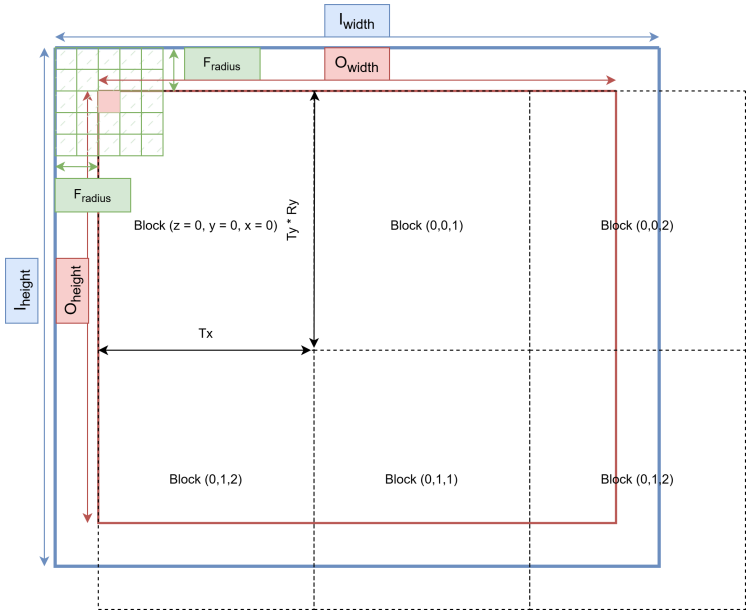
UNIVERSITY OF COPENHAGEN
DEPARTMENT OF COMPUTER SCIENCE




# Masters Thesis

Emil Vilandt Rasmussen - `nbz406`
Jóhann Utne - `dvl176`

Optimizing Convolutions for GPU Execution in Futhark

June 2, 2025

Advisor: Cosmin Eugen Oancea

**Abstract**

2D discrete convolutions are the fundamental part of convolutional neural networks (CNN), and are responsible for 90% of the runtime. Due to the parallel structure of the convolution operation, it is suitable for GPU execution. Furthermore, due to high data re-use, it is a great candidate for locality of reference optimizations. Futhark is a purely functional language, with a compiler that generates efficient block-register tiling for GEMM programs; however, it does not transform convolution operations.

This thesis successfully explores such locality optimizations through block-register tiling of the convolution operation in CUDA. The CUDA versions were then used as a template for the transformations in the Futhark compiler. However, only a handwritten kernel transformation was achieved.

The CUDA optimizations and the handwritten kernel were evaluated through benchmarking and performed well. The CUDA optimizations achieved between 89% and 96.3% of the peak TFLOPs of the GPU, and the transformed kernel shows great promise, achieving a $\times 3$ speedup compared to the non-transformed Futhark convolution.

# Contents

# 1 Introduction

The convolution is a mathematical operation that combines two functions in the continuous or discrete domain and produces a separate function by taking the integral of the product of the two functions when one function is reflected about the y-axis and shifted:

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

Convolutions have applications in many domains, one of which is image processing, and is a fundamental operation in Convolutional Neural Networks (CNNs), where the convolution is applied in what is called the convolutional layer. The convolution in CNNs usually takes as input a discrete 2-dimensional matrix $I(x, y)$ and filter kernel $F$ and produces an output matrix, $O(x, y)$ using a 2-dimensional discrete convolution:

$$O(x, y) = \sum_{i=1}^{F_{width}} \sum_{j=1}^{F_{height}} F(i, j)I(x - i, y - j)$$

$I$ is usually an image with possibly multiple channels, and several filters may be applied to each image. The convolutional layer takes up to 90% of the runtime of CNNs [21], making it a prime candidate for optimization.

The discrete convolution is inherently parallel as it can process the elements of the output matrix independently. It lends itself well to implementation in languages that target highly-parallel hardware, such as general-purpose graphics processing units (GPGPUs). CUDA [16] and Futhark [8] are the languages focused on in this paper.

## 1.1 Project Overview

The goal of this project was to implement an optimizing pass in the Futhark compiler that identifies convolution-like expressions in the source language and produces efficient GPU code through a series of transformations on the Futhark intermediate representation (IR), similar to the way Futhark recognizes and optimizes matrix-matrix multiplications [4].

We wanted a theoretical and practical baseline of the performance we could expect from the optimized Futhark code. To obtain this, we developed several implementations using CUDA, starting with a naive implementation and performing several optimizing transformations. In doing so, we discovered a simple and – to our knowledge – novel approach to convolutions on the GPU and achieved a highly performant implementation, reaching 90-96% of the theoretical FLOPS of an A100 GPU.

Due to time constraints, we did not finish the optimizing pass in the Futhark compiler. However, by applying a subset of the transformations from the CUDA versions manually to the Futhark IR, we achieved roughly 50% of the theoretical FLOPS on the A100, showing that applying these steps in the compiler pass is possible and may have significant performance benefits.

### 1.1.1 Report Structure

This report begins by presenting the convolution algorithm in its base form and defining our notation in section 2.1. Section 2.2 discusses existing implementations of convolutions on GPUs. We also provide a background section on the CUDA programming model in 2.3, which provides necessary context for the choices made in the CUDA implementation.

We show how we transformed the base convolution algorithm to a more parallel structure in section 3.1, followed by the concrete implementation steps in CUDA in 3.6. We also provide an overview of the work we did concerning the Futhark version, although this work is incomplete due to time constraints. The evaluation section shows the performance of the CUDA implementations and compares the effects of the applied transformations.

# 2 Background and Related Work

## 2.1 Convolutions

### 2.1.1 Discrete Convolutions

Since CNNs operate on 2D images, a discrete 2D convolution is applied, which has the following formula.

$$(f * g)[m, n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} f[i, j]g[n - i, n - j]$$

In most CNNs, the convolution step is done using a cross-correlation operation instead of convolution, which has the same formula as convolution, with the filter input being flipped. The formula of cross-correlation is

$$(f \otimes g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[m - n]$$

However, since the filter weights are updated across iterations in CNNs, the choice of doing convolution and cross-correlation is arbitrary. This results in most CNNs being implemented using cross-correlation, as the simpler implementation of cross-correlation does not flip the filter, yet it is still considered a convolution operation. We will adopt the same convention and refer to our implementation as a convolution despite implementing the operation as cross-correlation.

### 2.1.2 CNN Convolution Step

Since Convolutional Neural Networks handle images, the inputs are 2D matrices, and therefore, we focus solely on the 2-dimensional convolution. Each

input image consists of multiple channels that convey the information of the picture. For example, color images can have four channels to represent the red, green, blue, and alpha values of the picture, referred to as feature maps [24] [10]. Thus, the shape of an input image tensor $I$ will be.

$$I_c \times I_{height} \times I_{width}$$

Where $I_c$ is the number of input channels for the picture. Each channel requires a separate filter, so for a filter with size $F_{height} \times F_{width}$, the shape of the filter's tensor $F$ becomes

$$I_c \times F_{height} \times F_{width}$$

To compute the convolution step in a CNN, a convolution operation is performed for each image and filter pair, and the results are summed over, resulting in one output channel.

$$\sum_{i=0}^{I_c-1} conv2d(I_i, F_i)$$

To handle multiple output channels at each convolution step of a CNN, we need a set of filters with shape $I_c \times F_{width} \times F_{height}$ for every output channel. We can then describe the shape of the filters needed by denoting the number of output channels as $O_c$.

$$O_c \times I_c \times F_{height} \times F_{width}$$

Pseudocode for the convolution step is shown in Figure 1 in a C-like language.

```
convolution_step(
    inputs[I_c, I_width, I_Iheight],
    filters[O_c, I_c, F_width, F_height],
    outputs[O_c, O_width, O_Iheight]
  ) :
  for (o = 0; o < O_c; o++):
    for (y = 0; y < O_height; y++)
      for (x = 0; x < O_width; x++)
        tmp = 0
        for (n= 0; n < I_c; n++)
          for (i = 0; i<F_height; i++):
            for (j = 0; j<F_width; j++)
              tmp += inputs[n, y+i, x+j] *
                  filters[n,o,i,j]
        outputs[o, y, x] = tmp
```

Figure 1: Convolution step

,

Where $O_{height}$ and $O_{width}$ depend on the input and filter sizes, and what convolution "mode" is performed. The output shape and modes are defined in section 2.1.3

### 2.1.3 Convolution Modes

For convolutions, there are different "modes" that the convolution can do, that alter the shape of the output. Depending on how the convolution should be used, a different convolution method is necessary. Some of the most common modes are: full- , valid- and same mode [14] The modes are implemented by padding or trimming the inputs and outputs to fit the shape needed. The modes do not alter how a convolution is calculated, only what is retained in the output.

To describe the modes, we will refer to the input size as $I_{height}$ and $I_{width}$, the filter sizes as $F_{height}$ and $F_{width}$ and the output size as $O_{height}$ and $O_{width}$.

**2.1.3.1 Full Convolution Mode** For full convolution mode, the input is padded in all dimensions by $F_{height} - 1$ and $F_{width} - 1$ resulting in $I^{padded}_{width} = I_{width} + F_{width} - 1$ and vice versa for the height. This will result in an output of shape:

$$O_{height} = I^{padded}_{height}$$

$$O_{width} = I_{width}^{padded}$$

**2.1.3.2 Valid Convolution Mode** In the valid convolution mode, no padding is added to the input. The output shape will result is

$$O_{height} = I_{height} - F_{height} + 1$$

$$O_{width} = I_{width} - F_{width} + 1$$

Resulting in a smaller output than the input. This project will primarily focus on the valid convolution mode, which introduces the least amount of overhead. This mode is what we primarily deal with in this thesis.

**2.1.3.3 Same Convolution Mode** In the same convolution mode, padding is added to ensure the input and output sizes remain the same. The padding can be added to both dimensions equally or be skewed to be overloaded at the beginning or end. In the context of CNNs, *Same* convolution mode enables the output to be directly used as the input again. The implementation is the same, except for the padding and some indexing offsets.

## 2.2 Existing Implementations

Existing implementations of convolutions on GPUs can roughly be divided into those that transform the data first and those that don't.

**Direct convolution**: This naive implementation performs no transformation of the data and is our starting point for the implementation (see Figure 1). The naive implementation performs relatively poorly due to latencies from global memory accesses which will be discussed later. However, A CPU implementation from [25] shows that when implemented with some thought given to the memory layout of the input image and filters, a direct convolution can reach up to 89% of the peak theoretical floating point operations per second (FLOPS) on ARM architectures with zero memory overhead, rivaling other existing CPU implementations. They do not implement the operation on the GPU, but they believe it should be possible. We show that it is and attain up to 96% of the theoretical FLOPS on the A100 GPU.

**Im2col** [2] produces the output image coordinate by coordinate by overlaying the filter over the input image producing a sort of window of the input and filters to be processed. It then flattens these windows on which it performs a GEMM operation, transforming the resulting matrix to extract the output. This method is commonly used in existing deep learning frameworks and often acts as a performance benchmark. A downside to the algorithm is that it requires preprocessing of the input and copying it into a separate buffer. Many of the input images are duplicated in the transformed data, and thus, the algorithm has large memory requirements. However since GEMM operations are optimized on tensor cores, it's still very useful, if that hardware is available. For GPGPUs, the memory overhead may not be worth the ease of using existing GEMM implementations.

**cuDNN** [5] is a GPU deep learning library developed by NVIDIA, supporting various implementations for convolutions, including the Fast Fourier Transform (FFT), GEMM-based, implicit GEMM, Winograd, and direct methods. cuDNN supports selecting the appropriate algorithm depending on the dimensions of the input and filters, but shows high memory usage in its GEMM version, because it uses the im2col, and in its FFT version, because it uses complex numbers [11]. The Winograd is efficient for small convolution kernels.

**Im2win** [11] presents a variation of the im2col method that also rearranges the input data but with a lower memory footprint from a more compact data arrangement and higher performance than im2col. It shows comparable performance with the implicit GEMM-based convolution, the FFT convolution, and the Winograd convolution in cuDNN with a much lower memory footprint.

**Separable Filters** [20]. Some optimizations can be applied when the filter of a convolution is separable, meaning it can be written as the product of two or more simple filters (1-dimensional). Each simple filter can be applied individually, removing the need to process large windows at a time. However, this is not a general optimization that can be applied to all convolutions.

Our implementation is a variation of the direct convolution with several optimizations applied. It has the benefit of being a direct convolution since it uses no additional memory other than the additional padding for the

allocations, which is a small constant factor.

## 2.3 CUDA Overview

To write efficient CUDA code, a thorough understanding of the NVIDIA CUDA Computing Platform and Programming Model is necessary. The following sections cover the level of understanding necessary for implementing the convolution in CUDA. The information is applicable across most newer compute capabilities. We did not use any of the fancier functionality of newer compute capabilities[1].

### 2.3.1 Programming Model

CUDA allows the user to define functions called *kernels* that execute on the GPU. The kernel specifies, the computation that each of the $N$ available threads on the GPU performs as opposed to the single threads in regular CPU functions [17]. This allows large workloads to be processed partially by each thread, resulting in a larger computation happening in parallel. In order for the user to be able to map all the threads on the GPU to a given part of the workload, the threads can be uniquely identified in the following way:

The threads are divided into blocks of threads with `blockDim` threads in each block. For a given thread, its block index is given by `blockIdx`. Within a block, each thread gets assigned a unique `threadIdx`. These indices and dimensions are 3-dimensional vectors $(x, y, z)$ for the convenience of working with vectors, matrices, or volumes. When calling a kernel, the user specifies how many threads are in a block in each dimension and how many blocks to spawn in each dimension, which defines a "grid" of blocks.

When processing one-dimensional workloads, one can simply ignore the $Y$ and $Z$ dimensions, and likewise with $Z$ when processing two-dimensional workloads. The thread ids are laid in such a way that the $X$-dimension is the innermost, followed by $Y$ and $Z$. The unique ID of a thread is `threadIdx.x + blockDim.x * threadIdx.y + blockDim.x * blockDim.y * threadIdx.z`. In one-dimensional workloads, `threadIdx.y` and `threadIdx.z` are always 0. The unique ID of a thread corresponds to its physical location on the GPU,

[1] "The compute capability of a [GPU] ... identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU." – [17]

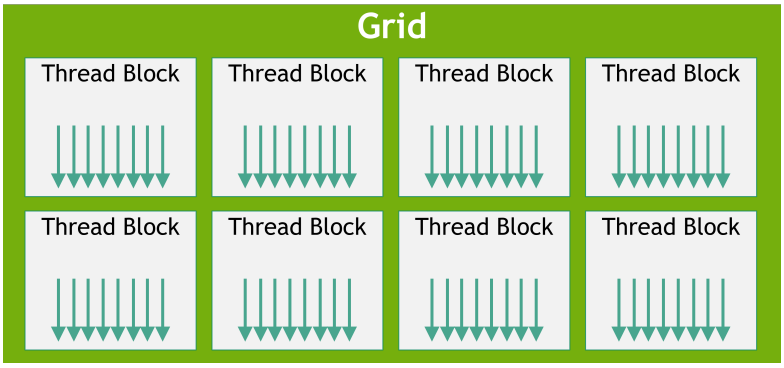



Figure 2: CUDA Thread hierarchy [17]

which will become relevant later. In Figure 2 we can see how threads are arranged in a grid of blocks, each of which has several threads.

There is a maximum number of threads that can fit in a block. Each GPU has several streaming multiprocessors (SM), which can process a number of blocks at a time. Since no specific order of execution on the SM is guaranteed by the thread execution model, thread blocks are required to be able to execute independently.

### 2.3.2 Memory Hierarchy

CUDA uses a "heterogeneous" programming model wherein the CUDA threads execute on a separate *device* (the GPU) as a coprocessor to the *host* program running on the CPU. Each has its own available memory but allocation of GPU memory and execution of GPU kernels is orchestrated by the CPU by using CUDA's allocation APIs and invoking kernels.

The CUDA threads on the device have access to a number of different levels of memory. Each thread has a private set of registers and private memory that only it can access. Private memory is used mostly when the number of variables used by a thread cannot be satisfied by the registers and must spill to private memory. The threads in a block have access to a shared region of memory aptly named *shared memory* that is private across blocks, but shared among threads in a block. Lastly, all threads across blocks have access to *global memory*. Global memory accesses can be cached in L1 and L2 caches. Shared memory accesses are very fast compared to global, and on modern GPUs the same on-chip memory is used for both L1 and shared

memory. All global memory accesses are cached in L2.

