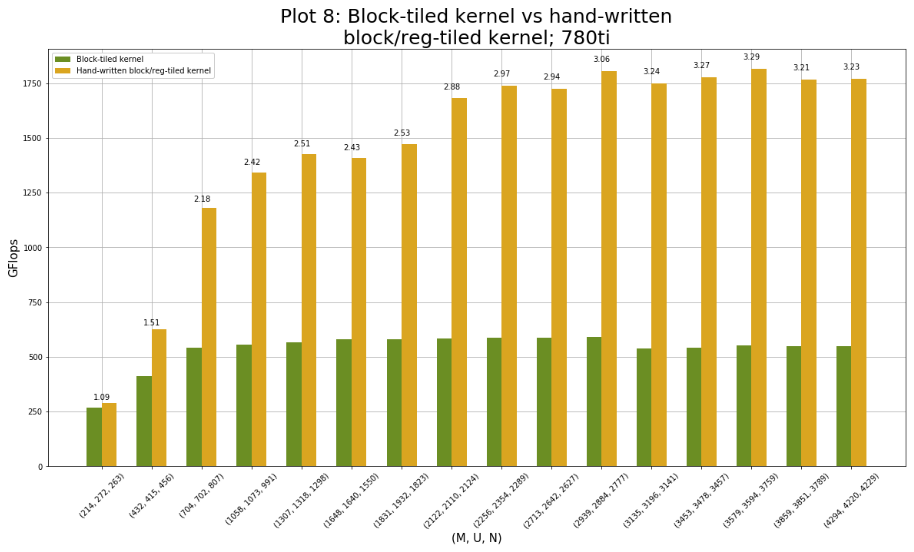


- **Plot 7:** On the 780ti, our implementation is beaten by a very consistent, roughly three times faster execution by the hand-written kernel on the larger half of the input sets.

What is perhaps most interesting about this plot is the fact that while our implementation failed to outperform the block-tiled kernel by a significant margin, there is now an *even greater* gap between the two block/register tiled kernels.

This plot would prove that block/register tiling *can be* a visible improvement over block-tiling (since the block tiled kernel's results can largely be subtituted for our implementation in this plot), thus putting into question our previous explanation of the results in plot 6.

We take this to be evidence that it must be possible to achieve better performance from a Futhark-generated block/register tiled kernel than we have managed.



- **Plot 8:** Since our block/register tiled kernel performed about as well as the existing block tiling kernel on the 780ti, plot 8 provides about the same information as plot 7.

6.3 Benchmarking: limitations and sources of error

Having presented and evaluated results of benchmarking, we will in this section discuss limitations and potential sources of error in our benchmarking.

- **Limitation: tile size combinations**

  As mentioned, we only test one set of tiling parameters per combination of program and GPU. We did not manage to test very many different types of input test sets, but when we do in the future, it will be equally meaningful to test a corresponding number of tile sizes. In fact, ideally, we would test *every* input size with parameters optimal for just that particular type of data. For example, if the data has a large `M` dimension, then it might not only be beneficial to test with large `Ty*Ry`, but to tweak *all* parameters.

  Futhark has a parameter auto-tuning utility called `futhark-autotune`. As of the time of writing this project, auto-tuning of register tile size parameters is still a matter of future research, but if ever implemented, then this limitation could more easily be levied.

- **Limitation: test input sets**

  Our benchmarks are *very* limited in that we only use a single test set containing only one type of input (near-square matrices). The benchmarks are therefore not very representative of the actual performance of our program - in testing by hand, we have for example witnessed our program not perform very well on inputs with long inner dimensions (but this is the matter of future work; see section 7.3).

  In the future, we would like to benchmark programs using all types of input sets besides (near-)square matrices: two dimensions fixed and one varying; all dimensions small or large, et cetera. Since each type of test set requires a different set of tiling parameters, this is another thing which would be easier given automated tuning of register tile size parameters from `futhark-autotune`.

- **Limitation: the GPUs used**

  As puny bachelor's students, we don't have exclusive access to (nor do we ourselves own) expensive GPU hardware. A consequence of this fact that we are limited to running benchmarks on the GPUs provided to us by the department, greatly limiting generalizability [24].

  As seen in section 6.2, the speedups gained on the 780ti were not nearly as significant (if even significant at all) as those gained on the 2080ti; as dicussed, one possible explanation is that the overhead of block/register tiling might not be as greatly amortized on the slower device.

[24]While the 2080ti is significantly faster and more capable than the 780ti (as evidenced by the benchmark results), the two devices are *relatively* near each other in technical hardware specifications.

It would be very meaningful to test and benchmark our implementation on both older, aswell as newer, more high-end hardware, as compared to the 780ti and 2080ti, to examine this further - perhaps our implementation is not even viable on smaller hardware.

Testing on a broader range of hardware could also teach us something about tuning, as optimal tiling parameters for such hardware might look completely different from those we found.

- **Source of error: occupied GPUs and imprecise measurements**

  Another consequence of the fact that we do not have exclusive access to GPUs are other people using the machines. Before a given benchmark, we would use `nvidia-smi` to assert that the particular GPU was not currently occupied running someone else's computations, but we cannot account for what happened between that and the end of a given set of benchmark runs, which typically take a couple of minutes. This is typically not a big problem, as Futhark's benchmark mechanic reports deviations in measurements, so these errors are easy to spot.

- **Source of error: "optimal" tiling parameters**

  Another very plausible explanation for the results we saw on the 780ti is that the tiling parameters found by our automated script were not actually optimal for that kernel on the 780ti - we were, in fact, surprised to see the script determine a `2*2` register tile to be optimal on the 780ti, considering that optimal register tile dimensions for the hand-written kernel were `8*4` for both GPUs tested.

  This may very well be due to source of error which is related to the prior. Our script for finding optimal tile sizes does not take measurement deviation into account - in fact, it discards this information completely and simply compares measurements. This is problematic with the number of factors in play in the search for optimal tile sizes [25]: the search took more than 8 hours to run overnight, and we have no way of accounting for whether other users' work might have occupied the GPUs in the mean-time, meaning the tile sizes we found may not actually be optimal.

  This limitation would of course not be affected by the support of register tile parameter tuning in `futhark-autotune`, as occupation of a partciular GPU would still mean irrepresentative measurements.

- **Limitation: different programs**

  As stated in section 4.5, our implementation does not support very many programs besides ordinary matrix multiplication, hence we only benchmark this. If or when block and register tiling is implemented and generalized for eg. GEMM, then we should of course benchmark these programs as well, because other types of programs may not necessarily benefit from the transformation simply because they apply.

[25] 1024 parameter combinations, two GPUs, two programs, and 16 different inputs make 65536 tests; each was run 1000 times for a total of some 65 million runs.

7 Conclusions and future work

We have argued for the theoretical benefits to temporal locality of reference in block and register tiling: Where block tiling alone reduces the number of global accesses by a factor of block tile sizes, register tiling further reduces this by a factor of register tile sizes.

We have expanded the IR of the Kernels stage of the Futhark compiler with `RegTileReturns`, enabling support for register tiling.

We have succesfully reached the project goal of implementing block and register tiling for ordinary matrix multiplication in the Kernels stage of the Futhark compiler.

In benchmarking programs optimized by our block and register tiling pass, we have seen our implementation outperform block-tiling for most inputs and with an early sweet-spot on the 2080ti, whilst on the 780ti we have not seen as significant speedups and a later sweet-spot.

Our implementation falls short of the performance of a hand-written, human-optimized kernel by a large margin, indicating opportunity for improvement in future work.

Our implementation also greatly lacks in genericity, limiting its practical applications. Much work is required before it is ready to be deployed alongside the existing block-tiling pass. However, we argue that is serves as a proof of concept - or, at the least, a grounds for future research and testing.

7.1 Future work: alleviating limitations

In the future, we will want to look into all those limitations to our implementation discussed in section 4.5, as well as all those limitations and sources of error we determined in benchmarking in section 6.3. The former has priority as it is a prerequisite to the latter, as well to any implementation into an eventual release build of Futhark.

7.2 Future work: `collectiveCopy` in the IR

In section 3, we introduced the abstraction of a `collectiveCopy` to our simplified version of the Kernels IR. Since the Kernels IR has no knowledge of the manifestation of arrays in memory, this was a nice abstraction, which we unfortunately had to get rid of during implementation. Here, we saw ourselves forced to use `WriteReturns` to handle cases of non-writes to local memory, which produced extra, redundant branches in the generated code.

It would be very nice to see such a `collectiveCopy` construct added to the IR, which could then be resolved at later compiler stages (eg. in the Explicit Memory Allocations stage) such that a guarantee of (or at least a best-effort attempt at) coalesced access to arrays could be made early in the compiler, without the need for redundant guards in the generated imperative code.

7.3 Future work: future optimization

7.3.1 Parallelizing the redomap

In a footnote in section 2.1, we briefly mentioned the possibility of parallelizing the inner **redomap**, but omitted it from the code transformation as it was (and is) outside the scope of the project. We discuss it in brief here.

In our implementation of the transformation, a grid of $(M/Ry) \cdot (N/Rx)$ threads cooperate in computing the matrix product. In such cases where `M` and `N` are large, GPU resources are easily saturated; however, when for example `M` and `N` are small, but `U` is very large, a very few threads spend a lot of time sequentially iterating the `U` dimension.

We know that **redomap** is a parallel operator [1], and so there is an opportunity in tiling the `U` dimension.

This tiling transformation introduces a sixth parameter, `Rk`, which is not a register tile size. Instead, it is used to compute `Tk*Rk`, the factor of sequentialization of the common dimension, and `U / (Tk*Rk)`, the factor of parallelization of the common dimension.

This transformation has a slight lopside and generally small trade-off. The lopside is that to compute the **redomap** in parallel, threads must compute local results and these must then sequentially be accumulated after the MM kernel, and the trade-off being that in order to accomodate this, the kernel needs to expand the result array `C` by the parallelization factor. Neither of these are big concerns, however, as they both are proportional to the size of `C`, ie. `M*N`, which, as stated, is typically small when we choose this optimization.

Once we begin testing inputs other than (near-)square matrices, it would be interesting to see this optimization in play.

7.3.2 Special case optimization: ordinary matrix multiplication

We saw the handwritten, human-optimized MM kernel outperform all other implementations over all input sets and on both GPUs, with speedup factors in the range of ~2.1-3 over our implementation on the 780ti. In section 6.1.1 we presented some of the optimizations used in the handwritten kernel, and mentioned that it also employs some which are specific for MM. These optimizations are in fact valid in many similar programs, but for this discussion we shall stick to MM.

The optimization in question pertains to boundary checks surrounding local and register memory, and checks surrounding computations made in register memory.

During code transformation of ordinary MM in section 2.1, we learned that when a loop is stripmined, boundary checks are inserted to assert that we never exit the bounds of the original loop when tile sizes do not divide input dimensions. Recall that padding values are sometimes inserted into local memory during reading from global memory; without the aforementioned checks, we would sometimes be making computations using padding values.

Now, since Futhark is a functional language, arrays are always written fully, and when the data type in question is integral, the padding value is zero.

Consider the line of code which accumulates the dot product in MM:

```
acc[i, j] += as[i] * bs[j],
```

where `as` and `bs` are register tile slices of `A` and `B`. If either `as[i]` or `b[j]` (or both) is not an actual value, but rather a padding value of zero, then the result of the multiplication will be zero and nothing is accumulated in `acc[i, j]`.

As such, mapping and accumulating padding values have no effect, and boundary checks are not necessary when accessing local and register memory. Here, one can simply make the computations, and any invalid results are then guarded against by the boundary check surrounding the write to `C` - because if out of bounds, then `acc[i, j]` does not correspond to an actual element of `C`.

To generalize, this optimization is safe whenever:

```
p map_op x = x map_op p = p map_op p = red_ne
```

where `p` is the padding value used; `map_op` is the **map** operator; `x` is an arbitrary, actual input value; and `red_ne` is the neutral element to some **reduce** operator, and whenever `map_op` is well-defined for all possible values of `p`.

However, in a more general case where `map_op` and the **reduce** operator are something other than for example multiplication and addition, it is difficult to reason about how to choose `p` such that the result of `map_op` becomes the neutral element to the reduction. It can also be hard to determine whether the `map_op` is even well-defined: consider for example the case `map_op = (\x y -> x/y)`, where source arrays for `x` and `y` are integral. In this case, we can definitely not pad with zeroes as this would very likely mean imposing integer zero division exceptions upon a program which would not be the fault of the user.

In fact, this problem is not exclusive to the padding of the second operand matrix in local memory; consider `(\x y -> y/x)`. Choosing a padding value different from zero does not guarantee non-failure either: For any padding value `k` hard-coded into code generation, a `map_op` could always be chosen which could potentially produce an error, eg. `(\x y -> x/(y - k))`.

As such, there are infinitely many programs for which this optimization is illegal, but at the same time there are infinitely many programs for which it is perfectly safe and beneficial. Since this optimization is only legal for those specific **redomap** lambdas described above, we would need some method of identifying this in pattern matching of the AST of the **redomap**.

In any case, these optimizations specific to matrix multiplication, while incredibly powerful (as evidenced by our benchmark results), may not be as big of a priority since the benefits here will only be felt by a small subset of programs, and because implementing pattern matching in the compiler to detect all eligible programs might prove difficult.

7.3.3 Handling `M` and `N` dimension residuals in an "epilogue"

In section 4.2 we saw how the number of boundary checks in the handling of residual input in the common dimension could be optimized via an epilogue. A very similar optimization can safely be applied to residuals in the `M` and `N` dimensions - except here, it is not so much an epilogue as it is different versions of the code; nevertheless, we shall call it an epilogue because it is conceptually similar.

As discussed only groups at the edges of the grid experience these residuals, so where we previously unrolled the last iteration of the `kk` loop, here, we would let individual groups branch off right after initialization of register memory, to a version of the same kernel but with `M` (or `N`) boundary checks omitted.

The implementation is symmetric in the `M` and `N` dimensions, so as an example, let's consider an epilogue in the `M` dimension. Here is what that program would look like:

```
if (group_index_y == grid_dim_y - 1 &&  // if last group in y-direction.
    M % Ty*Ry)                          // if residual in M-dimension.
  /* kernel body */

else /* epilogue: kernel body, with M boundary checks omitted */
```

As with the epilogue in the common dimension, the code nearly doubles in size. However, one considerable difference between this epilogue and the type of epilogue we saw for residuals in the `U` dimension is that not all groups benefit from this optimization [26], and the entire result is of course not ready before all groups finish execution [27].

The optimization can of course be implemented simultaneously in the `M` *and* `N` dimension. This program would look like so:

```
if (group_index_y == grid_dim_y - 1 &&  // if last group in both directions.
    group_index_x == grid_dim_x - 1 &&
    M % Ty*Ry && N % Tx*Rx)             // if residual in both directions.

  /* kernel body, with all boundary checks intact */

else if (group_index_y == grid_dim_y - 1 && M % Ty*Ry)
  /* kernel body, with M boundary checks omitted */

else if (group_index_x == grid_dim_x - 1 && N % Tx*Rx)
  /* kernel body, with N boundary checks omitted */

else /* kernel body, with both M and N boundary checks omitted */
```

and would obviously result in a quadruple in code size. The combined number of `M` and `N` boundary checks omitted here is *larger* than the number of `U` boundary checks, so this should be a welcomed optimization.

However, simple benchmarks (which we have not documented) showed an immense slowdown very likely caused by this quadrupling and so we did not pursue it further. We did not look into combining the `U` dimension epilogue with one or both of the `M` and `N` dimension epilogues.

[26]Because `group_index_y == grid_dim_y - 1` for `grid_dim_x` number of groups.

[27]Due to the limit on the number of scheduled groups at a given time, this should be amortized as `M` and `N` grow, since the number of bottlenecking "epilogue" groups is propotional to `N` while the grid size is proportional to `M*N`.

References

[1] Cosmin E. Oancea. *Lecture Notes for the Software Track of the PMPH Course.* 2018.

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

[3] Nvidia. Cuda C++ Programming Guide, Table 15: Technical Specifications per Compute Capability. CUDA Toolkit Documentation, 2019. URL `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[4] Mark Harris. How to Access Global Memory Efficiently in CUDA C/C++ Kernels. Nvidia developer blog post, 2013. URL `https://devblogs.nvidia.com/how-access-global-memory-\efficiently-cuda-c-kernels/`.

[5] Steffen Holst Larsen. Multi-GPU Futhark Using Parallel Streams. Msc thesis, 2019.

[6] Troels Henriksen. Design and Implementation of the Futhark Programming Language (Revised). PhD thesis, 2017.

[7] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin E. Oancea. Incremental Flattening for Nested Data Parallelism. PPoPP, 2019.