Since the GPU's memory is separate from the CPU's, the user needs to copy memory between them. Copying from the CPU onto the GPU copies it into global memory. Device memory can be allocated as linear memory using `cudaMalloc`, similar to a regular general-purpose allocator in C. With these basic constructs in place, we can begin programming for the GPU.

## 2.4 Performance Considerations

The performance of a given application or system using GPUs is highly dependent on whether or not the code strike an adequate balance of the following performance strategies:

1. Maximize hardware utilization
2. Maximize memory throughput
3. Maximize instruction throughput
4. Minimize memory thrashing

### 2.4.1 Maximize Hardware Utilization

There are different levels to consider when maximizing utilization: Application, device, and multiprocessor. At the application level, the expectation is that several GPU kernels are executed by launching them from the CPU code. Device operations (kernel executions, among other things) can be enqueued and executed at the appropriate time by the CUDA driver when device resources are available. If no data dependencies exist between kernels, no synchronization is needed between them and the CUDA driver is free to schedule them as it wishes, maximizing application-level utilization.

At the device level, multiple different kernels can be executed simultaneously on the same device. Since our project focuses on a single kernel implementation, we mostly consider maximizing utilization at the multiprocessor level.

**2.4.1.1 Multiprocessor Utilization** There is a maximum number of blocks that can reside on a given multiprocessor, as well as a maximum *shared memory* size and number of registers. The number of blocks launched

by a kernel invocation is controlled by the user specifying the launch bounds of the kernel. The number of registers used by a kernel depends on the variable usage of the kernel. The shared memory size is specified by the user when invoking the kernel.

Balancing shared memory size, register use, and number of blocks can increase occupancy (the number of threads residing on a multiprocessor), but higher occupancy does not always translate to higher performance. It is commonly recommended to run more threads per multiprocessor and more threads per thread block to increase occupancy, since this in theory means more workers to process units of work. However, high instruction-level parallelism and low memory latency can be achieved with fewer threads, and this is often enough to fully utilize the hardware [22]:

At the multiprocessor level, threads are divided into units of execution called warps (usually 32 threads). At every instruction issue time, a warp scheduler selects an instruction that is ready to execute. It might be an instruction for the same warp if the instruction is independent of a currently executing instruction (exploiting instruction-level parallelism) or it might be an instruction of another warp (exploiting thread-level parallelism). It then issues the instruction to the active threads of the warp: those that are participating in the current instruction. Branches may cause individual threads within a warp to execute a different set of instructions. Those not participating in the current instruction are said to be inactive.

Full utilization is achieved when all warp schedulers always have an instruction to issue for some warp during the latency of another issued instruction. When throughput is maximized like this, it hides the latency of instructions, maximizing utilization. If all warp schedulers always have an instruction to issue, it doesn't matter if the occupancy is low.

The most common reason that an instruction is not ready to execute is if its operands are not ready yet. If the operands all reside in registers already, throughput is dependent on the number of independent instructions, that can be scheduled to run. Often, and in our case, the largest operand waiting times come from memory fetches, often taking several hundreds of clock cycles. This is discussed in the next section. Another reason for instructions not being ready is if they are guarded by some memory fence or synchronization point. For instance, if a piece of code depends on other threads in the block completing some action, a `__syncThreads()` call needs to be inserted, causing the finished threads to wait for unfinished ones.

### 2.4.2 Maximize Memory Throughput

To maximize memory throughput, it's important to minimize the amount of memory transfers with low bandwidth. One aspect of this is reducing the amount of data transfer from the CPU to the GPU. To avoid this, you want to perform as much of a given algorithm on the GPU as possible.

Reads from and writes to device global memory also suffer from low bandwidth compared to arithmetic instruction throughput and on-chip bandwidth. The instruction throughput of reads and writes to global memory is highly dependent on the memory addresses accessed by the threads.

**2.4.2.1 Coalesced Access** Global memory is accessed via 32-, 64- or 128-byte transactions. Addresses must be naturally aligned, meaning that the address of the read must be divisible by the size of the data-type read, i.e. divisible by 4 for floats for instance. A 32-byte transaction will be generated even if only a single thread reads a single float, effectively dividing throughput by 8, as the remaining 28 bytes are unused. However when adjacent threads in a warp access adjacent elements in memory, it coalesces the accesses into one or more of these potentially 128-byte transactions, where all of the elements are used. Say, a warp of 32 threads reads 32 consecutive 4-byte floats. The warp coalesces the memory accesses into a single $32 \cdot 4 = 128$ byte transaction with zero wastage. Therefore, we want threads with consecutive `threadIdx.x`'s to read adjacent memory locations. To ensure that this is possible, the decision of how data is laid out in memory must take into account the access pattern of the algorithm.

Generally, you would want all your transactions to have the largest possible size of 128 bytes, which, for instance, would serve all 32 threads in a warp with a 4-byte float read. However, applications often need to read an amount of contiguous values that is not divisible by the warp size, causing additional and smaller-sized transactions with some values not being used. For example, an algorithm may need to read 10 contiguous 4-byte floats, resulting in a 64-byte transaction with 24 bytes being unused, and a wastage of 37.5%.

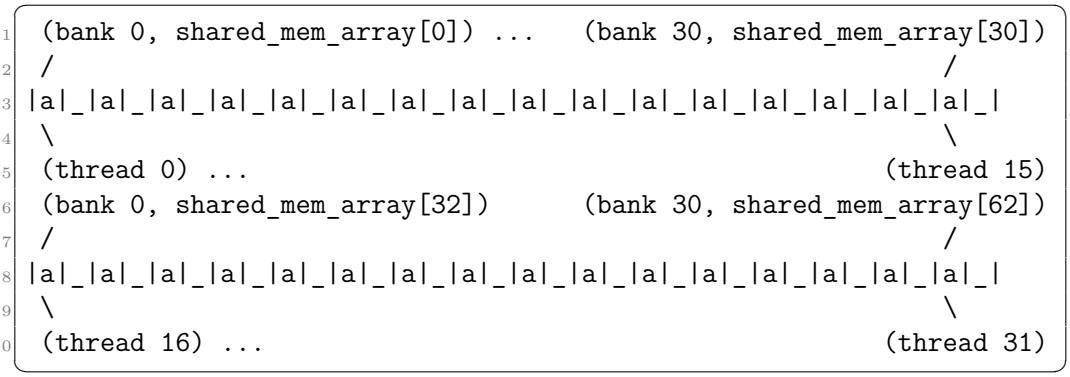
**2.4.2.2 Shared Memory** While coalesced access increases global memory throughput, it's still low compared to on-chip memory. Therefore, keeping these transfers to a minimum and reusing data by keeping it in registers or caches is crucial. One way to achieve this is through shared memory.

Shared memory can be seen as a user-managed cache local to each thread block. If threads in a block read a lot of the same data, caching it explicitly by copying it into shared memory makes sense. Shared memory is split into evenly sized memory banks consisting of $n$ 32-bit words. $n$ is compute capability dependent but 32 for 2.0 and up. Accesses to the $n$ banks can be served simultaneously, if different threads in a warp access different banks. If two or more threads in a warp try to access the same bank, the accesses must be serialized. However, if *all* threads in a warp read from the same bank, the read can be broadcast to the threads simultaneously.

Below is a diagram of an optimal shared memory access pattern. No two threads access the same bank, which is the case if threads access shared memory like this: `shared_mem_array[threadIdx.x]` `|a|` means that the bank is accessed and `|_|` means that it is not. Note that the accesses in the second line are from a different warp and will not cause a bank conflict.

```
 (bank 0, shared_mem_array[0]) ...   (bank 31, shared_mem_array[31])
 /                                                                 /
|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|
 \                                                                 \
 (warp 0, thread 0) ...                              (warp 0, thread 31)
 (bank 0, shared_mem_array[32])      (bank 31, shared_mem_array[63])
 /                                                                 /
|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|
 \                                                                 \
 (warp 1, thread 0) ...                              (warp 1, thread 31)
```

Below is a suboptimal access pattern: a 2-way bank conflict, where two threads access the same bank. This can occur if threads access shared memory like this: `shared_mem_array[threadIdx.x * 2]`. Note that threads 0 and 16 both access bank 0. The requests must be split into twice as many conflict-free requests, halving the effective bandwidth.




### 2.4.3 Maximize Instruction Throughput

Maximizing instruction throughput can be achieved by either reducing the number of instructions in the PTX[2] code, minimizing low throughput instructions or minimizing divergent thread execution within a warp due to branches.

**Reducing Number of Instructions**. Pointer aliasing can prevent the CUDA compiler from performing optimizations like instruction reordering, reusing common sub-expressions and such. This can be alleviated by marking pointers guaranteed not to point to the same location with the `__restrict__` keyword. Synchronization points via `__syncThreads()` can force some threads to idle as mentioned, so they should only be used when necessary.

**Minimizing Branching**. As mentioned in 2.4.1.1, control flow instructions can cause threads within a warp to take different execution paths. At the worst case, each thread takes a different path causing the rest of the threads to wait, effectively making execution sequential. If a conditional only differs across warps or there is no conditional at all, no divergence occurs.

Branch predication may also cause both paths of short `if`s or `switches` to be scheduled but only have their results written if the condition evaluates to true. Loops with a statically known size will often be unrolled automatically by the CUDA compiler, but the user can force the unrolling by using the `#pragma unroll` directive. In these cases, warp divergence can never occur, greatly increasing instruction throughput.

[2]"PTX defines a virtual machine and ISA [Instruction Set Architecture] for general purpose parallel thread execution. PTX programs are translated at install time to the target hardware instruction set." [18]

**Minimizing Low Throughput Instructions** can be achieved by using lower precision instructions when the higher precision is not needed or by using intrinsic functions to have more control over the generated PTX code.

Lastly, as mentioned in 2.4.1.1, instructions not waiting for operands from memory fetches or other dependent instructions are important.

### 2.4.4 Minimize Memory Thrashing

Memory thrashing can occur if an application makes a lot of memory allocations and deallocations. The program may find allocation times slowing down due to the nature of releasing memory back to the operating system. In addition, allocations are generally slow.

It is therefore important to size the allocations upfront to the problem at hand and avoid dynamically allocating when more memory is needed.

# 3 Implementation

## 3.1 Convolution Optimization Transformations

To go through the optimizations to the convolution operation on the GPU, we will look at the pseudo code for a convolution, that has $N$ input channels, $O$ output channels, and square filter matrices with size$F_{width}$,

```
for (z = 0; z < O; z ++):
  for (y = 0; y < O_height; y++)
    for (x = 0; x < O_width; x++)
      tmp = 0
      for (n= 0; n < N; n++)
        for (i = 0; i < F_width; i++):
          for (j = 0; j < F_width; j++)
            tmp += inputs[n, y+i, x+j] *
                          filters [n, z, i, j]
      outputs[z, y, x] = tmp
```

Figure 3: Pseudo code of convolution step

where *input*, *filters* and *outputs* are arrays of inputs, filters and outputs respectively. This pseudo-code maps a filter to each output channel, which is applied to each of the N inputs, and sums up the results.

We can see that this has a perfect loop-nest structure for the outer dimensions and an inner map-reduce (redomap) composition, which is the sum of the element-wise product between the slice of the input and the filter.

## 3.2 Arithmetic Intensity

From the pseudo code 3, we can obtain an overview of the number of floating-point operations and memory accesses required, which can be used to estimate the arithmetic intensity. The arithmetic intensity can then be used for a Roofline Model performance estimate [23], which can be used to determine whether our implementation will be compute or memory-bound.

The number of floating point operations, $FP$, is determined solely by the redomap in the innermost loop. The only FPs are a multiplication followed by an add, so 2 FP operations in the innermost loop. Therefore, the total number of FP becomes.

$$FP = O \cdot N \cdot O_{height} \cdot O_{width} \cdot (F_{width})^2 \cdot 2FP$$

We access all elements of the input and write to all elements of the output. Therefore, the amount of bytes accessed by distinct memory accesses, with all elements assumed to be floats, and thus having a size of $4B$, is

$$Bytes = (N \cdot N_{height} \cdot N_{width} + O \cdot O_{width} \cdot O_{height} + O \cdot N \cdot (F_{width})^2) \cdot 4B$$

We can simplify the equation a bit by noting that we are working on a valid convolution, and therefore, the output dimensions are upper-bounded by the input dimensions.

$$Bytes < ((N + O) \cdot N_{height} \cdot N_{width} + O \cdot N \cdot (F_{width})^2 \cdot 4B$$

We can calculate the arithmetic intensity of the convolution operation as

$$Ai = \frac{O \cdot N \cdot O_{height} \cdot O_{width} \cdot (F_{width})^2 \cdot 2FP}{((N + O) \cdot N_{height} \cdot N_{width} + O \cdot N \cdot (F_{width})^2) \cdot 4B}$$

From this, we can see that if the filter size is small, and there are few input and output channels, the arithmetic intensity becomes quite small, and we are memory-bound.

However, as the problem size grows, the amount of FP grows much faster than the number of memory accesses, so for a greater number of channels and

bigger filters, the problem becomes compute-bound. Furthermore, we can observe that the filter width contributes more to memory accesses than floating-point operations, for a constant width and height of the input. Therefore, for very high filter widths, the convolution will be memory-bound.

The Roofline model gives an estimate for the maximum performance of the algorithm on the GPU given by

$$P_{max} = \min(P_{Peak}, Ai \cdot b_{max})$$

where $P_{Peak}$ is the theoretical peak FLOPS of the GPU, $Ai$ is the arithmetic intensity, and $b_{max}$ is the maximum bandwidth of the GPU. This gives the relation that the higher the arithmetic intensity, the better FLOPS performance can be achieved by the GPU.

## 3.3 Dependency Analysis

To reason about the opportunities for GPU optimization, we perform a loop dependency analysis on the convolution implementation [19] [6] A loop dependency is defined as:

**Definition 3.1** (Loop dependence)**.** For two statements $S1$ and $S2$ in a loop nest, there is a dependence *iff* $\exists$ iterations $\vec{k}, \vec{l}$ such that

1. $\vec{k} < \vec{l}$ or $\vec{k} = \vec{l}$ and $\exists$ an execution path from S1 to S2 such that:
2. S1 accesses memory location $M$ on iteration $\vec{k}$
3. S2 accesses memory location $M$ on iteration $\vec{l}$
4. One of the accesses is write

With S1 being the source of the dependency and S1 being the sink. We are only interested in cross-iteration dependencies for determining parallel loops. To analyze the direction of the dependency, we use *Dependency-direction vectors*.

To determine the dependency direction, we analyze the dependency from the sink S1 in iteration $\vec{k}$ to the sink S2 in iteration $\vec{l}$ ($\vec{k} \leq \vec{l}$), and the dependence direction vector $\vec{D}(\vec{k}, \vec{l})$

1. $\vec{D}(\vec{k}, \vec{l})_m$ = "=" if $\vec{k}_m = \vec{l}_m$
2. $\vec{D}(\vec{k}, \vec{l})_m$ = ">" if $\vec{k}_m > \vec{l}_m$
3. $\vec{D}(\vec{k}, \vec{l})_m$ = "<" if $\vec{k}_m < \vec{l}_m$
4. $\vec{D}(\vec{k}, \vec{i})_m$ = "*" if $\vec{k}_m$ is uncomparable to $\vec{i}_m$

Where $m$ denotes the depth of the loop nest. We can then use the resulting direction vectors to form a direction matrix, which will help us determine the parallel loops.

A loop is parallel if the execution of the loop does not cause any dependencies across its iterations. Furthermore, it can also be determined by the aforementioned direction matrix, which requires all elements in a column to either be =, or if there is an outer loop on the row that carries the dependency noted with < direction.