Appendix

A Full, unsimplified IR sketch

Here comes the full, unsimplified sketch of block and register tiled (ordinary) matrix multiplication using an epilogue to handle residual input, with a near-one-to-one correspondence to the IR as it should look when it leaves the loop tiling step of the Kernels stage.

```
-- host code goes here. assume B transposed at this point, such that A :: [M][U], B :: [N][U].
let {[M][N]t res} = segmap group (gid_y < gridDim_y, gid_x < gridDim_x) {
  let {i32 iii} = gid_y * Ty * Ry
  let {i32 jjj} = gid_x * Tx * Rx

  -- threads each allocate [Ry][Rx] of private memory and
  -- initialize with neutral elements wrt. the reduction.
  let {[Ty][Tx][Ry][Rx]t grp_acc_init} =
    segmap threadPrivate (ltid_y < Ty, ltid_x < Tx) {
      let {[Ry][Rx]t thd_acc_init} = scratch([Ry][Rx], t)
      in loop (thd_acc_merge = thd_acc_init) for i < Ry do {
        loop (thd_acc_merge' = thd_acc_merge) for j < Rx do {
          thd_acc_merge' with [i, j] <- 0
        }
      }
    }

  -- allocate local memory.
  let {[Ty*Ry][Tk]t A_loc_init} = scratch([Ty*Ry][Tk], t)
  let {[Tx*Rx][Tk]t B_loc_init} = scratch([Tx*Rx][Tk], t)

  -- START prologue
  let {i32 num_full_tiles} = U/Tk
  let {[Ty][Tx][Ry][Rx]t prologue_res} =
  loop (grp_acc_merge = grp_acc_init) for kk0 < num_full_tiles do {
    let {i32 kk} = kk0 * Tk

    -- each group collectively copies 2D slices A[iii : iii + Ty*Ry, kk : kk+Tk]
    -- and B[jjj : jjj + Tx*Rx, kk : kk+Tk] from global to local memory.

    -- A_loc :: [Ty][Tx][Ry][Tk/Tx] <=> [Ty*Ry][Tk].
    let {[Ty*Ry][Tk]t A_loc} =
      segmap thread (ltid_y < Ty, ltid_x < Tx) {
        loop (A_loc_merge = A_loc_init) for i0 < Ry do {
          let {i32 i} = ltid_y + i0 * Ty
          in loop (A_loc_merge' = A_loc_merge) for k0 < ceil(Tk/Tx) do {
            let {i32 k} = ltid_x + k0 * Tx
            let {t a_elem} = if (iii+i < M) then A[iii + i, kk + k] else 0
            in A_loc_merge'
              with [i, k] <- a_elem if (k < Tk) -- this check is necessary in case Tx does not divide Tk.
          }
        }
      }

    -- B_loc :: [Ty][Tx][Rx][Tk/Ty] <=> [Tx*Rx][Tk].
    let {[Tx*Rx][Tk]t B_loc} =
      segmap thread (ltid_y < Ty, ltid_x < Tx) {
        loop (B_loc_merge = A_loc_init) for j0 < Rx do {
          let {i32 j} = ltid_x + j0 * Tx
          in loop (B_loc_merge' = B_loc_merge) for k0 < ceil(Tk/Ty) do {
            let {i32 k} = ltid_y + k0 * Ty
            let {t b_elem} = if (jjj+j < N) then B[jjj + j, kk + k] else 0
            in B_loc_merge'
              with [j, k] <- b_elem if (k < Tk) -- this check is necessary in case Ty does not divide Tk.
          }
        }
      }

    let {[Ty][Tx][Ry][Rx]t thread_res} =
    loop (grp_acc_merge' = grp_acc_merge) for k < Tk do {

      -- each thread copies 1D slices A_loc[ltid_y*Ry : ltid_y*Ry + Ry]
      -- and B_loc[ltid_x*Rx : ltid_x*Rx + Rx, k] from local to private mem.
      let {[Ty][Tx][Ry]t asss, [Ty][Tx][Rx]t bsss} =
        segmap threadPrivate (ltid_y < Ty, ltid_x < Tx) {
          let {[Ry]t as_init} = scratch([Ry], t)
          let {[Rx]t bs_init} = scratch([Rx], t)
          let {[Ry]t as} = loop (as_merge = as_init) for i < Ry do {
                             as_merge with [i] <- A_loc[ltid_y*Ry + i, k]
                           }
          let {[Rx]t bs} = loop (bs_merge = bs_init) for j < Rx do {
                             bs_merge with [j] <- B_loc[ltid_x*Rx + j, k]
                           }
          in (as, bs)
        }

      -- each thread computes redomap over its register tile.
      in segmap threadPrivate (ltid_y < Ty, ltid_x < Tx) {
        let {[Ry]t as} = asss[ltid_y, ltid_x]
        let {[Rx]t bs} = bsss[ltid_y, ltid_x]
        let {[Ry][Rx]t thd_acc_init} = grp_acc_merge'[ltid_y, ltid_x]

        in loop (thd_acc_merge = thd_acc_init) for i < Ry do {
          loop (thd_acc_merge' = thd_acc_merge) for j < Rx do {
            if (iii + i < M && jjj + j < N) then {
              let {t map_res} = as[i] * bs[j]
              let {t red_res} = thd_acc_merge'[i, j] + map_res
              in thd_acc_merge' with [i, j] <- red_res
                }
            else {
              thd_acc_merge'
            }
          }
        }
      }
    }
    in thread_res
  }
  -- END prologue
  -- START epilogue
  let {[Ty][Tx][Ry][Rx]t epilogue_res = {
    let {i32 kk} = num_full_tiles * Tk

    let {[Ty*Ry][Tk]t A_loc} =
      segmap thread (ltid_y < Ty, ltid_x < Tx) {
        loop (A_loc_merge = A_loc_init) for i0 < Ry do {
          let {i32 i} = ltid_y + i0 * Ty
          in loop (A_loc_merge' = A_loc_merge) for k0 < ceil(Tk/Tx) do {
            let {i32 k} = ltid_x + k0 * Tx
            let {t a_elem} =
              if (iii+i < M) && (kk+k < U) -- extra bounds check in the common dim.
              then A[iii + i, kk + k] else 0
            in A_loc_merge'
              with [i, k] <- a_elem if (k < Tk)
          }
        }
      }

    let {[Tx*Rx][Tk]t B_loc} =
      segmap thread (ltid_y < Ty, ltid_x < Tx) {
        loop (B_loc_merge = A_loc_init) for j0 < Rx do {
          let {i32 j} = ltid_x + j0 * Tx
          in loop (B_loc_merge' = B_loc_merge) for k0 < ceil(Tk/Ty) do {
            let {i32 k} = ltid_y + k0 * Ty
            let {t b_elem} =
              if (jjj+j < N) && (kk+k < U) -- extra bounds check in the common dim.
              then B[jjj + j, kk + k] else 0
            in B_loc_merge'
              with [j, k] <- b_elem if (k < Tk)
          }
        }
      }

    let {[Ty][Tx][Ry][Rx]t thread_res} =
    loop (grp_acc_merge' = grp_acc_merge) for k < Tk do {
      if (kk + k < U) then { -- extra bounds check in the common dim.

        let {[Ty][Tx][Ry]t asss, [Ty][Tx][Rx]t bsss} =
          segmap threadPrivate (ltid_y < Ty, ltid_x < Tx) {
            let {[Ry]t as_init} = scratch([Ry], t)
            let {[Rx]t bs_init} = scratch([Rx], t)
            let {[Ry]t as} = loop (as_merge = as_init) for i < Ry do {
                               as_merge with [i] <- A_loc[ltid_y*Ry + i, k]
                             }
            let {[Rx]t bs} = loop (bs_merge = bs_init) for j < Rx do {
                               bs_merge with [j] <- B_loc[ltid_x*Rx + j, k]
                             }
            in (as, bs)
          }

        in segmap threadPrivate (ltid_y < Ty, ltid_x < Tx) {
          let {[Ry]t as} = asss[ltid_y, ltid_x]
          let {[Rx]t bs} = bsss[ltid_y, ltid_x]
          let {[Ry][Rx]t thd_acc_init} = grp_acc_merge'[ltid_y, ltid_x]

          in loop (thd_acc_merge = thd_acc_init) for i < Ry do {
            loop (thd_acc_merge' = thd_acc_merge) for j < Rx do {
              if (iii + i < M && jjj + j < N) then {
                let {t map_res} = as[i] * bs[j]
                let {t red_res} = thd_acc_merge'[i, j] + map_res
                in thd_acc_merge' with [i, j] <- red_res
              }
              else {
                thd_acc_merge'
              }
            }
          }
        }
      }
      else { -- if out of bounds of common dim, keep current acc.
        grp_acc_merge'
      }
      in thread_res
    }
  }
  in RegTileReturns [(M, Ty, Ry), (N, Tx, Rx)] epilogue_res
}
```

B Test reproduction

To reproduce our validation and/or benchmark tests, do:

- log onto or be running a machine with a CUDA-enabled device.
- clone our branch of the Futhark compiler and install it per the instructions given in 4.
- to benchmark the block-tiled kernel, install Futhark from the master branch and do:

  ```
  $ cp ~/.local/bin/futhark ~/.local/bin/futhark_block_tiled
  ```

  before switching back to our branch and rebuilding. This should comply with our `Makefiles`.
- clone our public project repository at:

  ```
  https://github.com/sortraev/nybachelorprojekt_public.git
  ```

- navigate to either of the `testing/validation` or `testing/benchmarks` directories inside the repo.
- from here, browse our test suite; run `make` in either directory to run corresponding tests; or browse through our `Makefiles` to see additional options.