We can quickly see that the three outer loops are parallel, since the *tmp* value is privatized, and the write to *outputs[z,y,x]* happens in iteration (x,y,z), meaning that the loops are independent of each other and can be parallelized. Therefore, all elements of the vector have direction =. The inner loops all have a RAW dependency on *tmp*, since *tmp* is written to and read from in every iteration. This results in an across-loop read-after-write (RAW) dependency and an intra-iteration write-after-read (WAR). This means that all the inner loops have the direction <, and can therefore not be parallelized, since there is no outer loop with direction <. In the pseudo code below, the dependency vectors have been noted.

```
1  for (z = 0; z < O; z ++):
2    for (y = 0; y < O_height; y++)
3      for (x = 0; x < O_width; x++)
4        tmp = 0                                  // [=,=,=]
5        for (n= 0; n < N; n++)
6          for (i = 0; i < F_width; i++):
7            for (j = 0; j < F_width; j++)
8              tmp += inputs[n, y + i, x + j] *
9                  filters [n, z, i, j]       // [=,=,=,<...]
10       outputs[z, y, x] = tmp                  // [=,=,=]
```

Figure 4: Dependency vectors

This means that we can safely parallelize the outer three loops while keeping the inner loops sequential:

```
for (z = 0; z < O; z ++):  // parallel
  for (y = 0; y < O_height; y++): // parallel
    for (x = 0; x < O_width; x++): // parallel
      tmp = 0
      for (n= 0; n < N; n++): // sequential
        for (i = 0; i < F_width; i++): // sequential
          for (j = 0; j < F_width; j++): // sequential
            tmp += inputs[n, y + i, x + j] *
                   filters [n, z, i, j]
      outputs[z, y, x] = tmp
```

Figure 5: Parallel loops

The inner loops create a perfect loop nest of redomap construction, and therefore can be resolved by privatizing results and reducing the partial values. This, however, is not explored in this paper.

## 3.4 Optimizing the Convolution Operation

To optimize the code, we can utilize temporal locality by using Block and register tiling [19].

Block and register tiling consists of strip mining outer loops, interchanging them inwards, and distributing parallel loops.

### 3.4.1 Transformation Techniques

**3.4.1.1 Loop Stripmining** Loop stripmining consists of splitting a normalized loop into two loops within a perfect loop nest. The first loop iterates with a stride $T$ and the inner loop iterates with a stride 1. This transformation is always safe, since the resulting transformation always executes the same statements, in the same order.

```
        for (int i=0; i< N; i++){
            loop body
        }

```

⇓

```
        for (int ii=0; ii < N; ii += T){
            for (int i = ii; i < min(ii+T, N); i++ {
                loop body
            }
        }
```

Figure 6: Strip-mining

**3.4.1.2 Loop Interchange** Loop interchanges consist of changing outer loops inward. Interchange is safe by Theorem 7 in [19] *iff* the permuting the direction matrix does not result in a $>$ direction as the leftmost non $=$ direction in a row.

Furthermore, Corollary 2 to Theorem 7 states that a parallel loop can always be interchanged inwards.

```
for ( i = 0; i<N; i++) { // parallel
    for ( j = 0; j < $F_{width}$; j++) {
        loop body
    }
}
```

⇓

```
for ( i = 0; j < $F_{width}$; j++) {
    for ( i = 0; i<N; i++) { // parallel
        loop body
    }
}
```

Figure 7: Loop interchange example

**3.4.1.3 Loop Distribution** Loop distribution distributes a loop across its statements. The transformation can be proved safe according to Theorem 9 in [19] if its dependency graph contains no cycles. However, it states as a

corollary that a parallel loop can always be distributed across its statements. This requires array expansion of local variables.

```
        for ( i = 0; i<N; i++) { // parallel
           tmp;
           Statement 1
           Statement 2
        }

```

```
        tmp[N];
        for ( i = 0; i<N; i++) {
           Statement 1
        }
        for ( i = 0; i<N; i++) { // parallel
           Statement 2
        }

```

Figure 8: Loop distribution example

### 3.4.2 Optimization Transformations of the Convolution Step

We block tile the three outermost parallel loops and leave the innermost loop completely sequential. To block tile, we stripmine the consecutive loops and interchange the resulting stride-1 loops inward. This is safe since the outer loops are parallel.[3] We block tile by tile sizes $T_z$, $T_y$, and $T_x$. The resulting code after block tiling and code interchanging becomes:

[3]Any loop interchange would be safe, since no $>$ dependency direction is present in the code, and can therefore not be the leftmost element in any row, in a direction matrix

```
for (zz=0; zz < O; zz += T_z):
  for (yy=0; yy < O_height; yy += T_y
    for (xx=0; xx < O_width; xx += T_x
      for (z = zz; z < min(zz + T_z, O); z++):
        for (y = yy; y < min(yy + T_y, O_width); y++):
          for (x = xx; x < min(xx + T_x, O_width); x++):
            tmp = 0
            for (n= 0; n < N; n++): // sequential
              for (i = 0; i < F_width; i++): // sequential
                for (j = 0; j < F_width; j++): // sequential
                  tmp += inputs[n, y + i, x + j] *
                          filters [n, z, i, j]
            outputs[z, y, x] = tmp
```

Figure 9: Blk-tiling

Where $T_z$, $T_y$, and $T_x$ will also denote the tile sizes of the GPU. After loop normalization and unrolling of the inner loops, this code will be mapped as the "naive" GPU version, where each thread directly maps one output value of one output channel.

To further optimize the convolution step, we can register tile the resulting implementation. Register tiling consists of strip mining one or more of the outer loops in the perfect loop nest, and then interchanging them to an innermost position and unrolling the loop, loop distributing, and array-expanding as necessary [19].

The transformation is safe if in the original program, it is safe to interchange any of the loops to the innermost position, and it is safe to distribute the loops. We have already seen that we can always interchange the loops, and since it is a parallel loop that is loop distributed, it is always safe.

The transformation improves temporal locality by storing and reusing values from registers if the original loop contains invariant data across some of the loops. In the convolution example, there is a perfect loop nest, and the filter is invariant to both the $y$ and $x$ outer loops, and the *inputs* array is invariant to the $O$ outer loop. Giving us good opportunities for improving temporal locality

We start by register-tiling the $y$ loop by a tiling size of $R_y$. This also results in further stripmining the outermost *yyloop*. We then array-expand

*tmp* and loop distribute the inner loops (*z*,*yy*,*x*,). The pseudo code becomes

```
for (zz=0; zz < 0; zz += T_z): // dim-z grid
  for (yyy=0; yyy < O_height; yy += T_y * R_y): // dim-y grid
    for (xx=0; xx < O_width; xx += T_x): // dim_x grid
      tmp[T_z][T_y][T_x][R_y];
      for (yy = yyy; yy < min(yyy + T_y * R_y, O_width); yy+= R_y): // y-Tblock
        for (z = zz; z < min(zz + T_z, O); z ++):  // z-Tblock
          for (x = xx; x < min(xx + T_x, O_width); x++): // x-Tblock
            for (y = yy; y < min(yy + R_y, O_width); y++): // unroll
              tmp[z - zz,y - yyy, x - xx, y - yy] = 0
      for (yy = yyy; yy < min(yyy + T_y * R_y, O_width); yy += R_y): // y-Tblock
        for (z = zz; z < min(zz + T_z, O); z++): // z-Tblock
          for (x = xx; x < min(xx + T_x, O_width); x++): // x-Tblock
            //conv redomap
            for (n= 0; n < N; n++)
              // stride 1 loop interchanged inwards
              for (y = yy; y < min(yy + R_y, O_width); y++): //unroll
                for (i = 0; i<F_width; i++):  // unroll
                  for (j = 0; j<F_width; j++) // unroll
                    tmp[z - zz, y - yyy, x - xx, y - yy] +=
                        inputs[n, y+i, x+j] * filters [n, z, i, j]

      // loop distribution
      for (yy = yyy; yy < min(yyy + T_y * R_y, O_width); yy+= R_y): // y-Tblock
        for (z = zz; z < min(zz + T_z, O); z ++): // z-Tblock
          for (x = xx; x < min(xx + T_x, O_width); x++): // x-Tblock
            for (y = yy; y < min(yy + R_y, O_width); y++): // unroll
              outputs[z, y, x] = tmp[z - zz, y - yyy, x - xx, y - yy]
```

Figure 10: Reg-tiling of the y dimension

When register-tiling in the y dimension, we can reuse most of the filter values, as multiple overlapping calculations exist. For the inner loops to be unrolled, we normalize the loops for implementation on the GPU. Then each thread would store $R_y$ size tile in private memory, and each thread maps $R_y$ outputs of one output channel.

We tile in the y dimension to avoid strided access of threads, such that the reads to global memory are still coalesced across threads. We can further optimize with register-tiling by tiling over the $O$ dimension, which the input was invariant to. This way we compute the update for multiple output channels, reusing the same input. We tile the outer dimension by $R_Z$. This

will require extending the register usage so that we can store immediate results for $R_y * R_Z$ values. The transformed is shown as.

```
for (zzz=0; zzz < 0; zz+= T_z * R_Z ): // dim-z grid
  for (yyy=0; yyy < O_height; yy += T_y * R_y): // dim-y grid
    for (xx=0; xx < O_width; xx += T_x): // dim_x grid
      tmp[T_z][T_y][T_x][R_y][R_Z];

      for (zz = zzz; zz < min(zzz + T_z * R_Z, O); zz += R_Z):  // z-Tblock
        for (yy = yyy; yy < min(yyy + T_y * R_y, O_width); yy+= R_y): // y-Tblock
          for (x = xx; x < min(xx + T_x, O_width); x++): // x-Tblock
            for (y = yy; y < min(yy + R_y, O_width); y++): // unroll
              for (z = zz; z < min(yy + R_Z, O); z++)
                tmp[z - zz, y - yyy, x - xx, y - yy, z - zz] = 0

      for (zz = zzz; zz < min(zzz + T_z * R_Z, O); z += R_Z): // z-Tblock
        for (yy = yyy; yy < min(yyy + T_y * R_y, O_width); yy+= R_y): // y-Tblock
          for (x = xx; x < min(xx + T_x, O_width); x++): // x-Tblock
          //conv redomap
          for (n= 0; n < N; n++)
            // stride 1 loop interchanged inwards
            for (y = yy; y < min(yy + R_y, O_width); y++): //unroll
              for (z = zz; z < min(zz + R_Z, O); z++): //unroll
                for (i = 0; i < F_width; i++):  // unroll
                  for (j = 0; j < F_width; j++) // unroll
                    tmp[z - zz, y - yyy,x - xx, y - y y, z - zz] +=
                      inputs[n, y + i, x + j] * filters [n, z, i, j]

            // loop distribution
      for (zz = zzz; zz < min(zzz + T_z * R_z, O); zz += R_Z): // z-Tblock
        for (yy = yyy; yy < min(yyy + T_y * R_y, O_width); yy += R_y): // y-Tblock
          for (x = xx; x < min(xx + T_x, O_width); x++): // x-Tblock
            for (y = yy; y < min(yy + R_y, O_width); y++): // unroll
              for (z = zz; z < min(yy + R_y, O); z + +) : // unroll
                outputs[z, y, x] = tmp[z - zz, y - yyy, x - xx, y - yy, z - zz]
```

Figure 11: Reg-tiling of 2 dimensions

We can then map the parallel loops to the GPU, with further loop normalization, inserting bound checks as necessary. The outer loops marked with grid comments will be mapped to the grid, and the inner loops with block comments will be mapped to the block. To further optimize the convolution, we will explore effectively copying the tiles of input and filter data

from global memory to shared memory. Thus, when performing the convolution operation, we save the reads from global memory, allowing for more efficient memory loads. Other optimizations, such as vector loads, will also be explored and described in detail in section 3.6.3.2.

### 3.4.3 Grid and Block dimensions

The grid dimensions are determined by the choice of the block and register tile parameters.

$$G_{dimz} = \left\lceil \frac{\#\text{out channels}}{T_z * R_z} \right\rceil \tag{1}$$

$$G_{dimy} = \left\lceil \frac{O_{height}}{T_y * R_y} \right\rceil \tag{2}$$

$$G_{dimx} = \left\lceil \frac{O_{width}}{T_x} \right\rceil \tag{3}$$

And the number of blocks must be given by

$$G_{dimz} \cdot G_{dimy} \cdot G_{dimx}$$

The block dimensions correspond directly to the tile sizes $T_z, T_y, T_x.$, meaning that the size of a block is

$$Block_{size} = T_x \cdot T_y \cdot T_z$$

CUDA also requires that $Block_{size} \leq 1024$. Furthermore, each thread will use $R_y \cdot R_z$ registers to store intermediate results, and since there is a hardware limit of a maximum of 255 registers per thread, we must further have that $R_y \cdot R_z \leq 255$.

The choice of these tiling parameters is limited by the hardware resources, such as the amount of shared memory size, the number of threads per block, and registers. When choosing the parameters, it can be challenging to do so statically, as all bounds must be respected simultaneously. Even if the bounds are respected, it can often be beneficial to decrease or increase certain tiling parameters, depending on the problem instance. For example, if working with a small set of output channels but with large dimensions, it might be better to favor tiling in the y direction.

## 3.5 Mapping to the GPU

### 3.5.1 Block-tiled Version

We can transform the sequential code to a simplified CUDA code for the three different versions based on the tiling described in the previous section. We get the following CUDA kernel for the version that was only block-tiled.

```
ConvNaive(inputs, filters, outputs):
  gidx = blockIdx.x * blockDim.x + threadIdx.x;
  gidy = blockIdx.y * blockDim.y + threadIdx.y;
  gidz = blockIdx.z * blockDim.z + threadIdx.z;
   // boundary check for loop normalization of the parallel dimensions
  if (gidz < 0 && gidy < O_height && gidx < O_width) {
  tmp = 0
  for (n = 0, n < N, n++)
    for (i = 0; i<F_width; i++)
      for (j = 0; j < F_width; j++)
        tmp += input[n, gidy+i, gidx+j] * filters[n, gidz, i, j]
  output[gidz, gidy, gidx] = tmp
}
```

Figure 12: Direct mapped threads

Here, the three strip-mined loops correspond to the block dimensions. To normalize the loops, we insert bounds checks on the thread IDs.

### 3.5.2 Register-tiling in y-Dimension

```
ConvNaive(inputs, filters, outputs):
  gidx = blockIdx.x * blockDim.x + threadIdx.x;
  gidy = blockIdx.y * blockDim.y + threadIdx.y;
  yy = gidy * R_z
  gidz = blockIdx.z * blockDim.z + threadIdx.z;

  tmp[R_y];
  // register initialization
  for (i = 0; i < R_y; i++)  // normalized loop
    if (i + yy < O_height && gidz < 0 && gidx < O_width)
      tmp[i] = 0

  // convolution redomap
  for (n = 0, n < N, n++)
    for (iy = 0; iy < R_y; iy++)
      y = iy + yy              // normalized loop
      if (y < O_height && gidz < 0 && gidx < O_width) {
        for (i = 0; i < F_width; i++)
          for (j = 0; j < F_width; j++)
            tmp[iy] += input[n, y + i, gidx + j] * filters[n, gidz, i, j]
      }
  // write to output
  for (iy = 0; iy < R_y; iy++)
    y = iy + yy              // normalized loop
    if (y < O_height && gidz < 0 && gidx < O_width)
      output[gidz, y, gidx] = tmp[iy]
```

Figure 13: Register tiling in y dimension

The result of the transformation where the y-dimension has been register-tiled, after normalization with the necessary boundary checks is shown in Figure 13.

### 3.5.3 Register-tiling in y- and z-Dimensions

The result of the final transformation, where we tile in both the y- and z-dimension is shown in Figure 14.

```
ConvNaive(inputs, filters, outputs):
  gidx = blockIdx.x * blockDim.x + threadIdx.x;
  gidy = blockIdx.y * blockDim.y + threadIdx.y;
  yy = gidy * R_z
  gidz = blockIdx.z * blockDim.z + threadIdx.z;
  zz = gidz * R_Z

  tmp[R_y][R_Z];

  for (i=0; i < R_y; i++) :
    for (j=0; j < R_Z; i++):
      if (i + yy < O_height && j + zz < 0 && gidx < O_width)
        tmp[i][j] = 0

  for (n = 0, n < N, n++)
    for (iy = 0; iy < R_y; iy++)
      y = iy + yy              // normalized loop
      for (iz = 0; iz < R_z; iz++)
        z = iz + zz
        if (y < O_height && z < 0 && gidx < O_width) {
          for (i = 0; i < F_width; i++)
            for (j = 0; j < F_width; j++)
              tmp[iy][iz] += input[n, y+i, gidx+j] * filters[n, z, i, j]
        }

  for (iy = 0; iy < R_y; iy++)
    y = iy + yy              // normalized loop
    for (iz = 0; iz < R_z; iz++)
      z = iz + zz
      if (y < O_height && z < 0 && gidx < O_width)
      output[z, y, gidx] = tmp[iy][iz]
```

Figure 14: GPU reg tiled in the y- and z-dimension.