C Benchmark results

C.1 Benchmark results: 2080ti

(M,	U,	N)	(µs,	GFlops)	(RSD,	min,	max)
(214,	272,	263)	(61,	501.92)	(0.023,	-4%,	+18%)
(432,	415,	456)	(233,	701.73)	(0.132,	-14%,	+26%)
(704,	702,	807)	(1106,	721.20)	(0.112,	-4%,	+42%)
(1058,	1073,	991)	(2591,	868.40)	(0.084,	-3%,	+42%)
(1307,	1318,	1298)	(4894,	913.76)	(0.062,	-2%,	+42%)
(1648,	1640,	1550)	(9195,	911.19)	(0.052,	-3%,	+40%)
(1831,	1932,	1823)	(34109,	378.13)	(0.039,	-2%,	+39%)
(2122,	2110,	2124)	(61504,	309.25)	(0.033,	-4%,	+35%)
(2256,	2354,	2289)	(95689,	254.07)	(0.120,	-4%,	+37%)
(2713,	2642,	2627)	(170354,	221.07)	(0.309,	-60%,	+24%)
(2939,	2884,	2777)	(223283,	210.84)	(0.016,	-3%,	+3%)
(3135,	3196,	3141)	(312417,	201.47)	(0.018,	-3%,	+7%)
(3453,	3478,	3457)	(419639,	197.87)	(0.032,	-1%,	+59%)
(3579,	3594,	3759)	(499253,	193.70)	(0.002,	-1%,	+1%)
(3859,	3851,	3789)	(587707,	191.62)	(0.003,	-1%,	+1%)
(4294,	4220,	4229)	(721880,	212.31)	(0.018,	-1%,	+10%)

Table 6: Benchmark results: naive, un-tiled MM kernel; square matrices; 2080ti

(M,	U,	N)	(µs,	GFlops)	(RSD,	min,	max)
(214,	272,	263)	(43,	712.03)	(0.023,	-3%,	+8%)
(432,	415,	456)	(142,	1151.43)	(0.025,	-2%,	+8%)
(704,	702,	807)	(528,	1510.70)	(0.129,	-15%,	+13%)
(1058,	1073,	991)	(1205,	1867.25)	(0.158,	-10%,	+31%)
(1307,	1318,	1298)	(2301,	1943.48)	(0.129,	-6%,	+34%)
(1648,	1640,	1550)	(4194,	1997.72)	(0.103,	-3%,	+37%)
(1831,	1932,	1823)	(6290,	2050.51)	(0.083,	-2%,	+38%)
(2122,	2110,	2124)	(9341,	2036.19)	(0.071,	-2%,	+38%)
(2256,	2354,	2289)	(11914,	2040.63)	(0.061,	-2%,	+38%)
(2713,	2642,	2627)	(18396,	2047.15)	(0.046,	-1%,	+38%)
(2939,	2884,	2777)	(23021,	2044.92)	(0.059,	-2%,	+39%)
(3135,	3196,	3141)	(31085,	2024.84)	(0.057,	-2%,	+41%)
(3453,	3478,	3457)	(40640,	2043.16)	(0.026,	-2%,	+34%)
(3579,	3594,	3759)	(47638,	2029.97)	(0.033,	-2%,	+15%)
(3859,	3851,	3789)	(55477,	2029.97)	(0.034,	-2%,	+32%)
(4294,	4220,	4229)	(75500,	2030.00)	(0.036,	-2%,	+36%)

Table 7: Benchmark results: block tiled MM kernel; square matrices; 2080ti

(M,	U,	N)	(µs,	GFlops)	(RSD,	min,	max)
(214,	272,	263)	(142,	215.62)	(0.114,	-14%,	+21%)
(432,	415,	456)	(202,	809.42)	(0.096,	-13%,	+26%)
(704,	702,	807)	(379,	2104.62)	(0.130,	-7%,	+44%)
(1058,	1073,	991)	(1007,	2234.39)	(0.085,	-3%,	+40%)
(1307,	1318,	1298)	(1525,	2932.42)	(0.073,	-4%,	+44%)
(1648,	1640,	1550)	(2502,	3348.69)	(0.059,	-2%,	+40%)
(1831,	1932,	1823)	(4692,	2748.87)	(0.157,	-17%,	+23%)
(2122,	2110,	2124)	(6865,	2770.59)	(0.121,	-20%,	+14%)
(2256,	2354,	2289)	(6971,	3487.60)	(0.066,	-3%,	+39%)
(2713,	2642,	2627)	(10870,	3464.52)	(0.039,	-4%,	+37%)
(2939,	2884,	2777)	(13315,	3535.57)	(0.027,	-6%,	+33%)
(3135,	3196,	3141)	(17566,	3583.19)	(0.019,	-1%,	+38%)
(3453,	3478,	3457)	(22269,	3728.68)	(0.017,	-1%,	+39%)
(3579,	3594,	3759)	(26245,	3684.64)	(0.016,	-1%,	+38%)
(3859,	3851,	3789)	(31444,	3581.50)	(0.014,	-2%,	+32%)
(4294,	4220,	4229)	(41925,	3655.69)	(0.013,	-1%,	+30%)

Table 8: Benchmark results: block and register tiled MM kernel; square matrices; 2080ti

(M,	U,	N)	(µs,	GFlops)	(RSD,	min,	max)
(214,	272,	263)	(92,	332.80)	(0.020,	-2%,	+20%)
(432,	415,	456)	(132,	1238.66)	(0.011,	-1%,	+11%)
(704,	702,	807)	(267,	2987.46)	(0.121,	-11%,	+30%)
(1058,	1073,	991)	(579,	3886.07)	(0.115,	-10%,	+43%)
(1307,	1318,	1298)	(736,	6076.00)	(0.090,	-3%,	+52%)
(1648,	1640,	1550)	(1253,	6686.70)	(0.074,	-2%,	+53%)
(1831,	1932,	1823)	(1910,	6752.72)	(0.056,	-2%,	+43%)
(2122,	2110,	2124)	(2759,	6893.83)	(0.046,	-2%,	+43%)
(2256,	2354,	2289)	(3937,	6175.27)	(0.102,	-27%,	+7%)
(2713,	2642,	2627)	(5308,	7094.83)	(0.036,	-1%,	+44%)
(2939,	2884,	2777)	(5478,	8593.67)	(0.033,	-2%,	+46%)
(3135,	3196,	3141)	(7952,	7915.27)	(0.030,	-2%,	+44%)
(3453,	3478,	3457)	(10445,	7949.63)	(0.038,	-1%,	+45%)
(3579,	3594,	3759)	(11219,	8619.62)	(0.022,	-1%,	+41%)
(3859,	3851,	3789)	(14183,	7940.26)	(0.021,	-1%,	+41%)
(4294,	4220,	4229)	(18251,	8397.61)	(0.017,	-2%,	+37%)

Table 9: Benchmark results: hand-written MM kernel using MM-specific optimizations; square matrices; 2080ti

C.2 Benchmark results: 780ti

(M,	U,	N)	(µs,	GFlops)	(RSD,	min,	max)
(214,	272,	263)	(288,	106.31)	(0.006,	-1%,	+3%)
(432,	415,	456)	(1066,	153.38)	(0.031,	-2%,	+6%)
(704,	702,	807)	(5751,	138.70)	(0.055,	-21%,	+8%)
(1058,	1073,	991)	(18616,	120.87)	(0.178,	-15%,	+22%)
(1307,	1318,	1298)	(41264,	108.37)	(0.176,	-12%,	+30%)
(1648,	1640,	1550)	(82875,	101.10)	(0.009,	-1%,	+6%)
(1831,	1932,	1823)	(132064,	97.66)	(0.034,	-3%,	+5%)
(2122,	2110,	2124)	(192540,	98.79)	(0.050,	-22%,	+7%)
(2256,	2354,	2289)	(261039,	93.14)	(0.179,	-15%,	+31%)
(2713,	2642,	2627)	(408035,	92.29)	(0.177,	-12%,	+31%)
(2939,	2884,	2777)	(514456,	91.51)	(0.005,	-1%,	+3%)
(3135,	3196,	3141)	(694050,	90.69)	(0.032,	-2%,	+5%)
(3453,	3478,	3457)	(918711,	90.38)	(0.053,	-22%,	+8%)
(3579,	3594,	3759)	(1071606,	90.24)	(0.179,	-16%,	+32%)
(3859,	3851,	3789)	(1324856,	85.00)	(0.176,	-12%,	+30%)
(4294,	4220,	4229)	(1691224,	90.62)	(0.010,	-1%,	+6%)