## 3.6 CUDA Implementation

This section details the implementation of convolutions in CUDA. We start with the naive implementation and move on to more optimized versions. We assume that the dimensions of the filters are known at compile-time, as well as being square and having odd-numbered dimensions. That is, $F_{width} = F_{height} = 2F_{radius} + 1$. All versions use C++ templated arguments for $F_{radius}$, denoted `FilterRadius`, the data-type `T` and the tiling parameters `Tx, Ty,`

`Tz, Ry, Rz`.

### 3.6.1 Naive Convolution

The naive implementation corresponds to the final register-tiled version shown in Figure 14.

**3.6.1.1 Data Layout** The algorithm deals with 3 tensors: one for the input image $I$, one for the output image $O$ and one for the filters $F$. They have the shape shown below, as mentioned in section 2.1.2:

$$\begin{aligned} I &: I_c \times I_{height} \times I_{width} \\ F &: I_c \times O_c \times F_{height} \times F_{width} \\ O &: O_c \times O_{heigth} \times O_{width} \end{aligned} \tag{4}$$

We allocate three flat arrays for the respective tensors. In the naive version, we can simply allocate the input and output tensors with the following:

```
T *d_input;
T *d_output;
T *d_filters;
cudaMalloc((void**)&d_input,
    in_num_channels * in_height * in_width * sizeof(T));
cudaMalloc((void**)&d_output,
    out_num_channels * out_height * out_width * sizeof(T));
cudaMalloc((void**)&d_filters,
    in_num_channels * out_num_channels * out_height * out_width * sizeof(T));
```

Listing 1: Allocating input tensor

The memory is laid out in the format specified in Equation 4 such that the width is the innermost dimension for all tensors.

In Figure 15 is a view of the work performed by each thread block for a single input channel. Blocks are marked by dashed lines. In this example, the block-tiling in the x- and y-dimensions does not evenly divide $O_{width}$ and $O_{height}$ respectively, which can be seen by the dashed lines overflowing the red output-matrix. We check for these out-of-bounds accesses in Figure 14. The input array is overlaid in blue on the figure to show which elements are read from the input array to produce the result in the output array. An example

for a single thread and Y-tile is shown in the top left. The red square is the element produced by the kernel, and the blue grid represents the values read from the input array. The arrays do not overlap in memory, of course. The filter radius, which is 2 in this case, is marked in green.

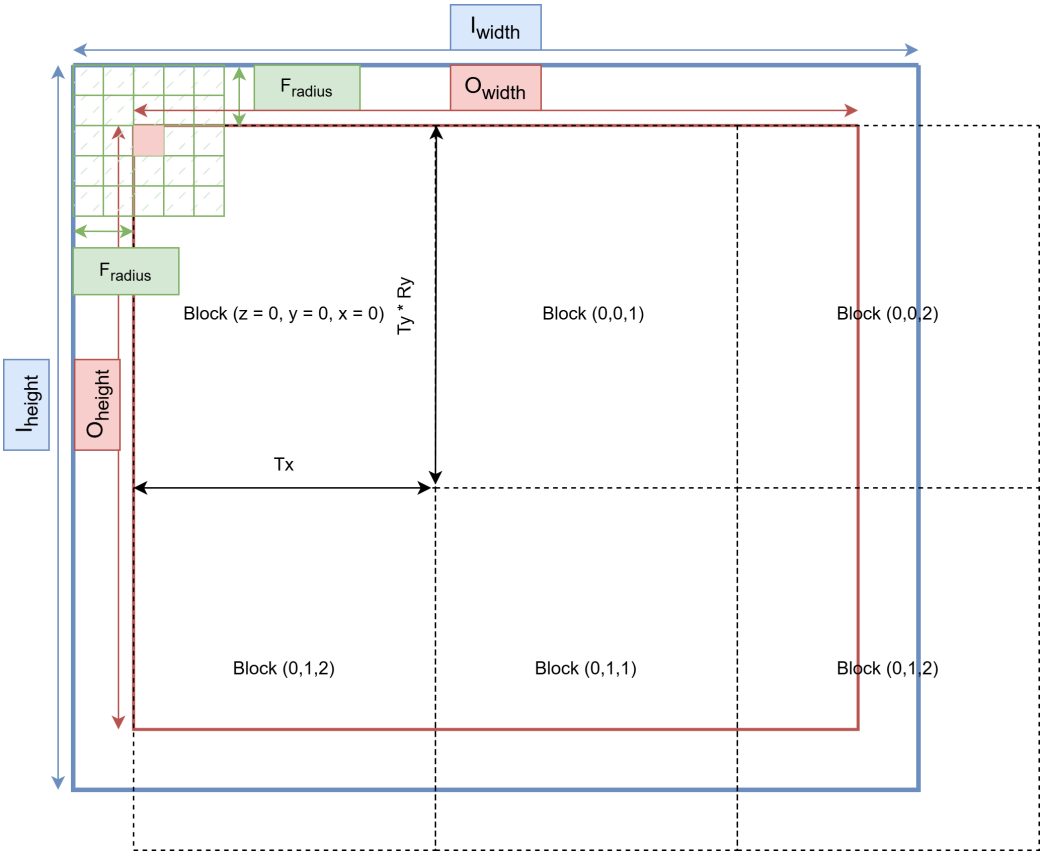



Figure 15: View of work performed by each thread-block for a single input channel. The dashed lines indicate the region of the output array operated on by each block.

#### 3.6.1.2 Spacial Locality via Coalesced Access

With this data layout, we ensure coalesced access when reading from the input and filter tensors and writing to the output tensor. Consider the following lines:

```
...
tmp[iy][iz] += input[n, y+i, gidx+j] * filters[n, z, i, j]
...
```

```
4 output[z, y, gidx] = tmp[iy][iz]
```

The `input[n, y+i, gidx+j]` read by the threads of the kernel will be coalesced, as in each iteration of the nested for loops on `i` and `j`, threads with adjacent `gidx`, i.e. the innermost thread dimension, will read adjacent elements. In the `filters[n, z, i, j]`, all threads within a warp read the same element, given that `blockDim.x * blockDim.y >= warp_size`, resulting in a broadcast. If `blockDim.x * blockDim.y < warp_size`, one warp may have two or more `z`-values, but the values are still broadcast to the threads within the warp with the same `z`. That is, we only get an amount of transactions equal to `warp_size / blockDim.x * blockDim.y`. We ensure that `blockDim.x` and `blockDim.y` are always powers of two, so `blockDim.x * blockDim.y` is as well. The `output[z, y, gidx]` writes are coalesced as adjacent threads write adjacent elements.

**3.6.1.3 Temporal Reuse and Caching** In addition to the accesses being coalesced, the algorithm also exhibits a great deal of temporal reuse, even in the naive version. As mentioned, all global memory accesses are always cached in L2, and often in L1, although we can't guarantee that without shared memory. Imagine that we have a `blockDim.x` of 32 and a `FilterRadius` of 1: a $3 \times 3$ filter. This means that the first `input[n, y+i, gidx+j]` access will result in a $32 \times 4 = 128$ byte memory transaction, which is cached at least in L2. For the block with index $(0, 0, 0)$ and `threadIdx.y = 0` and `n = 0`, which means that the elements `input[0, 0, 0:31]` are already in the cache from the previous transaction from the same block. In addition, the block with index $(0, 0, 1)$ will have loaded `input[0, 0, 32:63]`. Therefore, in the next iteration all the threads of the first block will be able to access their required memory via the cache, since they access `input[0, 0, 1:32]`, of which 31 of the values were loaded in by the same block and 1 by the next block.

One would be tempted to think that we would not benefit from shared memory, since so many values should be cached in L1. However, our shared memory version is significantly faster, especially for larger filter sizes.

### 3.6.2 Branchless Version

As mentioned in section 2.4.3, to maximize instruction throughput, we want to eliminate warp divergence, making sure threads within a warp execute the

same instructions. This can be done by eliminating branching within warps. The bounds checking in Figure 14 is one such source of warp divergence. It's a subtle point, but the only check that is needed for correctness is the check on the write to `output` on line 30. If this check passes, we know that the other checks have also passed, and correctness is guaranteed. We also found that eliminating this check did not improve performance, while eliminating the other checks did.

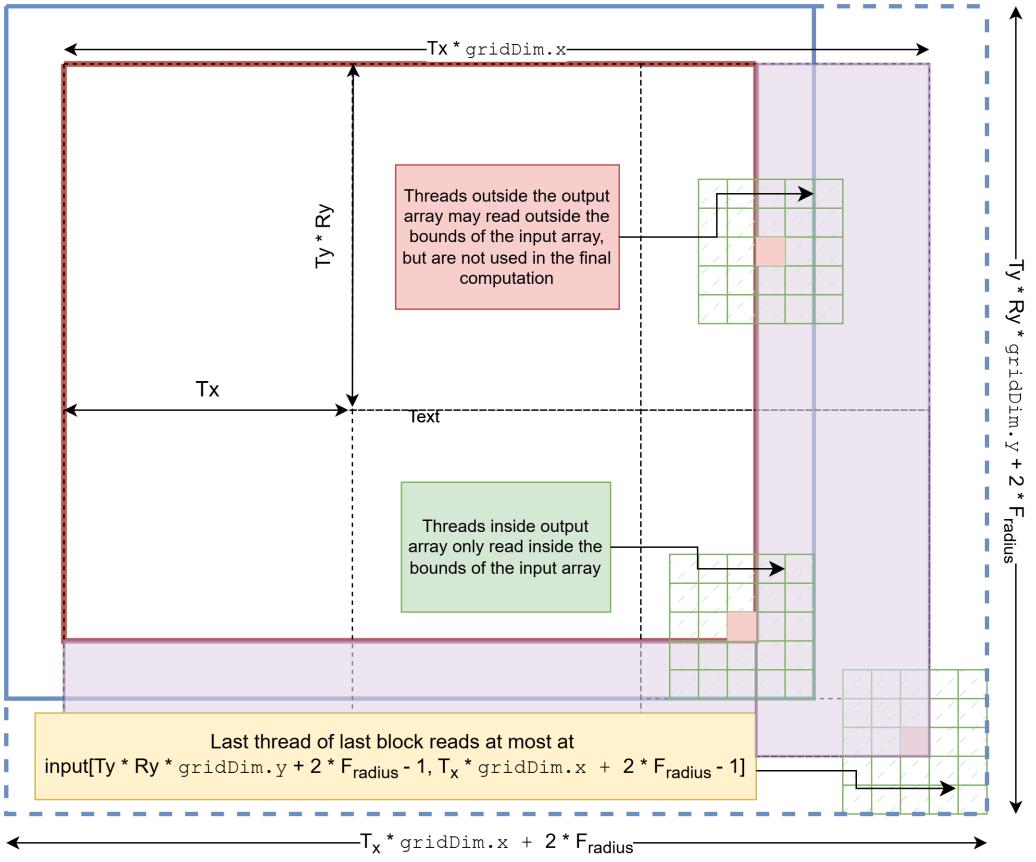



Figure 16: All threads in the purple area will access the input array out of bounds, but do not contribute to the output array.

The checks on lines 12 and 20 are not needed for correctness, but they do prevent reads outside the bounds of our allocation. Without the checks, the threads marked by the purple area in Figure 16 will at some point access outside the bounds of the input array, but do not contribute to the output

array. What the last thread of the last block accesses can be seen in the bottom right, marked by the blue grid as in Figure 15. We can calculate that the threads try to access the input array as if it had the following dimensions, also shown by the dashed blue matrix in Figure 16:

$$T_y \star R_y \star \text{gridDim.y} + 2F_{radius} \times T_x \star \text{gridDim.x} + 2F_{radius}$$

Therefore, to avoid undefined behavior, we need to pad the input tensor to fit this size, but *only* for the final input channel. This is because each input matrix is laid out in a row-major format, and each input channel comes right after the previous in the same allocation. Therefore, for all but the final input channel, the threads will simply access some element on another row for the same input channel or some element of the next input channel. In any case, the values are not used. To make the allocation, we simply calculate its size as in the naive version and align it up to the nearest multiple of the final padded input channel: $|I^{padded}| = (T_y \star R_y \star \text{gridDim.y} + 2F_{radius}) \star (T_x \star \text{gridDim.x} + 2F_{radius})$ as shown in Listing 2.

```
const u64 in_size = ROUND_UP_TO_MULTIPLE_OF(
    (u64)in_width * (u64)in_height * (u64)in_num_channels,
    (Ty * Ry * dim_y + 2 * FilterRadius) * (Tx * dim_x + 2 * FilterRadius));
```

Listing 2: Calculating input size with padding

Since the results are never written by these out-of-bounds threads, the padding can be left uninitialized.

The same can be done for the filters tensor. If #out channels is less than $T_z * R_z$, for some threads and some iterations, the threads may access out of bounds. As with the input tensor, we can alleviate this problem by padding. For the filters, we align the allocation up to be a multiple of $T_z * R_z * F_{width} * F_{width}$.

Another simple optimization is unrolling for-loops with statically known sizes, maximizing instruction parallelism. This is often done automatically by the CUDA compiler, but in some of our testing, we found that it did not always unroll all of our loops, and a `#pragma unroll` was needed. Therefore, we chose to insert it at all points where loop unrolling was possible. Loop unrolling increases instruction parallelism, but will also increase register usage, possibly leading to lower occupancy. This tradeoff is worth it in a lot of cases [22], including ours.

### 3.6.3 Shared Memory

The next version utilizes shared memory to increase the speed of access of commonly accessed elements in the algorithm. Each thread block will copy a region of the input tensor as well as some of the filters to shared memory. After the data in the shared memory has been used, thread synchronization is necessary, before the next batch is copied to avoid race conditions among threads.

**3.6.3.1 Copying Input Tensor to Shared Memory** Another direct convolution implementation shown in [20] maps each thread in the block to a single element in the input array. Since the output array is $2F_{radius}$ smaller than the input array in each dimensions, their algorithm has an apron of threads of area $2F_{radius} * T_x + 2F_{radius} * T_Y * R_y - 4F_{radius}^2$ in each block that are inactive in the computation phase. We copy to shared memory in a way that avoids these inactive threads in exchange for slightly less coalesced access.

In each of the sequential iterations on the input channel, we need to copy an area of $N_{shared} = (T_y * R_y + 2F_{radius}) * (T_x + 2F_{radius})$ at an x-offset of blockIdx.x $* T_x$ and a y-offset of blockIdx.y $* T_y * R_y$. We re-map the $N_{threads} = T_x * T_y * T_z$ available threads by calculating a flattened thread-id like so:
`threadIdx.x + threadIdx.y * Tx + threadIdx.z * Tx * Ty`. The copy is shown in Listing 3.

```
for (int i = 0; i < (T_y * R_y + 2F_radius) * (T_x + 2F_radius); i += N_threads) {
  int id = i + threadIdx.x + threadIdx.y * Tx + threadIdx.z * Tx * Ty;
  int y = id / Tx;
  int x = id % Tx;
  in_shared[id] = input[n, blockIdx.y * T_y * R_y + y, blockIdx.x * T_x + x)];
}
```

Listing 3: Copying the input matrix to shared memory.

If $N_{threads}$ doesn't evenly divide the amount of elements to copy, we may read even further out of bounds of the input array than before, so we need to pad the input array with an additional $N_{threads}$ elements. Also, if $T_x + 2F_{radius}$ is not a multiple 8, 16 or 32 (the amount of float-values to generate 32, 64, or 128 byte memory transactions), then we will not have completely coalesced

access. $T_x + 2F_{radius}$ is almost never a multiple of 8, 16, or 32, since $T_x$ must be a power of 2, and is always $\geq 8$ in our implementation, so we cannot get completely coalesced access unless $T_x + 2F_{radius} = 16 + 2 * 8 = 32$ for instance.

This way of copying to shared memory introduces another source of out-of-bounds accesses. In the worst case, the last iteration of the copy reads $N_{threads} - 1$ elements outside the region that it should. This manifests in threads reading extra rows. We can calculate the max size of these extra rows as follows:

$$\#extrarows = \left\lceil \frac{\lceil N_{shared}/N_{threads} \rceil * N_{threads} - N_{shared}}{T_x + 2F_{radius}} \right\rceil$$

The upper part of the fraction calculates the amount of extra elements being copied, which we then divide by the row width and round up, leading to any amount greater than 0 producing an extra row. We adjust our allocation of the final padded input image to be

$$|I^{padded}| = (T_y \star R_y \star \text{gridDim.y} + 2F_{radius} + \#extrarows) \star (T_x \star \text{gridDim.x} + 2F_{radius})$$

An illustration of the copy process for a single block is shown in Figure 17.

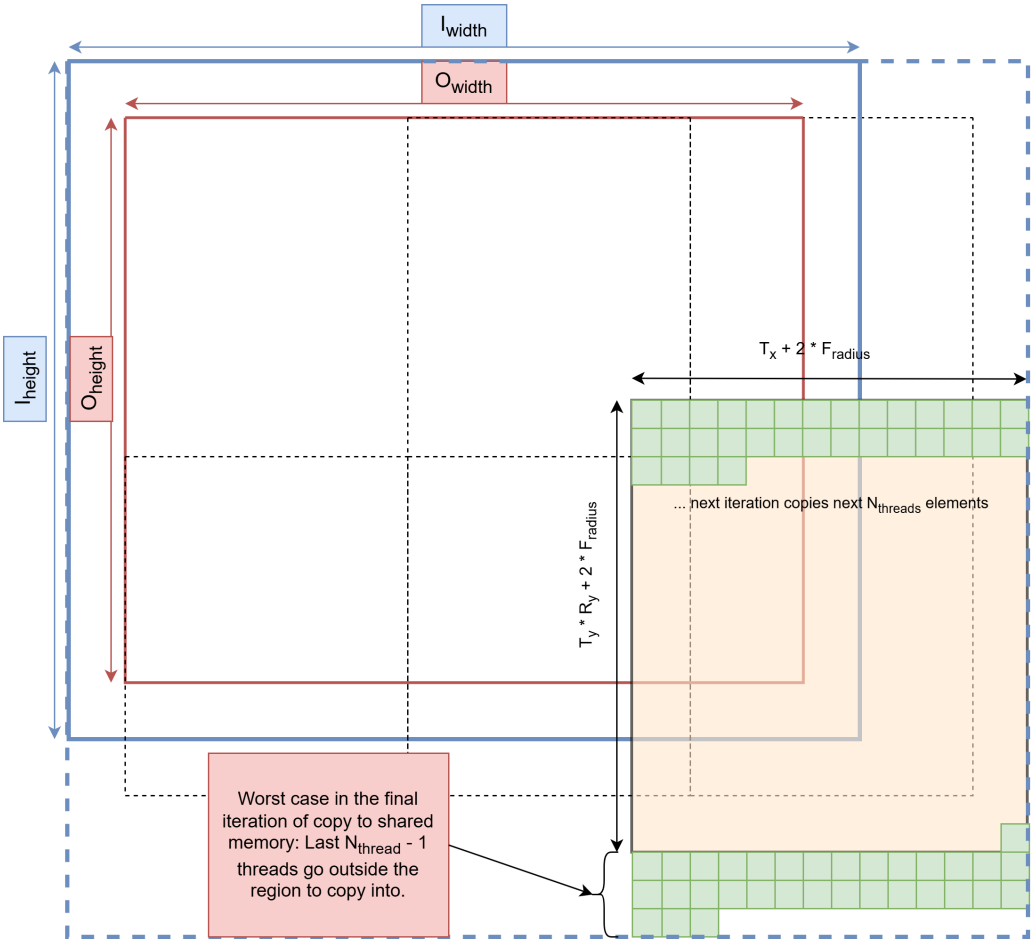



Figure 17: Shared memory copy. In each iteration, all threads copy a section (marked by green tiles) of the input array into shared memory (the whole orange area). 32 threads copy one value each in every iteration. The sizes in the figure are relatively representative, except $T_x = 10$, which is not possible as it must be a power of 2 and the small $O$ and $I$ dimensions.

**3.6.3.2 Copying Filter to Shared Memory** We use the same shared memory allocation for the input and the filters, where the filters come directly after the input matrix at index $(T_y * R_y + 2F_{radius}) * (T_x + 2F_{radius})$. The copy of the input to shared memory may, as mentioned, write slightly out of bounds, overwriting the filters in shared memory. To rectify this, we either need to sync the threads in the block to make sure the input writes are

finished and the filter writes won't be overwritten or round up the reserved size for the input in shared memory to the nearest multiple of the number of elements that are copied in each iteration such that no overwrites can occur regardless of the order. We chose the latter approach as this reduced the amount of synchronization necessary.

We need to copy $T_z * R_z * F_{width} * F_{width}$ elements from the filters into shared memory. For each input channel, there are #out channels filters. They are laid out sequentially in memory, so we can copy them into shared memory using all of our threads, as for the input matrix. We don't need to worry about rows and columns here, so we get fully coalesced accesses in each iteration. Here we can just pad the filter tensor linearly with the amount of elements all the threads in a block can copy in each iteration: $N_{threads}$. We also similarly pad the shared memory size: aligning the section of shared memory devoted to the filters up to $N_{threads}$.

**3.6.3.3 Transforming the Computation to Work with Shared Memory** Now, the computation needs to work with the format of the data as it is in shared memory. This means using the width of the copied matrix slice of the input array, called `InSharedPitch`, to calculate the flattened indices and not using the `blockDim` to account the the blocks offset, since this was handled by the copy. We also insert a `__syncthreads()` call after the computation to ensure that no threads begin the copies of the next input channel iteration and overwrite values that are being used for the current computation. The computation is shown in Listing 4.

```
for (int loop_y = 0; loop_y < Ry; loop_y++) {
  const int y = threadIdx.y * Ry + loop_y;
  for (int thread_cout = 0; thread_cout < Rz; thread_cout++) {
    for (int i = 0; i < FilterWidth; i++) {
      for (int j = 0; j < FilterWidth; j++) {
        tmp[loop_y][thread_cout] +=
          in_shared[InSharedPitch * (y + i) + threadIdx.x + j] *
          filters_shared[FilterWidth * FilterWidth * (Rz * threadIdx.z +
    thread_cout) + FilterWidth * i + j];
      }
    }
  }
}
__syncthreads();
```

Listing 4: Computation using shared memory.

### 3.6.4 Vector Loads

The final optimization uses a trick in which each thread copies several elements by using CUDA's 128-bit load and store instructions. This increases the number of elements copied per thread and therefore reduces the number of iterations taken to copy the input and filters. Our primary resource was an article [12] from NVIDIA. The article has a figure that shows, at best, a 12.5% higher bandwidth of memory transfers using 128-bit loads. However, "...using vectorized loads increases register pressure and reduces overall parallelism"[12]. This is not a problem in our case, since we often have too few threads per block to copy the tensors in a low amount of iterations.

To make use of these 128-bit load and store instructions, we can use the CUDA built-in `float4` data-type, which is 4 floats packed into one structure: 16 bytes or 128 bits. To be generic, we can use C++ templates to write the code to load using different data types with a single function. We call the template parameter `LoadType`, which is then instantiated to e.g. `float4`, `float` or any other 4-16 byte data-type. The amount of elements of `T` loaded by each `LoadType` is `ElementsPerLoadType = sizeof(LoadType) / sizeof(T)`. We can cast all loads to this and multiply the thread offsets into the arrays by `ElementsPerLoadType`.

**3.6.4.1 Alignment Requirements** Reading N-byte elements requires the address to be naturally aligned as mentioned in section 2.4.2.1, meaning

that the address is divisible by $N$. All allocations returned by CUDA are guaranteed to be aligned to at least 256 bytes, so the first element of each tensor will always be aligned to 16 bytes. We can also guarantee that the second block will read at an address aligned to 16 bytes, since we always use $T_x \geq 8$ and $T_x$ is a power of 2, so the horizontal offset of a block is always divisible by 16 bytes: $\underbrace{T_x * \texttt{sizeof(T)}}_{\text{divisible by 16}} *$blockIdx.x.

We cannot, however, guarantee that reading the first element of a row is aligned to 16 bytes, since the input array may not have a width that is divisible by 16 bytes (4 floats). This means that we need to pad our input array allocation by making the width of each row align up to 16 bytes. This gives us a new value: the pitch of the row, `row_pitch`, which is however many elements we need to skip to get to the next row in the allocation. We also need to pad the filters tensor such that each section of $T_z * R_z * F_{width} * F_{width}$ elements is aligned to a 16 byte boundary, since each block will start its read at this boundary. It takes $\left\lceil \frac{\#\text{out channels}}{T_z * R_z} \right\rceil$ filters to fill out an input channel. This, multiplied by $F_{width}^2$, is also aligned up to 16 byte boundary, giving us `filters_in_channel_pitch`: how many elements to skip to get to the next input channel.

We also need to pad the shared memory by aligning the width of input matrix block up to be aligned to 16 bytes. The other padding requirements explained in the previous sections can be adapted by always dealing with the number of elements each thread can load. For instance, every place we pad with $N_{threads}$ is changed to pad with $N_{threads} *$ `ElementsPerLoadType` instead. The load code for the input array and filters now looks as shown in Listing 5. The `device_index_arr_of_arr` is a helper function for indexing into an array of arrays with a given row pitch. `NumLoadTypes` and `FiltersNumLoadTypes` are the amount `LoadType`s that need to be loaded into shared memory.

The full algorithm can be seen in the appendix at subsection B.1.

```
// Input matrix load
for (int i = 0; i < NumLoadTypes; i += NumThreads) {
    const int id = i + thread_id;
    const int y = id / InSharedLoadTypePerRow;
    const int x = (id % InSharedLoadTypePerRow) * ElementsPerLoadType;
    ((LoadType*)in_shared)[id] =
        *(LoadType*)&input[device_index_arr_of_arr(
            row_pitch,
            height,
            in_channel,
            block_offset_y + y,
            block_offset_x + x
        )];
}
// Filter load
for (int i = 0; i < FiltersNumLoadType; i += NumThreads) {
    const int id = i + thread_id;
    ((LoadType*)filters_shared)[id] =
        *(LoadType*)&filters[
              filters_in_channel_pitch * in_channel
            + filters_block_offset
            + id * ElementsPerLoadType
        ];
}
__syncthreads();
```

Listing 5: Copy to shared memory using `LoadType`, typically `float4`

### 3.6.5 Same Mode Convolution

For the same mode convolution mentioned in 2.1.2, there are two differences: One, we pad the input array in both the dimensions of the input image to be $I^{padded}_{width} \times^{padded}_{height} = I_{width}+2F_{radius} \times I_{height}+2F_{radius}$ and make the output-image have the same dimensions. Two, we write at an offset of $F_{radius}$ in the x and y dimensions of the output tensor. We don't have to worry about alignment of these accesses, since they are not vectorized. With this implementation, the output array can be used as the input for the next iteration of a multistep convolution operation.

## 3.7 Futhark

### 3.7.1 Futhark Background

Futhark is a purely functional programming language that is statically typed, aiming to provide a high-level functional language that compiles efficient code for parallel hardware via CUDA, OpenCL, or multi-threaded CPU code. [8]. Parallel programs in Futhark consist of a combination of Second-Order Array Combinators (SOACs). These features enable the Futhark compiler to generate correct and efficient parallel code, eliminating the need to explicitly manage memory and handle data races. This is handled later in the compiler stages.

The Futhark compiler follows a fairly conventional architecture consisting of three major parts: the frontend, the middle-end, and the backend. [7]

**3.7.1.1 Frontend** The frontend parses, type-checks, and then transforms the Futhark source language into the Intermediate Representations (IRs) used by the compiler's middle-end.

**3.7.1.2 Middle-end** The middle-end consists of passes that accept a program as input and produce a new program. A combination of these passes is a *pipeline*.

The Middle-end starts with a program in SOAC representation, given by the front-end. The code is transformed, depending on the pipeline, into another representation. To produce GPU CUDA code, it is transformed to a *GPU* representation.

In the tile-loop pass, a block-register tiling of General Matrix Multiplication already exists, and it is the same pass where the optimization transformation for the convolution operation should be implemented.

**3.7.1.3 Backend** The backend takes a program in some core IR representation, which always contains memory information, and then translates the program to imperative IR, also known as ImpCode. Finally, the ImpCode is translated to a real executable language.

### 3.7.2 Overview of the Compiler

We have created a sparse overview of the Futhark compiler structure, based on the information and figures from [9]. The GPU optimization has been highlighted green, and the red highlight denotes the pass where the convolution optimization should be implemented.

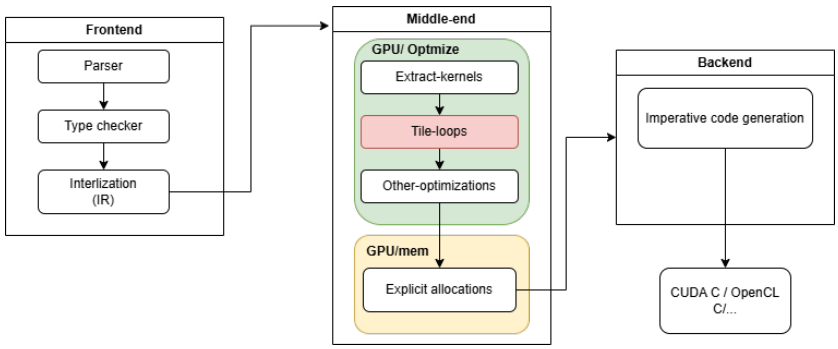



Figure 18: Simplified compiler overview

## 3.8 Incremental Flattening & Auto Tuning

When generating GPU code, Futhark will generate several different versions. The best version is then chosen at runtime, based on specific heuristics. This generation of different code versions is referred to as *incremental flattening* [3]. When passing through the code, it inserts guarded code versions, each time it encounters a **map** operator, which can be mapped to some level $l$ of hardware parallelism. It then generates multiple versions, each of which explores one more level of inner parallelism. The guards are implemented with a threshold that determines the amount of parallelism at runtime. This will generate semantically equal code versions that can utilize all possible top-level parallelism.

Furthermore, the thresholds are autotuned using a simple stochastic autotuner on the threshold parameters.

When applying incremental flattening, the different versions of the code are structured in a tree-like way. A representation of this is shown in the figure below from [3].

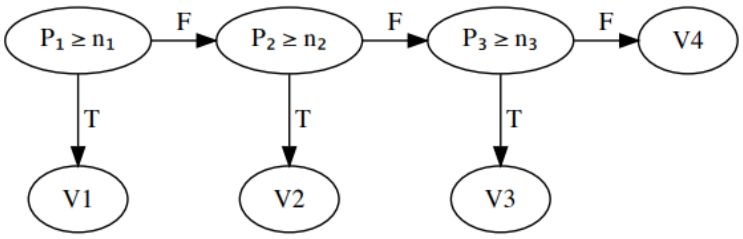



Figure 19: Visualization of branching structure of Incremental flattening [3]

In the figure $V_1..V_4$ is the different code versions, where each version utilizes a different level of parallelization.

## 3.9 Memory in Futhark

Even though Futhark does not have explicit memory allocation for the user, due to being a functional language, the compiler utilizes an extended functional language with a notion of explicitly allocating memory through LMAD representations or transforming into lower-level code [15]. The ability to allocate explicit memory is crucial for our optimization, since it relies on each thread in a kernel storing multiple intermediate results.

## 3.10 Kernel Representation

We provide a short overview of two of the most important internal representations in the Futhark compiler for the convolution example. The overview is based on the description from[13] and the Hackage comments [7]

### 3.10.1 SegOps

The GPU kernel always consists of an outer map, determining the parallel structure. This is represented as a *SegOp* structure, which is either a *segmap*, *segscan*, *seghist*, or a *segred*. It is semantically a stack of perfectly nested maps, on top of some computation. The Segspace encodes the map structure.

SegOps has parameters describing the representation of the body, as a *level*. The *level* describes for GPU representations, whether the SegOp is expected to run at the thread or block level.

```
data SegOp lvl rep
= SegMap lvl SegSpace [Type] (KernelBody rep)
| SegRed lvl SegSpace [SegBinOp rep] [Type] (KernelBody rep)
| ...
deriving (eq,ord,show)
```

Figure 20: Definition of SegOp

For our naive direct-mapped convolution step implementation, we would have a SegMap or SegRed construction at the thread level, where the three outer dimensions correspond to the block dimensions.