Table 10: Benchmark results: naive, un-tiled MM kernel; square matrices; 780ti

(M, U, N)	(µs, GFlops)	(RSD, min, max)
(214, 272, 263)	(114, 268.57)	(0.004, -1%, +2%)
(432, 415, 456)	(395, 413.93)	(0.031, -2%, +6%)
(704, 702, 807)	(1473, 541.52)	(0.056, -21%, +8%)
(1058, 1073, 991)	(4046, 556.11)	(0.179, -16%, +31%)
(1307, 1318, 1298)	(7861, 568.88)	(0.176, -12%, +30%)
(1648, 1640, 1550)	(14454, 579.66)	(0.004, -1%, +2%)
(1831, 1932, 1823)	(22179, 581.53)	(0.032, -2%, +9%)
(2122, 2110, 2124)	(32538, 584.55)	(0.058, -24%, +8%)
(2256, 2354, 2289)	(41456, 586.45)	(0.179, -16%, +33%)
(2713, 2642, 2627)	(64139, 587.15)	(0.179, -15%, +33%)
(2939, 2884, 2777)	(79766, 590.18)	(0.177, -13%, +30%)
(3135, 3196, 3141)	(116818, 538.81)	(0.011, -1%, +5%)
(3453, 3478, 3457)	(152900, 543.06)	(0.032, -2%, +8%)
(3579, 3594, 3759)	(175179, 552.03)	(0.020, -1%, +6%)
(3859, 3851, 3789)	(204795, 549.90)	(0.176, -16%, +33%)
(4294, 4220, 4229)	(279209, 548.92)	(0.179, -13%, +30%)

Table 11: Benchmark results: block tiled MM kernel; square matrices; 780ti

(M, U, N)	(µs, GFlops)	(RSD, min, max)
(214, 272, 263)	(114, 268.57)	(0.006, -0%, +4%)
(432, 415, 456)	(394, 414.98)	(0.032, -2%, +5%)
(704, 702, 807)	(1465, 544.47)	(0.040, -24%, +7%)
(1058, 1073, 991)	(4030, 558.32)	(0.181, -16%, +33%)
(1307, 1318, 1298)	(7833, 570.91)	(0.177, -13%, +30%)
(1648, 1640, 1550)	(14369, 583.09)	(0.007, -1%, +6%)
(1831, 1932, 1823)	(22049, 584.96)	(0.033, -2%, +11%)
(2122, 2110, 2124)	(32428, 586.53)	(0.023, -1%, +8%)
(2256, 2354, 2289)	(41411, 587.09)	(0.179, -16%, +32%)
(2713, 2642, 2627)	(64110, 587.42)	(0.177, -16%, +22%)
(2939, 2884, 2777)	(79748, 590.31)	(0.178, -12%, +31%)
(3135, 3196, 3141)	(106909, 588.75)	(0.005, -1%, +2%)
(3453, 3478, 3457)	(140577, 590.67)	(0.032, -2%, +9%)
(3579, 3594, 3759)	(162687, 594.41)	(0.040, -21%, +8%)
(3859, 3851, 3789)	(190198, 592.10)	(0.175, -16%, +21%)
(4294, 4220, 4229)	(260776, 587.73)	(0.177, -12%, +31%)

Table 12: Benchmark results: block and register tiled MM kernel; square matrices; 780ti

(M, U, N)	(μs, GFlops)	(RSD, min, max)
(214, 272, 263)	(105, 291.59)	(0.004, -0%, +2%)
(432, 415, 456)	(261, 626.45)	(0.027, -1%, +6%)
(704, 702, 807)	(676, 1179.96)	(0.060, -21%, +9%)
(1058, 1073, 991)	(1675, 1343.30)	(0.180, -16%, +32%)
(1307, 1318, 1298)	(3132, 1427.82)	(0.177, -13%, +30%)
(1648, 1640, 1550)	(5939, 1410.75)	(0.008, -1%, +6%)
(1831, 1932, 1823)	(8753, 1473.52)	(0.033, -3%, +6%)
(2122, 2110, 2124)	(11313, 1681.26)	(0.031, -22%, +7%)
(2256, 2354, 2289)	(13975, 1739.68)	(0.179, -15%, +32%)
(2713, 2642, 2627)	(21833, 1724.88)	(0.179, -16%, +31%)
(2939, 2884, 2777)	(26098, 1803.82)	(0.175, -12%, +30%)
(3135, 3196, 3141)	(36000, 1748.40)	(0.009, -1%, +6%)
(3453, 3478, 3457)	(46700, 1778.03)	(0.032, -2%, +6%)
(3579, 3594, 3759)	(53228, 1816.78)	(0.055, -22%, +8%)
(3859, 3851, 3789)	(63756, 1766.37)	(0.177, -16%, +21%)
(4294, 4220, 4229)	(86554, 1770.74)	(0.176, -13%, +30%)

Table 13: Benchmark results: hand-written MM kernel; square matrices; 780ti

D Implementation source code: BlkRegTiling.hs

In this section of the appendix is the entire contents of Futhark.Optimise.BlkRegTiling.hs as it appears in our branch of the Futhark repository at the time of hand-in.

Highlighted lines are our own additions, and lines not highlighted are courtesy of Cosmin, and in part Troels, our supervisors.

```haskell
-- | Perform a restricted form of block+register tiling corresponding to
--   the following pattern:
--     * a redomap is quasi-perfectly nested inside a kernel with at
--       least two parallel dimension (the perfectly nested restriction
--       is relaxed a bit to allow for SGEMM);
--     * all streamed arrays are one dimensional;
--     * all streamed arrays are variant to exacly one of the two
--       innermost parallel dimensions, and conversely for each of
--       the two innermost parallel dimensions, there is at least
--       one streamed array variant to it;
--     * the stream's result is a tuple of scalar values, which are
--       also the "thread-in-space" return of the kernel.
--   Test code can be found in "tests/mmm/sgemm.fut".
module Futhark.Optimise.BlkRegTiling
       ( mm_BlkRegTiling )
       where
import Control.Monad.State
import Control.Monad.Reader
import qualified Data.Map.Strict as M
import qualified Data.Sequence as Seq
import Data.List
import Data.Maybe
import Debug.Trace

import Futhark.MonadFreshNames
import Futhark.Representation.Kernels
import Futhark.Tools
import Futhark.Transform.Rename

type TileM = ReaderT (Scope Kernels) (State VNameSource)
type VarianceTable = M.Map VName Names

mm_BlkRegTiling :: Stm Kernels -> TileM (Maybe (Stms Kernels, Stm Kernels))
mm_BlkRegTiling (Let pat aux (Op (SegOp (SegMap SegThread{} seg_space ts old_kbody))))
  | KernelBody () kstms _ <- old_kbody,

    -- build the variance table, that records, for
    -- each variable name, the variables it depends on
    initial_variance <- M.map mempty $ scopeOfSegSpace seg_space,
    variance <- varianceInStms initial_variance kstms,

    -- check that the code fits the pattern having:
    -- some `code1`, followed by one Screma SOAC, followed by some `code2`
    (code1, Just screma_stmt, code2) <- matchCodeStreamCode kstms,

    Let pat_redomap _ (Op _) <- screma_stmt,

    -- checks that the Screma SOAC is actually a redomap and normalizes it
    Just (common_dim, arrs, (_, red_lam, red_nes, map_lam)) <- isTileableRedomap screma_stmt,

    -- checks that the input arrays to redomap are variant to
    -- exactly one of the two innermost dimensions of the kernel
    Just _ <- isInvarTo1of2InnerDims mempty seg_space variance arrs,

    -- get the variables on which the first result of redomap depends on
    fst_res : _      <- patternValueElements pat_redomap,
    Just res_red_var <- M.lookup (patElemName fst_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', _) <- foldl (processIndirections (namesFromList arrs) res_red_var)
                                 (Just (Seq.empty, M.empty)) code1,

    null code2 && null code2', -- TODO: remove the need for these assumptions !

    -- we get the global-thread id for the two inner dimensions,
    --   as we are probably going to use it in code generation
    (gtid_x, width_B) : (gtid_y, height_A) : rem_outer_dims <- reverse $ unSegSpace seg_space,

    null rem_outer_dims, -- TODO: remove the need for this assumption !

    -- sanity check that the reduce part is not missing
    not $ null red_nes = do
      let load_A : load_B : _ = stmsToList code1 -- TODO: unsafe in general, since first two
                                                 --       elements of code1 may be something else.
      let inp_A  : inp_B  : _ = arrs
      let map_t1 : map_t2 : _ = map (elemType . paramAttr) (lambdaParams map_lam)
      let red_ne : _ = red_nes
      red_t <- subExpType red_ne

      ---- in this binder: host code and outer seggroup (ie. the new kernel) ----
      (new_kernel, host_stms) <- runBinder $ do -- host code

        tk_name    <- nameFromString . pretty <$> newVName "Tk"
        tx_name    <- nameFromString . pretty <$> newVName "Tx"
        ty_name    <- nameFromString . pretty <$> newVName "Ty"
        rx_name    <- nameFromString . pretty <$> newVName "Rx"
        ry_name    <- nameFromString . pretty <$> newVName "Ry"
        tk         <- letSubExp "Tk" $ Op $ SizeOp $ GetSize tk_name SizeTile
        tx         <- letSubExp "Tx" $ Op $ SizeOp $ GetSize tx_name SizeTile
        ty         <- letSubExp "Ty" $ Op $ SizeOp $ GetSize ty_name SizeTile
        rx         <- letSubExp "Rx" $ Op $ SizeOp $ GetSize rx_name SizeRegTile
        ry         <- letSubExp "Ry" $ Op $ SizeOp $ GetSize ry_name SizeRegTile

        tk_div_tx  <- letSubExp "tk_div_tx" =<< ceilDiv tk tx
        tk_div_ty  <- letSubExp "tk_div_ty" =<< ceilDiv tk ty

        tx_rx      <- letSubExp "TxRx" =<< toExp (primFromSe tx * primFromSe rx)
        ty_ry      <- letSubExp "TyRy" =<< toExp (primFromSe ty * primFromSe ry)

        a_loc_sz   <- letSubExp "a_loc_sz" =<<
                        toExp (primFromSe ty * primFromSe ry * primFromSe tk)

        b_loc_sz   <- letSubExp "b_loc_sz" =<<
                       toExp (primFromSe tk * primFromSe tx * primFromSe rx)

        gridDim_x  <- letSubExp "gridDim_x"  =<< ceilDiv width_B  tx_rx
        gridDim_y  <- letSubExp "gridDim_y"  =<< ceilDiv height_A ty_ry
        grid_size  <- letSubExp "grid_size"  =<< toExp (primFromSe gridDim_x * primFromSe gridDim_y)
        group_size <- letSubExp "group_size" =<< toExp (primFromSe ty * primFromSe tx)
        let segthd_lvl = SegThread (Count grid_size) (Count group_size) SegNoVirtFull

        gid_x      <- newVName "gid_x"
        gid_y      <- newVName "gid_y"
        gid_flat   <- newVName "gid_flat"

        ---- in this binder: outer seggroup ----
        (ret_seggroup, stms_seggroup) <- runBinder $ do

          iii <- letExp "iii" =<< toExp (LeafExp gid_y int32 * primFromSe ty_ry)
          jjj <- letExp "jjj" =<< toExp (LeafExp gid_x int32 * primFromSe tx_rx)

          -- initialize register mem with neutral elements.
          cssss_list <- segMap2D "cssss" segthd_lvl ResultPrivate (ty, tx) $ \_ -> do
            css_init <- scratch "css_init" (elemType red_t) [ry, rx]
            css <- forLoop ry [css_init] $ \i [css_merge] -> do
              css' <- forLoop rx [css_merge] $ \j [css_merge'] -> do
                css'' <- update' "css" css_merge' [i, j] red_ne
                resultBodyM [Var css'']
              resultBodyM [Var css']
            return [Var css]
          let [cssss] = cssss_list

          a_loc_init <- scratch "A_loc" map_t1 [a_loc_sz]
          b_loc_init <- scratch "B_loc" map_t2 [b_loc_sz]

          let kkLoopBody kk0 (thd_res_merge, a_loc_init', b_loc_init') epilogue = do
               kk <- letExp "kk" =<< toExp (LeafExp kk0 int32 * primFromSe tk)
               a_loc <- forLoop ry [a_loc_init'] $ \i0 [a_loc_merge] -> do
                 loop_a_loc <- forLoop tk_div_tx [a_loc_merge] $ \k0 [a_loc_merge'] -> do

                   scatter_a_loc <- segScatter2D "A_glb2loc" a_loc_sz a_loc_merge'
                                      segthd_lvl (ty, tx) $ \(thd_y, thd_x) -> do

                       k <- letExp "k" =<< toExp (LeafExp thd_x int32 +
                              LeafExp k0 int32 * primFromSe tx)
                       i <- letExp "i" =<< toExp (LeafExp thd_y int32 +
                              LeafExp i0 int32 * primFromSe ty)

                       letBindNames_ [gtid_y] =<< toExp (LeafExp iii int32 + LeafExp i int32)
                       a_col_idx <- letExp "A_col_idx" =<< toExp (LeafExp kk int32 + LeafExp k int32)

                       a_elem <- letSubExp "A_elem" =<<
                                eIf (toExp $ LeafExp gtid_y int32 .<. primFromSe height_A .&&.
                                             if epilogue then
                                               LeafExp a_col_idx int32 .<. primFromSe common_dim
                                             else true)
                                    (do addStm load_A
                                        res <- index "A_elem" inp_A [a_col_idx]
                                        resultBodyM [Var res])
                                    (eBody [eBlank $ Prim map_t1])
                       a_loc_ind <- letSubExp "a_loc_ind" =<<
                                eIf (toExp $ LeafExp k int32 .<. primFromSe tk)
                                    (toExp (LeafExp k int32 + LeafExp i int32 * primFromSe tk)
                                      >>= letTupExp' "loc_fi" >>= resultBodyM)
                                    (eBody [pure $ BasicOp $ SubExp $ intConst Int32 (-1)])
                       return (a_elem, a_loc_ind)
                   resultBodyM $ map Var scatter_a_loc
                 resultBodyM [Var loop_a_loc]

               -- copy B from global to shared memory
               b_loc <- forLoop tk_div_ty [b_loc_init'] $ \k0 [b_loc_merge] -> do
                 loop_b_loc <- forLoop rx [b_loc_merge] $ \j0 [b_loc_merge'] -> do
                   scatter_b_loc <- segScatter2D "B_glb2loc" b_loc_sz b_loc_merge'
                         segthd_lvl (ty, tx) $ \(thd_y, thd_x) -> do

                     k <- letExp "k" =<< toExp (LeafExp thd_y int32 +
                            LeafExp k0 int32 * primFromSe ty)
                     j <- letExp "j" =<< toExp (LeafExp thd_x int32 +
                            LeafExp j0 int32 * primFromSe tx)
```

```haskell

                letBindNames_ [gtid_x]  =<< toExp (LeafExp jjj int32 + LeafExp j int32)
                b_row_idx <- letExp "B_row_idx" =<< toExp (LeafExp kk int32 + LeafExp k int32)

                b_elem <- letSubExp "B_elem" =<<
                           eIf (toExp $ LeafExp gtid_x int32 .<. primFromSe width_B .&&.
                                        if epilogue then
                                          LeafExp b_row_idx int32 .<. primFromSe common_dim
                                        else true)
                               (do addStm load_B
                                   res <- index "B_elem" inp_B [b_row_idx]
                                   resultBodyM [Var res])
                               (eBody [eBlank $ Prim map_t2])

                b_loc_ind <- letSubExp "b_loc_ind" =<<
                          eIf (toExp $ LeafExp k int32 .<. primFromSe tk)
                              (toExp (LeafExp j int32 + LeafExp k int32 * primFromSe tx_rx)
                                 >>= letTupExp' "loc_fi" >>= resultBodyM)
                              (eBody [pure $ BasicOp $ SubExp $ intConst Int32 (-1)])
                return (b_elem, b_loc_ind)
              resultBodyM $ map Var scatter_b_loc
            resultBodyM [Var loop_b_loc]

          -- inner loop updating this thread's accumulator (loop k in mmm_kernels).
          thd_acc <- forLoop tk [thd_res_merge] $ \k [acc_merge] -> do
            resultBodyM =<< letTupExp' "foo" =<<
              eIf (toExp $ if epilogue then LeafExp kk int32 + LeafExp k int32
                                              .<. primFromSe common_dim
                             else true) -- if in prologue, always compute redomap.
                  (do reg_mem <- segMap2D "reg_mem" segthd_lvl
                                   ResultPrivate (ty, tx) $ \(ltid_y, ltid_x) -> do
                        asss_init <- scratch "asss_init" map_t1 [ry]
                        bsss_init <- scratch "bsss_init" map_t2 [rx]

                        asss <- forLoop ry [asss_init] $ \i [asss_merge] -> do

                          a_loc_ind <- letExp "a_loc_ind" =<< toExp (LeafExp k int32 +
                                          (LeafExp ltid_y int32 * primFromSe ry +
                                           LeafExp i int32) * primFromSe tk)

                          asss <- index "A_loc_elem" a_loc [a_loc_ind]
                                    >>= update "asss" asss_merge [i]
                          resultBodyM [Var asss]

                        bsss <- forLoop rx [bsss_init] $ \j [bsss_merge] -> do

                          b_loc_ind <- letExp "b_loc_ind" =<< toExp (LeafExp j int32 +
                                          LeafExp k int32 * primFromSe tx_rx +
                                          LeafExp ltid_x int32 * primFromSe rx)

                          bsss <- index "B_loc_elem" b_loc [b_loc_ind]
                                    >>= update "bsss" bsss_merge [j]
                          resultBodyM [Var bsss]
                        return $ map Var [asss, bsss]

                      let [asss, bsss] = reg_mem

                      -- the actual redomap.
                      redomap_res <- segMap2D "redomap_res" segthd_lvl
                                       ResultPrivate (ty, tx) $ \(ltid_y, ltid_x) -> do

                        as <- index "as" asss [ltid_y, ltid_x]
                        bs <- index "bs" bsss [ltid_y, ltid_x]
                        css_init <- index "css_init" acc_merge [ltid_y, ltid_x]