### 3.10.2 Screma

Screma (ScanReduceMap) is a value constructor for the SOAC type, representing a combination of scans, reduces, and maps. The functionality of a Screma is specified through a ScremaForm. The ScremaForm consists of a map part, denoted as *Lambda*, and then an arbitrary number of scans and reductions. If the ScremaForm only consists of map information and reductions, it would represent a redomap construct. For the convolution example, we would only expect the loops to be in redomap form and deeply nested. A figure illustrating the Screma construct is provided below:

```
Screma SubExp [Vname] (ScremaForm rep)
.
.
.
data ScremaForm rep = ScremaForm
{
  ScremaLambda :: Lambda rep,
  ScremaScans :: [Scan rep]
  ScremaReduces :: [Reduce rep]
}
deriving (Eq, Ord, Show)
```

Figure 21: ScremaForm definition

## 3.11 Convolution in Futhark

To demonstrate our desired code transformation in Futhark, we will work with an example of a convolution with multiple input and output channels as discussed in section 2.1.2.

```
def conv2d [l][m][n][o][k] (inputs : [l][m][n]f32) (kernels : [l][o][k][k]f32) =
  let out_m = m-k+1 -- output height
  let out_n = n-k+1 -- output width
  in
  -- indexes and zero initialization of outputs
  tabulate3d_ o (out_m) out_n (\out_channel y x ->
    -- reduction across input channels
    reduce (+) 0 (
      map_ (iota l) (\in_channel ->
        -- reduction across rows
        reduce (+) 0 (
          -- slice of input
          map2_ (inputs[in_channel, y:(y+k),x:(x+k)] :> [k][k]f32)
          (kernels[in_channel,out_channel] :> [k][k]f32) (\in_row kernel_row ->
            -- redo-map
            reduce (+) 0 (
              map2_ in_row kernel_row (\in_val kernel_val ->
                in_val * kernel_val
              )
            )
          )
        )
      )
    )
  )
```

Figure 22: Convolution of multiple input-output channels in Futhark

Where we have used a modified *map* and *tabulate* where the input is the first argument as opposed to the lambda for a better overview of the input arrays:

```
map_ xs f ≡ map f xs
```

Figure 23: Map transformation

.

## 3.12 Transformation of Futhark code

To transform our Futhark code, listed in Figure 22, to implement some of the optimization transformations, such as those shown in Figure 14, we start by inspecting the IR form of the code generated by the Futhark GPU compiler pass.

We use the command

```
futhark dev --gpu "conv.fut"
```

This generates multiple versions of possible kernel representations, according to the general flattening principle of Futhark. To implement our optimizations, we would want a kernel that is "parallel" in the three outer dimensions: the number of output channels, the height of the output image, and the width of the output image. The inner reduction should be kept sequential. This would result in an intragroup kernel, which consists of a segOp on the thread level, with three dimensions. To generate the desired kernel, we gave Futhark the following attribute over the innermost map of the 3D tabulate construct.

```
def tabulate3d_ (dim1) (dim2) (dim3) (f) =
    tabulate_2d dim1 dim2 (\i j ->
    #[sequential_inner]
     map (\k -> f i j k) (iota dim3) )
```

To ensure the generation of kernels with sequential inner execution.

However, the Tile-loop pass would optimize our desired kernel away. To circumvent this, we disabled the tile-loop pass, and then the Futhark compiler generated the correct kernel with the following shape:

```
let {defunc_0_map_res_pretr_pretr_9606 :
       [o_7512][out_m_9210][out_n_9212]f32} =
   segmap(thread; ;
   grid=segmap_usable_groups_9605;
   blocksize=segmap_tblock_size_9604)
   (gtid_9607 < o_7512, gtid_9608 < out_m_9210,
       gtid_9609 < out_n_9212)
   (~phys_tid_9610) : {f32} { ...
```

Figure 24: Segmap IR representation

Which has the desired SegOp arguments.

We transform the kernel by hand to create a prototype for the final version. Disabling the tile loop pass is fine, as it is the same pass where our optimizations would be implemented. Therefore, the compiler should be able to recognize the convolution construct and apply the convolution transformations before being optimized away by another transformation.

To transform the segmap construct in Figure 24, we must define the new register tile sizes and redefine the shapes of the input and output. The input is modified by adding the two register dimensions and dividing the original sizes $Dim_{out} = \frac{o}{reg_Z}$ and $Dim_{height} = \frac{out_m}{reg_Z}$, resulting in the input shape of:

$$[Dim_{out}][Dim_{height}][out_{width}][reg_Y][reg_Z]f32$$

For the output, in the original kernel version, each thread returns one element, and in the transformed version, each thread will return a tile of shape.

$$[reg_Y][reg_Z] = reg_Y \times reg_Z$$

Resulting in a new segmap

```
let {Dim_height : i64} =
   sdiv_up64(out_height, 8i64)
 let {Dim_out : i64} =
   sdiv_up64(num_out_ch, 4i64)
 let {to_rearrange_9707 :
        [Dim_out][Dim_height][out_width][8i64[4i64]f32} =
   segmap(thread; ; grid=segmap_usable_groups_9706;
   blocksize=segmap_tblock_size_9705)
   (tIdz_9708 < Dim_{out}, tIdy_9709 < Dim_{height,
   tIdx_9710 < out_width_9313) (~phys_tid_9711) :
    {[8i64][4i64]f32} {
```

Figure 25: Segmap IR representation, with $reg_Y = 8$ and $reg_Z = 4$

The convolution operation is defined by nested loop constructs, precluded by several boundary checks. In the original kernel, they have the following nested loop reduction shape:

```
let {loop_reduce_N : f32 } = -- outer N loop
  loop {loop_res_N : f32} = {0.0}
      for i_1 < N do {
        ...
        -- boundary validity check
        ...
        -- inner filter loop
      let {loop_reduce_filter1 : f32} = {
        loop {loop_res_F1} = {0.0}
        for i_2 < f_width do {
          let {slice_index1 : i64} =
            -- the index into the input slice
            add(gtid_y, i_2)
          -- inner filter loop
          let {loop_reduce_filter2 : f32} = {
            loop {loop_res_F2} = {0.0}
              for i_3 < f_height do {
                let {slice_index2 : i64} =
                -- the index into the input slice
                add(gtid_x, i_3)
                let {input_val : f32} =
                    inputs[i_1, slice_index1, slice_index2]
                let {kernel_val : f32} =
                    kernels[i_1, gtid_z, i_2, i_3]
                let {mul_val : f32} =
                    fmul32(input_val, kernel_val)
                let {inner_red : f32} =
                    fadd32(mul_val, loop_res_F2)
                in {inner_red}
          }
          let {middle_red} =
          ↪  fadd32(loop_reduce_filter2,loop_res_F1)
          in {middle_red}
        }
        let {outer_red} = fadd32(loop_reduce_filter1,loop_res_N)
        in {outer_red}
        }
        return {returns loop_reduce_N }
```

Figure 26: Loop construct of convolution

Where we would like the loop constructs to return several elements, instead of only one, by adding the necessary loops from the ”strip mining” of the outer parallel dimensions.

To transform it with our optimizations, we start by zero-initializing the return results of each thread, using replicate, followed by defining the start point of each thread, with *Tidy, Tidz* specified in Figure 25.

```
let {init : [reg_y][reg_z]f32  = replicate([reg_y][reg_z], 0) }
let {y_thread_start : i64} = mul(Tidy, reg_y)
let {z_thread_start : i64} = mul(Tidz, reg_z)
```

Now we need to add the two loops, following the structure shown in Figure 14, and specify that the outer loop should now return a 2D array instead of a single element. The outer loop definition becomes:

```
let {loop_reduce_N : [reg_y][reg_z]f32 }
    let loop {loop_res_N : *[reg_y][reg_z]f32} = init
```

Before the first filter loop, we add the two new loops with the structure

```
let {reg_y_loop_res : [reg_y][reg_zf32] }=
  loop {Reg_y_loop}: *[reg_y][reg_z]f32 = {loop_res_N}
    for loop_y < reg_y do {
      let {thread_current_y} = add(y_thread_start, loop_y)
    }
    let {reg_z_loop_res : [reg_y][reg_zf32] }=
      loop {Reg_z_loop}: *[reg_y][reg_z]f32 = {reg_y_loop_res}
      for loop_z < reg_z do {
        let {thread_current_z} = add(z_thread_start, loop_z)
      }
```

The inner filter loops are still kept to calculate one result at a time, but now with the updated indexing based on the added inner loops, and extra code to write to the correct accumulator of the thread.

```
./modifed_ver --dump-cuda modified_ker.cu
```

Figure 28: Command for generating CUDA kernel from Futhark

```
let {outer_filter_loop : f32} =
  loop {outer_filter_tmp : f32} = {0.0f32}
  for i < filter_width do {
    let {y : i64} =
      add_nw64(thread_current_y, i)
    let {inner_filter_loop: f32} =
      loop {inner_filter_loop_tmp: f32} = {0.0f32}
      for j:i64 < filter_width do {
        let {x : i64} =
          add_nw64(tIdx, j)
        let {input_val : f32} =
          inputs[in_channel, y, x]
        let {filter_val : f32} =
          filters[in_channel, thread_current_z, i, j]
```

Figure 27: Updated inner loop

Finally, to adhere to the original shape of the transformation, we rearrange the output. This might be avoided with smarter transformation and dimension extraction in the compiler.

A limitation of this handwritten version is that no additional bounds checking was inserted, due to the amount of extra logic for the prototype. This limits the input to only sizes that tile perfectly; otherwise, we will encounter illegal out-of-bounds accesses. This limitation should be handled when doing the transformation in the compiler.

After recompiling the hand-modified Futhark kernel and verifying the validity of the transformations by comparing it to the naive version, we got Futhark to generate the CUDA kernel, to inspect it, and measure the performance of the generated CUDA kernel. This will be elaborated further on in the evaluation section 4.2.4. The CUDA kernel was generated with the following command: The CUDA kernel itself is way too long to include. Upon inspection of the generated kernel, it followed the desired structure of

declaring and then initializing a register array for each thread, with the desired dimensions, followed by the desired loop-nest structure of the sequential loops.

# 4 Evaluation

## 4.1 Testing

### 4.1.1 CUDA versions

To test the correctness of the CUDA convolution implementations, we compared them to the only block-tiled version, where threads are directly mapped to the output values. The direct-mapped version was first tested with small test cases to confirm the correctness of the program. The kernels were tested on different inputs, with varying numbers of input and output channels as well as with different tiling-parameters to ensure no illegal out-of-bounds accesses could occur. The following parameters were used for testing:

| **Parameters (Inp, InChannels, OutChannels)** | **Description** |
|---|---|
| (2048, 32, 32) | Well-formed medium input |
| (4096, 64, 64) | Well formed big input |
| (1500, 22, 22) | Malformed smaller input |
| (2048, 1, 32) | Well formed, one single input channel |

Table 1: Testing inputs

These inputs were tested on several different permutations of register and block tiling parameters and filter radii. A few are listed below.

| (Radius, Tx, Ty, Ry, Tz, Rz) | Purpose |
|---|---|
| (1, 16, 16, 3, 2, 3) | Balanced tiling of x,y. odd register tiling |
| (1, 32, 8, 8, 4, 1) | x-dim favored tiling, even register tiling |
| (1, 256, 1, 8, 1, 8) | Only x, and register tiling |
| (2, 8, 2, 32, 2, 2) | Heavy y-register tiling |
| (5, 64, 4, 4, 2, 8) | Big radius, and maximum threads. |

Table 2: Few test cases

All tests for the direct-mapped and only-y register-tiled version validate for all inputs. For the register-tiled version in both the y and z dimensions, certain parameter selections cause a *GPU ERROR: too many resources requested for launch*, which indicates that too many registers are being used by the SMs. This occurs for parameters where the number of threads equals the maximum number of threads for each block, with register tile sizes greater than 1. For example, the parameters $(5, 64, 4, 4, 2, 8)$ cause the kernel not to run at all, most likely since the total amount of registers used by the kernel exceeds the ones available. This indicates that the tiling parameters should be kept rather small to avoid this issue.

For all input, where we do not exceed the hardware resources of the GPU, all kernels validate on all inputs. To reproduce the tests, please navigate to our Github.

## 4.2 Benchmarking

The original goal of the project was to compare the transformed Futhark convolution with the optimized versions created in CUDA. We do this for the kernel generated by the hand-transformed Futhark code, mentioned in section 3.12. We were not able to generate the transformations in the compiler, so no benchmarking of this could be created.

We instead created a comprehensive comparison between our different optimizations in CUDA to analyze the potential performance of the convolution operation. To achieve the highest performance, we evaluated a range of different permutations of block and register tiling parameters, as well as various configurations of filter sizes, input sizes, and the number of input and output channels, to assess their impact on performance. We measure on arithmetic output, in TFLOPS[4] due to the general compute-bound nature of the convolution operation.

### 4.2.1 Hardware Description

All benchmarking was done on the Futhark machines, provided through DIKU. The machine has an A100 NVIDIA GPU with the following relevant specifications:

[4]Tera ($10^{12}$) floating point operations per second

| Peak FP32 | 19.5 TFLOPS |
|---|---|
| GPU memory | 40GB |
| GPU BandWidth | 1555 GB/s |

Table 3: A100 specifications

For our optimizations, we want to get as close to the peak floating-point operations as possible.

### 4.2.2 Parameter Search

To benchmark our different versions, we want to use the optimal tiling parameters. For our register-titled optimized convolution, we have five different tiling parameters to optimize, and we also use the Filter size as a parameter, since for different convolutions in CNN, this is a known filter with a fixed size.

The tiling parameters are $T_x, T_y, T_z, R_y, R_z$, and for the filter size we range over its "radius" $r$. For the tiling parameters, it is almost impossible to find all combinations, since they will also depend very much on the input; therefore, we limit the search space for the parameters.

$$(T_x, T_y, T_z) = \{1..4..64\}$$

$$(R_y, R_z) = \{1..16\}$$

We used a brute-force approach to analyze which combination would yield the best performance by trying a number of different combinations. We found that the choice of tiling parameters had a significant impact, and an incorrect choice would result in a substantial slowdown. It was quickly discovered that small values of $Ty$ and $Tz$ were preferable and that $Tx$ should be around $\approx 32$ for the best performance, across most problem instances. Therefore, the following block tiling parameters were chosen.

$$T_x = 32$$
$$T_y = 2$$
$$T_z = 2$$

The register-tiling parameters were more volatile across the problem instances and were, in general, chosen for the specific input. However, a general trend was to pick $R_y \approx 10$ and $R_z \approx 4$

The radius was tested for the range $1-8$. We did not test greater radii as there is no real benefit of such big filter dimensions in the context of CNNs [1]. Most filter sizes vary from $3 \times 3$ to $9 \times 9$, which corresponds to $r = 1-4$.

The best performance was in general observed for radii of $r \approx 4$,

### 4.2.3 CUDA Version Comparison

To analyze the impact of the different optimizations, we benchmarked the performance of each version on various input, channel count, and filter sizes. The filter sizes had varying radii from 1..8. For all versions the tile sizes mentioned in previous sections were used, and the register-tiles varied slightly between the different radius sizes, however all versions were tested with the same set of register tiles, $T_y \in \{5, 6, 8, 10, 13, 16\}$ and $T_z \in \{2, 4\}$. Each performance measure was done by first running a warm-up run and then taking the average over 5 GPU runs.

The first performance benchmarking was on a "big" input, with the following dimensions.

$$\begin{aligned} I_{height} &= 4096 \\ I_{width} &= 4096 \\ I_c &= 64 \\ O_c &= 64 \end{aligned}$$

We chose to keep the image size the same across tests, since we could reduce the parallelism by reducing the output channel count as long as $T_z * R_z \leq O_c$.

The results can be seen in the Figure 29. In all figures, the max theoretical FLOPS of the A100 (19.5 TFLOPS) is shown with a red line. When the vector load version is at its best, it reaches $18.879/19.5 = 96.3\%$ of that. All the optimized versions perform better than their previous counterpart, but the vector load only has a slight edge over the non-vector load. Higher radii are especially impacted by the shared memory optimization, with a sharp dropoff in performance of the branchless version at $R = 8$.