```

```
                     css <- forLoop ry [css_init] $ \i [css_merge] -> do
                       css <- forLoop rx [css_merge] $ \j [css_merge'] -> do
                         resultBodyM =<< letTupExp' "foo" =<<
                           eIf ( toExp $ LeafExp iii int32 + LeafExp i int32 +
                                           primFromSe ry * LeafExp ltid_y int32
                                           .<. primFromSe height_A .&&.
                                         LeafExp jjj int32 + LeafExp j int32 +
                                           primFromSe rx * LeafExp ltid_x int32
                                           .<. primFromSe width_B
                               )

                               ( do a <- index "a" as [i]
                                    b <- index "b" bs [j]
                                    c <- index "c" css_merge' [i, j]

                                    map_res  <- newVName "map_res"
                                    map_lam' <- renameLambda map_lam
                                    red_lam' <- renameLambda red_lam

                                    addStms $ rebindLambda map_lam' [a, b] map_res
                                           <> rebindLambda red_lam' [c, map_res] c

                                    css <- update "css" css_merge' [i, j] c

                                    resultBodyM [Var css])
                               ( resultBodyM [Var css_merge'] )
                     return [Var css]

                   resultBodyM $ map Var redomap_res
               )
               (resultBodyM [Var acc_merge])
       return [thd_acc, a_loc, b_loc]

  -- build prologue.
  full_tiles <- letExp "full_tiles" $ BasicOp $ BinOp (SQuot Int32) common_dim tk
  prologue_res_list <-
    forLoop' (Var full_tiles) [cssss, a_loc_init, b_loc_init] $
    \kk0 [thd_res_merge, a_loc_merge, b_loc_merge] -> do

    process_full_tiles <-
      kkLoopBody kk0 (thd_res_merge, a_loc_merge, b_loc_merge) False

    resultBodyM $ map Var process_full_tiles

  let prologue_res : a_loc_reuse : b_loc_reuse : _ = prologue_res_list

  -- build epilogue.
  epilogue_res_list <- kkLoopBody full_tiles (prologue_res, a_loc_reuse, b_loc_reuse) True

  let epilogue_res : _ = redomap_res_list

  -- TODO: to support gemm and other programs with non-empty code2 and/or
  -- TODO: code2', this should be implemented here with something like:
  -- TODO: segmap (ltid_y < ty, ltid_x < tx) {
  -- TODO:   for i < ry do
  -- TODO:     for j < rx do
  -- TODO:       addStms code2 <> code2'
  -- TODO:       final_res <- some function of epilogue_res
  -- TODO:       return final_res

  -- TODO: RegTileReturns is still missing boundary checks.
  return [RegTileReturns [(height_A, ty, ry), (width_B, tx, rx)] epilogue_res]

let level' = SegGroup (Count grid_size) (Count group_size) SegNoVirt
    space' = SegSpace gid_flat [(gid_y, gridDim_y), (gid_x, gridDim_x)]
```

```haskell
            kbody' = KernelBody () stms_seggroup ret_seggroup
        return $ Let pat aux $ Op $ SegOp $ SegMap level' space' ts kbody'

      return $ Just (host_stms, new_kernel)

mm_BlkRegTiling _ = do return Nothing

primFromSe :: SubExp -> PrimExp VName
primFromSe se = primExpFromSubExp int32 se

ceilDiv :: MonadBinder m => SubExp -> SubExp -> m (Exp (Lore m))
ceilDiv x y = eDivRoundingUp Int32 (eSubExp x) (eSubExp y)

scratch :: MonadBinder m => String -> PrimType -> [SubExp] -> m VName
scratch se_name t shape = letExp se_name $ BasicOp $ Scratch t shape

-- index an array with indices given in outer_indices; any inner
-- dims of arr not indexed by outer_indices are sliced entirely
index :: MonadBinder m => String -> VName -> [VName] -> m VName
index se_desc arr outer_indices = do
  arr_t <- lookupType arr
  let shape = arrayShape arr_t

  let inner_dims = shapeDims $ stripDims (length outer_indices) shape
  let inner_slices = map (\inner_dim -> DimSlice  (intConst Int32 0)
                                        inner_dim (intConst Int32 1)) inner_dims

  let indices = map (DimFix . Var) outer_indices ++ inner_slices
  letExp se_desc $ BasicOp $ Index arr indices

update :: MonadBinder m => String -> VName -> [VName] -> VName -> m VName
update se_desc arr indices new_elem = update' se_desc arr indices (Var new_elem)

update' :: MonadBinder m => String -> VName -> [VName] -> SubExp -> m VName
update' se_desc arr indices new_elem =
  letExp se_desc $ BasicOp $ Update arr (map (DimFix . Var) indices) new_elem

forLoop' :: SubExp            -- loop var
        -> [VName]            -- loop inits
        -> (VName -> [VName]  -- (loop var -> loop inits -> loop body)
             -> Binder Kernels (Body Kernels))
        -> Binder Kernels [VName]
forLoop' i_bound merge body = do
  i <- newVName "i"     -- could give this as arg to the function

  let loop_form = ForLoop i Int32 i_bound []

  merge_ts   <- mapM lookupType merge
  loop_inits <- mapM (\merge_t -> newParam "merge" $ toDecl merge_t Unique) merge_ts

  loop_body <- runBodyBinder $ inScopeOf loop_form $
    localScope (scopeOfFParams loop_inits) $ body i $ map paramName loop_inits

  letTupExp "loop" $ DoLoop [] (zip loop_inits $ map Var merge)
                          loop_form loop_body

forLoop :: SubExp -> [VName] -> (VName -> [VName] -> Binder Kernels (Body Kernels))
        -> Binder Kernels VName
forLoop i_bound merge body = do
  res_list <- forLoop' i_bound merge body
  return $ head res_list


-- given a lambda "lam", a list "new_params" of new
-- parameters which should be applied to the lambda,
-- and a VName "res_name" which the lambda result should
-- be bound to:
--   creates Stms corresponding to binding of new_params,
--   lambda body, and binding of lambda result to res_name.
rebindLambda :: Lambda Kernels
             -> [VName]
             -> VName
             -> Stms Kernels
rebindLambda lam new_params res_name =
  (stmsFromList $
    map (\(ident, new_param) ->
              mkLet [] [ident] $ BasicOp $ SubExp $ Var new_param)
        $ zip idents new_params)
  <> bodyStms lam_body
  <> oneStm (mkLet [] [Ident res_name lam_ret_type] $ BasicOp $ SubExp lam_res)
  where
    (lam_params, lam_body, lam_ret_type : _) =
      (lambdaParams lam, lambdaBody lam, lambdaReturnType lam)
    idents = map (\param -> Ident (paramName param) (paramAttr param))
                 lam_params
    lam_res : _ = bodyResult lam_body

-- | Tries to identify the following pattern:
--   code followed by some Screma followed by more code.
matchCodeStreamCode :: Stms Kernels ->
                       (Stms Kernels, Maybe (Stm Kernels), Stms Kernels)
matchCodeStreamCode kstms =
  let (code1, screma, code2) = foldl (\acc stmt ->
              case (acc, stmt) of
                ((cd1, Nothing, cd2), Let _ _ (Op (OtherOp (Screma _ _ _)))) ->
                 (cd1, Just stmt, cd2)