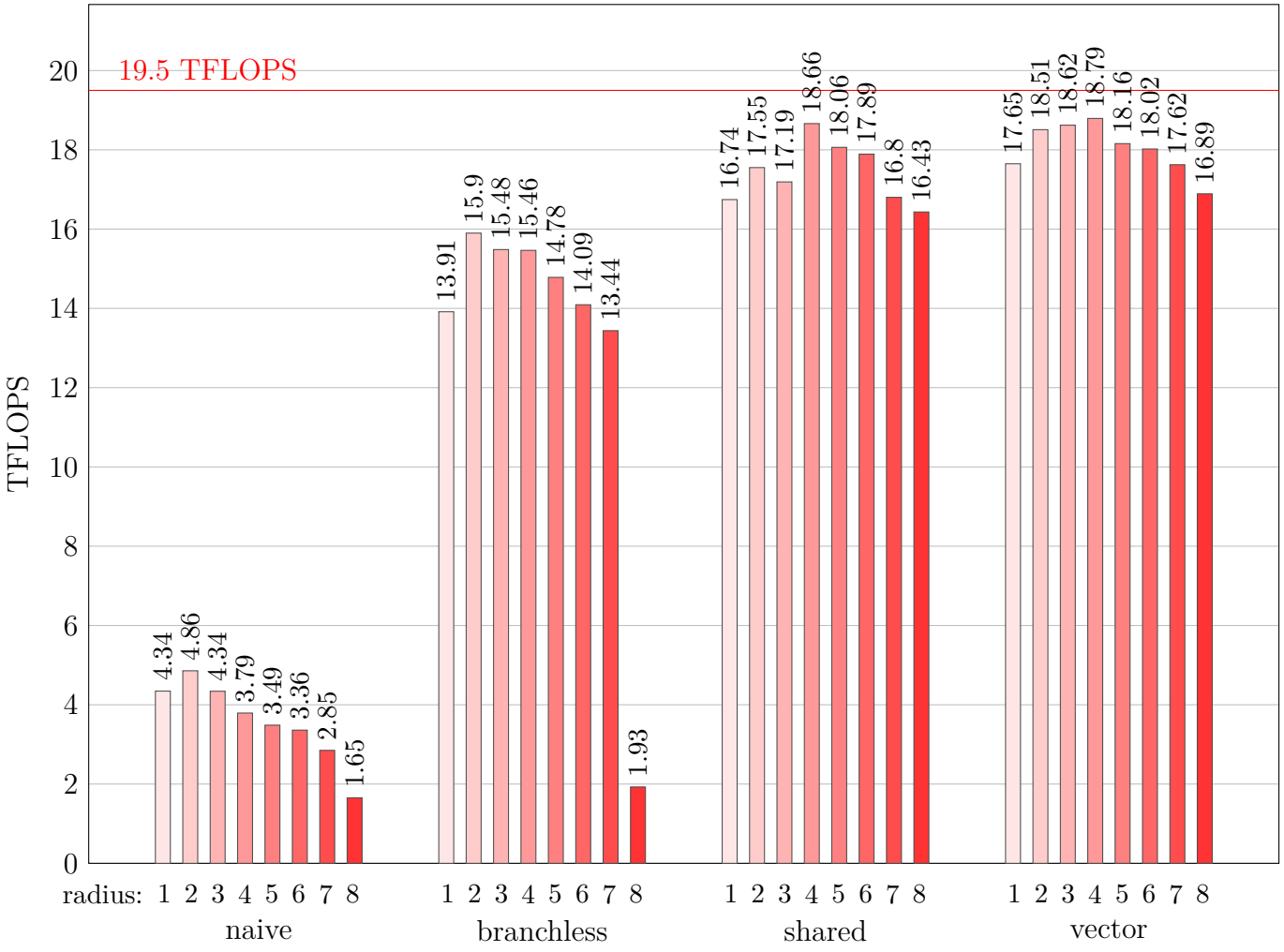



Figure 29: Impact of each performance optimization for input size of $I \times O \times I_{height} \times I_{width} = 64 \times 64 \times 4096 \times 4096$ and varying radii (shown above version name).

To measure the impact of the channel count, additional benchmarking was done on the same versions, but now with channel counts of 64, 32, 8, and 1, respectively, and the radius frozen to $R = 6$. This radius performed the best for lower channel counts, with all versions handling 1x1 poorly due to the lower inherent parallelism. The results are shown in Figure 30. Again, the optimizations all improve the performance, but the most impactful are

the branchless and shared memory versions.

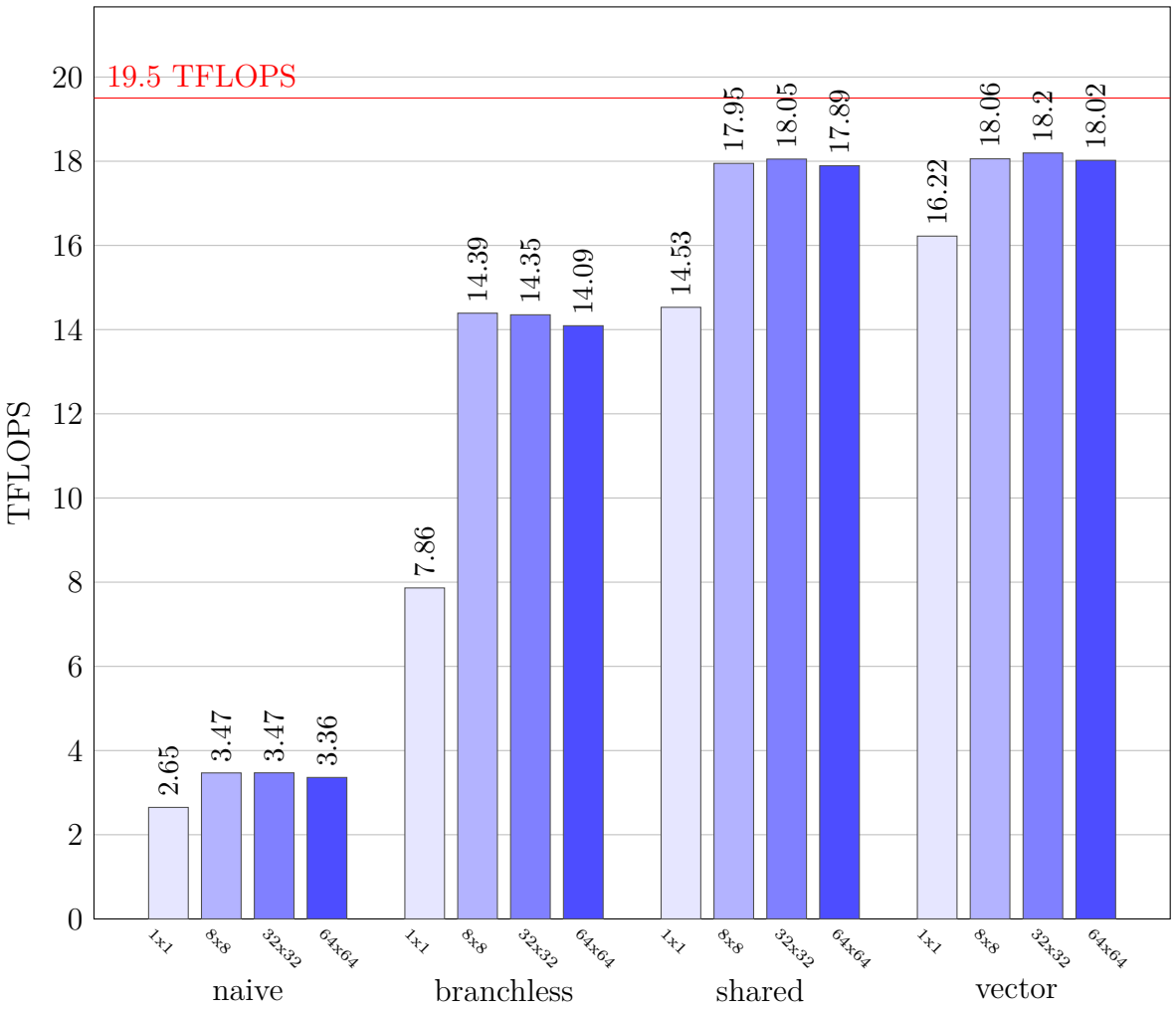



Figure 30: Impact of each performance optimization for input image size of $I_{height} \times I_{width} = 4096 \times 4096$ and varying input and output channel counts: $1 \times 1$, $8 \times 8$, $32 \times 32$ and $64 \times 64$ (shown above version name). Radius is locked at $R = 6$.

We chose $R = 6$ for the last comparison, as it performed optimally. For the sake of a fair comparison, we have also included a figure showing the vector load version across different radiiand channel counts. We show 1x1, 4x4, 8x8, and 32x32, as the performance jump from 1x1 is quite large, and the

jump from 32x32 to 64x64 is almost nothing. Changing the channel counts is roughly equivalent to changing the input-image dimensions. Results are shown in Figure 31. The vector load version performs relatively poorly at $R = 1$ and a 1x1 channel count, but is already performing well at 4x4 and higher radii. The tiling was changed to have $T_z * R_z \leq O_c$ where needed.

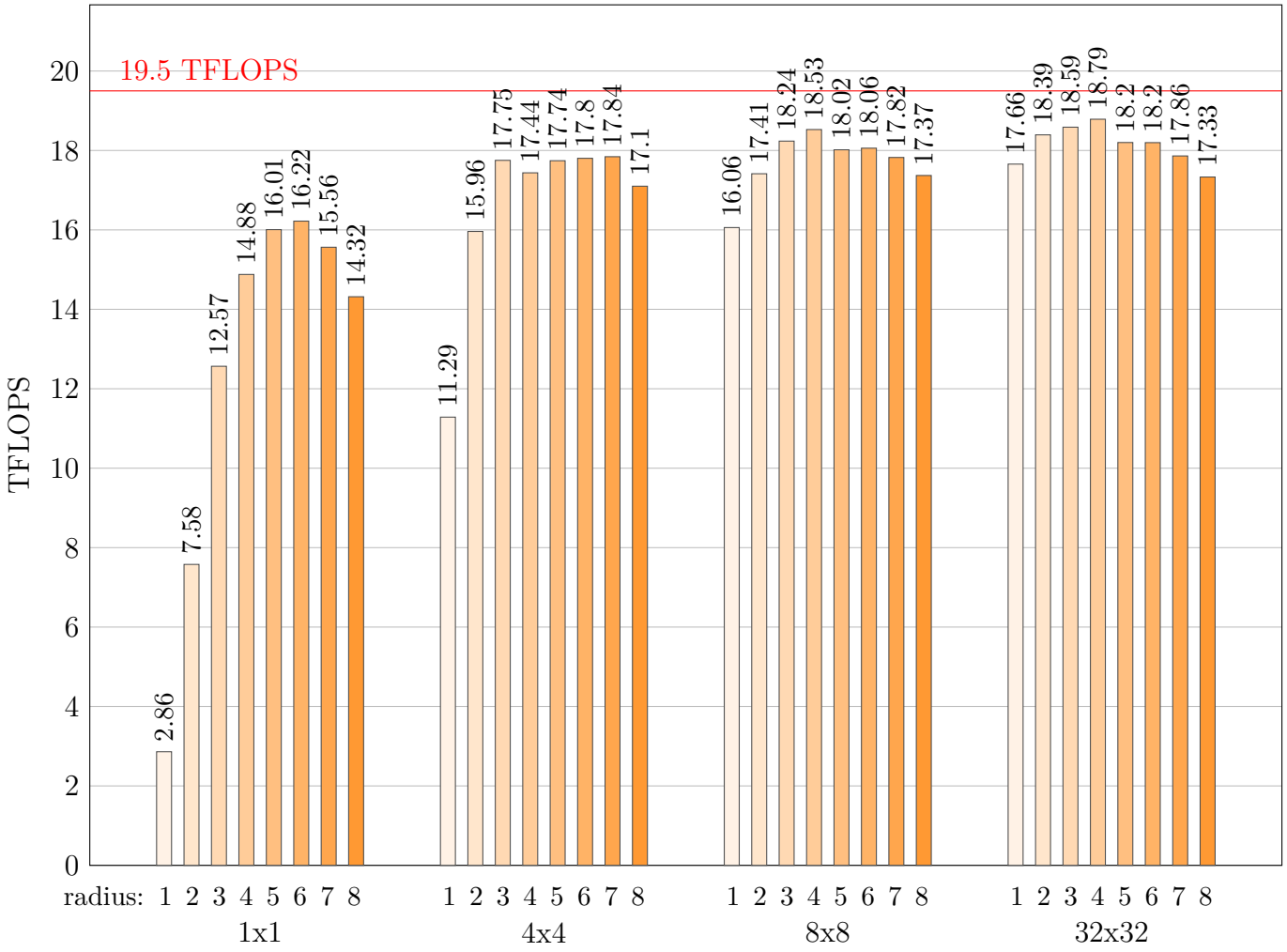



Figure 31: Impact of channel count and radius on performance for the vector load version with an input image size of $I_{height} \times I_{width} = \times 4096 \times 4096$.

### 4.2.4 Futhark Evaluation

To benchmark the Futhark kernel, generated by our hand-modified Futhark version discussed in Section 5.13, we utilize Futhark's utility to load the kernel on execution and time several runs.

```
./Mod_ver --load-cuda Fut_ker.cu -r 10 -t /dev/stderr -n  < data/dataset_ker.in
```

Figure 32: Benchmarking the Futhark kernel

The kernel was tested on an input size of $2052 \times 2052 \times 64 \times 64$ and auto-generated tiling parameters, and hard-coded register tiles of $R_y = 8$ and $R_z = 4$. The input for this version must result in output sizes that perfectly tile into the register parameters. Otherwise, the kernel will access out-of-bounds data due to the missing boundary checks.

The kernel was measured to obtain run times of $83.5ms$ on average, which corresponds to

$$Fut_{ker} \approx 10.2TFLOPS$$

Which corresponds to about 51% of the peak performance.

We benchmarked the non-optimized version we wrote in Futhark for comparison, with the same input, using the *futhark-bench utility* with the CUDA backend, and got a run time of $269.4ms$ on average, and this corresponds to

$$Fut_{orig} \approx 3.19TFLOPS$$

Which corresponds to about 16% of the peak performance. This shows that the modified version gets a speedup of

$$SpeedUp \approx \times 3.2$$

The benchmarking was also done for a smaller output, $1028 \times 1028 \times 32 \times 32$ and yielded similar results.

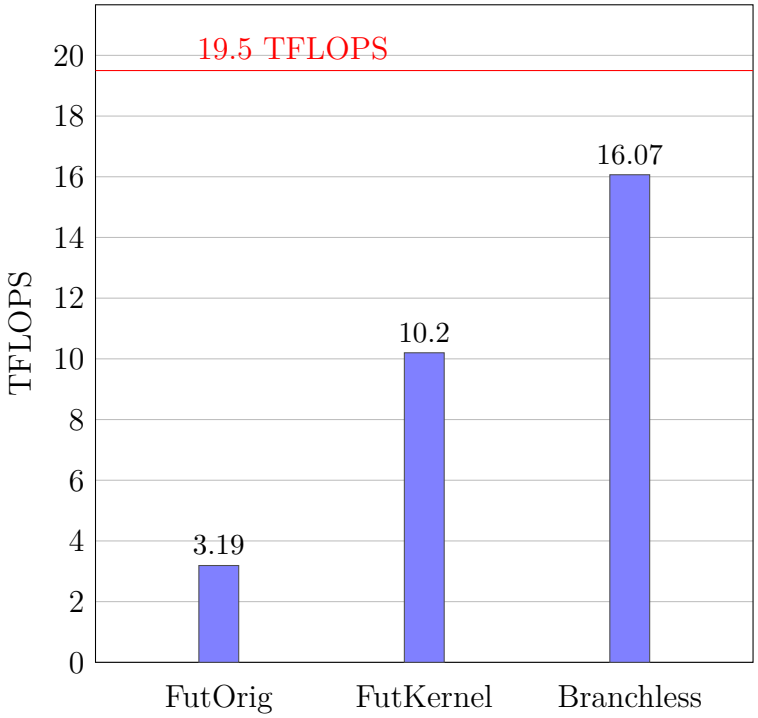



Figure 33: Impact of Futhark hand kernel generated from hand transformed Futhark IR, for input image size of $I_{height} \times I_{width} = 2052 \times 2052$ and input and output channel count $64 \times 64$. Radius is locked at $R = 2$.

# 5 Discussion and Further Work

## 5.1 Discussion of CUDA Results

### 5.1.1 Work Size

It was found through benchmarking that the performance of the kernels gets worse if the parallel work size is too small. For instance, if the total dimensions of $O \times I_{width} \times I_{height}$ go below a certain threshold. For example, for the vectorized load version, performance starts to suffer once you go down to small images, such as 256x256 images, with four input channels. For $R = 1$ it only reaches 2.2 TFLOPs, but for $R = 6$ it still reaches 12.3 TFLOPS with the same tiling parameters as used for larger images. It's possible that choosing different tiling parameters could improve performance

even further.

## 5.2 Impact of Radius

We can see from figure 29, that as the radius value increases above $r > 2$ the nonshared memory versions start to decrease in performance. This is especially evident for $r = 8$, where the branchless versions get $\approx \times 7$ performance slowdown. However, this aligns with the arithmetic intensity previously mentioned, where we can observe that the convolution operation becomes more memory-bound as the radius increases. This also explains why the effect is not as evident for the shared memory versions, since the increased number of global memory accesses does not affect them. For the shared memory versions, we see that $r = 4 - 5$ is optimal.

## 5.3 Impact of Channels

From figure 30 and figure 31 we can see that the choice of input and output channels does not make a great impact as long as the number of channels is $> 1$. If $channels == 1$ then we cannot tile in the $z$ dimension of the kernel, and the optimization becomes only tiling in the $y$ dimension. Furthermore, it may not fully saturate the GPU due to the reduced input size.

## 5.4 Discussion of Fuhtark Kernel results

We demonstrated, through benchmarking of the generated Futhark kernel, that implementing block and register tiling can yield a $\times 3$ speedup compared to implementing it naively in Futhark. Futhark automatically determined the kernels' tiled dimensions. However, the register tiles were pre-emptively chosen. This provides an opportunity for even better performance, as the choice of these parameters can significantly impact performance and would greatly benefit from auto-tuning.

There is also potential for even more performance if further optimizations could be implemented, such as using shared memory.

## 5.5 Further Work

There is still work to be done to complete the optimizations in Futhark, as we did not manage to add the transformations to the compiler.

When the transformations are added, we could further explore the use of shared memory for optimization, as seen in CUDA, which significantly contributes to performance. Other future work could involve alternative methods to calculate the convolution operation, such as transforming the data to utilize GEMM, or handling special cases, and therefore special kernels for separable filters, as discussed in [20].

### 5.5.1 Compiler Transformations

As mentioned, the compiler transformations were never added to the Futhark compiler, and this would be an obvious next step. This would involve pattern matching the convolution operation, extracting the required dimensions, and handling additional necessary boundary checks. For this project, further exploration of pattern matching should be conducted to extract the dimensions from the loops. The boundary checks could be handled by creating an "epilogue," as demonstrated for block and register tiling of tensor core computation, as shown in [4]. This would allow the main part of the convolution to be calculated efficiently without handling out-of-boundary checks, as it would be perfectly tiled. Then, a small part of the calculations would be handled sequentially.

### 5.5.2 Shared Mem in Futhark

It would also be beneficial to explore the exploitation of shared memory in the Futhark compiler transformations. This could eliminate certain boundary checks and has shown promise in improving performance. The shared memory would be expressed through the use of LMADS [15].

# 6 Conclusion

This thesis has explored how to optimize the convolution calculation, used primarily in Convolutional Neural Networks, for GPU execution, with the goal of implementing these optimizations in the Futhark compiler. The optimization exploration was achieved successfully by exploiting temporal reuse through block and register tiling of the convolution operation in two dimensions: the height of the output image and the number of output channels. Several iterations of optimizations were explored and implemented, including

the use of shared memory and vector loads. The optimizations were successfully validated and evaluated. The Optimized convolution CUDA kernels achieved high arithmetic performance, reaching up to 96.3% of the GPU's peak theoretical FLOPS.

For the Futhark language, a hand-modified kernel was created and evaluated against the naive implementation in Futhark, achieving approximately three times the speed of the naive implementation, reaching up to 51% of the GPU's peak theoretical FLOPS.

The handwritten kernel does, however, have strict restrictions on the input dimensions, as additional boundary checking was not implemented, and was left for the implementation in the compiler. Also, the implementation of the optimizing transformations in the Futhark compiler was unfortunately not completed. This leaves room for significant improvement and further work in the compiler, including applying transformations, boundary checking, and utilizing shared memory.

However, the performance increase shown by both the CUDA kernels and the handwritten Futhark kernel compared to naive implementations demonstrates the potential benefits of implementing optimizations in the Futhark compiler, and we believe it is a worthwhile project for the future.

# References


[1] Yunus Camgözlü and Yakup Kutlu. Analysis of filter size effect in deep learning, 2020. URL: https://arxiv.org/abs/2101.01115, arXiv:2101.01115.

[2] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. 10 2006.

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

[4] Anders L. Holst. Optimizing tensor contractions for gpu execution in futhark. *DIKU*, 2024.

[5] Marc Jordà, Pedro Valero-Lara, and Antonio J. Peña. Performance evaluation of cudnn convolution algorithms on nvidia volta gpus. *IEEE Access*, 7:70461–70473, 2019. doi:10.1109/ACCESS.2019.2918851.

[6] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[7] The Futhark Programming Language. Futhark-hackage comments. URL: https://hackage.haskell.org/package/futhark-0.24.3/docs/Futhark.html.

[8] The Futhark Programming Language. Why futhark?, n.d. URL: https://futhark-lang.org.

[9] Steffen Holst Larsen. Multi-gpu futhark using parallel streams. *DIKU*, 2019.

[10] Yann Lecun, Larry Jackel, L. Bottou, A. Brunot, Corinna Cortes, John Denker, Harris Drucker, Isabelle Guyon, Urs Muller, E. Sackinger, Patrice Simard, and V. Vapnik. Comparison of learning algorithms for handwritten digit recognition. 01 1995.

[11] Shuai Lu, Jun Chu, Luanzheng Guo, and Xu T. Liu. Im2win: An efficient convolution paradigm on gpu, 2023. URL: https://arxiv.org/abs/2306.14316, arXiv:2306.14316.

[12] Justin Luitjens. CUDA Pro Tip: Increase Performance with Vectorized Memory Access. https://developer.nvidia.com/blog/cuda-pro-tip-increase-performance-with-vectorized-memory-access/. [Accessed 28-05-2025].

[13] Christian Marslev and Jonas Grønborg. Efficient sequentialization of parallelism. *DIKU*, 2024.

[14] Brian McFee. *Digital Signals Theory.* 09 2023. doi:10.1201/9781003264859.

[15] Philip Munksgaard, Cosmin Oancea, and Troels Henriksen. Compiling a functional array language with non-semantic memory information. In *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages*, IFL '22, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3587216.3587218.

[16] NVIDIA. Cuda, release: 10.2.89, 2020. URL: https://developer.nvidia.com/cuda-toolkit.

[17] NVIDIA. Cuda c++ programming guide v12.9, May 2025. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide.

[18] NVIDIA. Parallel thread execution isa version 8.8, May 2025. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[19] Cosmin Eugen Oancea. *PMPH Lecture Notes for the Software Track Vol. 1, Article 1.* 2018.

[20] Victor Podlozhnyuk. Image convolution with cuda. *NVIDIA*, 2007.

[21] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. Efficient processing of deep neural networks: A tutorial and survey, 2017. URL: https://arxiv.org/abs/1703.09039, arXiv:1703.09039.

[22] Vasily Volkov. Better performance at lower occupancy. *Proceedings of the GPU Technology Conference, GTC*, 10, 01 2015.

[23] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009. doi:10.1145/1498765.1498785.

[24] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning, 2023. URL: https://arxiv.org/abs/2106.11342, arXiv:2106.11342.

[25] Jiyuan Zhang, Franz Franchetti, and Tze Meng Low. High performance zero-memory overhead direct convolutions. *CoRR*, abs/1809.10170, 2018. URL: http://arxiv.org/abs/1809.10170, arXiv:1809.10170.

# A AI-Declaration

**Declaration of using generative AI tools (for students)**

☐ **I/we have used generative AI as an aid/tool** *(please tick)*

☑ **I/we have NOT used generative AI as an aid/tool** *(please tick)*

*If generative AI is permitted in the exam, but you haven't used it in your exam paper, you just need to tick the box stating that you have not used GAI. You don't have to fill in the rest.*

**List which GAI tools you have used and include the link to the platform (if possible):**

*Example: [Copilot with enterprise data protection (UCPH license), https://copilot.microsoft.com]*

**Describe how generative AI has been used in the exam paper:**

1) *Purpose (what did you use the tool for?)*
2) *Work phase (when in the process did you use GAI?)*
3) *What did you do with the output? (including any editing of or continued work on the output)*

*Please note:* Content generated by GAI that is used as a source in the paper requires correct use of quotation marks and source referencing. Read the guidelines from Copenhagen University Library at KUnet.



Figure 34: AI-declaration

# B CUDA-versions

Figure 35: Direct mapped

```
template <class ElTp, int radius>
__global__ void convNaive (ElTp* input, ElTp* kernel, ElTp*
↪  output, int height, int widthIn, int widthOut, int nIn,
↪  int N_out){
    int gidx = blockIdx.x * blockDim.x + threadIdx.x;
    int gidy = blockIdx.y * blockDim.y + threadIdx.y;
    int gidz = blockIdx.z * blockDim.z + threadIdx.z;
    if (gidx >= widthOut || gidy >= widthOut || gidz >= N_out){
        return;
    }
    ElTp sum = 0.0f;
    int r_size = 2*radius+1;
    for (int c_in = 0; c_in < nIn; c_in++) {
        for (int row = 0; row < r_size; row += 1) {
            for (int col = 0; col < r_size; col += 1 ) {
               int InCol = gidx + col;
               int InRow = gidy + row;
                sum += input[c_in * widthIn * widthIn + InRow *
                ↪  widthIn + InCol] *
                kernel[c_in * N_out * r_size * r_size + gidz *
                ↪  r_size * r_size + row * r_size + col];
            }
        }
    }
    output[gidz * widthOut * widthOut + gidy * widthOut + gidx]
    ↪  = sum;
}


template<class ElTp, int radius, int Ry,int Rz>
__global__ void conv2DTiledRM (ElTp* input, ElTp* kernel, ElTp*
↪  output, int height, int widthIn, int widthOut, int N_in, int
↪  N_out) {
```

```
int gidx  = blockIdx.x * blockDim.x + threadIdx.x;
int y0 = (blockIdx.y * blockDim.y + threadIdx.y) * Ry;
int c_out0 = blockIdx.z * blockDim.z;
if (blockDim.z != 1) {
    c_out0 += threadIdx.z;
}
c_out0 = c_out0 * Rz;

ElTp tmp[Ry][Rz];
#pragma unroll
for (int i =0; i< Ry*Rz; i++){
    #pragma unroll
    for (int j =0; j<Rz; j++)
        tmp[i][j] = 0.0;
}
if (gidx >= widthOut || y0 >= widthOut || c_out0 >= N_out) {
    return;
}
int r_size = 2*radius+1;
for (int c_in = 0; c_in < N_in; c_in++) {
    #pragma unroll
    for (int iy = 0; iy < Ry; iy ++) {
        int y = y0 + iy;
        #pragma unroll
        for (int ic = 0; ic < Rz; ic++) {
            int c_out =  c_out0 + ic;
            #pragma unroll
            for (int i = 0; i < r_size; i ++) {
                #pragma unroll
                for (int j = 0 ; j < r_size; j++ ){
                    if (y < widthOut && c_out < N_out){
                        tmp[iy][ic] += input[c_in * widthIn *
                        ↪  widthIn + (y+i) * widthIn +
                        ↪  (gidx+j)] *
                        kernel[c_in * N_out * r_size * r_size +
                        ↪  c_out * r_size * r_size + i *
                        ↪  r_size + j];
                    }
                }
            }
```

```
            }
        }
    }
    #pragma unroll
    for (int iy = 0; iy < Ry; iy ++) {
        int y = y0 + iy;
        #pragma unroll
        for (int ic = 0; ic < Rz; ic++) {
            int c_out = c_out0 + ic;
            if (c_out < N_out && y < widthOut){ // can be removed
            ↪  with padding
                output[c_out * widthOut * widthOut + y * widthOut +
                ↪  gidx] = tmp[iy][ic];
            }
        }
    }
}
```

## B.1 Shared Memory Vector Load Version

```
template <class T, class LoadType, int FilterRadius, int Tx, int Ty, int Tz, int Rz, int Ry>
__global__ void convolution_shared_with_reduction(
  T* __restrict__ input,
  T* __restrict__ filters,
  T* __restrict__ out,
  int in_num_channels,
  int in_height,
  int in_row_pitch,
  int filters_in_channel_pitch,
  int filters_block_pitch,
  int out_num_channels,
  int out_width,
  int out_height,
  int out_row_pitch
) {
  constexpr int CoutPerBlock = Tz * Rz;
  constexpr int YPerBlock = Ty * Ry;
  constexpr int ElementsPerLoadType = sizeof(LoadType) / sizeof(T);

  constexpr int FilterWidth = 2 * FilterRadius + 1;
  constexpr int FiltersSharedSize = CoutPerBlock * FilterWidth * FilterWidth;
  constexpr int InSharedWidth = Tx + 2 * FilterRadius;
  constexpr int InSharedPitch = ROUND_UP_TO_MULTIPLE_OF(InSharedWidth, ElementsPerLoadType);
  constexpr int InSharedHeight = YPerBlock + 2 * FilterRadius;
  constexpr int InSharedSize = InSharedPitch * InSharedHeight;
  constexpr int NumThreads = Tx * Ty * Tz;

  constexpr int FiltersNumLoadType = (FiltersSharedSize + ElementsPerLoadType - 1) / ElementsPerLoadType;

  constexpr int InSharedLoadTypePerRow = (InSharedWidth + ElementsPerLoadType - 1) / ElementsPerLoadType;
  constexpr int NumLoadTypes = InSharedLoadTypePerRow * InSharedHeight;

  volatile extern __shared__ int shared_convolution_shared_load_LoadType[];
  T* in_shared = (T*)shared_convolution_shared_load_LoadType;
  T* filters_shared = &in_shared[ROUND_UP_TO_MULTIPLE_OF(InSharedSize, NumThreads * ElementsPerLoadType)];

  const int block_offset_x = blockIdx.x * Tx;
  const int block_offset_y = blockIdx.y * YPerBlock;
  const int filters_block_offset = blockIdx.z * filters_block_pitch;

  const int thread_id = threadIdx.x + threadIdx.y * Tx + threadIdx.z * Tx * Ty; // [0 : num_threads_per_block)

  T tmp[Ry][Rz];
#pragma unroll
  for (int i = 0; i < Ry; i++) {
#pragma unroll
    for (int j = 0; j < Rz; j++) {
      tmp[i][j] = 0.0;
    }
  }

  for (int in_channel = 0; in_channel < in_num_channels; in_channel++) {
#pragma unroll
    for (int load_start = 0; load_start < NumLoadTypes; load_start += NumThreads) {
      const int id = load_start + thread_id;
      const int block_y = id / InSharedLoadTypePerRow;
      const int block_x = (id % InSharedLoadTypePerRow) * ElementsPerLoadType;
      ((LoadType*)in_shared)[id] =
        *(LoadType*)&input[device_index_arr_of_arr(in_row_pitch, in_height, in_channel, block_offset_y + block_y,
        ↪  block_offset_x + block_x)];
    }
#pragma unroll
    for (int load_start = 0; load_start < FiltersNumLoadType; load_start += NumThreads) {
      const int id = load_start + thread_id;
      ((LoadType*)filters_shared)[id] =
        *(LoadType*)&filters[filters_in_channel_pitch * in_channel + filters_block_offset + id * ElementsPerLoadType];
    }
    __syncthreads();

#pragma unroll
    for (int loop_y = 0; loop_y < Ry; loop_y++) {
      const int y = threadIdx.y * Ry + loop_y;
#pragma unroll
      for (int thread_cout = 0; thread_cout < Rz; thread_cout++) {
#pragma unroll
```

```
        for (int i = 0; i < FilterWidth; i++) {
#pragma unroll
          for (int j = 0; j < FilterWidth; j++) {
            tmp[loop_y][thread_cout] +=
              in_shared[InSharedPitch * (y + i) + threadIdx.x + j] *
              filters_shared[FilterWidth * FilterWidth * (Rz * threadIdx.z + thread_cout) + FilterWidth * i + j];
          }
        }
      }
    }
    __syncthreads();
  }

#pragma unroll
  for (int loop_y = 0; loop_y < Ry; loop_y++) {
    const int y = block_offset_y + Ry * threadIdx.y + loop_y;
#pragma unroll
    for (int thread_cout = 0; thread_cout < Rz; thread_cout++) {
      const int cout = CoutPerBlock * blockIdx.z + Rz * threadIdx.z + thread_cout;
      if (cout < out_num_channels && y < out_height && block_offset_x + threadIdx.x < out_width) {
        out[device_index_arr_of_arr(out_row_pitch, out_height, cout, y, block_offset_x + threadIdx.x)] =
        ↪  tmp[loop_y][thread_cout];
      }
    }
  }
}
```