                ((cd1, Nothing, cd2), _) ->
                 (cd1 ++ [stmt], Nothing, cd2)

                ((cd1, Just strm, cd2), _) ->
                 (cd1, Just strm, cd2 ++ [stmt])
            ) ([], Nothing, []) (stmsToList kstms)
  in (stmsFromList code1, screma, stmsFromList code2)


isTileableRedomap :: Stm Kernels
         -> Maybe (SubExp, [VName],
                   (Commutativity, Lambda Kernels, [SubExp], Lambda Kernels))
isTileableRedomap stm
  | Op (OtherOp (Screma w form arrs)) <- stmExp stm,
    Just (reds, map_lam)              <- isRedomapSOAC form,
    Reduce red_comm red_lam red_nes   <- singleReduce reds,
    all (primType . rowType . paramType) $ lambdaParams red_lam,
    all (primType . rowType . paramType) $ lambdaParams map_lam,
    lambdaReturnType map_lam == lambdaReturnType red_lam, -- No mapout arrays.
    not (null arrs),
    all primType $ lambdaReturnType map_lam,
    all (primType . paramType) $ lambdaParams map_lam =
      Just (w, arrs, (red_comm, red_lam, red_nes, map_lam))
  | otherwise =
      Nothing


-- | Checks that all streamed arrays are variant to exacly one of
--   the two innermost parallel dimensions, and conversely, for
--   each of the two innermost parallel dimensions, there is at
--   least one streamed array variant to it. The result is the
--   number of the only variant parallel dimension for each array.
isInvarTo1of2InnerDims :: Names -> SegSpace -> VarianceTable -> [VName]
```

```haskell
                          -> Maybe [Int]
isInvarTo1of2InnerDims branch_variant kspace variance arrs =
  let inner_perm0 = map varToOnly1of2InnerDims arrs
      inner_perm  = catMaybes inner_perm0
      ok1 = elem 0 inner_perm && elem 1 inner_perm
      ok2 = length inner_perm0 == length inner_perm
  in  if ok1 && ok2 then Just inner_perm else Nothing
  where varToOnly1of2InnerDims :: VName -> Maybe Int
        varToOnly1of2InnerDims arr = do
          (j, _) : (i, _) : _ <- Just $ reverse $ unSegSpace kspace
          let variant_to       = M.findWithDefault mempty arr variance
              branch_invariant = not $ nameIn j branch_variant ||
                                         nameIn i branch_variant
          if not branch_invariant then Nothing -- if i or j in branch_variant; return nothing
          else if nameIn i variant_to && not (nameIn j variant_to) then Just 0
          else if nameIn j variant_to && not (nameIn i variant_to) then Just 1
          else Nothing


varianceInStms :: VarianceTable -> Stms Kernels -> VarianceTable
varianceInStms = foldl varianceInStm

-- just in case you need the Screma being treated differently than
-- by default; previously Cosmin had to enhance it when dealing with stream.
varianceInStm :: VarianceTable -> Stm Kernels -> VarianceTable
varianceInStm v0 bnd@(Let _ _ (Op (OtherOp (Screma _ _ _))))
  | Just (_, arrs, (_, red_lam, red_nes, map_lam)) <- isTileableRedomap bnd =
    let v = defVarianceInStm v0 bnd
        red_args  = lambdaParams red_lam
        map_args  = lambdaParams map_lam
        card_red  = length red_nes
        acc_lam_f = take (card_red `quot` 2) red_args
        arr_lam_f = drop (card_red `quot` 2) red_args
        stm_lam   = (bodyStms $ lambdaBody map_lam) <> (bodyStms $ lambdaBody red_lam)

        v' = foldl' (\vacc (v_a, v_fm, v_fr_acc, v_fr_var) ->
                       let vrc   = oneName v_a <> M.findWithDefault mempty v_a vacc
                           vacc' = M.insert v_fm vrc vacc
                           vrc'  = oneName v_fm <> vrc
                       in  M.insert v_fr_acc (oneName v_fr_var <> vrc') $ M.insert v_fr_var vrc' vacc'
                     ) v $ zip4 arrs (map paramName map_args) (map paramName acc_lam_f) (map paramName arr_lam_f)
    in varianceInStms v' stm_lam
  | otherwise = defVarianceInStm v0 bnd

varianceInStm v0 bnd = defVarianceInStm v0 bnd

defVarianceInStm :: VarianceTable -> Stm Kernels -> VarianceTable
defVarianceInStm variance bnd =
  foldl' add variance $ patternNames $ stmPattern bnd
  where add variance' v = M.insert v binding_variance variance'
        look variance' v = oneName v <> M.findWithDefault mempty v variance'
        binding_variance = mconcat $ map (look variance) $ namesToList (freeIn bnd)

-- alternatively, import TileLoops?
segMap2D :: String           -- desc
         -> SegLevel         -- lvl
         -> ResultManifest   -- manifest
         -> (SubExp, SubExp) -- (dim_x, dim_y)
         -> ((VName, VName)  -- f
             -> Binder Kernels [SubExp])
         -> Binder Kernels [VName]
segMap2D desc lvl manifest (dim_x, dim_y) f = do
  ltid_x    <- newVName "ltid_x"
  ltid_y    <- newVName "ltid_y"
  ltid_flat <- newVName "ltid_flat"
  let segspace = SegSpace ltid_flat [(ltid_x, dim_x), (ltid_y, dim_y)]

  ((ts, res), stms) <- runBinder $ do
    res <- f (ltid_x, ltid_y)
    ts  <- mapM subExpType res
    return (ts, res)
  Body _ stms' res' <- renameBody $ mkBody stms res

  letTupExp desc $ Op $ SegOp $
    SegMap lvl segspace ts $ KernelBody () stms' $ map (Returns manifest) res'

segScatter2D :: String   -- desc
             -> SubExp   -- arr_size
             -> VName
             -> SegLevel -- lvl
             -> (SubExp, SubExp) -- (dim_y, dim_x)
             -> ((VName, VName) -> Binder Kernels (SubExp, SubExp)) -- f
             -> Binder Kernels [VName]
segScatter2D desc arr_size updt_arr lvl (dim_x, dim_y) f = do
  ltid_x <- newVName "ltid_x"
  ltid_y <- newVName "ltid_y"
  ltid_flat <- newVName "ltid_flat"
  let segspace = SegSpace ltid_flat [(ltid_x, dim_x), (ltid_y, dim_y)]

  ((t_v, res_v, res_i), stms) <- runBinder $ do
    (res_v, res_i) <- f (ltid_x, ltid_y)
    t_v <- subExpType res_v
    return (t_v, res_v, res_i)

  Body _ stms' res' <- renameBody $ mkBody stms [res_i, res_v]
  let [res_i', res_v'] = res'
  let ret  = WriteReturns [arr_size] updt_arr [([res_i'], res_v')]
  let body = KernelBody () stms' [ret]

  letTupExp desc $ Op $ SegOp $ SegMap lvl segspace [t_v] body

processIndirections :: Names   -- input arrays to redomap
                    -> Names   -- variables on which the result of redomap depends on.
                    -> Maybe (Stms Kernels, M.Map VName (VName, Slice SubExp, Type))
                    -> Stm Kernels
                    -> Maybe (Stms Kernels, M.Map VName (VName, Slice SubExp, Type))
processIndirections arrs _ acc (Let patt _ (BasicOp (Index arr_nm slc)))
  | Just (ss, tab) <- acc,
    [p] <- patternValueElements patt,
    (p_nm, p_tp) <- (patElemName p, patElemType p),
    nameIn p_nm arrs,
    Array _ (Shape [_]) _ <- p_tp =
      Just (ss, M.insert p_nm (arr_nm, slc, p_tp) tab)

processIndirections _ res_red_var acc stm'@(Let patt _ _)
  | Just (ss, tab) <- acc,
    ps <- patternValueElements patt,
    all (\p -> not (nameIn (patElemName p) res_red_var)) ps =
      Just (ss Seq.|> stm', tab)
  | otherwise = Nothing
```